PipeUp: WebRTC basiertes tooling zur Ablösung dedizierter Zuschauer-Mikrofone bei Live-Events

Masterarbeit

im Studiengang
Computer Science and Media

vorgelegt von

Willi Kampe
Matr.-Nr.: 25873

am 1. August 2014
an der Hochschule der Medien Stuttgart

Erstprüfer/in:	Prof. W. Kriha
Zweitprüfer/in:	Dipl.-Vw. T. Hartmann

Abstract

Diese Arbeit beschäftigt sich mit dem Thema WebRTC, also der Peer-to-Peer-Datenübertragung im Webbrowser. Es wird untersucht ob diese junge Technologie bereits einen Entwicklungsstand erreicht hat, der einen produktiven Einsatz rechtfertigen würde.

Dazu ist ein Prototyp entwickelt worden, welcher während einem Live-Event den Einsatz von Zuschauermikrofonen überflüssig machen soll. Die digitale Aufzeichnung der Zuschauerfragen wird dabei von den mobilen Endgeräten (Smartphone, Tablet, Laptop, etc.) der Fragenden übernommen, da diese heutzutage weitestgehend mit Mikrofon und Kamera bestückt sind. Die damit aufgezeichneten Daten werden über eine WebRTC-Verbindung zu einem zentralen Host geleitet und von dort aus weiterverarbeitet.

Die Umsetzung drei unterschiedlich komplexer Szenarien in diesem Open-Source-Projekt hat gezeigt, dass die Technologie bereits einen sehr erwachsenen Stand erreicht hat und für diese Anwendungsfälle bereits alle Funktionen unterstützt.

Gefahren liegen in dem unfertigen Standard, der sich noch grundlegende Änderungen vorbehält und bei der noch fehlenden Unterstützung von Internet Explorer und Safari.

Verbesserungspotenzial gibt es bei der Manipulation von SDP-Paketen während des Verbindungsaufbaus, diese werden in einem JavaScript unfreundlichem Text-Format übertragen, was die Anpassung einzelner Parameter erschwert.

Schlagwörter: WebRTC, Web Real-Time Communications, PipeUp, Signaling, STUN, TURN, ICE, SDP, SIP, VoIP

Inhaltsverzeichnis

Abbildungsverzeichnis

Code-Beispiel-Verzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

WebRTC	Web Real-Time Communications
WebSocket	TCP basierendes Netzwerkprotokoll
RTP und SRTP	Real-Time Transport Protocol und Secure RTP
SDP	Session Description Protocol
NAT	Network Address Translation
STUN	Session Traversal Utilities for NAT
TURN	Traversal Using Relays around NAT
ICE	Interactive Connectivity Establishment
TLS	Transport Layer Security
TCP	Transmission Control Protocol
DTLS	Datagram TLS
UDP	User Datagram Protocol
SCTP	Stream Control Transport Protocol
W3C	World Wide Web Consortium
IETF	Internet Engineering Task Force
API	Application Programming Interface

Vorwort

Die Materie dieser Arbeit ist sehr technisch geprägt und die Fachtermini in diesem Bereich sind sehr von der englischen Sprache beeinflusst. Deshalb wird auf eine Übersetzung ins Deutsche an den Stellen verzichtet, die nicht dem allgemeinen Sprachgebrauch entsprechen.

Voraussetzungen

Für das Verständnis dieser Arbeit werden Grundkenntnisse in den Netzwerkprotokollen TCP, IP und UDP vorausgesetzt, außerdem ist ein Grundverständnis vom technischen Aufbau großer Netzwerksysteme wie dem World Wide Web wichtig.

Begriffsdefinition

An einigen Stellen dieser Arbeit wird von einer Q&A-Session an der Teilnehmer beteiligt sind gesprochen. Damit ist im weitesten Sinne das Anwendungsszenario der Prototypen gemeint und kann synonym für jede Art von Feedback-Runde nach einem Live-Event angesehen werden. Q&A steht für Question and Answer dt. Frage und Antwort.

Ein weiterer erläuterungswürdiger Begriff ist Peer, dessen sinngemäße Übersetzung „gleichberechtigter Teilnehmer“ ist. Im Kontext dieser Arbeit wird damit der Benutzer als Person bezeichnet welcher gewillt ist, eine WebRTC-Verbindung zu einem anderen Benutzer aufzubauen. Analog dazu ist aus technischer Sicht auch der Webbrowser, der an einer WebRTC-Session teilnimmt, mit Peer gemeint.

Danksagungen

Ich möchte mich an dieser Stelle bei allen bedanken, die mich bei diesem Projekt unterstützt haben.

Insbesondere Herrn Prof. Kriha für immer offene Ohren und wertvolle Anregungen.

Tobias Hartmann für das Einbringen seiner wertvollen Erfahrungen und die Betreuung dieser Arbeit.

Der Firma Weitclick, die mich nicht nur im Rahmen dieser Arbeit, sondern durch meine gesamte Master-Studienzeit hindurch, mit hoher Flexibilität und tollen Aufgaben, immer wieder neu inspiriert hat.

Überblick

Im ersten Teil dieser Arbeit wird zunächst ein Einstieg in das Thema WebRTC gegeben. Dabei wird zunächst die Motivation für diese Arbeit erklärt und anschließend wichtiges Grundlagenwissen vermittelt. Dazu kommt die Vorstellung des praktischen Teils dieser Arbeit, die Idee des entwickelten Prototyps mit dem Namen PipeUp wird theoretisch erläutert.

Der zweite Teil geht technisch ins Detail und erläutert alle Elemente, die unter dem Oberbegriff WebRTC zusammengefasst sind. Zusätzlich wird ein Blick auf mögliche Architekturen geworfen, die sich mit der neuen Technologie ermöglichen und auf das Thema Sicherheit eingegangen.

Im dritten Teil werden die theoretischen Aspekte aus dem zweiten Teil in einen praktischen Kontext gestellt. Dazu wurde das Projekt PipeUp prototypisch umgesetzt, dieses Kapitel erläutert alle wichtigen Details bis hin zu den Arbeitsschritten, die für eine Weiterentwicklung nötig wären.

Teil 4 beschäftigt sich mit dem, was sich in dieser Arbeit außer dem theoretischen Wissen über WebRTC zusätzlich als auffällig und nützlich erwiesen hat.

Der letzte Teil rundet diese Arbeit damit ab, dass die Zukunft dieser jungen Technologie betrachtet und ein Fazit gezogen wird.

1 Einleitung

1.1 Motivation

Seit einigen Jahren macht eine Technologie auf sich aufmerksam, mit der es möglich ist, den Webbrowser um Echtzeitkommunikationsfähigkeit zu erweitern. Dadurch werden Anwendungen wie Videotelefonie oder Filesharing-Dienste möglich, ohne das zusätzliche Plugins für den Webbrowser installiert werden müssen. Die Rede ist von WebRTC.

Hinter dieser als Web Real-Time Communications bezeichneten Technologie steckt der klassische Peer-to-Peer Gedanke, also einer Ende-zu-Ende-Verbindung zwischen zwei Computern in einem Netzwerk. Gegenüber der klassischen Client-Server-Architektur hat das einige Vorteile. Durch den direkten Weg zwischen zwei Webbrowsern werden Latenzzeiten deutlich verringert und es muss kein Server mit dem Weiterleiten von Daten beschäftigt werden. Dieses Plus an Geschwindigkeit lässt Szenarien denkbar werden, welche als Real-Time-Anwendungen bezeichnet werden, so zum Beispiel Videotelefonie oder Online-Gaming-Anwendungen.

Schon heute gibt es prominente Anwendungsbeispiele in denen WebRTC zum Einsatz kommt. In Googles Chromecast wird WebRTC für das Spiegeln von Webseiten auf Anzeigegeräte wie z.B. Fernseher oder Video-Beamer eingesetzt. [Derek Ross, 2013]

Amazon betreibt ein Customer-Service-Support-Tool namens Mayday, welches seit dem 22.07.2014 auch für den deutschen Markt gestartet worden ist. Damit ist es bei einem Amazon-Einkauf möglich Unterstützung von einem Mitarbeiter zu bekommen.

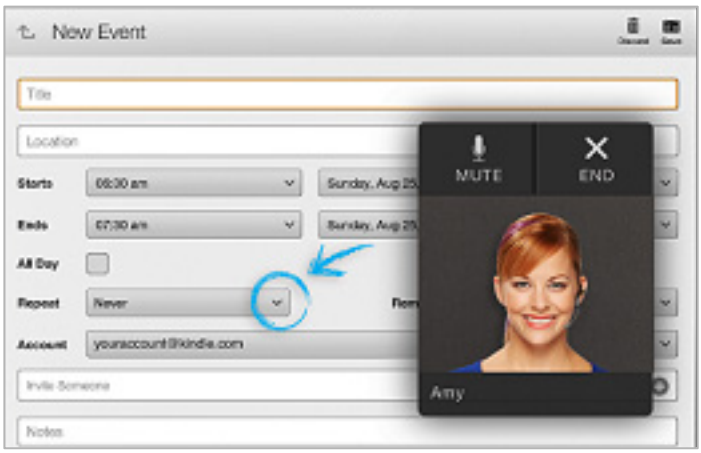


Abbildung 1: Amazon Mayday-Funktion

Dieser Mitarbeiter wird mit dem Kunden live und direkt verbunden. Wie in Abbildung 1 zu sehen, wird sogar ein Live-Video des Mitarbeiters übertragen. Da Amazon sich mit

Informationen zur technischen Umsetzung sehr zurückhält, können nur Vermutungen, mittels Reverse Engineering, angestellt werden. Es gilt allerdings als erwiesen, dass zumindest für die Übertragung des Video-Streams WebRTC eingesetzt wird. [Chad Hart, 2014]

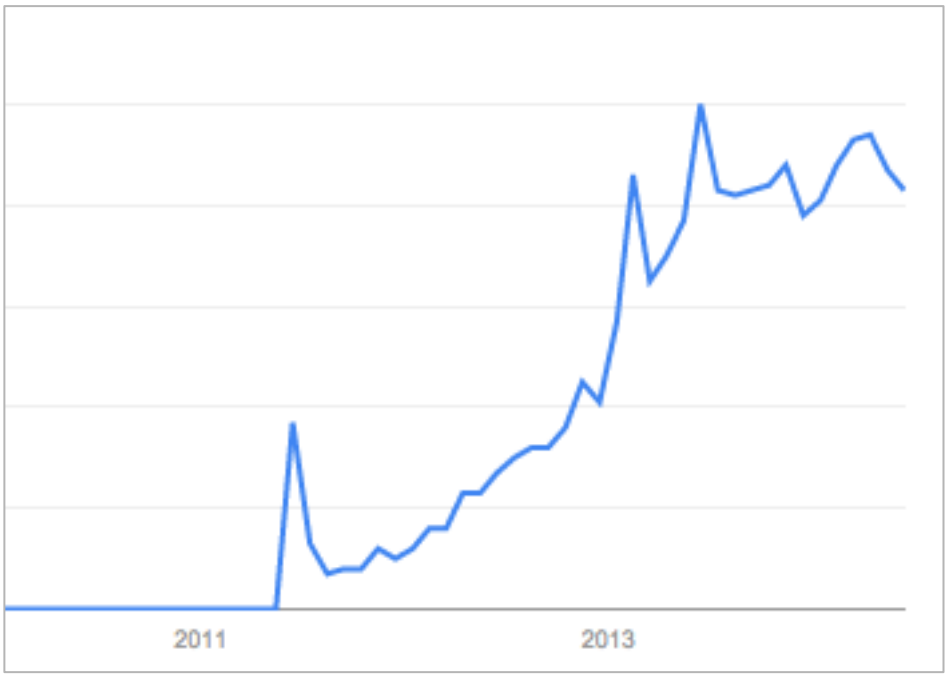


Abbildung 2: Google Trends Analyse für das Stichwort WebRTC (Stand: 28.06.14)

In Abbildung 2 ist zu sehen, wie die Aufmerksamkeit zu diesem Thema in den letzten Jahren enorm zugenommen hat. Die Google Trend Analyse stützt sich dabei auf die Häufigkeit der Google-Suchanfragen zu dem Begriff WebRTC.

Bei der Entwicklung dieses Standards wird sehr viel Wert auf Sicherheit, Offenheit und Unabhängigkeit gegenüber Dritt-Anbietern gelegt. Wobei es immer noch andauernde Probleme gibt, beispielsweise bei der Frage welcher Video-Codec eingesetzt werden soll. Der Standard ist also noch nicht als solcher, sondern vielmehr als Pre-Standard[1] einzustufen.

1.1.1 Geschichte

Begonnen hat die Geschichte von WebRTC Ende 2009, als Google sich nach Marktnischen umschaute, in denen Webbrowser noch nicht mit Desktop-Anwendungen konkurrieren konnten. Dabei war das Anwendungsgebiet der Real-Time-Communication (RTC) aufgefallen.

[1] Pre-Standard Implementation: Als solche werden Umsetzungen von Standards, die sich noch in der Entwicklung befinden bezeichnet, häufig auch als Proof-of-Conecpt.

Bis heute ist es nicht trivial, seinen Benutzern RTC-Funktionalität im Webbrowser anzubieten. Bisher wurden dafür eigene Plugins entwickelt, welche zunächst installiert werden mussten. Beispielsweise Flash, Google Hangouts (basierend auf dem Google Talk Service), Skype oder Facebook (basiert auf Skype).

Im Mai 2010 hat dann Google das Unternehmen Global IP Solutions (GIPS) aufgekauft. Dieses war maßgeblich für die Entwicklung wichtiger Komponenten, unter anderem Echo-Canceling und diverser Video- und Audio-Codecs verantwortlich. Diese hat Google dann unter Open-Source-Lizenzen veröffentlicht und erste Standards bei der IETF[2] und dem W3C[3] eingereicht.

Genau ein Jahr später im Mai 2011 hat dann Ericsson die erste Implementierung von WebRTC im WebKit GTK+[4] veröffentlicht. [Ericsson Labs, 2011]

Am 1. Juni 2011 veröffentlichte Google die Plattform http://webrtc.org/, um die Popularität dieser neuen Technologie zu steigern.

Im November 2012 wurde WebRTC das erste Mal in der stabilen Version von Google Chrome verfügbar. Im Juni 2013 dann zog der Firefox gleich und Opera folgte im November des Jahres 2013.

1.1.2 Ein wenig Statistik

Das Unternehmen Bistri versteht sich selbst als Aggregator von vielen Instant Messaging (IM) Protokollen und Distributionen und setzt seit 2012 auch auf WebRTC für Videoanrufe.

Waren es im März 2013 noch 13% aller Anrufe über Bistri, die über WebRTC abgewickelt wurden, sind es inzwischen schon 76% (Stand Mai 2014). Das zeigt wie schnell die Akzeptanz und Verbreitung der Technologie voranschreitet. [webrtcstats.com, 2014]

1.1.3 Elemente von WebRTC

Bei der Entwicklung dieses Standards werden auch Technologien berücksichtigt, die bisher nicht direkt mit Browsern zu tun hatten. Dazu gehören beispielsweise Gateways ins Festnetz, also in das Public Switched Telephone Network (PSTN) oder zu Voice-over-IP-Diensten (VoIP), wie SIP-Telefonen oder Jingle-Clients für XMPP.

WebRTC ist an dieser Stelle also ganz klar darauf ausgerichtet, die volle Bandbreite der Echtzeitkommunikation über das Internet abzudecken. In diese Arbeit liegt der Focus aber auf der Peer-to-Peer-Kommunikation zwischen zwei oder mehreren Webbrowsern.

---

2 IETF: The Internet Engineering Task Force - http://www.ietf.org/

3 W3C: World Wide Web Consortium - http://www.w3.org/

4 WebKit GTK+: GNOME Portierung der WebKit Rendering Engine

1.1.4 Die Browserfrage

Da WebRTC heute und in Zukunft ein sehr wichtiges Thema im Internet ist, bleibt die Frage offen, warum die beiden großen fehlenden Hersteller Microsoft (Internet Explorer) und Apple (Safari) noch keine Unterstützung liefern. Bei beiden Herstellern liegt die Vermutung nahe, dass sie ihren Produkten Skype und FaceTime mit der Unterstützung dieses Standards unnötig Konkurrenz machen würden, und diesen Schritt deshalb noch nicht gegangen sind.

	IE 11.0	Firefox 33.0	Chrome 38.0	Safari 8.0	Opera 24.0
UserMedia / RTCPeerConnection	Nein	Ja	Ja	Nein	Ja

Tabelle 1: WebRTC-Unterstützung Desktop-Webbrowser

	iOS Safari 8.0	Opera Mini 7.0	Opera Mobile 22.0	Chrome (Android) 35.0	Firefox (Android) 30.0	IE Mobile 10.0
UserMedia / RTCPeerConnection	Nein	Nein	Ja	Ja	Ja	Nein

Tabelle 2: WebRTC-Unterstützung Mobile-Webbrowser

1.2 PipeUp

Vor dem Hintergrund der wachsenden Popularität von WebRTC, ist die Idee zu PipeUp entstanden. PipeUp hat sich als Ziel gesetzt die Mikrofonproblematik bei Q&A-Sessions zu optimieren.

1.2.1 Das Problem

Das Problem ist an der Hochschule des Autors aufgetreten, kann aber vom Prinzip her überall dorthin übertragen werden, wo im Anschluss eines Live-Events das Publikum in einer Q&A-Session Fragen an den Referenten stellen kann. Damit auch die Zuschauer aus dem Live-Stream diese Fragen verstehen können, sollte alles was im Rahmen dieser Q&A-Session gesprochen wird, von einem Mikrofon erfasst werden. Für diesen Zweck wird bisher ein Handmikrofon herumgereicht, aber durch die ständigen Unterbrechungen und Verzögerungen ist dieser Prozess zeitaufwändig und behindert einen kreativen Gedankenaustausch. Es können keine lebhaften Diskussionen entstehen oder sie sind nicht vom Live-Stream aus zu verfolgen.

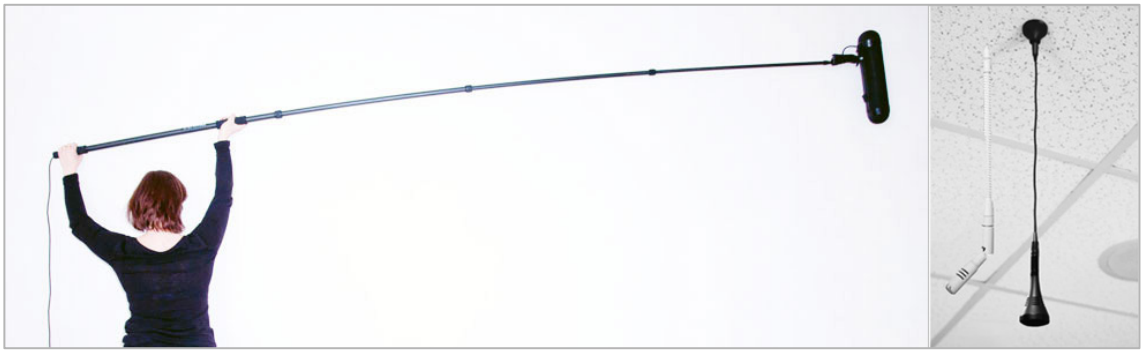


Abbildung 3: Tonangel (links) und Deckenmikrofone (rechts)

Auch professionelle Lösungen wurden bereits in Erwägung gezogen, sind aber aus Kostengründen im Hochschulkontext nicht umsetzbar. Eine Tonangel würde zusätzlich Personal- und Schulungsaufwand generieren. Bei Deckenmikrofonen kommt hinzu, dass sie stationär installiert werden müssen. Mobilität ist aber eine wichtige Anforderung des eingesetzten Equipments, da die Events nicht immer am gleichen Ort stattfinden.

1.2.2 Eine Lösung

Eine bereits von Prof. Kriha prototypisch umgesetzte Lösung für dieses Problem, ist ein Schaumstoffball mit integriertem Mikrofon. Dieser kann zwischen den Fragenden hin und her geworfen werden, was den Zeitaufwand deutlich reduziert. Zusätzlich sorgen die Bälle für eine positive User Experience (UX), da beim Anwender durch das Werfen bzw. Fangen des Balls eine Spannung erzeugt wird. Eine Art der Gamifizierung[5] des Problems.

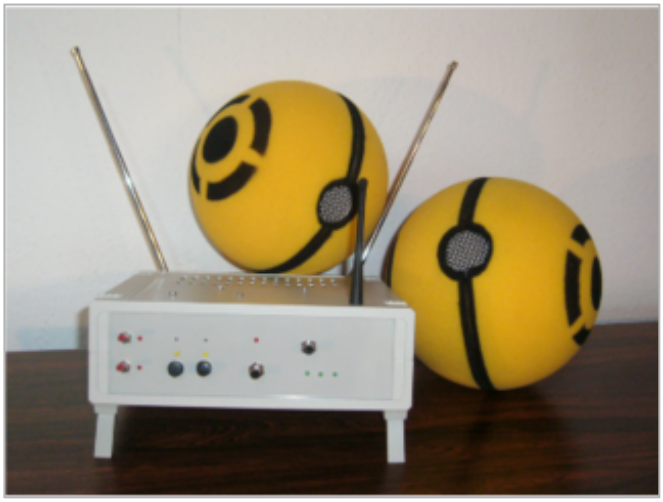


Abbildung 4: Mikrofon-Bälle [Kriha, 2013]

[5] Gamifizierung: Darunter wird die Anreicherung eines spielfremden Kontexts um spieltypische Elemente verstanden, um eine Motivationssteigerung beim Benutzer zu erzielen.

Allerdings hat diese Lösung auch ihre Schwächen, der Benutzer muss nach dem Fangen des Balls zunächst die Stelle suchen, an welcher er in das Mikrofon sprechen kann, außerdem sind häufige Anweisungen zur Handhabung nötig. Aus der Praxis hat sich auch die mangelnde Ballfertigkeit einiger Teilnehmer als problematisch erwiesen.

1.2.3 Die neue Lösung

PipeUp[6] ist eine Softwarelösung, welche das Publikum im Hörsaal und die Teilnehmer aus dem Live-Stream in einem virtuellen Raum versammelt. Dazu wird ein Smartphone, Tablet oder Laptop mit einem aktuellen Webbrowser und WebRTC-Unterstützung benötigt. Jeder Benutzer hat in der PipeUp-Session die Möglichkeit, an einem Text-Chat zu partizipieren und über die PipeUp-Funktion zu signalisieren, dass er oder sie sich zu Wort melden möchte. Diese Aktion wird in der Teilnehmerliste visuell kenntlich gemacht und es entsteht ganz nach dem FIFO-Prinzip[7] eine Reihenfolge der Wortmeldungen. In Abbildung 5 wird dies in der Teilnehmerliste mit einem roten Ausrufezeichen neben dem Benutzernamen symbolisiert. In diesem Beispiel möchte Andre etwas beitragen.

