Frankfurt University of Applied Sciences
Fachbereich 2: Informatik und Ingenieurwissenschaften
Studiengang Informatik (B.Sc)

Bachelorthesis

zur Erlangung des akademischen Grades Bachelor of Science

Einsatz von WebRTC für Browserbasierte Brettspiele im Vergleich zu Client-Server Architektur

---

Autor: Robin Buhlmann
Matrikelnummer.: 1218574
Referent: Prof. Dr. Eicke Godehardt
Korreferent: Prof. Dr. Christian Baun

Version vom: 11. April 2021

Eidesstattliche Erklärung

Hiermit erkläre ich, Robin Buhlmann, dass ich die vorliegende Bachelorarbeit selbstständig und ohne unerlaubte Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Friedrichsdorf, den

Datum	Unterschrift

Vorwort

Danke und so!

Textabschnitte in ROT sind temporär und müssen neu geschrieben werden, da ich die extremst schlecht formuliert habe.

> *"Die wohl absurdeste Art aller Netzwerke sind die Computernetzwerke. Diese Werke werden von ständig rechnenden Computern vernetzt und niemand weiß genau warum sie eigentlich existieren. Wenn man den Gerüchten Glauben schenken darf, dann soll es sich hierbei um werkende Netze handeln die das Arbeiten und das gesellschaftliche Miteinander fordern und fördern sollen. Großen Anteil daran soll ein sogenanntes Internet haben, dass wohl sehr weit verbreitet sein soll. Viele Benutzer des Internets leben allerdings das genetzwerkte Miteinander so sehr aus, dass das normale Miteinander nahezu komplett vernachlässigt wird (vgl. World of Warcraft)."*
>
> – Netzwerke – www.stupidedia.org

Zusammenfassung

In dieser Arbeit wird die Anwendbarkeit von WebRTC