Als zentrale Verwaltungsoberfläche dient ein eigenes User-Interface, welches dem Verwalter der Session, nachfolgend mit Host bezeichnet, erlaubt eine Audio/Video-Verbindung zu einem Benutzer aufzubauen.

Dieser Ansatz hat den großen Vorteil gegenüber den anderen Lösungen, dass für die Q&A-Session keine dedizierten Mikrofone mehr mitgebracht, vorbereitet, installiert, herumgereicht oder hingehalten werden müssen. Heutzutage tragen die meisten Teilnehmer mobile Endgeräte mit eingebauten Mikrofonen in ihren Hosentaschen und sind, gerade im Hochschulumfeld, vertraut damit diese Geräte zu bedienen.

Nun wäre diese Lösung auch schon seit ein paar Jahren denkbar gewesen, nur hätte man bisher von den Teilnehmern erwarten müssen, dass sie ein Plugin auf ihrem Endgerät haben oder nachträglich installieren. Möglichkeiten über Skype, Google Hangouts oder Flash gibt es nämlich schon seit einigen Jahren. Mit WebRTC kommt nun aber eine nativ integrierte Technologie in die Webbrowser, so dass keine Installation von Plugins mehr nötig ist. Der Aufwand für den Benutzer beschränkt sich dadurch auf den Besuch einer Webseite und das Eingeben seines Benutzernamens.

---

[6] PipeUp: Ursprung des Namens ist der englische Begriff „to pipe up“ dt. „sich zu Wort melden“

[7] FIFO: First In – First Out bedeutet soviel wie „der Reihe nach“

Ein weiterer erheblicher Vorteil gegenüber der etablierten Lösung ist die Möglichkeit, dem Benutzer, welcher das Event über den Internet-Live-Stream verfolgt, besser in die Diskussion einzubinden. Er wird durch den neuen Ansatz in die Lage versetzt, seine Frage live und persönlich zu stellen. Bisher hat diese Aufgabe ein Moderator für ihn übernommen, der seine Frage im Live-Chat gelesen hat.

Durch den Einsatz von WebRTC gibt es nun die Möglichkeit, neben dem Ton auch bewegte Bilder zu übertragen. Bisher ist es so, dass eine Kamera das Event aufgezeichnet hat und dieses dann von einem Laptop an einen Streaming-Server übertragen wird. Dieser übernimmt dann die Distribution der Aufzeichnung zu den verbundenen Zuschauern aus dem Internet. Es ist denkbar, dass in Zukunft die Aufzeichnung der Session von der PipeUp-Anwendung übernommen wird, da die Host-Anwendung ebenfalls auf die lokale Kamera zugreifen kann. PipeUp könnte als Multiplexer[8] der einzelnen Audio und Videosignale fungieren und den erzeugten Stream direkt an den Streaming-Server weiterleiten. Das würde positiver Weise zu einer Konsolidierung der eingesetzten Soft- und Hardware führen und das ganze Videostreaming um Live-Kommentare von Zuschauern in Bild und Ton erweitern.

1.2.4 Szenarien

Aus dem im Vorfeld beschriebenen Aufbau sind für diesen Anwendungsfall drei Szenarien genauer untersucht und technisch im Rahmen dieser Arbeit umgesetzt worden. Nachfolgend wird deren praktischer Ablauf einzeln erläutert, während im dritten Kapitel dann die technische Dimension hinzukommt.

[8] Multiplexverfahren: Der Begriff kommt aus der Signal- und Nachrichtenübertragung und beschreibt den Vorgang mehrere Datenströme zu einem zusammenzuführen.

Szenario 1:

In diesem Fall sind die Teilnehmer der Q&A-Session, auf das Publikum vor Ort beschränkt. Die Session wird mit Hilfe eines Laptops und der dort angeschlossenen Kamera und einem Mikrofon von der PipeUp-Hostanwendung aufgezeichnet.

Damit nun auch die Fragen der Teilnehmer mit aufgezeichnet werden, loggen diese sich im Vorfeld über die Eingabe einer URL in den Webbrowser ihres mobilen Endgerätes in die PipeUp-Session ein.

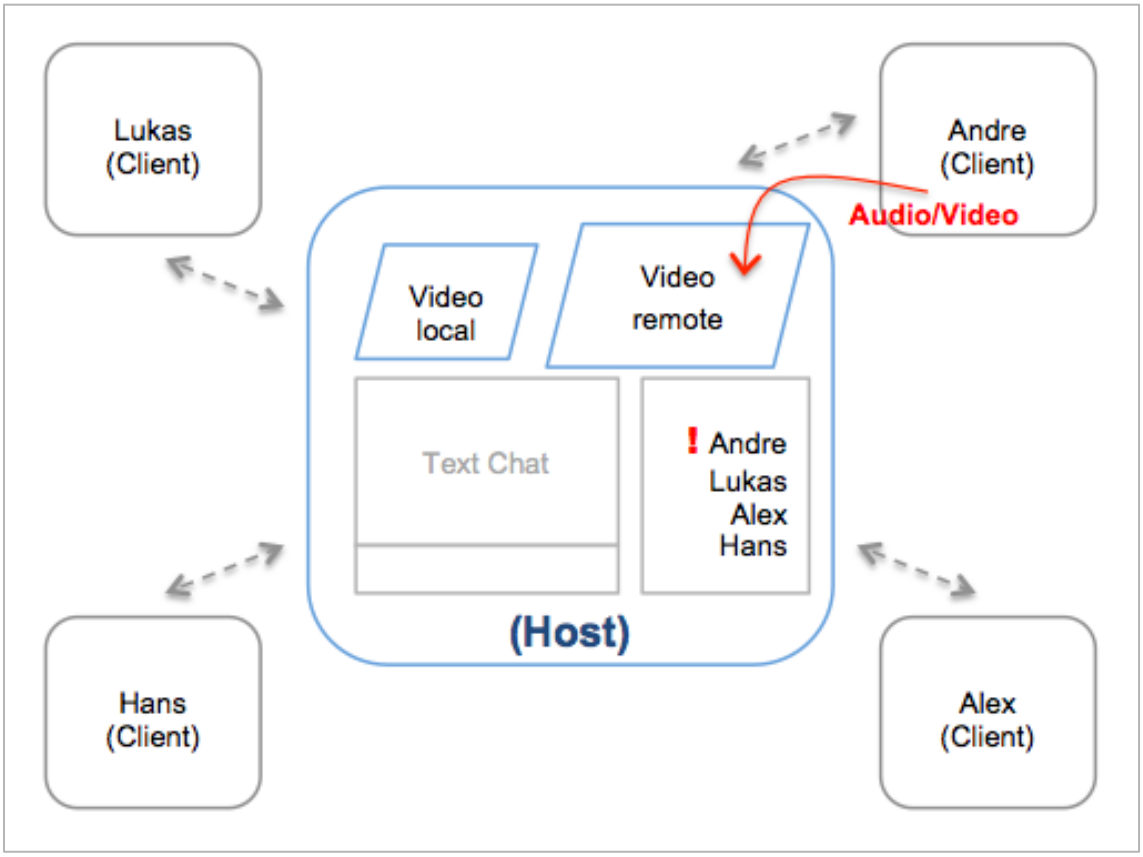


Abbildung 5: PipeUp Szenario 1

In Abbildung 5 ist eine PipeUp-Session schematisch dargestellt. Es sind bereits vier Teilnehmer in der Session, wobei Andre bereits die PipeUp-Funktion betätigt hat. Entscheidet sich nun der Moderator dazu, Andre zu Wort kommen zu lassen, wird über die WebRTC-Verbindung Andres Kamera und Mikrofon angefragt und in dem Bereich „Video remote“ angezeigt. Ist Andre nun fertig mit seinen Ausführungen, wird die Verbindung wieder abgebaut und der nächste Teilnehmer kann verbunden werden.

Szenario 2:

Das zweite Szenario baut auf das erste auf und erweitert den adressierten Benutzerkreis um Teilnehmer welche nicht vor Ort sind, sich aber trotzdem an dem Event beteiligen wollen.

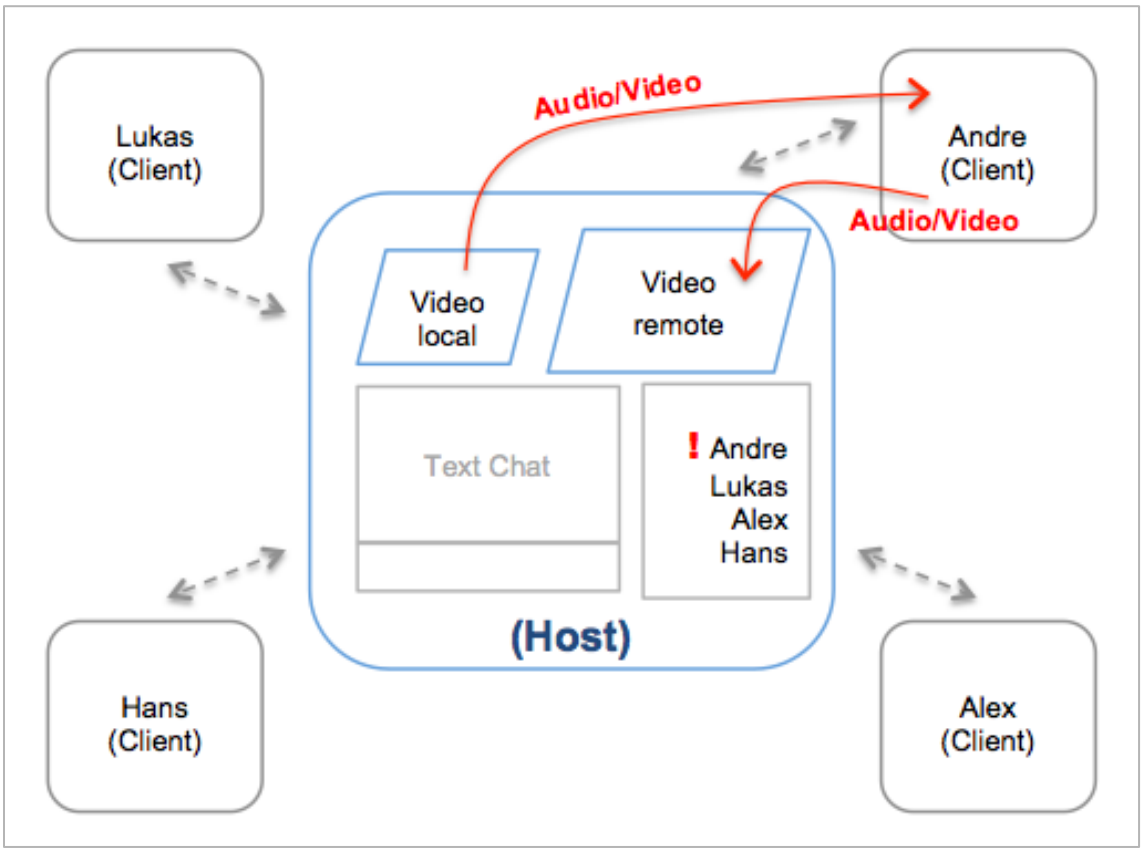


Abbildung 6: PipeUp Szenario 2

In Abbildung 6 ist nun Andre derjenige, welcher nicht Teil des Publikums vor Ort ist, sondern zu Hause vor dem Computer an seinem Schreibtisch sitzt. Um dem Event folgen zu können, bekommt Andre die lokale Aufzeichnung des Hosts gestreamt. Seinerseits sendet er auch Audio und Video an den Host. Diese beiden Streams sind voneinander losgelöst, Andre empfängt also auch die Daten aus dem Hörsaal, wenn er selbst nichts sendet.

Szenario 3:

Dieses Szenario setzt wiederum auf das erste auf. Hinzu kommt nun, dass ein Benutzer der nicht vor Ort ist, auch gern mitbekommen würde, was Andre zu sagen hat. Zu diesem Zweck wird Andres Stream über den Host geschleift und zu dem extern befindlichen Hans gesendet.

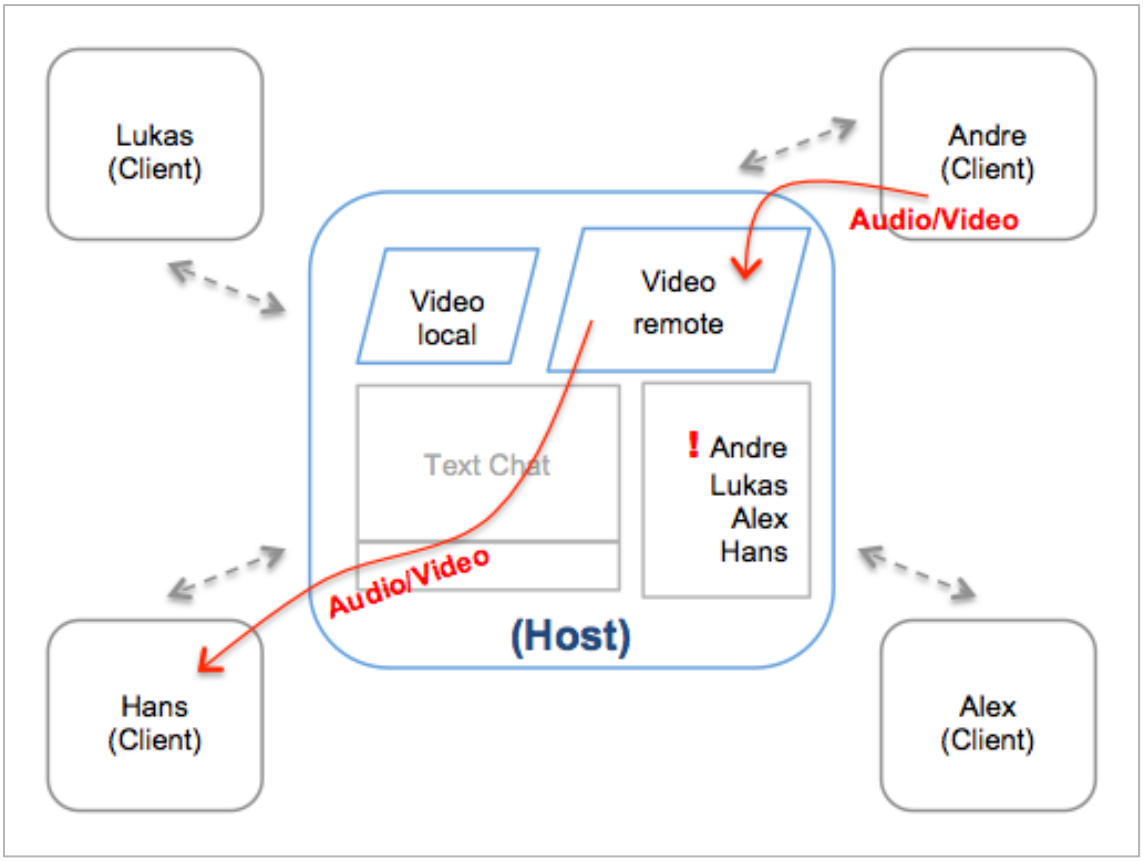


Abbildung 7: PipeUp Szenario 3

Abbildung 7 zeigt den nicht vor Ort sitzenden Hans, welcher die aufgezeichneten Daten von dem gerade sprechenden Andre weitergeleitet bekommt.

2 Technischer Hintergrund

WebRTC setzt sich aus einer Sammlung von Standards, Protokollen und JavaScript-APIs zusammen. Das folgende Kapitel beschäftigt sich mit den einzelnen Komponenten und setzt diese zu einem Gesamtbild zusammen, um dieses heterogene Umfeld verständlich zu machen.

2.1 RTC JavaScript-APIs

Als Schnittstellen werden dem Anwendungsentwickler vom Webbrowser drei JavaScript-Bibliotheken angeboten. Mit deren Hilfe kann er echtzeitfähige Kommunikationsmöglichkeiten in seine Anwendung integrieren.

MediaStream

Die MediaStream-API wird verwendet, um den Zugriff auf die Hardware des Benutzers (z.B. Kamera, Mikrofon) zu bekommen. Dafür wird die Funktion `getUserMedia()` vom Browser bereitgestellt. Bevor allerdings der Zugriff auf die Hardware erfolgen kann, muss vom Benutzer über eine Abfrage die Erlaubnis explizit erteilt werden.

RTCPeerConnection

Wie der Name schon verrät, kapselt RTCPeerConnection alles was mit der eigentlichen RTC-Verbindung zu tun hat, und stellt browserseitig eine JavaScript-API zur Verfügung. Seine Hauptaufgaben liegen darin, die Verbindung aufzubauen, zu verwalten und wieder zu beenden.

RTCDataChannel

Der RTCDataChannel ist in einer eigenen API gekapselt, da Daten, die hier übertragen werden, in der Regel einen anderen Anspruch an die Übertragungsqualität stellen. Über den DataChannel können alle möglichen Daten übertragen werden, also beispielsweise auch Dokumente im PDF-Format. Die Datenpakete dieser Dateien müssen natürlich in der richtigen Reihenfolge beim Gegenüber wieder ankommen und es dürfen keine verlorengehen. Um das zu gewährleisten, setzt der RTCDataChannel auf das Protokoll SCTP.

2.2 RTC-Funktionalität als Teil des Browsers

Der Webbrowser heutzutage hat eine Menge verschiedener Aufgaben zu bewältigen, dazu zählt das Regeln der Kommunikation zu einem Webserver oder das Interpretieren und Darstellen von Bildern, Videos, JavaScript, HTML und CSS-Dateien, um nur einige zu nennen. Diese Anforderungen sind historisch gewachsen und Stück für Stück in den Browser integriert worden. Nun kommt ein weiterer Teil hinzu, der bisher nur Browser-Erweiterungen (Plugins) vorbehalten war.

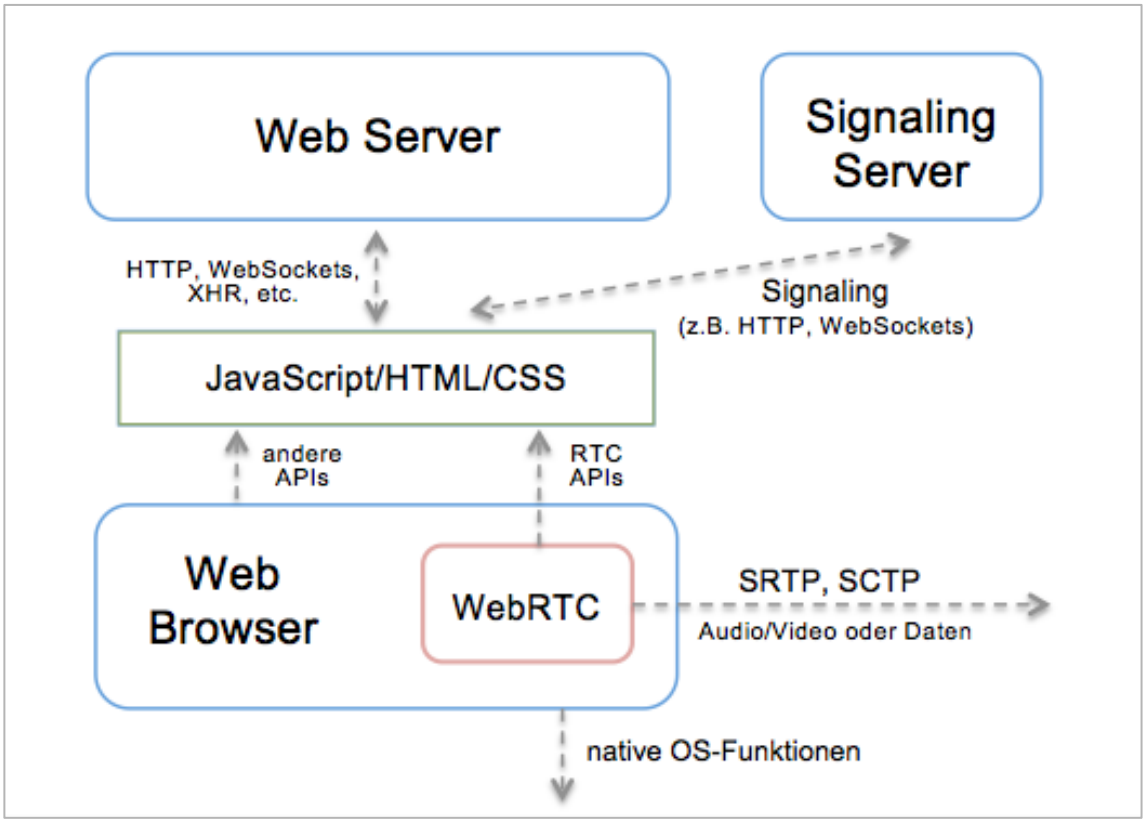


Abbildung 8: WebRTC im Technologie-Kontext

Abbildung 8 [Alan B. Johnston und Daniel C. Burnett, 2013] zeigt das Zusammenspiel der klassischen Web-Komponenten und die Eingliederung der RTC-Funktionalität in dieses Schema. WebRTC nutzt den Browser, um auf Betriebssystem-Funktionen zuzugreifen und kommuniziert mit der Web-Anwendung über standardisierte JavaScript-APIs. Neu ist, mit Hilfe der RTCPeerConnection-API, eine direkte Verbindung zu anderen Webbrowsern aufbauen zu können und dafür unabhängig von herkömmlichen Protokollen wie TCP, besser geeignete Protokolle wie SRTP und SCTP einzusetzen. Diese bauen auf das Protokoll UDP und haben deshalb einige Vorteile um Echtzeitanwendungen zu realisieren.

2.3 WebRTC Protokollaufbau

WebRTC setzt sich aus einer ganzen Reihe von Protokollen zusammen, dieses Kapitel gibt einen Überblick und zeigt die Zusammenhänge auf.

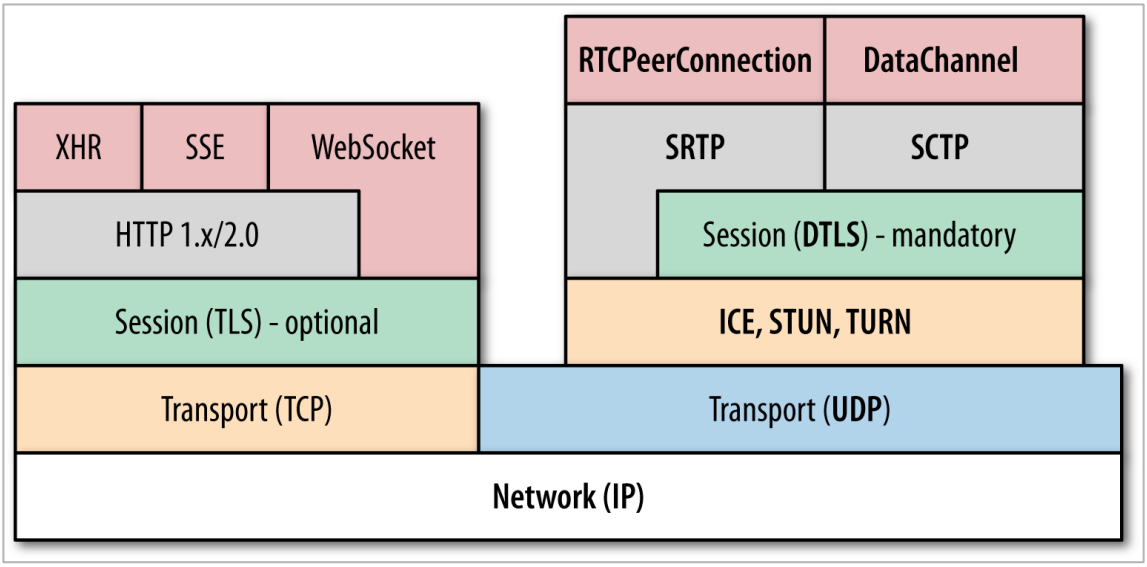


Abbildung 9: WebRTC Protokollaufbau [Ilya Grigorik, 2013]

Der Focus des linken Blocks (Abb. 9) liegt auf der Signaling-Problematik, während der rechte Block sich mit der eigentlichen RTC-Verbindung auseinandersetzt.

Die Grundlage jeder Kommunikation ist das Internet Protokoll (IP), für das Signaling wird normalerweise TCP verwendet, da die klassischen Anwendungsprotokolle wie XHR[9], SSE[10] oder WebSocket darauf aufbauen. „Normalerweise“ deshalb, weil der Standard für das Signaling nichts vorschreibt, also durchaus andere Konstellationen denkbar sind. Das ist auch der Grund, warum die Verschlüsselungsebene optional ist. An dieser Stelle lauert eine potenzielle Sicherheitslücke.

Die eigentliche RTC-Verbindung setzt dagegen auf UDP, ein verbindungsloses Protokoll, welches keine Garantie auf die Paketreihenfolge gibt und auch keine Kontrollmechanismen integriert, um zu versichern dass ein Paket angekommen ist. Dadurch können die Datenpakete ohne große Zeitverzögerung direkt auf die Reise zum Ziel geschickt werden. Durch diese Zeitersparnis ist es möglich Real-Time-Anwendungen zu realisieren.

[9] XHR: Mit XMLHttpRequest können Daten mit HTTP übertragen werden

[10] SSE: Die Simple Sharing Extensions ist ein Synchronisierungs-Algorithmus im XML-Format

Für den Verbindungsaufbau dient das ICE-Protokoll, das sich der beiden Technologien STUN und TURN bedient, Näheres dazu im Kapitel 2.4 RTC-Verbindungsaufbau. DTLS[11] sorgt für die Verschlüsselung sämtlicher Daten. Die Grafik zeigt, das SRTP die Verschlüsselung umgehen kann, dies ist aber nur deshalb möglich, weil SRTP selbst einen Verschlüsselungsmechanismus beherrscht. Es werden also über eine RTC-Verbindung niemals Daten unverschlüsselt übertragen.

Das Secure Real-Time Transport Protocol (SRTP) ist ein zuverlässiges und verbindungsorientiertes Transportprotokoll. Mit dem es möglich ist, mehrere Datenströme (Streams) über eine Verbindung zu leiten. Außerdem gibt es einen Heartbeat-Mechanismus, der Verbindungsabrisse erkennt. SRTP bietet somit alles was benötigt wird, um Medien-Daten von der RTCPeerConnection-API zu versenden.

Durch das Stream Control Transmission Protocol (SCTP) wird UDP befähigt, seine Pakete in einer verlässlichen Reihenfolge zu senden, außerdem kann damit festgestellt werden, ob Pakete auch zum Ziel gefunden haben. Dies ist sehr wichtig für die Nutzung des DataChannels, über den es auch möglich ist, ganze Dateien zu transportieren.

2.4 RTC-Verbindungsaufbau

Die RTCPeerConnection kapselt eine ganze Reihe von Aufgaben, die nötig sind, um eine Verbindung zwischen zwei Webbrowsern aufzubauen. Mit Hilfe des ICE-Protokolls wird zu Beginn herausgefunden, wie über das Netzwerk zwischen den beiden Endpunkten kommuniziert werden kann. Diese Informationen werden dann über den Signaling-Channel, mit Hilfe von SDP-Paketen, ausgetauscht und die Verbindung aufgebaut. In diesem Kapitel wird das komplexe Aufgabengebiet in seine Einzelteile zerlegt, erläutert und am Ende noch einmal ganzheitlich betrachtet.

2.4.1 Hole Punching

Der Hole-Punching-Vorgang [Bryan Ford, 2005] beschreibt wie sich zwei Peers über lokale Netzwerkadressen hinaus, also auch jenseits von Firewalls und NATs, direkt miteinander verbinden können, um dann über eine Peer-to-Peer-Verbindung Daten auszutauschen. In diesem Prozess wird also die netzwerktechnische Grundlage für einen Datenaustausch geschaffen.

[11] DTLS: Datagram Transport Layer Security ist ein auf TLS basierendes Verschlüsselungsprotokoll

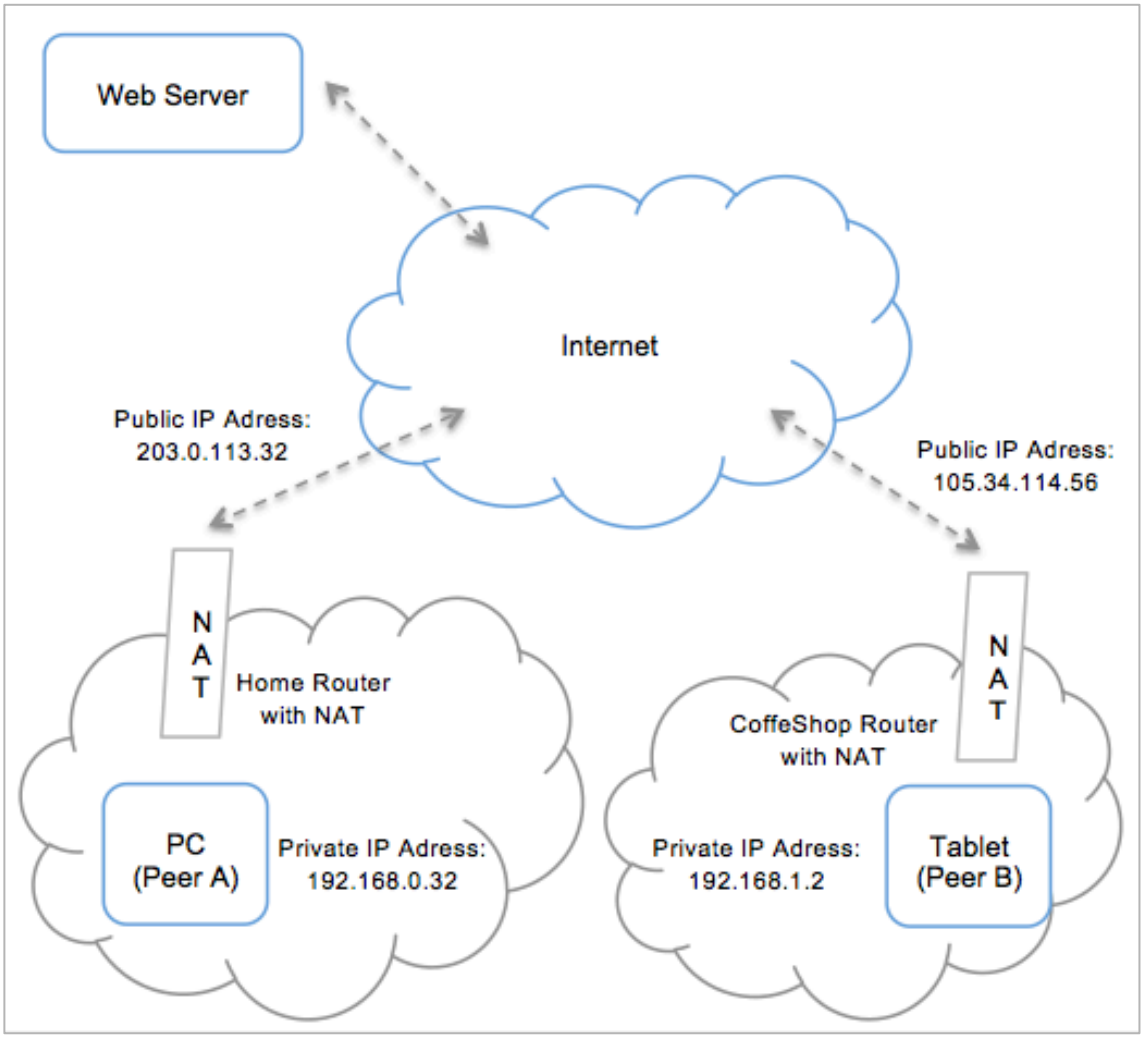


Abbildung 10: öffentliche und private IP-Adressen

Nahezu jede Verbindung ins Internet, sei es privat oder im Enterprise-Umfeld, führt zunächst vorbei an einem Network Adress Translator (NAT), wie in Abbildung 10 dargestellt. In komplexeren Netzwerkstrukturen ist aber auch eine Verschachtelung von mehreren NATs denkbar. Innerhalb des lokalen Netzwerks ist jeder Computer noch mit seiner eigenen lokalen IP-Adresse identifizierbar, hinter dem NAT allerdings bekommen alle Verbindungen ins Internet die öffentliche IP-Adresse vom Internet Service Provider (ISP).

Möchte nun Peer A mit Peer B kommunizieren, muss er zunächst die öffentliche IP-Adresse von Peer B kennen und dann wissen, was er dessen NAT sagen soll, um den richtigen Computer in seinem Netzwerk zu erreichen. Die Schwierigkeit liegt in letzterem Punkt, denn jedes Netzwerk ist anders konfiguriert und hat gewisse Restriktionen, die berücksichtigt werden müssen.

Interactive Connectivity Establishment (ICE)

Im Falle von WebRTC wird für das Hole Punching das ICE-Protokoll eingesetzt. Dafür kapselt es die beiden Technologien STUN und TURN. Mit der Hilfe von STUN wird einem Peer ermöglicht, sich darüber klar zu werden wie er selbst von außen erreichbar ist.

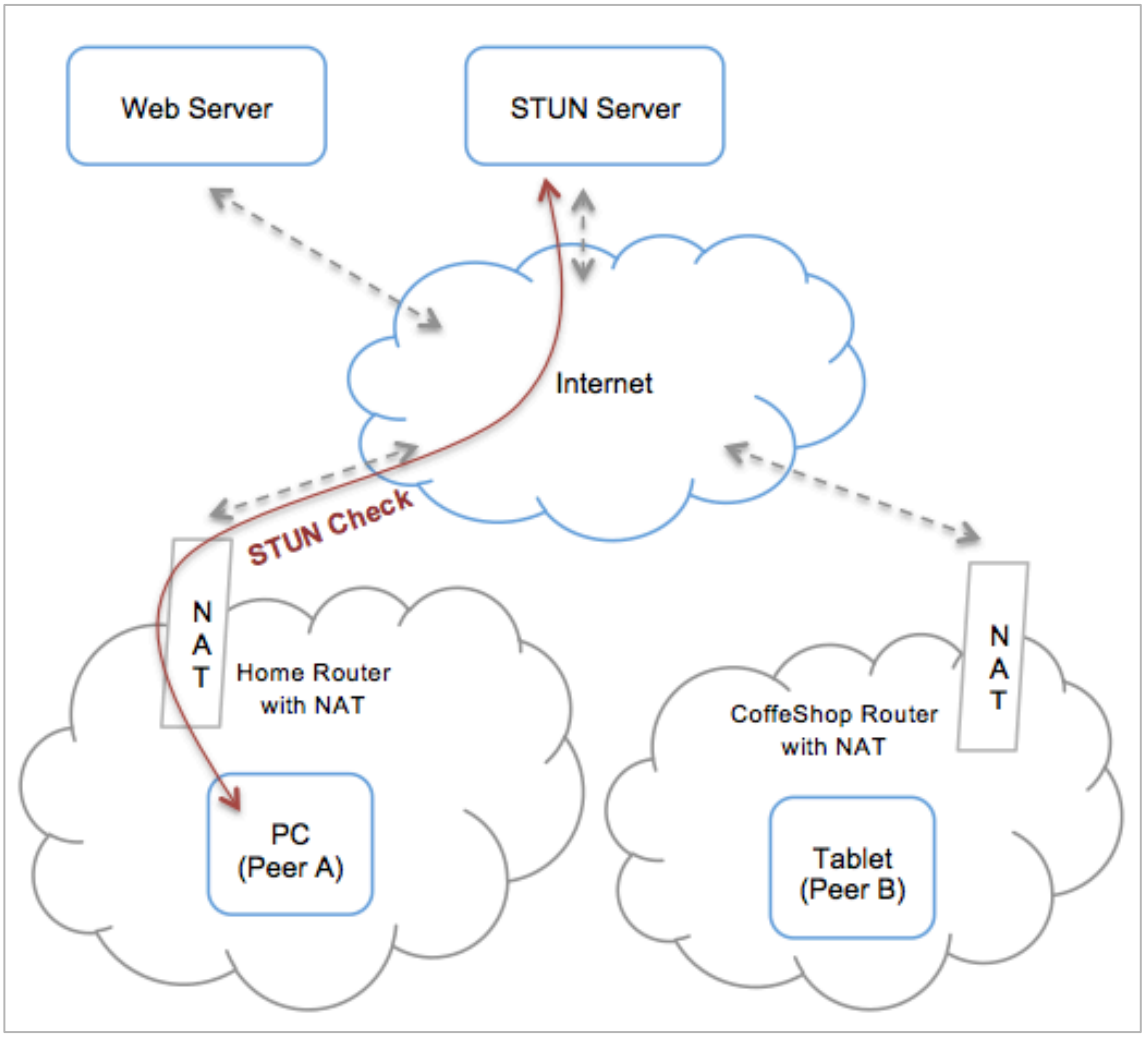


Abbildung 11: STUN-Check eines Peers

Zu diesem Zweck sendet der Peer eine Anfrage an einen STUN-Server und bekommt als Antwort die für den Server sichtbare IP-Adresse von sich. Da der Server hinter dem NAT, im Internet positioniert ist, bekommt Peer A seine öffentliche IP-Adresse mitgeteilt (Abb. 11). Dieser sog. STUN-Check wird nun mehrfach mit unterschiedlichen UDP-Ports ausgeführt, um möglichst viele gültige Candidates zu diesem Peer zu sammeln. Ein Candidate setzt sich aus einer IP-Adresse und einem bestimmten Port zusammen. Jeder gültige STUN-Check führt zu einem Eintrag in die Candidate-Liste. Dieser Vorgang wird auch als ICE-Gathering bezeichnet und sollte zeitlich immer während des Verbindungsaufbaus stattfinden, da sonst die Gefahr besteht, dass die gesammelten Candidates ihre Gültigkeit verlieren.

Diese Art des Hole Punchings für das UDP funktioniert in 82% der Fälle [Bryan Ford, 2005], bei den restlichen 18% sind die Restriktionen des Netzwerks so hoch, dass keine UDP-Verbindung von außen möglich sind. Diese Zahl ist allerdings als total anzusehen, sie wurde für eine große Anzahl von Netzwerken und Verbindungen erhoben, ob ein STUN-Check also Erfolg hat, ist von Netzwerk zu Netzwerk unterschiedlich und nicht von jeder einzelnen Verbindung.

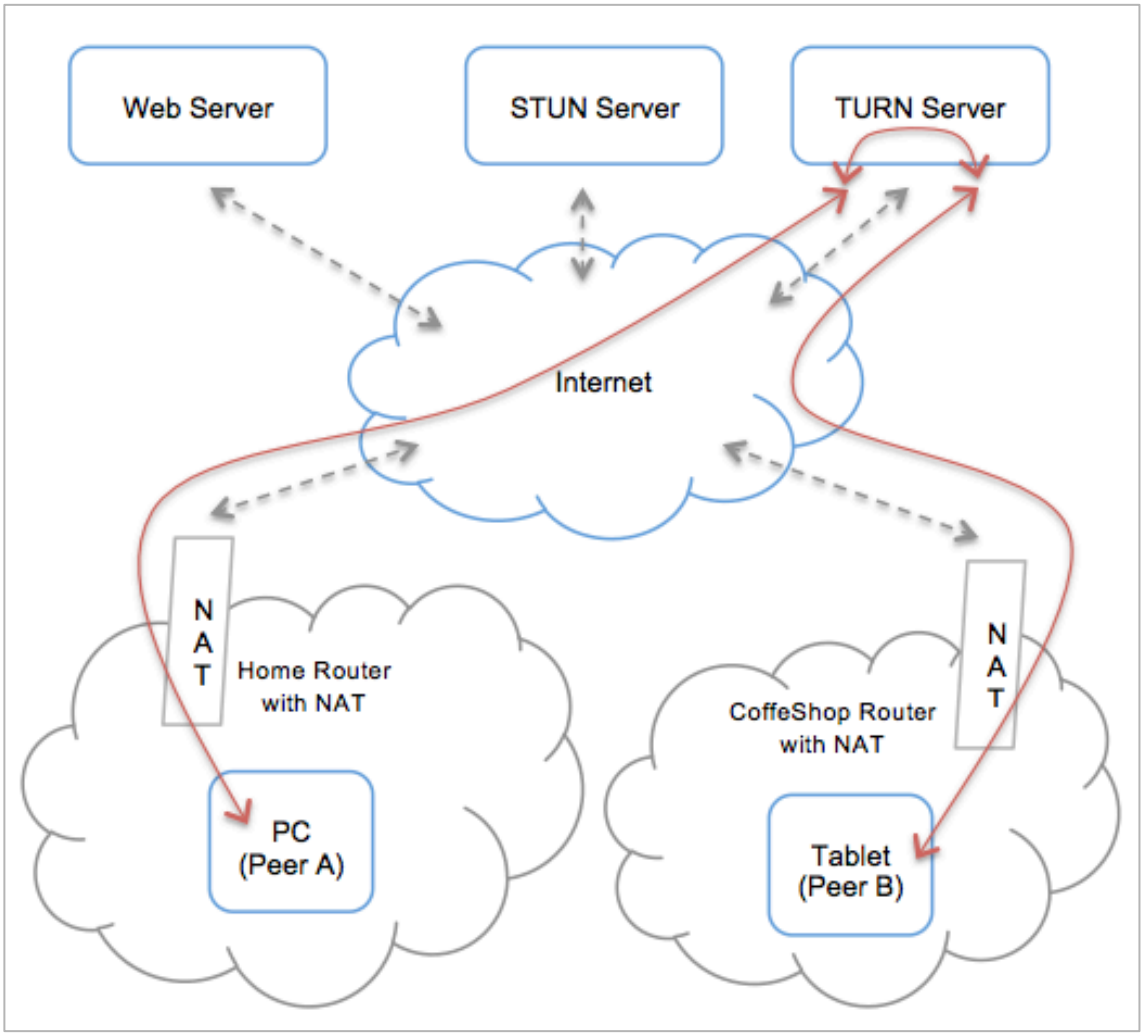


Abbildung 12: Datenfluss über einen TURN-Server

Für die 18% der Verbindungen, die nicht über STUN aufgebaut werden können, hält der ICE-Gathering-Prozess eine Fall-Back-Lösung über einen TURN-Server bereit (Abb. 12). Dieser fungiert als Man-In-The-Middle zwischen den Peers, die Datenpakete werden in diesem Fall über den TURN-Server geleitet, statt direkt zwischen den Peers. Das ist nicht optimal, da es dadurch automatisch zu höheren Latenzzeiten kommt. In der Praxis sind TURN-Server häufig STUN-Server mit TURN-Funktionalität, also kombiniert.

Typen verschiedener Candidates

Tabelle 3 zeigt eine Liste von möglichen Candidate-Typen, diese verschiedenen Typen sind wichtig, da deren Priorität konfigurierbar ist.

Candidate-Typ	Verwendung
Host	die lokale Verbindungsadresse von der Netzwerkkarte
Server Reflexive	Verbindungsadresse welche vom STUN-Server nach einem STUN-Check zurückgegeben wird
Peer Reflexive	Adresse bereitgestellt von einem anderen Peer, kommt während eines Connectivity Checks, nicht über den Signaling Channel
Relayed	Fall-Back über einen TURN-Server, die Verbindungsadresse entspricht der des TURN-Servers

Tabelle 3: Typen von ICE-Candidates

2.4.2 Signaling

Nachdem im ICE-Gathering-Prozess eine Liste von möglichen Candidates erstellt wurde, müssen diese Informationen dem Peer bereitgestellt werden, mit welchem eine Verbindung aufgebaut werden soll. Diese Aufgabe übernimmt der Signaling-Channel.

Der WebRTC-Standard spezifiziert nicht wie dieser Signaling-Channel aufgebaut wird und welche Technologie dafür eingesetzt werden soll. Abbildung 8 verdeutlicht dies indem der Signaling-Prozess völlig losgelöst von der RTC-Verbindung mit der Anwendung kommuniziert.

Den Grund für die fehlende Implementierung, dieser wichtigen Phase des Verbindungsaufbaus, beschreibt ein kurzer Auszug aus dem Internet-Draft von JSEP.

> *„... The rationale is that different applications may prefer to use different protocols, such as the existing SIP[12] or Jingle[13] call signaling protocols, or something custom to the particular application, perhaps for a novel use case ...“* [J. Uberti und C. Jennings, 2014]

[12] SIP: Session Initiation Protocol - Protokoll um Verbindungen für IP-Telefonate aufzubauen

[13] Jingle: Die Signaling-Erweiterung vom Extensible Messaging and Presence Protocol (XMPP)

Das Javascript Session Establishment Protocol (JSEP) ist die JavaScript-Schnittstelle der RTCPeerConnection für diesen Vorgang. Von ihr werden Callbacks wie z.B. `onicecandidate(cand);` zur Verfügung gestellt um den asynchronen Verbindungsaufbau steuern zu können.

Neben den im Zitat beispielhaft erwähnten Protokollen gibt es aber auch die Möglichkeit, als Anwendungsentwickler selbst diese Aufgabe zu übernehmen, wie im Rahmen dieser Arbeit für den PipeUp-Prototypen geschehen.

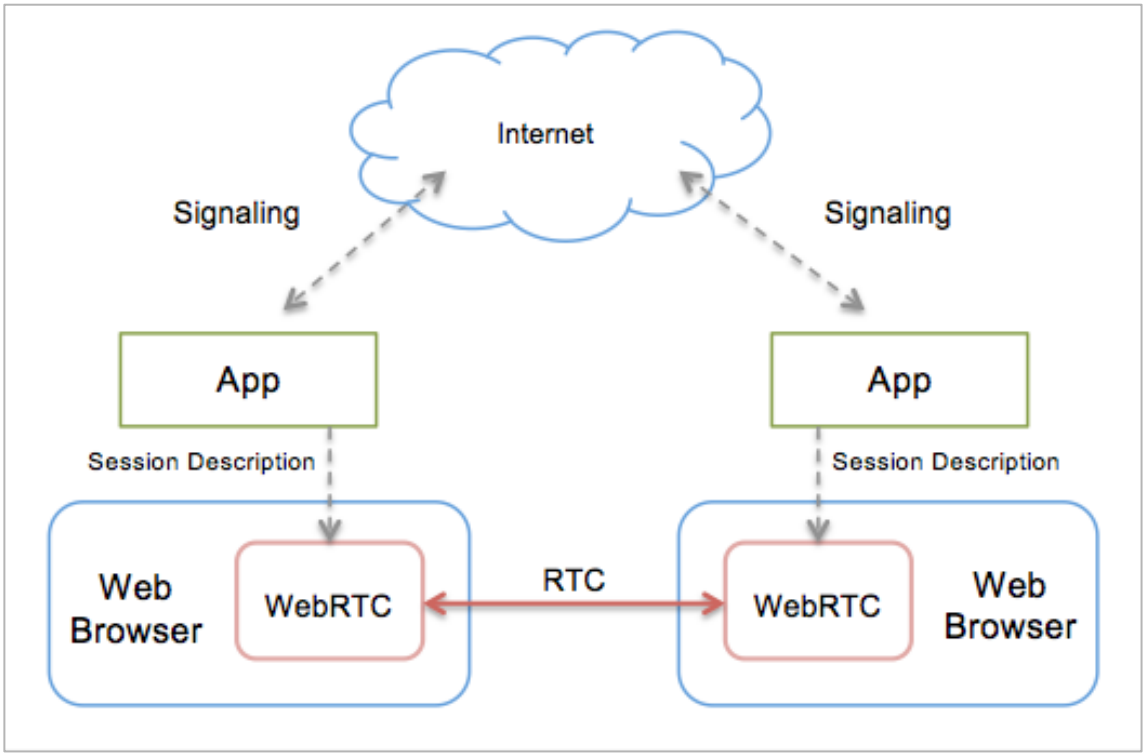


Abbildung 13: JSEP Signaling-Model [J. Uberti und C. Jennings, 2014]

Abbildung 13 verdeutlicht, dass es für den Verbindungsaufbau nicht wichtig ist, wie die Signaling Operationen von Peer zu Peer gelangen. Im Praxisteil dieser Arbeit wird ein möglicher Weg über WebSockets aufgezeigt.

2.4.3 Offer/Answer-Workflow

Nachdem zwei Peers einen Signaling-Channel zueinander aufgebaut haben, wird über diese Verbindung darüber verhandelt, welche technischen Eigenschaften die RTC-Verbindung haben soll. Dazu gehören unter anderem die Liste der Candidates, Bandbreiteninformationen, Audio/Video-Formate und Verschlüsselungsparameter. Dieser Prozess wird als Offer/Answer-Workflow bezeichnet.

Zu diesem Zweck setzt JSEP auf das Session Description Protocol (SDP). SDP ist kein Kommunikationsprotokoll, vielmehr eine grammatikalische Syntax, die im Voice-Over-IP-Umfeld sehr verbreitet ist.

```
type: offer, sdp: v=0
o=- 8926912939805629217 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE audio video data
a=msid-semantic: WMS
m=audio 1 RTP/SAVPF 111 103 104 0 8 126
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
a=ice-ufrag:Ypm4BNpyeocraCTp
a=ice-pwd:gJH2MUS7nk+ve+vwOOSgkCnk
a=ice-options:google-ice
a=fingerprint:sha-256 C5:24:C8:81:EB:51:B1:67:A3:48:B5:B3:39:D6:64:19:46:F
a=setup:actpass
a=mid:audio
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=extmap:3 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
a=recvonly
a=rtcp-mux
a=rtpmap:111 opus/48000/2
a=fmtp:111 minptime=10
a=rtpmap:103 ISAC/16000
a=rtpmap:104 ISAC/32000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:126 telephone-event/8000
a=maxptime:60
m=video 1 RTP/SAVPF 100 116 117 96
c=IN IP4 0.0.0.0
a=rtcp:1 IN IP4 0.0.0.0
```

Abbildung 14: Beispielsyntax einer SDP-Offer

Abbildung 14 zeigt einen Auszug einer SDP-Offer, hier kann der Anwendungsentwickler Änderungen vornehmen, um die Verbindung seinen Wünschen anzupassen. Mit JavaScript ist dies leider sehr umständlich, da die Syntax keiner JavaScript bekannten Form, wie z.B. JSON, entspricht. Deshalb wird der Standard an dieser Stelle auch oft kritisiert. Als Grund dafür wird die weite Verbreitung von SDP angeführt. Neue Features und Änderungen werden somit direkt unterstützt [J. Uberti und C. Jennings, 2014].

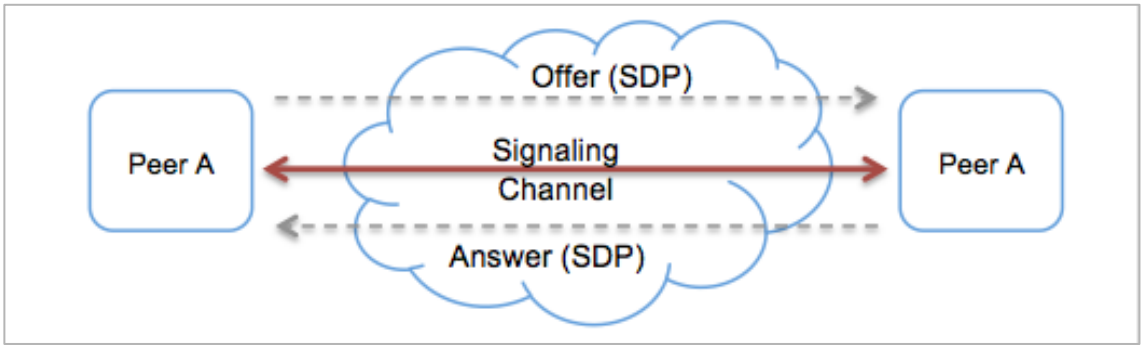


Abbildung 15: Offer/Answer-Workflow

Abbildung 15 skizziert diesen Offer/Answer-Workflow und vereinfacht diesen Prozess sehr stark. Detaillierter wird auf diesen Vorgang, in der ganzheitlichen Betrachtung (Kapitel 2.4.4) des Verbindungsaufbaus, noch eingegangen.

Peer A ist in diesem Fall der Initiator der RTC-Session und sendet Peer B sein Offer-Paket mit allen Informationen dazu, unter welchen Voraussetzungen er zu Peer B eine Verbindung aufbauen würde. Peer B entscheidet sich nun, ob er das Angebot annimmt oder ablehnt. Nimmt Peer B das Angebot an, sendet er ein Answer-Paket zurück an Peer A. Dieses ist angereichert mit den für ihn in Frage kommenden Parametern für den Verbindungsaufbau.

Dieser Prozess kann im Laufe einer WebRTC-Session mehrmals, auch für bestehende Verbindungen, wiederholt werden. So z.B. wenn der Verbindung ein weiterer Stream hinzugefügt werden soll.

2.4.4 Ganzheitliche Betrachtung

Nachdem nun die wichtigsten Komponenten erläutert wurden, wird an dieser Stelle der Verbindungsaufbau als Ganzes betrachtet.

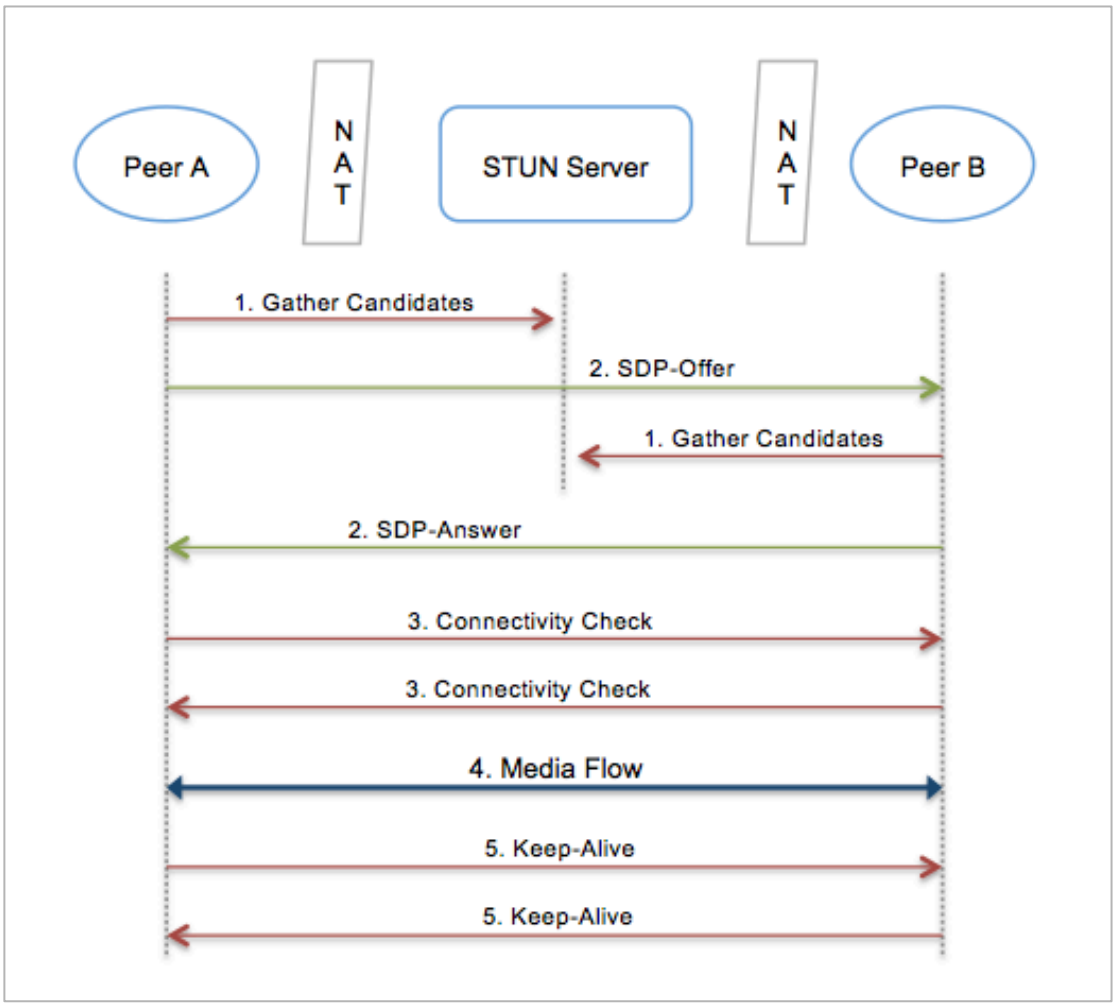


Abbildung 16: Gesamtübersicht des RTC-Verbindungsaufbaus

Das Flussdiagramm in Abbildung 16 schematisiert, welche Aktionen von Peer A und Peer B ausgelöst werden, um eine RTC-Verbindung zueinander aufzubauen.

STUN-Checks

Peer A ist der Initiator dieser Verbindung und beginnt damit geeignete Candidates von sich zu beschaffen (1. Gather Candidates), dazu befragt er den STUN-Server mit STUN-Checks.

Offer/Answer-Workflow

Ohne auf den Abschluss des Candidates-Gathering-Prozesses zu warten, wird ein Offer-Paket mit den Verbindungsparametern über den Signaling-Channel zu Peer B gesendet (2. SDP-Offer). Nimmt Peer B das Angebot an, beginnt auch er mit Hilfe von STUN-Checks nach geeigneten Candidates zu suchen und sendet das SDP-Answer-Paket zurück zu Peer A.

Sobald diese Verbindung zueinander besteht, werden die gesammelten Candidates über den Signaling Channel miteinander ausgetauscht und sortiert. Die Reihenfolge, in der die verschiedenen Candidates später überprüft werden, kann in den Verbindungsparametern des Offer/Answer-Workflows festgelegt werden, in der Regel möchte man aber eine möglichst kurze Verbindung zwischen den Peers aufbauen. Deshalb haben Host-Candidates meist die höchste Priorität, gefolgt von Server und Peer Reflexive Candidates. Erst dann, wenn alle Möglichkeiten ausgereizt sind, kommt der Fallback mit den Relayed-Candidates in Frage. Außerdem ist es möglich auch IPv4 und IPv6 unterschiedlich zu priorisieren.

Eine Besonderheit von WebRTC ist, die Bestrebung alle seine Datenströme wie Audio, Video oder Daten über eine Verbindung zu transportieren. Andere Technologien, die auch auf das ICE-Protokoll setzen, bauen dazu meist separate Verbindungen auf.

Connectivity Checks

Da nun beide Peers alle möglichen Candidates kennen, werden daraus anhand ihres Typs und ihrer Eigenschaften (z.B. IP-Adresstyp IPv4 oder IPv6) Pärchen gebildet. Diese Paare werden anschließend nach und nach mit einem Connectivity Check überprüft. Um eine Entscheidung darüber zu treffen, welches Paar für die Verbindung verwendet wird, legt ein Algorithmus im ICE-Protokoll einen Controlling Peer fest. Dieser trifft dann die Entscheidung, indem er ein Flag im Antwort-Packet eines Connectivity Checks setzt. Dies wird dann vom kontrollierten Peer noch einmal bestätigt und die Datenübertragung kann beginnen (4. Media Flow).

Keep-Alives

Bei Inaktivität der Verbindung sendet das ICE-Protokoll alle 15 Sekunden Keep-Alive-Signale, um die Verbindung aufrecht zu erhalten (5. Keep-Alive).

Session Rehydration

Sollte während einer aktiven RTC-Verbindung der Webbrowser einen Reload der Webseite auslösen oder die Applikation an sich in irgendeiner Form neugestartet werden, gibt es einen Mechanismus, der sich Rehydration nennt. Dieser setzt voraus, dass der Session-Status in irgendeiner Form persistent ist, sodass nach einem Reload die wichtigsten Informationen für einen erneuten automatisierten Verbindungsaufbau erhalten bleiben. Dies kann beispielsweise mit Hilfe von Local Storage[14] oder einer Datenbank erreicht werden.

An dieser Stelle wird nicht weiter darauf eingegangen, da über dieses Feature noch viel diskutiert wird und es in dieser Arbeit auch keine Anwendung findet. Nähere Informationen sind dem JSEP-Draft in Sektion 3.6 zu entnehmen [J. Uberti und C. Jennings, 2014].

Beenden der Verbindung

Um eine RTC-Verbindung letztendlich zu beenden, stellt die RTCPeerConnection-API die Funktion `close();` zu Verfügung. Wird diese Funktion ausgeführt, werden folgende Dinge unternommen:

1. Wenn der Signaling-Status des RTCPeerConnection-Objekts bereits `closed` ist, mache nichts weiter.
2. Beende sofort alle ICE-Aktivitäten, jegliches Streaming und gebe alle reservierten Ressourcen in diesem Zusammenhang wieder frei.
3. Setze den Signaling-Status des RTCPeerConnection-Objekts auf `closed.`

Durch das Setzten des Signaling-Status wird beim Verbindungspartner der `signalingstatechange()` ausgelöst, sodass auch dieser die Information über das Ende der Verbindung mitbekommt.

2.5 Multiparty Architekturen

Bisher lag der Focus dieser Arbeit auf einer RTC-Verbindung zwischen zwei Teilnehmern. In Abbildung 17 als Direct Connection bezeichnet. Die Architektur von WebRTC lässt es aber auch zu, dass mehrere RTC-Verbindungen pro Webbrowser aufgebaut werden können und somit komplexere Strukturen zwischen mehreren Peers denkbar werden. Allerdings muss je nach Anwendungsfall überlegt werden, welche Architektur am besten passt, damit das Nutzungserlebnis für den Anwender nicht beeinträchtigt wird.

[14] Local Storage: Ein HTML5-Feature um Daten im Webbrowser sessionübergreifend zu persistieren.

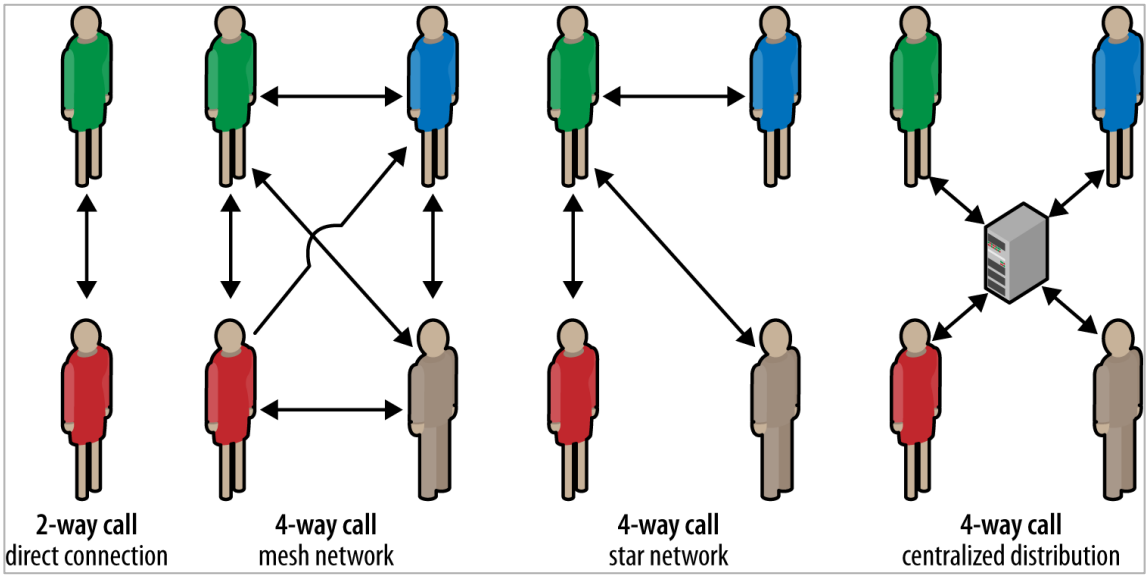


Abbildung 17: Architekturkonzepte für Multiparty-WebRTC [Ilya Grigorik, 2013]

Bei einer WebRTC-HD-Video-Konferenz zwischen zwei Teilnehmern kommen bereits 2024kbps bei einem Peer an und müssen gleichzeitig an den anderen versandt werden, dazu kommen noch die Audio-Daten von ca. 320kbps. Bei FullHD stünden bereits 4096kbps für das Videostreaming zu Buche. Diese Zahlen mögen im ersten Moment nicht praxisnah erscheinen, aber Deutschland liegt im Moment bei einer durchschnittlichen Internet-Bandbreite von 8,1Mbit/s [Akamai, 2014] und damit nur auf Platz 26 im internationalen Vergleich.

Wird nun die Teilnehmerzahl (N) an einer WebRTC-Session erhöht, steigt die benötigte Bandbreite allerdings um das (N-1)-fache, da jeder Teilnehmer mit jedem anderen eine RTC-Verbindung aufbaut und Daten überträgt. Eine Vollvermaschte-Netztopologie (mesh network) stößt hier also schnell an Grenzen.

Eine Alternative dazu ist das Stern-Netzwerk (star network), hier wird ein Teilnehmer als Host bestimmt, dieser hält dann als Einziger alle Verbindungen zu den anderen Teilnehmern. Diese Topologie macht die Kommunikationswege zwischen den Teilnehmern zwar komplizierter, da immer der Umweg über den Host genommen werden muss, aber sie beschränkt den Aufwand für jeden Nicht-Host auf nur eine Verbindung. Diese Netzstruktur wird auch für PipeUp angewendet, mehr dazu im dritten Teil dieser Arbeit.

Eine weitere Möglichkeit ist es, einen Server einzusetzen, zu dem sich alle Teilnehmer verbinden (centralized distribution). Diese Variante widerspricht zwar dem Peer-to-Peer-Gedanken, ist aber sehr gut auch für eine große Anzahl von Teilnehmern skalierbar. Für die Benutzer bleibt der große Vorteil von WebRTC, keine Plugins installieren zu müssen und der Server kann klassische Aufgaben übernehmen, so z.B. die Video-Streams der einzelnen Peers sammeln, zu einem Stream multiplexen und diesen dann wieder verteilen. Nur einen Stream zu versenden, schont die Bandbreite, CPU- und GPU-Zeit aller Teilnehmer enorm.

2.6 Security und Privacy

Im Hinblick auf die aktuellen Geschehnisse rund um den NSA-Skandal sind viele Menschen, was ihre Privatsphäre im Internet angeht, verunsichert und stehen neuen Technologien eher skeptisch gegenüber. Deshalb werden die Themen Security und Privacy, im Zusammenhang mit WebRTC immer wieder hervorgehoben und haben einen hohen Stellenwert bei der Entwicklung des Standards.

Dieses Kapitel soll einen Einblick in diese Thematik geben und häufig gestellte Fragen klären, erhebt dabei, aufgrund des Umfangs der eingesetzten Technologien, aber nicht den Anspruch auf Vollständigkeit.

Sicherheitsmodell

Das Sicherheitsmodell eines Webbrowsers ist grundsätzlich so ausgelegt, dass der Benutzer ihm volles Vertrauen schenken kann. Er soll dabei davon ausgehen können, dass alles was an Interaktion direkt mit dem Browser passiert, keine negativen Auswirkungen auf die Sicherheit seiner Daten hat. Anders verhält es sich bei Webseiten, die ein Benutzer mit dem Webbrowser besucht. Natürlich kann versucht werden, kompromittierte Webseiten nicht zu besuchen, dennoch kann dieses unabsichtlich passieren, indem z.B. einer verkürzten URL gefolgt wird. In diesem Falle muss ein Browser die Rahmenbindungen schaffen, dass der Benutzer die Situation erkennen und unterbinden kann, ohne weitreichende Probleme entstehen zu lassen. Ein gutes Beispiel dafür, ist die Nachfrage des Browsers, ob eine Datei wirklich heruntergeladen werden soll, bevor der Download startet.

Durch die Integration von WebRTC in den Browser, ist die Wahrung dieses Vertrauens, eine der Hauptaufgaben neben der eigentlichen Funktionalität. Bisher brauchten Webbrowser sich nur gegen böswillige Webseiten rüsten, durch die neue RTC-Verbindung zu anderen Browsern, muss sich nun allerdings auch gegen böswillige Browser geschützt werden. Es ist nicht undenkbar, dass ein veränderter Open-Source-Browser in Umlauf gebracht wird, der dann manipulierte WebRTC-Verbindungen herstellt, um dem Benutzer zu schaden.

Jede neue API oder Schnittstelle, die eine Software bekommt birgt auch ein neues Sicherheitsrisiko in sich. Aus diesem Grund ist ein gut durchdachter und erprobter Standard sehr wichtig für das Vertrauen, und somit auch den Erfolg so einer Neuerung.

Erlaubnis erteilen

Möchte der Webbrowser für eine RTC-Verbindung auf Medien, wie die Kamera oder das Mikrofon, zugreifen, muss er dazu die Erlaubnis des Benutzers einholen, Abb. 26 zeigt wie dies im Chrome-Browser aussieht.

Damit der Benutzer einen Überblick seiner Freigaben behält, ist die Erlaubnis immer an die Domain gebunden. Denn es gibt Konstellationen, wo nicht immer klar ist, von wem eine solche Anfrage nun tatsächlich gestellt wird, z.B. Pop-Ups. Außerdem ist so eine

Erlaubnis immer Session gebunden. Bei einer Webseite die über `http` erreichbar ist, bedeutet das, diese Abfrage wird nach einem Seiten-Reload erneut gestellt. Wird `https` verwendet, kann man diese Umständlichkeit umgehen, hier wird nur einmal nachgefragt.

Der Standard sieht für eine Verbindung bei der nur der DataChannel genutzt wird, keine Notwendigkeit für das Einholen einer Erlaubnis beim User. Das ist damit zu begründen, dass über den DataChannel nur Daten versendet werden, die sonst auch über eine HTTP-Verbindung von einem Webserver empfangen werden können.

2.6.1 Verschlüsselung der RTC-Verbindung

Für jeden Channel der zwischen zwei Peers aufgebaut wird, wird vorher ein DTLS-Handshake vollzogen, damit die Übertragung verschlüsselt erfolgt.

> *„Datagram Transport Layer Security (DTLS) ist ein auf TLS basierendes Verschlüsselungsprotokoll, das im Gegensatz zu TLS auch über unzuverlässige Transportprotokolle wie UDP übertragen werden kann.“*[15]

Die eigentliche RTC-Verbindung wird also stets verschlüsselt. Dieser Vorgang findet direkt nach den Connectivity Checks und der Wahl des Candidate-Paares statt. Der ganze Prozess wird in Kapitel 2.4.4 ganzheitlich erklärt, Abbildung 16 muss unter diesen Gesichtspunkten allerdings um den DTLS-Handshake erweitert werden.

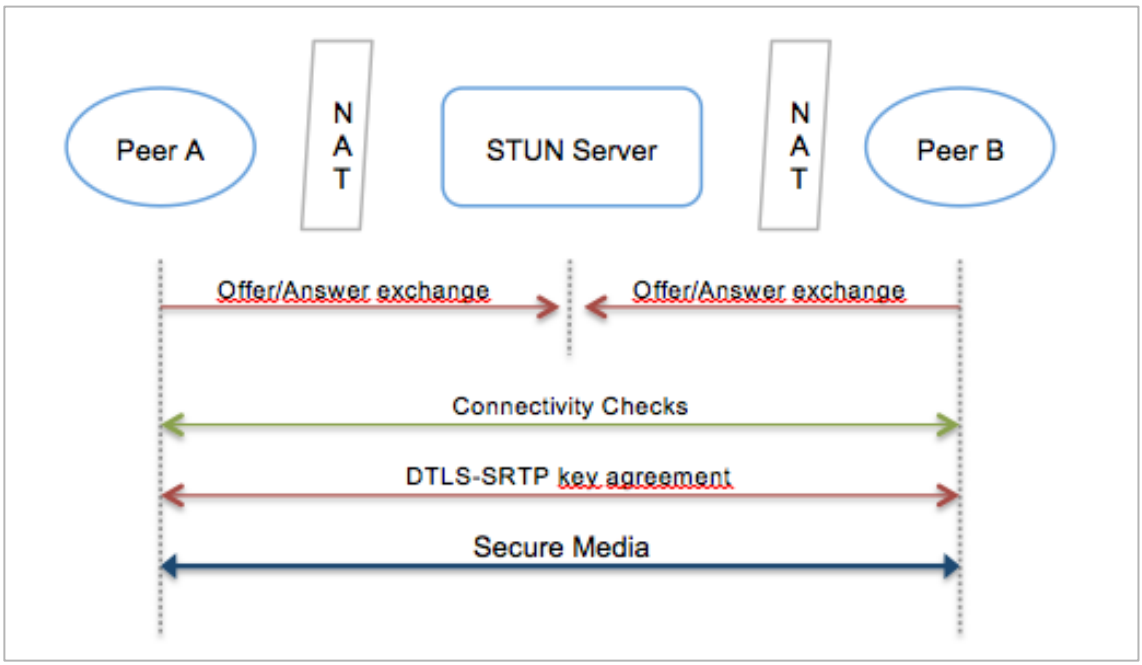


Abbildung 18: DTLS-SRTP Handshake

[15] Quelle: Wikipedia - http://de.wikipedia.org/wiki/Datagram_Transport_Layer_Security

Abbildung 18 vereinfacht den Prozess ein wenig, zeigt dafür aber, an welcher Stelle der DTLS-Handshake vollzogen wird. Wichtig ist, dass dieser Public-Key-Austausch bereits über die direkte Verbindung vollzogen wird und nicht den Signaling-Channel benutzt.

2.6.2 Verschlüsselung des Signaling-Channels

Auch wenn der Standard keine Vorschriften dazu macht, wie der Signaling-Mechanismus in eine Anwendung implementiert werden soll, so sollte darauf geachtet werden, dass dieser stets mit verschlüsselten Übertragungsmethoden realisiert wird. Dafür kommen beispielsweise die, um Verschlüsselung erweiterten, Protokolle HTTPS oder Secure WebSocket in Frage.

2.6.3 Mögliche Angriffsziele

In Verbindung mit WebRTC ergeben sich einige mögliche Angriffsziele, wie die bereits erwähnten und neu eigeführten JavaScript-APIs. Sind diese nicht bis ins letzte Detail durchdacht oder falsch von den Webbrowsern implementiert, können sich daraus Sicherheitslücken ergeben, die z.B. die Zugriffsabfrage für die Hardware umgehen. Dadurch könnte eine Webseite auf die Kamera und das Mikrofon des Besuchers zugreifen, ohne dass dieser davon erfährt.

Ein Angriff auf den Signaling-Channel ist eine andere Möglichkeit. Da dieser nicht grundsätzlich standardisiert ist, kommt es hier darauf an, welche Technologie gewählt wurde und welche Sicherheitslücken sich daraus ergeben. Hat der Angreifer Zugriff auf den Signaling-Channel, kann dieser z.B. verhindern, dass eine Verbindung aufgebaut wird. Oder er initiiert eine Man-In-The-Middle-Attacke, um die später übertragenen Daten abzufangen.

Auch das ICE-Protokoll, für den Aufbau einer RTC-Verbindung, stellt ein angreifbares Ziel dar. Während des Wole-Punching-Vorgangs werden ICE-Pakete von egal welchem Server akzeptiert und erst nach dem ICE-Handshake wird die Verbindung auf eine bestimmte IP-Adresse beschränkt. Hier sind auch Man-In-The-Middle-Attacken denkbar.

Zuletzt sei noch gesagt, dass grundsätzlich jedes eingesetzte Protokoll und jede API ein Sicherheitsrisiko darstellen. Im Falle von WebRTC kommen da einige dieser Technologien zusammen, deshalb besteht ein Risiko, Lücken in diesen Systemen für komplexe Angriffe zu kombinieren. Um bekannt werdende Probleme so schnell es geht zu beheben, sollten Webbrowser mit WebRTC-Funktionalität auto-update fähig sein.

3 Hands On: PipeUp

Im ersten Kapitel wurde das Konzept von PipeUp bereits vorgestellt, in diesem Abschnitt soll es seine technische Tiefe bekommen.

Dazu wird zunächst auf das User Interface eingegangen und die Funktionen im Detail erklärt. Anschließend wird die Software-Architektur genauer untersucht und auf einige Besonderheiten eingegangen. Am Ende werden noch die Aufgaben erläutert, die im Falle einer Weiterentwicklung des Prototypen, umgesetzt werden müssten.

3.1 User Interface

Das User Interface ist sehr minimalistisch ausgerichtet und hat einen starken Focus auf Funktionalität, es gibt keine grafischen Elemente und alle Funktionen sind ohne einen Page-Reload zu erreichen.

3.1.1 Host-Ansicht

Wie Abbildung 5 schematisch zeigt, stellt die Host-Anwendung den Kern von PipeUp dar, jeglicher Informationsfluss führt über den Webbrowser, der die Host-Anwendung geladen hat.

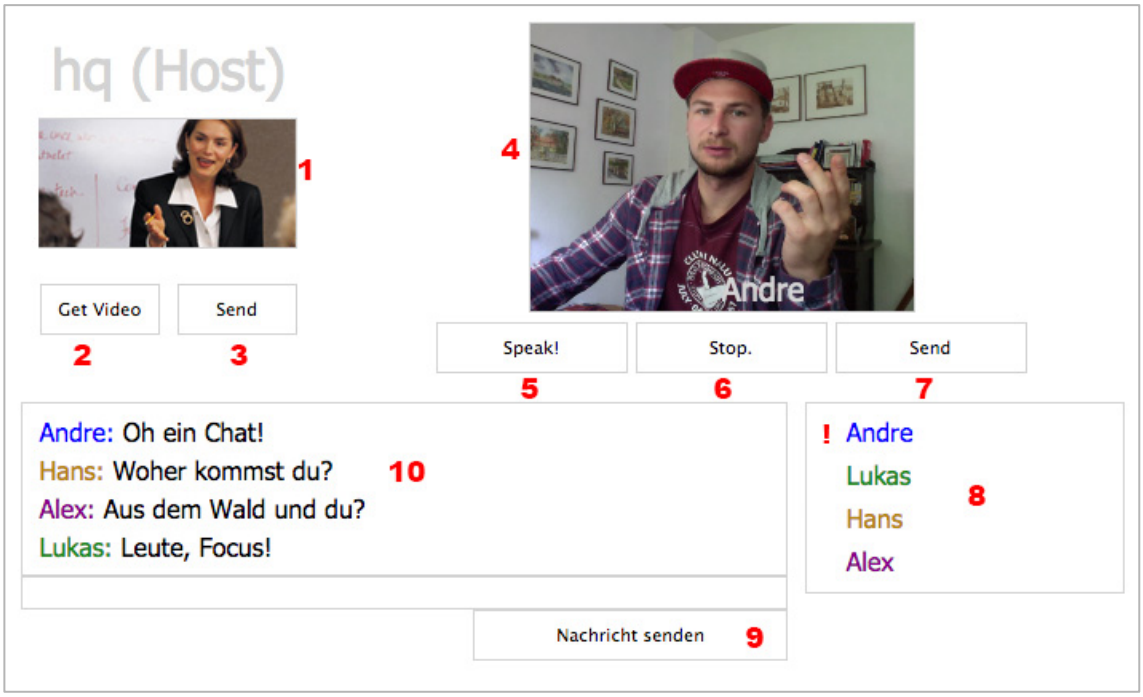


Abbildung 19: User Interface des PipeUp-Host

Aus diesem Grund bietet die Benutzeroberfläche des Hosts auch einige Zusatzfunktionen um die PipeUp-Session zu steuern. Diese werden nachfolgend einzeln erläutert.

1. Hier wird das lokale Video dargestellt, z.B. der Referent während eines Live-Events.
2. Mit diesem Button lässt sich das Video initial starten.
3. Mit dem Send-Button lässt sich das lokale Video an den aktuell ausgewählten Teilnehmer senden.
4. An dieser Stelle wird das Video, des gerade ausgewählten Teilnehmers, dargestellt.
5. Der Speak-Button veranlasst bei dem aktuell ausgewählten Benutzer die Abfrage, ob er seine Kamera und sein Mikrofon freigeben möchte. Bestätigt er diese, wird sein Bild und Ton zum Host übertragen und im Video-Bereich dargestellt.
6. Dieser Button stoppt die Übertragung des aktuell verbundenen Benutzers.
7. Wird, während eine Verbindung zu einem Benutzer besteht und seine Video und Audio-Daten übertragen werden, der Benutzer gewechselt, so kann dieser Stream über den Send-Button an den neuerlich ausgewählten Benutzer weitergeleitet werden (siehe dazu Szenario 3 aus Kapitel 1.2.4 Szenarien).
8. Dies ist die Liste aller Teilnehmer dieser Session, das Ausrufezeichen visualisiert, dass dieser Benutzer die PipeUp-Funktion betätigt hat und etwas sagen möchte. Die Farben der einzelnen Benutzer sind zufällig gewählt und dienen nur der besseren Unterscheidung. Durch einen Klick auf einen Benutzer, wird dieser ausgewählt und sein Name steht in dem Videobereich (Punkt 4).
9. Um eine Textnachricht abzusenden, lässt sich einfach die Enter-Taste betätigen, oder es wird auf den „Nachricht senden“-Button geklickt.
10. Der Verlauf des Text-Chats, zur Unterscheidung wer was geschrieben hat. Benutzernamen werden farblicher unterschieden und vorangestellt.

Die Elemente 4 bis 7 erscheinen erst wenn sich bereits mindestens ein Benutzer in der Session befindet und dieser dann über einen Klick aus der Benutzerliste ausgewählt wurde.

Die Bezeichnung „hq“ für den Host, steht für Headquarter, also Hauptquartier oder auch logistisches Planungs-, Befehls- und Koordinationszentrum. Hat aber ansonsten keinen tieferen Sinn.

3.1.2 Client-Ansicht

Die Benutzeroberfläche des Clients unterscheidet sich nur geringfügig von der Host-Ansicht, es fehlen lediglich einige Funktionen.

Abbildung 20: PipeUp-Client Anmelde-Ansicht

Doch bevor der Client auf die eigentliche Benutzeroberfläche gelangt, wird er zunächst aufgefordert, einen wahlfreien Benutzernamen einzugeben, mit dem er dann im Verlauf der Session identifiziert werden kann.

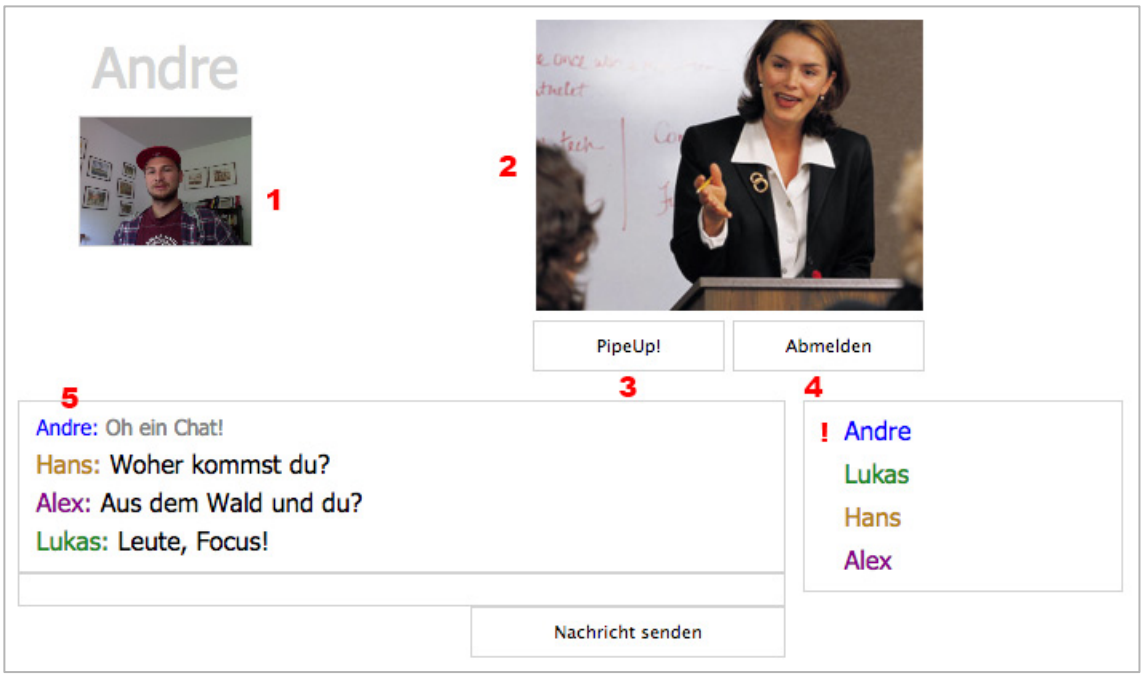


Abbildung 21: User Interface des PipeUp-Clients

Nachfolgend werden die Funktionen des PipeUp-Clients näher erläutert.

1. Das lokale Video während einer aktiven Verbindung, wird der Client also aufgefordert etwas zu sagen, kann er an dieser Stelle sehen, welches Bild er zum Host überträgt und ggf. die Kamera justieren.
2. An dieser Stelle wird das Remote-Video vom Host angezeigt. Ist der Client nicht vor Ort, besteht die Möglichkeit, ihm das Video der Session zu streamen.
3. Der PipeUp-Button sorgt dafür, dass dieser Client in der Benutzerliste nach oben wandert und mit einem Ausrufezeichen versehen wird. Dadurch kann jeder sehen, wer etwas beitragen möchte. Hat bereits im Vorfeld jemand den PipeUp-Button betätigt, reihen sich alle, die danach kommen, hinter diesem Benutzer ein, sodass eine Reihenfolge nach dem FIFO-Prinzip entsteht.
4. Mit dem Abmelden-Button können sich Benutzer wieder von der Session abmelden und landen wieder in der Anmeldeansicht.
5. Nachrichten, die selbst verfasst worden sind, werden in einer kleineren Schrift und grau dargestellt.

3.2 Architektur

An dieser Stelle wird näher auf die programmiertechnischen Abläufe von PipeUp eingegangen. Dazu wird zunächst der grundsätzliche Aufbau mit allen zusammenwirkenden Komponenten erklärt, die Dateistruktur aufgezeigt und ein Blick auf das Klassenmodel geworfen. Anschließend werden noch einige spezielle Abläufe erörtert.

3.2.1 Grundsätzlicher Aufbau

Die Idee einer direkten Peer-to-Peer-Verbindung klingt zunächst sehr simpel und einfach zu realisieren. Bei näherem Hinsehen wird aber ein komplexes Gebilde von vier unterschiedlichen Verbindungstypen sichtbar und auf einen Server kann leider nicht verzichtet werden.

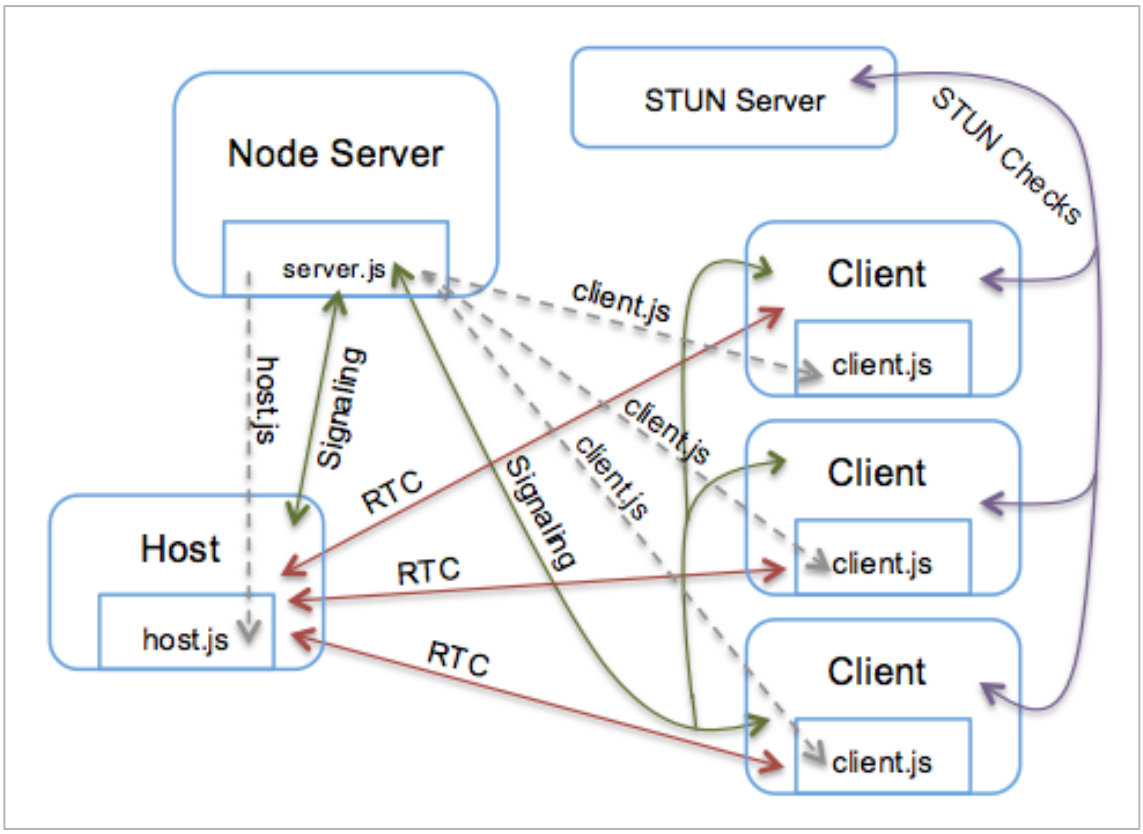


Abbildung 22: Verbindungsübersicht in PipeUp

PipeUp setzt einen Node-Server voraus auf dem die *server.js* läuft. Dieser Server ist in der Lage http-Anfragen entgegenzunehmen und verteilt so die client.js bzw. host.js, dieser Vorgang wird in Abb. 22 mit grau-gestrichelten Pfeilen dargestellt.

Führt nun der Webbrowser die client.js und host.js aus, wird eine Socket-Verbindung zum Server aufgebaut. Diese Stern-Topologie, mit der server.js als zentralem Element, stellt den Signaling-Channel dar. Hierüber werden die Offer/Answer-Pakete für den Aufbau der RTC-Verbindung gesendet. Die Verbindungen zum Server sind mit grünen

Pfeilen versehen und überlappen nur der Übersicht halber an einigen Stellen. Es ist keinesfalls so, dass die Clients direkt untereinander verbunden sind.

Einen weiteren Verbindungstypen stellen STUN-Checks an einen STUN-Server dar, diese sind violett gekennzeichnet und auch hier überlappen die einzelnen Verbindungen nur der Übersicht halber. Auf die Darstellung der Fall-Back-Lösung über einen TURN-Server ist in dieser Übersicht verzichtet worden.

Als rote Pfeile werden die einzelnen RTC-Verbindungen in der Übersicht dargestellt. Diese werden aufgebaut, nachdem der Offer/Answer-Workflow erfolgreich beendet, gültige Candidates über die STUN-Checks ermittelt und sich über die Connectivity Checks auf einen Transportweg geeinigt wurde.

3.2.2 Technologie-Übersicht

Trotz des Bestrebens, bei der Entwicklung möglichst auf Fremdkomponenten, welche in den Signaling-Prozess bzw. Verbindungsaufbau eingreifen, zu verzichten, ist es für moderne Web-Applikationen nicht ungewöhnlich, dass eine Vielzahl von Einzel-Komponenten zur Lösung komplexer Anwendungsfälle eingesetzt werden. Dieses Kapitel visualisiert die eingesetzten Technologien in einem Schichtenmodell und erläutert diese.

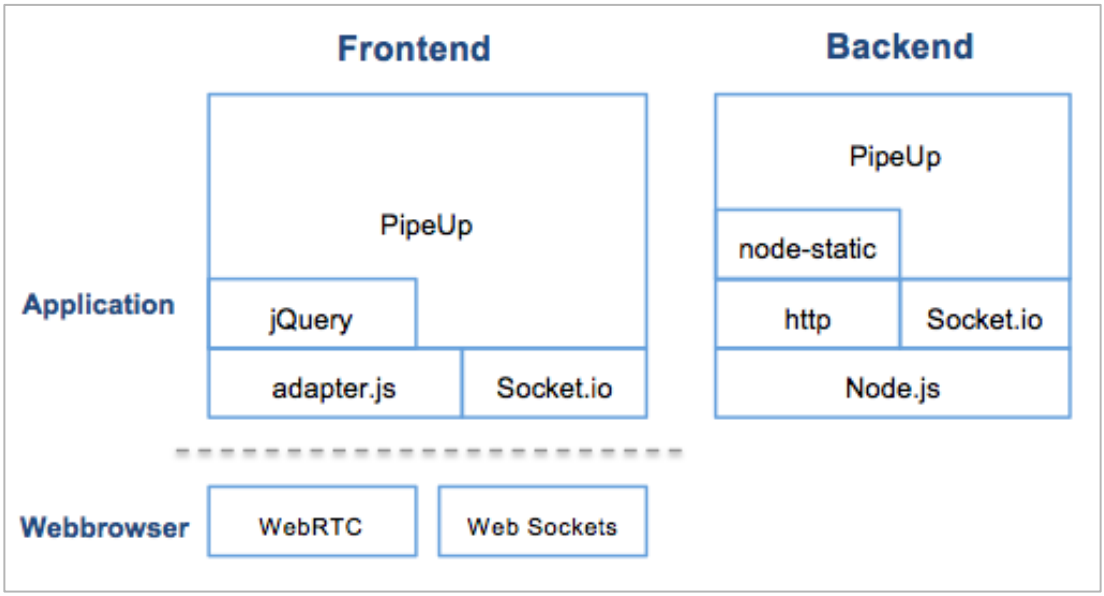


Abbildung 23: PipeUp Technologie-Übersicht

Frontend

Die beiden Frontend-Anwendungen, für den Client und den Host, haben denselben technologischen Hintergrund. Sie bauen auf die beiden Webbrowser-APIs WebRTC und Web Sockets auf. Web Sockets wird aus Anwendungssicht wiederum von dem Socket.io-Framework und WebRTC von der adapter.js-Bibliothek gekapselt. Im Falle

von Socket.io lässt sich von einem klassischen Framework sprechen, welches dem Anwendungsentwickler eine Menge Aufgaben abnimmt und neue Funktionen bereitstellt. Adapter.js ist da etwas anders geartet, es begradigt die unterschiedlichen Vendor-Prefixes der Webbrowser, stellt aber keine neuen Funktionen bereit.

Zusätzlich kommt jQuery zum Einsatz um im Frontend Benutzerinteraktionen bequemer und schlanker zu handhaben.

Backend

Der Aufbau des Backends ist ungleich simpler, da als Grundlage *`Node.js`* eingesetzt wird und darauf die beiden Node-Plugins *`http`* und *`node-static`* aufsetzen. Für den Aufbau der Socket-Verbindungen mit den Client-Webbrowsern wird die Server-Komponente von *`Socket.io`* eingesetzt.

adapter.js

Die einzige Ausnahme zu dem Verzicht auf RTC-Frameworks stellt adapter.js dar. Diese Bibliothek sorgt dafür, dass Vendor-Prefixes gekapselt werden und ein einheitliches Interface für die Entwicklung mit WebRTC bereitgestellt wird.

Vendor-Prefixes werden bei Neueinführungen von Technologien eingesetzt, die noch nicht die Reife erlangt haben um als Standard zu gelten, diese Funktionen werden dann häufig mit einem Prefix versehen.

So wird von der Gecko Engine (Firefox) beispielsweise das RTCPeerConnection-Objekt mit dem Prefix *`moz`* angesprochen und von Webkit (Chrome) mit dem Prefix *`webkit`*.

Durch das Einbinden von adapter.js werden diese unterschiedlichen Handhabungen vereinheitlicht, sodass hier keine Unterscheidung im Anwendungscode mehr gemacht werden müssen.

Socket.IO

Socket.IO ist eine JavaScript-Bibliothek für Echtzeit-Anwendungen im Webbrowser. Sie besteht aus einem clientseitigen JavaScript, welches im Webbrowser ausgeführt wird und einem serverseitigen Node.js-Plugin.

Der Focus der Bibliothek liegt darin, das WebSocket-Protokoll zu kapseln, es gibt aber auch Fall-Back-Lösungen die mit Adobe Flash Sockets[16], JSONP Polling[17] oder AJAX Long Polling[18] arbeiten.

Das WebSocket-Protokoll[19] ermöglicht es, eine vollduplex Verbindung zwischen einem Client mit nicht vertrauenswürdigen JavaScript-Code und einem Host nach einem Opt-In-Prozess aufzubauen. Um das Aufbauen mehrerer HTTP-Verbindungen zu vermeiden, dient als Protokollgrundlage das TCP.

Ein Socket.IO-Server organisiert seine verbundenen Clients in Gruppen die sich danach richten, welche Raumnamen der Client beim Verbinden angibt. Dadurch können mehrere Räume parallel von einem Server verwaltet werden. Ein Benutzer ist nicht in der Lage raumübergreifend mit anderen Benutzern zu kommunizieren.

node-static und http

Das http-Modul kümmert sich sehr low-level darum, dass Requests und Responses an und von der Node-Instanz möglich werden. Um mit den Requests umgehen zu können, wird zusätzlich noch *node-static* benötigt, damit ist es dann möglich, statische Dateien an einen Webbrowser auszuliefern.

```
var static = require('node-static'),
    http = require('http'),
    file = new(static.Server)(),
    app = http.createServer(function (req, res) {
            file.serve(req, res);
          }).listen(2013);
```

Code-Beispiel 1: Initialisieren eines Node-Web-Servers

Code-Beispiel 1 zeigt die ersten sechs Zeilen der server.js und wie einfach es ist, einen Server zu starten. Zunächst werden mit der Node.js eigenen *require*-Funktion die beiden Module geladen. Dann wird in die Variabel *file* der Server initialisiert, welcher später die statischen Dateien ausliefert. In die Variabel *app* wird anschließend der http-Server erstellt, dem in der Callback-Funktion der File-Server übergeben wird. Zusätzlich wird der http-Sever noch angewiesen auf dem Port 2013 zu lauschen.

Wird dieser Sechs-Zeiler nun ausgeführt, könnten bereits über die URL http://localhost:2013 statische Dateien von einem Webbrowser angefordert werden.

---

16 http://en.wikipedia.org/wiki/Adobe_Flash
17 http://en.wikipedia.org/wiki/JSONP
18 http://en.wikipedia.org/wiki/Comet_(programming)#Ajax_with_long_polling
19 http://tools.ietf.org/html/rfc6455

jQuery

Um möglichst bequem das DOM zu manipulieren, wurde für einige Funktionen in der Benutzeroberfläche auf jQuery gesetzt. Diese Aufgaben sind allerdings losgelöst von allem was mit WebRTC und dem Verbindungsaufbau zu tun hat, deshalb steht hier die Übersichtlichkeit des Codes im Vordergrund.

3.2.3 Dateistruktur

PipeUp besteht aus drei wichtigen Komponenten, die erste ist der Node-Server, welcher für die Distribution der html-Seiten und das Signaling über Socket.IO zuständig ist. Die Logik für diesen Server ist in der *server.js* gekapselt.

Die zweite Komponente ist die Host-Anwendung, welche über das Anfordern der *host.html* gestartet werden kann. Über das HTML werden dann die Assets *host.js*, *pipeUp.js* aus dem js-Ordner und die Dritt-Komponenten aus dem lib-Ordner nachgeladen und die *host.js* ausgeführt.

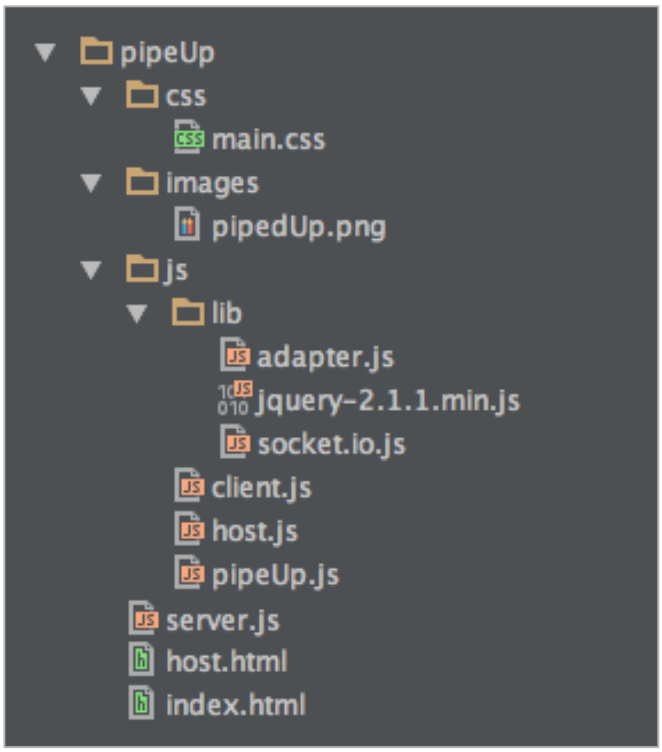


Abbildung 24: PipeUp-Dateistruktur

Nun fehlt noch die Client-Anwendung, welche über die *index.html* ausgeliefert wird. Diese lädt dann die client.js, in der die JavaScript-Logik für den Client steckt, nach.

Außerdem wird von beiden HTML-Dokumenten die *main.css,* in der die Style-Informationen für das Frontend gespeichert sind, geladen.

Das einzige Bild in dieser Anwendung ist *pipedUp.png*, es stellt ein kleines rotes Ausrufezeichen dar und dient der Visualisierung eines Benutzers in der Teilnehmerliste, wenn er die PipeUp-Funktion ausgelöst hat.

3.2.4 Objekt-Hierarchie

JavaScript implementiert kein Klassenkonzept, aber es erlaubt objektorientierte Programmierung und bietet durch seine flexible Struktur dafür gleich mehrere Konzepte. Eines dieser Konzepte wurde für diese Arbeit eingesetzt um die Komplexität der Host-Anwendung übersichtlich zu halten. Im Kapitel 4 Lessons Learned wird auf das eingesetzte Konzept noch intensiver eingegangen. An dieser Stelle soll nur die Objekt-Hierarchie erläutert werden.

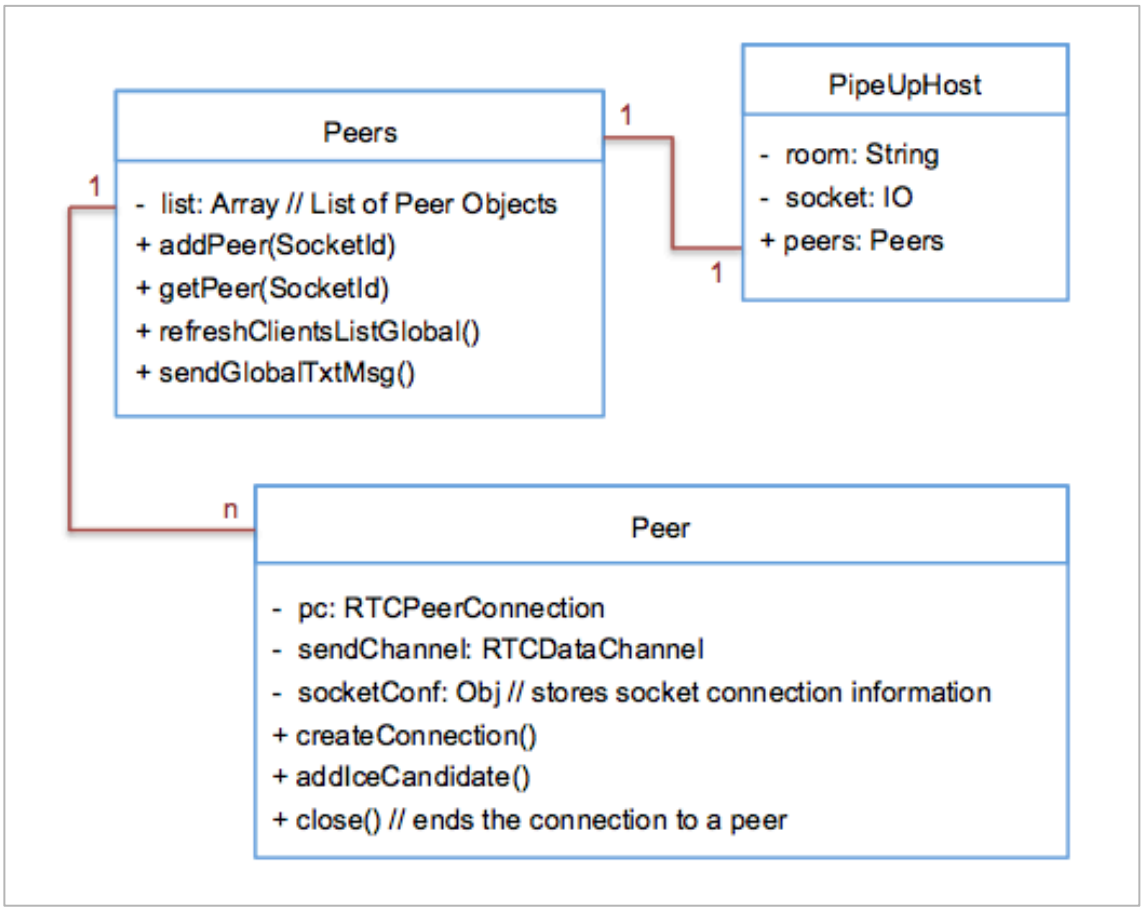


Abbildung 25: PipeUp.js Objekt-Hierarchie

Die Host-Anwendung von PipeUp instanziiert in der host.js das Objekt PipeUpHost aus der pipeUp.js. Es herrscht eine klare Aufgabenteilung, PipeUpHost ist für den Auf- und Abbau von RTC-Verbindungen zuständig und verwaltet die einzelnen Benutzer. Wie und von welchem UI-Element eine Aktion an das PipeUpHost-Objekt weitergeleitet wird, entscheidet sich in der host.js.

Abbildung 25 zeigt die drei Objekte der Datei PipeUp.js. Zu den Objekten wurden jeweils markante Variablen und Funktionen mit in die Übersicht aufgenommen, diese geben einen Einblick in die Aufgaben der Objekte. Das Minus- (-) und Plus-Symbol (+) vor den Einträgen bedeutet, diese sind privat bzw. öffentlich zugänglich.

Das Objekt PipeUpHost kümmert sich um die Kommunikation mit dem Socket-Server. Dieser wird, sobald die Host-Anwendung aufgerufen wird, mit dem Raumnamen gestartet, der in der *room*-Variabel angegeben wurde. In *socket* wird das IO-Objekt gespeichert, welches den Nachrichtenfluss vom und zum Socket-Server steuert. Das *peers*-Objekt ist eine Auslagerung der Verwaltung aller RTC-Verbindungen zu den Clients.

Alle RTC-Verbindungen werden im Peers-Objekt in das *list*-Array gespeichert. Dessen Reihenfolge stellt auch die aktuell dargestellte Reihenfolge der Benutzer im Frontend dar. Findet eine Veränderung in dieser Liste statt, wird *onListChanged()* ausgeführt, diese Funktion ist ein Callback für das Frontend und wird auch von diesem implementiert.

Verbinden sich neue Clients auf den Socket-Server, wird diese Nachricht an die Host-Anwendung gesendet. Mit dessen SocketId wird dann über *addPeer()* eine RTC-Verbindung zu diesem neuen Client aufgebaut. Die SocketId ist die primäre Identifikationsmöglichkeit eines Clients in PipeUp und wird für jeden Peer in der *socketConf* des Peer-Objektes gespeichert.

Im Peer-Objekt werden die eigentlichen WebRTC-Funktionen eingesetzt, hier werden das PeerConnection- und DataChannel-Objekt vorgehalten, der Offer/Answer-Workflow durchgespielt, ICE-Candidates verhandelt und die RTC-Verbindung auf und abgebaut.

3.3 Technische Abläufe im Detail

An dieser Stelle wird ein praxisbezogener Blick auf WebRTC geworfen, dazu werden typische Anwendungsszenarien von PipeUp untersucht und mit Code-Beispielen unterstützt.

3.3.1 Herstellen der WebSocket-Verbindung zum Server

Das Erste was passiert, wenn die Host-Anwendung gestartet wurde, ist der WebSocket-Verbindungsaufbau zum Server. Dabei ist die Host-Anwendung von PipeUp die einzige Instanz welche einen Raum[20] auf dem Socket-Server starten kann, da nur das PipeUpHost-Objekt die Create-Anweisung implementiert.

```
3  ////////////////////////////////////////////////////////////
4  /////////////// PipeUpHost Object //////////////////////////
5  ////////////////////////////////////////////////////////////
6
7  function PipeUpHost() {
8  ...
9    var self = this,
10       room = 'pipeUp',
11 ...
18       socket = io.connect();
19
20 ...
22
23   log('Create room', room);
24   socket.emit('create', room);
25
26   socket.on('created', function (room){
27     log('Created room ' + room);
28     // save socketId of host
29     hqSocketConf.socketId = this.socket.sessionid;
30   });
```

Code-Beispiel 2: PipeUpHost - WebSocket-Verbindungsaufbau zum Server

In Zeile 10 von Code-Beispiel 2 wird festgelegt, wie der Raum, zu dem sich später die Clients verbinden, heißt. Danach wird die Verbindung zum SocketServer über die Anweisung `io.connect()` hergestellt. Anschließend lässt sich mit `socket.emit()` eine Nachricht an den Socket-Server senden. Der erste Parameter gibt dabei an, welche Trigger-Funktion der Server ausführen soll und der zweite Parameter wird dieser Funktion übergeben. Zeile 24 löst also die `create`-Funktion aus und übergibt den Raumnamen. Wenn dies nicht zu Problemen führt, löst der Server seinerseits die Trigger-Funktion `created` in Zeile 26 aus und gibt ebenfalls den Raumnamen als Bestäti-

[20] Eine Erläuterung zu dem Begriff „Raum“ findet sich im Kapitel 3.2.2 Technologie-Übersicht Abschnitt Socket.IO

gung wieder zurück. Dies ist das Zeichen für PipeUpHost, die SocketId des Hosts in seinem Konfigurations-Objekt *hqSocketConf* zu speichern, Zeile 29.

Die *log*-Funktion in Zeile 23 und 27 wird im Abschnitt Debugging des Kapitels 4.3 Tooling näher erläutert.

Die Client-Sicht

Aus Sicht des Client wird die io.connect()-Funktion direkt nach Aufruf der Client-Anwendung ausgeführt und somit die Verbindung zum Server hergestellt. Allerdings wird erst nach Eingabe und Bestätigung des Benutzernamens die Funktion *join()* mit einem Konfigurations-Objekt als Parameter emittiert. Dieses enthält Informationen wie den Raum- und Benutzernamen.

```
26  socket.on('join', function (conf) {
27    if (conf.room == roomName) {
28  ...
30
31      if (masterSocket.id) { // masterSocket has to be present
32        socket.join(roomName);
33        io.sockets.socket(socket.id).emit('joined', {
34          roomName: roomName,
35          masterSocket: masterSocket.id});
36
37        io.sockets.socket(masterSocket.id).emit('joined', {
38          username: conf.username,
39          socketId: socket.id});
40      } else {
41        io.sockets.socket(socket.id).emit('denied', roomName);
42      }
43    }
44  });
```

Code-Beispiel 3: Client-Verbindungsaufbau zum Socket-Server

Die join()-Funktion des Servers checkt nun zunächst, ob der angegebene Raumname mit dem vom Host erstellten übereinstimmt (Z. 27). Dann wird überprüft, ob in dem globalen Konfigurations-Objekt *masterSocket*, die SocketId der Host-Anwendung hinterlegt ist. Denn nur, wenn auch eine Host-Anwendung auf Clients wartet, macht eine Verbindung Sinn. Ist dies nicht der Fall, wird beim Client der *denied*-Trigger ausgelöst (Z. 41). Andernfalls wird der Socket in den Raum mit aufgenommen (Z. 32) und sowohl beim Client, als auch dem Host werden die *joined*-Funktionen emittiert. Das führt beim Client dazu, dass er sich die übergebene SocketId der Host-Anwendung, Z. 35, speichert. Beim Host führt es dazu, dass er mit dem Aufbau der RTC-Verbindung beginnt, dazu mehr im nächsten Kapitel.

3.3.2 RTC-Verbindung zu neuem Client aufbauen

Nachdem nun die Socket-Verbindung zwischen Host-, Server- und Client-Anwendung eingerichtet ist, kann über diesen Signaling-Channel die RTC-Verbindung aufgebaut werden.

```
36  socket.on('joined', function (conf){
37    self.peers.addPeer(conf);
38    // save peer in peers Obj
39    log(conf.username + ' has joined');
40  });
41
42  ////////////////// SIGNALING ////////////////////
43
44  socket.on('message', function (message, from){
45    log('Received message:', message);
46    var peer = self.peers.getPeer(from);
47    if (message === 'got user media') {
48      //Verbindung erneuern
49      peer.createOffer();
50    } else if (message.type === 'answer') {
51      peer.setRemoteDescription(message);
52    } else if (message.type === 'candidate') {
53      peer.addIceCandidate(message);
54    } else if (message === 'close') {
55      // end session to peer
56      self.peers.closePeer(from);
57    }
58  });
```

Code-Beispiel 4: PipeUpHost – Signaling

Wenn die Trigger-Funktion *`joined()`* vom Socket-Server ausgelöst wird, wird im *`peers`*-Objekt die *`addPeer`*-Funktion aufgerufen (Code-Beispiel 4 - Z. 37). *`addPeer()`* wiederum erstellt ein neues *`Peer`*-Objekt, fügt dieses seinem *`list`*-Array hinzu und führt direkt die Funktion *`createConnection()`* auf dem *`Peer`*-Objekt aus. Auf diese Funktion wird später noch im Detail eingegangen. An dieser Stelle ist nur wichtig, dass außer dem Erstellen des PeerConnection-Objekts auch ein Offer-Paket über den Signaling-Channel an den Client gesandt wird. Hier beginnt der Offer/Answer-Workflow. Der Client antwortet auf die Offer über den Signaling-Channel, indem er die *`message`*-Funktion triggert (Z. 44). Diese Funktion deckt den kompletten Signaling-Prozess ab, indem je nach übergebenem *`message.type`* hier die entsprechende Funktion auf dem Peer-Objekt ausgelöst wird.

```
259   this.createConnection = function () {
260     pc = new RTCPeerConnection(pc_config, pc_constraints);
261     pc.onicecandidate = function (event) {
262       log('handleIceCandidate event: ', event);
263       if (event.candidate) {
264         pipeUpContext.sendSocketMessage({
265 ...
269         }, self.getSocketId());
270       } else {
271         log('End of candidates.');
272       }
273     };
274
275 ...
282
283     this.createSendChannel();
284     this.createOffer();
285   }
```

Code-Beispiel 5: PipeUpHost - createConnection()

Die *createConnection*-Funktion des Peer-Objektes instanziiert zunächst das RTCPeerConnection-Objekt und übermittelt ihm dabei die Konfigurationsparameter für diese Verbindung, Code-Beispiel 5 Z. 259. In *pc_config* werden ICE- und TURN-Server angegeben und *pc_constraints* beinhaltet z.B. Informationen über die Verschlüsselungsmethode oder ob der DataChannel in dieser Verbindung verwendet werden soll.

Zeile 260 zeigt, wie vom PeerConnection-Objekt (pc) die Trigger-Funktion *onicecandidate* gesetzt wird. Diese wird ausgelöst, wenn ein Connectivity-Check erfolgreich vom STUN-Server zurückgekommen ist. Ist das der Fall, wird mit *sendSocketMessage* dieser Candidate über den Signaling-Channel an den Client geschickt und verarbeitet.

Als nächstes muss noch der DataChannel, der zu jedem Client aufgebaut wird, vorbereitet werden. Dieser wird später verwendet, um Chatnachrichten direkt zum Server zu senden und nicht über den Signaling-Chanel. Dies passiert in Zeile 283 mit der Funktion *createSendChannel()*.

```
287   this.createSendChannel = function () {
288     sendChannel = pc.createDataChannel(self.getSocketId(), {reliable: true});
289     sendChannel.onmessage = self.handleAction;
290 ...
291
292     sendChannel.onopen = pipeUpContext.onPeerAdded(self);
293     sendChannel.onclose = function () {
294       log('Send channel state is: closed');
295     };
296   }
```

Code-Beispiel 6: PipeUpHost - createSendChannel()

Der DataChannel wird vom PeerConnection-Objekt erstellt (Code-Beispiel 6 Z. 288), dabei werden ihm zum Einen eine Bezeichnung in diesem Fall die SocketId und zum

Anderen ein Objekt mit Konfigurationsparametern übergeben. Hier nur die Anweisung, dass eine zuverlässige Verbindung aufgebaut werden soll – *`reliable: true`*.

Es folgen die Zuweisungen einiger Trigger-Funktionen, wie *`onmessage`*, welche ausgelöst wird, wenn der Client etwas über den DataChannel sendet, Z. 289. *`onopen`* wird ausgelöst, wenn die Verbindung zum Client hergestellt wurde und ist in diesem Fall der Moment, wo dem Frontend über die Funktion *`onPeerAdded()`* mitgeteilt wird, dass ein neuer Client sich verbunden hat. Daraufhin wird dann unter anderem die Liste der Teilnehmer im Frontend aktualisiert.

Nachdem nun also fast alle Vorbereitungen für den Verbindungsaufbau getroffen wurden, kann eine Offer für den Client vorbereitet und versendet werden. Die createOffer-Funktion wird in Code-Beispiel 5 Z. 284 aufgerufen.

```
297   this.createOffer = function () {
298   ...
303
304     if (localStream)
305       pc.addStream(localStream);
306
307     pc.createOffer(function (sessionDescription) {
308       // Set Opus as the preferred codec in SDP if Opus is present.
309       sessionDescription.sdp = preferOpus(sessionDescription.sdp);
310       pc.setLocalDescription(sessionDescription);
311       // send Offer to Client
312       pipeUpContext.sendSocketMessage(sessionDescription, self.getSocketId());
313     }, this.onError, constraints);
314   }
```

Code-Beispiel 7: PipeUpHost - createOffer()

In dieser Funktion wird nun geschaut, ob dem Peer der lokale Stream zugewiesen wurde, Zeile 304 von Code-Beispiel 7. Dies ist aber beim initialen Verbindungsaufbau nie der Fall. Diese Überprüfung spielt erst später eine Rolle, wenn das Video des Hosts zum Client übertragen werden soll.

In Zeile 307 wird dann die *`createOffer`*-Funktion aufgerufen, welche wiederum die übergebene Callback-Funktion mit dem Session-Description-Paket (SDP) als Parameter aufruft. Dies ist die Stelle an der das SDP manipuliert werden kann, in Zeile 309 wird das auch getan. In *`preferOpus()`* wird der Audio-Codec Opus, falls vorhanden, gesetzt. Dieses nun veränderte SDP wird über *`setLocalDescription()`* wieder in *`pc`* gespeichert und anschließend über den Signaling-Channel zum Client versandt.

Verbindungsaufbau aus Client-Sicht

Aus Client-Sicht laufen nach Eingang der SDP-Offer ganz ähnliche Vorgänge wie beim Host ab.

- createPeerConnection: Erstellen einer RTCPeerConnection.
- setRemoteDescription: das in der Offer mitgegebene SDP wird als Remote-Description im gerade erstellten PeerConnection-Objekt gespeichert
- pc.onicecandidate: Connectivity-Checks werden ausgeführt um ICE-Candidates zu finden. Wird einer gefunden wird onicecandidate ausgeführt und so dem Server gesendet.
- createAnswer: Eine Antwort auf die Offer wird gesendet. Widersprechen sich diese Konfigurationsparameter zwischen Offer und Answer nicht, sind sich beide Seiten einig zu welchen Bedingungen die Verbindung aufgebaut werden soll.

Ab dieser Stelle hat der Anwendungsentwickler nichts weiter zu tun, den Rest des Verbindungsaufbaus übernimmt das WebRTC-Framework. Da sich beide Seiten nun mögliche Candidates hin und her schicken, werden nach erfolgreichem Offer/Answer-Workflow die Connectivity-Checks ausgeführt. Dabei wird überprüft ob es möglich ist, eine Verbindung zwischen Candidates aufzubauen. Gelingt dies, wird bei beiden Verbindungspartnern *`oniceconnectionstatechange`* auf dem PeerConnection-Objekt aufgerufen und der Status *`succeeded`* mitgegeben.

Anschließend wird auf dem DataChannel-Objekt die Funktion *`onopen()`* ausgeführt, was für PipeUp bedeutet, dass ein neuer Client mit dem Host über eine RTC-Verbindung kommunizieren kann.

3.3.3 Anfordern der lokalen Media-Streams

Beim initialen Verbindungsaufbau zu einem Client wird nicht über Media-Streams, also Audio- oder Video-Daten, verhandelt. Dies geschieht erst später wenn der Host explizit von einem Client den Audio/Video-Stream anfordert. Das macht der Host-User indem er einen Benutzer aus der Liste auswählt und dann den Speak!-Button betätigt.

```
127   speakButton.click(function() {
128     log('ask speaker to speak: ' + currentSelectedUser);
129     pipeUp.peers.getPeer(currentSelectedUser).sendAction('getVideoAudio');
130   });
```

Code-Beispiel 8: PipeUpHost – speakButton.click()

Code-Beispiel 8 zeigt in Zeile 129 wie zunächst der Peer ausfindig gemacht wird. Dazu wird mit der SocketId aus der currentSelectedUser-Variabel *getPeer()* auf dem *peers*-Objekt ausgeführt und dann führt die sendAction-Funktion beim Client die Aktion *getVideoAudio* aus. *sendAction()* verwendet für die Übertragung der Anweisung den DataChannel und nicht den Signaling-Channel.

```
207  function handleAction(event) {
208    var action = JSON.parse(event.data);
209
210    log('Received Action: ' + action.text);
211
212    switch (action.type)
213    {
214  ...
224
225      case "getVideoAudio":
226        getUserMedia(constraints, handleUserMedia, handleUserMediaError);
227        break;
228  ...
238    }
239
240  }
```

Code-Beispiel 9: PipeUpClient - handleAction()

Je nach dem welcher *action.type* beim Client ankommt, reagiert die handleAction-Funktion unterschiedlich, s. Code-Beispiel 9. Im Falle von getVideoAudio() wird die getUserMedia-Funktion ausgeführt. Diese wird vom Webbrowser bereitgestellt und sorgt für den Zugriff auf die Kamera und das Mikrofon des Benutzers.

Abbildung 26: Sicherheitsabfrage für Kamera- und Mikrofon-Freigabe (Chrome)

Wie diese Abfrage aussieht, zeigt Abbildung 26 am Beispiel des Chrome-Webbrowsers.

Nach Bestätigung der Abfrage wird der Callback *handleUserMedia()* ausgelöst, Code-Beispiel 9 Z. 226. Dieser Funktion werden die Video- und Audio-Daten, welche live von der Hardware kommen, als Stream-Objekt übergeben. Dieses Objekt wird nun an das lokale Video-Element im DOM weitergereicht und an die Variabel

localStream übergeben. Anschließend wird noch der Host-Anwendung die Nachricht *got user media* gesendet.

Die Host-Anwendung baut daraufhin die RTC-Verbindung erneut auf, indem createOffer() erneut ausgeführt wird und somit der Offer/Answer-Workflow gestartet wird, Code-Beispiel 4 Z. 49. Dieser Schritt ist notwendig um der RTC-Verbindung einen Stream hinzuzufügen, es gibt Bestrebungen diese Maßnahme im Standardisierungsprozess in Zukunft zu umgehen.

Ist die Verbindung neuerlich aufgebaut, wird *onaddstream()* auf dem PeerConnection-Objekt der Host-Anwendung ausgeführt und ihm ebenfalls das Stream-Objekt mitgegeben. Dieses kann die Anwendung nun dem remoteVideo-DOM-Element hinzufügen und das Video starten.

3.3.4 Audio/Video-Stream vom Host zum Client

Die nachfolgend beschriebenen Anwendungsfälle beziehen sich auf die Szenarien 2 und 3, beschrieben im Kapitel 1.2.4. Szenarien. Um redundanten Inhalt zu vermeiden wird auf einige Parallelen zu vorherig beschriebenen Abläufen referenziert.

Szenario 2

Ein Teilnehmer der PipeUp-Session ist nicht Vorort und soll deshalb den Stream des Live-Events angezeigt bekommen. Dafür wird der Zugriff auf die Kamera und das Mikrofon des Hosts benötigt. Dazu wird der „Get Video“-Button in der Benutzeroberfläche betätigt, wie auch beim Client, muss der Zugriff auf die Hardware vom Benutzer bestätigt werden (s. Abbildung 26).

```
90    sendLocalVideoButton.click(function() {
91      pipeUp.peers.getPeer(currentSelectedUser).setRemoteVideo(pipeUp.getLocalStream());
92    });
```

Code-Beispiel 10: PipeUpHost - sendLocalVideo()

Anschließend kann der Host den betreffenden Benutzer aus der Liste auswählen und den „Send“-Button betätigen. Code-Beipsiel 10 zeigt in Zeile 91 den Funktionsaufruf der dann folgt. Zunächst wird der lokale Stream über *pipeUp.getLocalStream()* geholt und der Funktion *setRemoteVideo()* übergeben. *setRemoteVideo()* wird auf dem zuvor ausgewählten Peer ausgeführt, fügt der PeerConnection zu diesem Peer den Stream hinzu und startet den Offer/Answer-Workflow mit *createOffer()*. Somit wird die Verbindung erneuert und der Stream dem Client übergeben, dieser stellt diesen dann in seinem *remoteVideo*-DOM-Element bereit.

Szenario 3

Ein Teilnehmer, der nicht Vorort ist, soll den Stream eines anderen Teilnehmers, der gerade etwas beiträgt, bekommen, dazu muss der remoteStream der Host-Anwendung zu dem entfernten Benutzer „durchgeschleift“ werden.

Dazu wird zunächst wie in Szenario 1 beschrieben, eine Audio/Video-Verbindung zu dem Benutzer aufgebaut, der etwas sagen möchte. Dieses wird dann im remoteVideo-DOM-Element des Hosts dargestellt. Jetzt kann über die Liste der Benutzer derjenige ausgewählt werden, welcher diesen Stream bekommen soll. Über den Send-Button wird die Aktion dann ausgeführt. In dem Moment passiert genau dasselbe wie in Szenario 2, mit der Ausnahme, dass nicht der localStream weitergereicht wird, sondern der remoteStream.

Anschließend wird wieder der Offer/Answer-Workflow angestoßen und somit der Stream von einem Benutzer an einen anderen weitergereicht.

3.3.5 Funktion von Text-Chat und PipeUp!-Button

Um die Funktionalität des Text-Chats und die des PipeUp!-Buttons zu gewährleisten, wird die RTC-Verbindung verwendet. Genauer gesagt der DataChannel, welcher auch schon für das Aktualisieren der Benutzerliste eingesetzt wird.

```
191  function sendAction(action, content) {
192    var action = JSON.stringify({
193      type: action,
194      text: (content) ? content : ''
195    });
196    sendChannel.send(action);
197  }
```

Code-Beispiel 11: PipeUpClient - sendAction()

Über den DataChannel lässt sich grundsätzlich jede Art von Daten versenden, im Falle von sendAction() ist es ein JSON-Objekt[21]. Code-Beispiel 11 zeigt die sendAction-Funktion der Client-Anwendung. Dieses hat zwei Attribute, den *type* um dem Empfänger den Zweck dieser Nachricht klarzumachen, und *text* um den Inhalt z.B. von einer Nachricht aus dem Text-Chat zu transportieren, s. Z. 193 und 194. Wird die PipeUp-Funktion betätigt, bleibt das Text-Attribute leer, da der *type* in diesem Fall bereits genug Information darstellt.

Um die Aktion nun zu versenden, wird die send-Funktion des DataChannels bemüht, dies geschieht in Zeile 196. Der Funktion wird das JSON-Objekt einfach mitgegeben und um den Rest kümmert sich WebRTC.

---

[21] JSON: JavaScript Object Notation - ein kompaktes Datenformat in einer einfach lesbaren Textform, für den Informationsaustausch zwischen (Web)-Anwendungen gedacht

Auf der Empfängerseite wird die Trigger-Funktion *onmessage()* ausgelöst und die übertragenen Daten an diese übergeben. Anhand des *types* kann nun der Empfänger entscheiden, was daraufhin passieren soll.

Im Falle der PipeUp-Funktion wird bei dem Client, der diese Aktion sendet, das Attribut *pipedUp* im Peer-Objekt gesetzt und die Benutzerliste aktualisiert, was dazu führt, dass dieser Benutzer ein kleines Ausrufezeichen vor seinem Namen in der Liste angezeigt bekommt. Zusätzlich wird noch eine Aktion an alle verbundenen Clients versendet, sodass auch diese ihre Benutzerliste aktualisieren und somit sehen können, wer sich gemeldet hat.

Für das Versenden von Textnachrichten funktioniert das analog, diese müssen auch, nachdem Senden an den Host, wiederum an alle Teilnehmer versendet werden, damit alle den gleichen Stand des Chat-Verlaufs sehen.

3.4 ToDo

In diesem Abschnitt sollen die nächsten logischen Schritte skizziert werden, um aus dem Prototyp eine für seinen Anwendungsfall verwendbare Software zu entwickeln.

3.4.1 Benutzeroberfläche optimieren

Der Fokus des Prototyps liegt darauf die Funktionalität dieser Technologie auszutesten. Aus diesem Grund ist die Benutzeroberfläche aus Sicht der Usability nicht vollständig selbsterklärend. Um dennoch eine positive User Experience zu erzeugen, wurde darauf geachtet, dass die Oberfläche responsiv[22] auf die unzähligen Displaygrößen reagiert.

Nicht optimal realisiert sind beispielsweise die Szenarien 2 und 3, hier müssen mehrere Klicks auf verschiedene Buttons getätigt werden, was ein gewisses Maß an Einarbeitung bedeuten würde. Eine mögliche Lösung wäre ein Kontextmenü, mit allen Funktionen, die für einen Benutzer möglich sind, zu integrieren. Dieses könnte bei einem Rechtsklick auf einen Eintrag in der Benutzerliste erscheinen.

Auch werden dem Host im Moment nur in der Konsole des Browser Statusinformationen zum Verbindungsaufbau mit Clients geliefert, denkbar wäre, diese Informationen direkt im Chat-Verlauf unterzubringen.

Die Lautstärke der Audio-Streams zu verändern und die Größe der Videos variabel zu gestalten, wäre ebenfalls eine sinnvolle Erweiterung.

[22] Reponsive Webdesign: Gestalterisches und technisches Paradigma zur Erstellung von Webseiten. Damit umgesetzte Webseiten, passen sich den Rahmenbedingungen des Endgerätes an, um dem Benutzer ein möglichst hochwertiges Nutzungserlebnis zu bieten.

3.4.2 AV-Stream-Weiterleitung an Streaming-Server

Damit PipeUp die aktuelle Streaming-Lösung für die Live-Events vollends ablösen kann, muss der Stream der Session vielen entfernten Benutzern zur Verfügung gestellt werden. Das über WebRTC von einem Laptop aus zu realisieren, würde die davon profitierende Anzahl von Nutzern erheblich einschränken. Aus diesem Grund wird bisher ein Streaming-Server eingesetzt, der die Distribution des aufbereiteten Streams über das Internet vornimmt.

Eine sinnvolle Erweiterung von PipeUp wäre es, nun den lokalen Stream des Hosts an diesen Streaming-Server weiterzugeben und somit das Szenario 3 zu umgehen. Als mögliche Übertragungstechnologie käme *http-post* in Frage oder aber der Streaming-Server ist in der Lage eine WebRTC-Verbindung zur Host-Anwendung aufzubauen.

3.4.3 Refactoring der Client.js

Bei der client.js gibt es die Besonderheit, dass die Logik des Clients im Zuge des letzten Code-Reviews noch nicht mit in die pipeUp.js-Bibliothek gewandert ist und somit noch eigenständig funktioniert. Das äußert sich in weniger gut lesbarem Code, der aber mit 360LOC[23] dennoch recht überschaubar ist. Diesen mit in die pipeUp.js-Bibliothek zu übernehmen, würde die ganze PipeUp-Funktionalität übersichtlich in einer Datei zusammenfassen.

[23] LOC: Lines of Code dt. Anzahl von Codezeilen

4 Lessons Learned

Neben den grundsätzlichen Mechanismen und Zusammenhängen in Verbindung mit WebRTC, wurden noch einige Erkenntnisse gewonnen, die es wert sind, in dieser Arbeit als Lessons Learned, mit einzufließen.

4.1 Classlike-Programming in JavaScript

JavaScript wird oft bezichtigt, nicht objektorientiert zu sein und es dadurch dem Entwickler schwer zu machen, seine Anwendung gut zu strukturieren. Dies ist aber nicht der Fall, es stimmt zwar, dass es keine Klassen in ECMAScript[24] gibt, aber die flexible Architektur der Sprache gibt dem Entwickler einige Möglichkeiten, klassenähnliche Strukturen zu nutzen und somit auch objektorientiert zu arbeiten.

Für diese Arbeit wurde eine Variante verwendet, die für OO-Programmierer leicht nachzuvollziehen ist [Douglas Crockford, 2001].

```
7   function PipeUpHost() {
8     // private
9     var self = this,
10  ...
80
81    this.close = function () {
82      self.peers.closeAllPeers();
83      socket.emit('close room');
84    }
85
86  ...
98  }
```

Code-Beispiel 12: PipeUpHost - Classlike-Programming

Code-Beispiel 12 zeigt das Objekt PipeUpHost, dieses wird als *`function`* deklariert, kann aus OO-Sicht aber als Klassendeklaration angesehen werden. Die Funktion *`close()`* wird in Zeile 81 öffentlich deklariert indem sie über den this-Operator an den Scope von PipeUpHost gebunden wird.

Innerhalb der Funktion *`close()`* wird der Scope von PipeUpHost allerdings auch benötigt, mit *`this`* würde in diesem Fall aber der Scope der close-Funktion angesprochen werden. An dieser Stelle ist es sehr nützlich, dass JavaScript mit Closures arbeitet. Dies bedeutet, eine Funktion kann auf alle Variablen zugreifen, die in ihrer Parent-Funktion privat deklariert wurden. Deshalb wurde in Zeile 9 der Scope von PipeUpHost

[24] ECMAScript ist der Name für den Standard ECMA-262 auf dem JavaScript basiert. Die derzeit aktuelle Version ist 5.1, Version 1.0 wurde im Juni 1997 von der Ecma International veröffentlicht

in der Variabel *self* gespeichert und kann von innerhalb der close-Funktion erreicht werden. In der Zeile 82 wird das peers-Objekt angewiesen, die Verbindung zu allen Peers zu schließen.

Dieses Verfahren stellt eine gute Möglichkeit dar, seine Anwendung „classlike“ also klassen-ähnlich zu strukturieren.

4.2 Best Practise

Folgende sinnvolle aber nicht standardisierte Best-Practises sind während der Entwicklungsphase aufgefallen und sollten beachtet werden.

4.2.1 Signaling über den DataChannel

Wird eine RTC-Verbindung für eine Videokonferenz oder ähnliches aufgebaut, ist es sinnvoll, zunächst eine RTC-Verbindung nur mit einem DataChannel herzustellen, um diesen anschließend als Signaling-Channel für die Audio/Video-Verbindung zu verwenden. Der Signaling-Aufwand ist bei Verbindungen, die keine Audio/Video-Daten übertragen wollen, deutlich geringer. Es lohnt sich also, die deutlich schnellere direkte Verbindung über den DataChannel, für die restlichen Signaling-Aufgaben zu nutzen. Zusätzlich wird mit dieser Maßnahme auch der Server entlastet, was bei hohen Nutzerzahlen durchaus wichtig sein kann.

4.2.2 Usability-Optimierung der getUserMedia-Abfrage

Bevor der Webbrowser auf die Hardware eines Benutzers zugreifen darf, wird dieser um Erlaubnis gebeten (s. Abb. 27). Diese Abfrage ist nicht grafisch veränderbar und wird von Webbrowser zu Webbrowser an unterschiedlichen Stellen angezeigt.

Da diese Abfrage aber meist, wie jede andere vom Browser kommende Abfrage aussieht, z.B. ein Update-Hinweis, ist für den Benutzer nicht eindeutig klar, dass es sich hierbei um einen Teil der Anwendung handelt, mit der gerade interagiert wird. Es besteht also die Gefahr, dass der Benutzer gar nicht auf die Abfrage eingeht und somit die Verbindung nicht aufgebaut werden kann.

Eine mögliche Lösung für dieses Problem ist es, die grafische Oberfläche der Webanwendung so zu verändern, dass der Benutzer eindeutig auf die Abfrage hingewiesen wird. Im Falle von PipeUp wäre es denkbar, den Abdunkelung-Effekt für den initialen Login des Clients wiederzuverwenden.

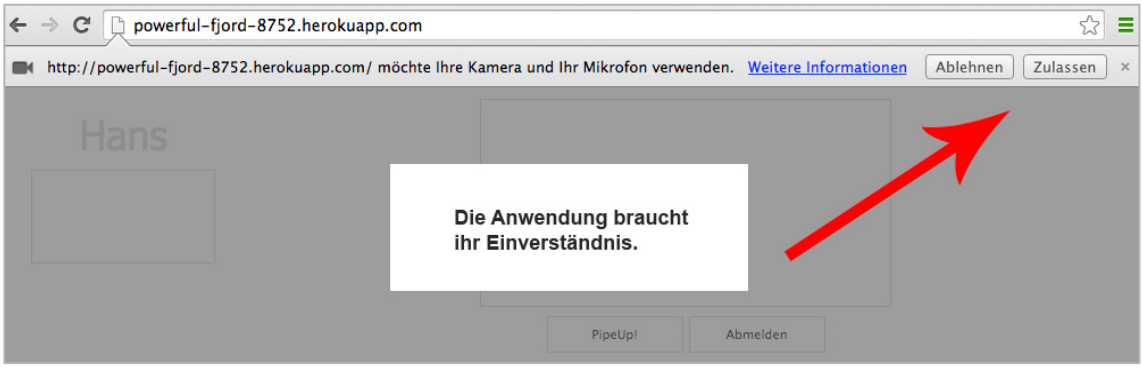


Abbildung 27: Usability-Optimierte getUserMedia-Abfrage

Abbildung 27 zeigt einen möglichen besseren Weg um dieses Problem zu umgehen. Ein Pfeil ist aber insofern schwer umzusetzen, da diese Nachricht in jedem Webbrowser woanders positioniert ist.

4.3 Tooling

Das Entwickeln im Umfeld von WebRTC unterscheidet sich nicht großartig von anderen JavaScript-Entwicklungsumgebungen, dennoch gibt es einige Werkzeuge, die einem das Programmieren erleichtern können.

Chrome: WebRTC-internals

Das Chrome-Team hat eine sehr ausführliche und aufschlussreiche Live-Übersicht von RTC-Verbindungen in den Chrome-Browser integriert. Dazu braucht nur die folgende URL in den Browser eingegeben werden:

chrome://webrtc-internals/

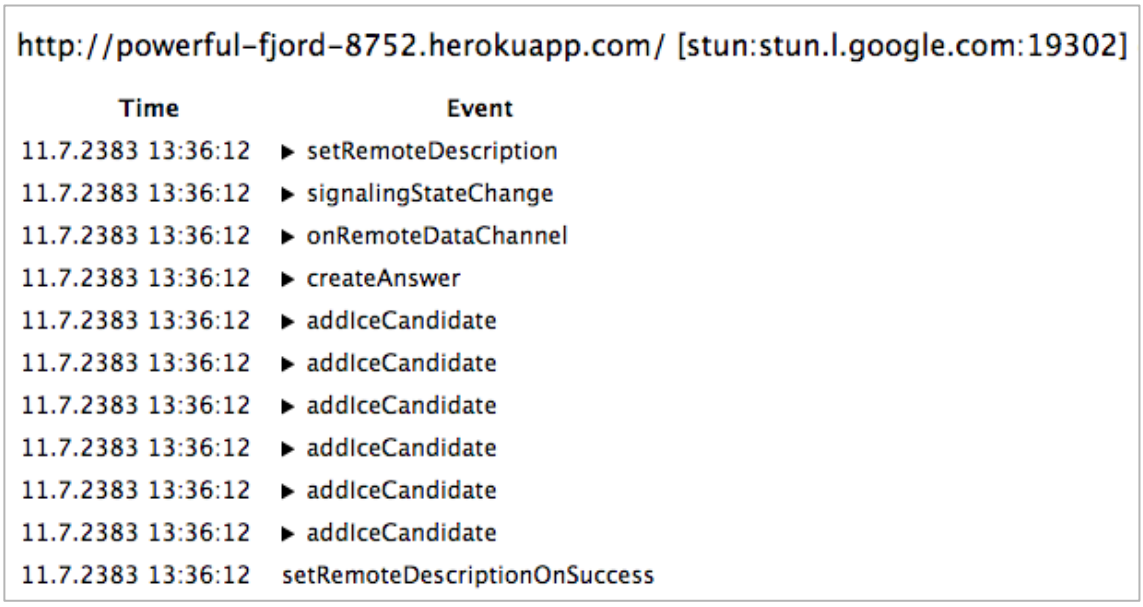


Abbildung 28: WebRTC-Internals - Offer/Answer-Pakete und ICE-Stati

Wenn eine RTC-Verbindung aufgebaut worden ist, werden auf dieser Seite die Offer-, Answer-Pakete und ICE-Stati zu dieser Verbindung angezeigt (Abb. 28). Dies kann bei der Fehlersuche mitunter sehr nützlich sein.

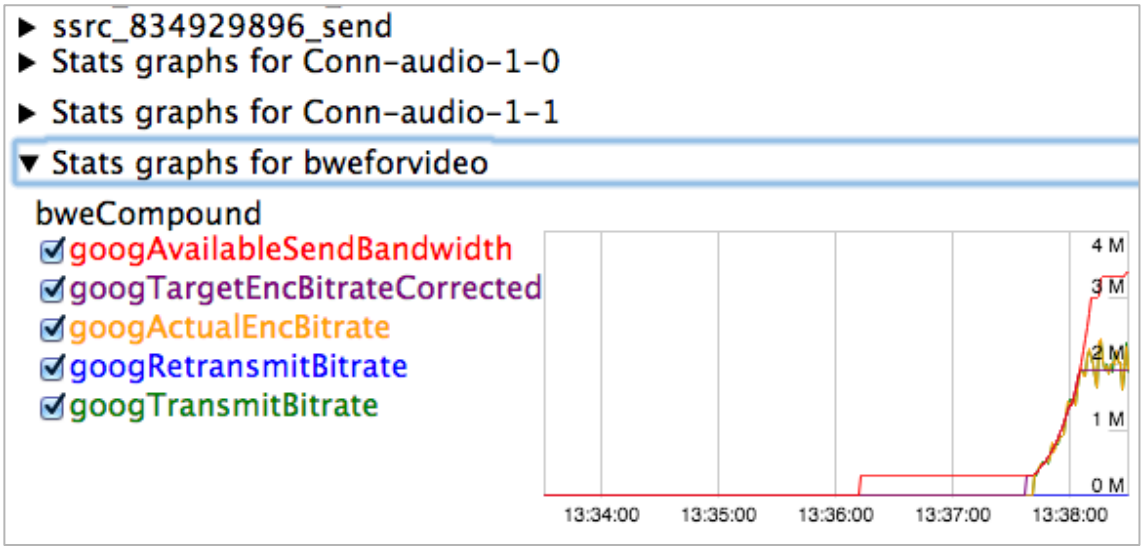


Abbildung 29: WebRTC-Internals – Verbindungsstatistiken

Außerdem werden einige Statistiken zu Verbindungsqualität, Bandbreite oder Anzahl versendeter Datenpakete angeboten.

Google Group: discuss-webrtc

Die Anlaufstelle für Fragen und aktuelle Entwicklungen rund um WebRTC ist die Google-Groups-Gruppe *`discuss-webrtc`*. Die häufigsten Fragen, die gerade am Anfang bei der Einarbeitung aufkommen, sind hier mit Sicherheit schon thematisiert und qualifiziert beantwortet worden.

WWW: https://groups.google.com/forum/#!forum/discuss-webrtc

Heroku

Um die Server-Applikation server.js öffentlich zugänglich und somit Tests über das Internet möglich zu machen, wurde der Cloud-Computing-Dienst Heroku verwendet. Bei Heroku bekommt man in der kostenlosen Variante genügend Credits, um seine Applikation mit einem Dyno, das ist die Einheit in der die Rechenleistung bei Heroku gemessen wird, rund um die Uhr zu betreiben. Für dieses Projekt hat das völlig ausgereicht.

WWW: https://www.heroku.com/

Um den Code auf den Heroku-Server zu bringen, wird ein Git-Hub-Projekt, von dem stets die aktuellen Ressourcen verwendet werden, verbunden. Zum Starten des Servers steht ein Command-Line-Tool zur Verfügung.

JS Frameworks

In dieser Arbeit wurde bewusst auf Frameworks verzichtet, welche dem Anwendungsentwickler den Blick „unter die Motorhaube“ versperren. Dennoch gibt es einige sehr gute Frameworks, von denen viel gelernt werden kann. Hier einige Beispiele.

- WebRTC-Experiment.com:
  Diese Seite bietet zu fast allen Features von WebRTC, eigene Frameworks und dazu Demo-Seiten an, um diese in Aktion zu sehen. RecordRTC.js ist in Hinblick auf die Weiterentwicklung des hier vorgestellten Prototypen besonders hervorzuheben.
  https://www.webrtc-experiment.com/
- SimpleWebRTC:
  Dieses Projekt versucht möglichst schnörkellos die Grundfunktionalität von WebRTC zu kapseln.
  https://github.com/HenrikJoreteg/SimpleWebRTC
- Peer.js:
  Ebenfalls ein Projekt mit dem Minimalismus-Ansatz und einer sehr guten Online-Dokumentation des Codes.
  http://peerjs.com/

Debugging

Für das Debugging von JavaScript hat sich eine eigene globale *log()*-Funktion als hilfreich herausgestellt, diese macht nichts weiter als zu überprüfen ob gelogged werden soll oder nicht und dann die übergebenen Parameter weiterzugeben an die *console.log()*–Funktion des Webbrowsers.

```
var doLog = true; // control logging
var log = function (data, data2) {
  if (doLog) {
    console.log(data, data2 || '');
  }
}
```

Code-Beispiel 13: Logging-Funktion

Der Switch für das Ein- und Ausschalten des Loggings ist die globale Variabel *doLog*.

5 Fazit und Ausblick

PipeUp

Der PipeUp-Prototyp hat aufgezeigt, dass mit dem aktuellen Stand der Technik alle im Vorfeld aufgestellten Anwendungsszenarien realisierbar sind. Einige Abstriche müssen allerdings bei der Qualität der Audio- und Video-Verbindung gemacht werden (QoS[25]). Der Echo-Canceling-Mechanismus, welcher Rückkopplungseffekte unterdrücken soll, könnte besser funktionieren und die automatische Anpassung des übertragenen Videos regelt trotz freier Ressourcen die Qualität konsequent nach unten. Diese Erkenntnisse sind allerdings nicht gemessen, sondern eher Subjektiver-Natur.

Aus technischer Sicht stehen der Umsetzung von PipeUp nur marginale Probleme im Weg. Ob der Benutzer sich auf so eine Neuerung einlassen wird, sollte in einem Akzeptanztest durch die Zielgruppe noch überprüft werden. Dabei wäre es ratsam, den Benutzern die Sicherheitsaspekte, welche WebRTC implementiert, klarzumachen.

WebRTC

Web-Realtime-Communication (WebRTC) hat sich von einer experimentellen Technologie zu einer legitimen Kommunikationsplattform entwickelt und hat dabei echtes Potenzial massenmarktfähig zu werden. Doch sind noch einige Hürden auf diesem Weg zu nehmen.

Allen voran fehlt die Unterstützung der beiden großen noch fehlenden Webbrowser Safari und Internet Explorer. Solange die Hersteller dieser weit verbreiteten Webbrowser nicht mit einer Unterstützung aufwarten, kann auch nicht von einem etablierten Internet-Standard die Rede sein. Des Weiteren stehen noch Rechtsfragen über das geistige Eigentum von unterstützten Video-Codecs im Raum.

[25] QoS: Quality of Service

Auch gibt es Kritik an dem Standardisierungsverfahren, im Moment wird dieses von zwei Organisationen vorangetrieben. Der IETF und dem W3C, wobei beide unterschiedliche Schwerpunkte setzen. Beim W3C steht die JavaScript-API von WebRTC im Focus und der IETF konzentriert seine Arbeit auf die Entwicklung von Kommunikations-Protokollen für WebRTC. Was von einem Standard allerdings erwartet wird, ist ein Dokument, welches alle Details der betreffenden Technologie beschreibt und konzentriert. Aus diesem Grund wurde innerhalb des W3C bereits ein weiterer Ausschuss gebildet, der ORTC (Object Real Time Communications)[26], eine Community-Group die das Geschehen um WebRTC beobachtet, ähnlich einer UN-Friedensdelegation. Diesem Ausschuss gehören unter anderem sowohl Google- als auch Microsoft-Mitarbeiter an. Die Teilnahme von Microsoft an diesem „runden Tisch“ lässt für die Zukunft von WebRTC hoffen.

Je weiter der Standardisierungsprozess voranschreitet, umso spezieller und umfangreicher werden die Frameworks, die den Anwendungsentwickler erwarten. Dieser muss sich dann vermutlich nicht mehr mit dem Signaling-Prozess, Offer/Answer-Workflow, SDP-Paketen oder dem ICE-Gathering auseinandersetzen. Ein gutes Beispiel dafür ist simpleWebRTC[27].

Da WebRTC als Standard sich allerdings nicht nur auf mobile Endgeräte und Webbrowser konzentriert, sondern den ganzen Kommunikations-Markt als Ganzes betrachtet, bin ich der Meinung, dass WebRTC sich noch viele weitere Einsatzgebiete erschließen wird. Vorstellbar sind WebRTC- gesteuerte Sicherheitskameras oder Bankautomaten.

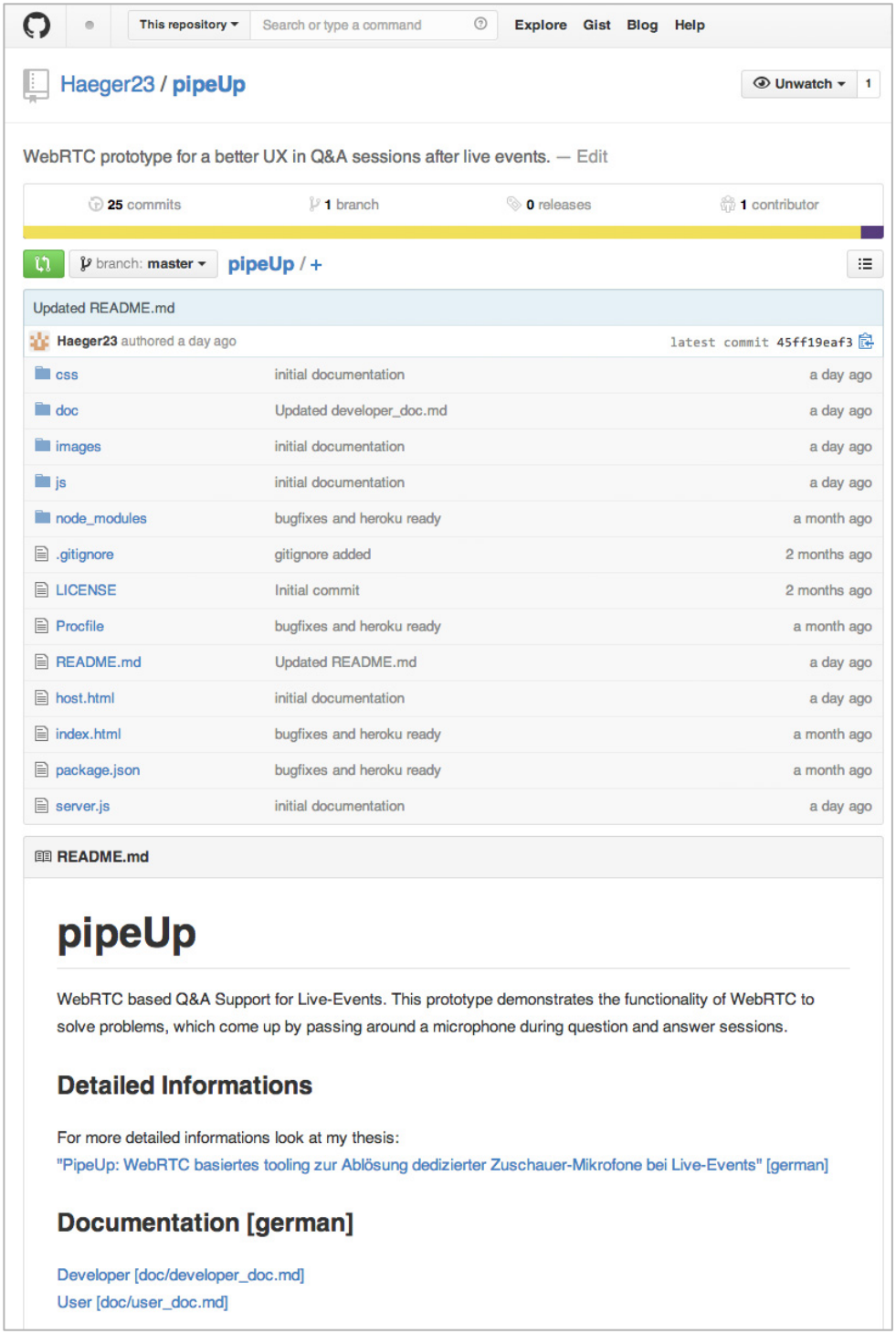
WebRTC wird in den nächsten Jahren als etabliertes Feature unserer Standard-Browser nicht mehr wegzudenken sein und neue Business-Ideen und Anwendungsfälle ins Leben rufen. Die Zukunft bleibt spannend!

[26] http://www.w3.org/community/ortc/

[27] http://simplewebrtc.com/

Anhang A: PipeUp GitHub-Repository

PipeUp ist ein Open Source Projekt und ist unter der GPL 2.0 auf GitHub veröffentlicht.



GitHub-Repository von PipeUp: https://github.com/Haeger23/pipeUp

Literaturverzeichnis

Ericsson Labs (2011):
Web Real-Time Communication
WWW: https://labs.ericsson.com/apis/web-real-time-communication/ (28.06.2014)

WebRTC Stats (2014):
WebRTC Revolution in progress!
WWW: http://webrtcstats.com/webrtc-revolution-in-progress/ (28.06.2014)

Alan B. Johnston und Daniel C. Burnett (2013):
WebRTC – APIs and RTCWEB Protocols oft he HTML5 Real-Time Web

Ilya Grigorik (2013):
High Performance Browser Networking

J. Uberti und C. Jennings (2014):
Javascript Session Establishment Protocol - draft-ietf-rtcweb-jsep-06
WWW: http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-06 (01.07.2014)

Bryan Ford (2005):
Peer-to-Peer Communication Across Network Address Translators
WWW: http://brynosaurus.com/pub/net/p2pnat/ (03.07.2014)

Alexandre Gouaillard (2014):
Overview of a peer connection lifetime
WWW: https://groups.google.com/forum/#!topic/discuss-webrtc/Z0W0zRNtrNE
(03.07.2014)

Douglas Crockford (2001):
Private Members in JavaScript
WWW: http://javascript.crockford.com/private.html (12.07.2014)

Akamai (Q1, 2014):
Akamai's State Of The Internet
WWW: http://www.akamai.com/dl/akamai/akamai-soti-a4-q114.pdf (04.07.2014)

Kriha (2013):
Multiparty Throwable Microphone, a mobile solution to capturing live discussions.
WWW: http://kriha.de/blog11.html#i157 (10.07.2014)

Chad Hart (2014):
Does Amazon's Mayday use WebRTC? A Wireshark analysis
WWW: http://webrtchacks.com/mayday-trace/ (28.07.2014)

Derek Ross (2013):
How Chromecast works: HTML5, WebRTC, and the technology behind casting
WWW: http://www.androidauthority.com/html5-and-webrtc-the-technology-behind-chromecast-248968/

Eidesstattliche Versicherung

Name:	Kampe	Vorname:	Willi
Matrikel-Nr.:	25873	Studiengang:	Computer Science and Media

Hiermit versichere ich, Willi Kampe, an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel **PipeUp: WebRTC basiertes tooling zur Ablösung dedizierter Zuschauer-Mikrofone bei Live-Events** selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO bzw. § 19 Abs. 2 Master-SPO der Hochschule der Medien Stuttgart) sowie die strafrechtlichen Folgen (siehe unten) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Auszug aus dem Strafgesetzbuch (StGB)

§ 156 StGB Falsche Versicherung an Eides Statt

Wer von einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Ort, Datum

Unterschrift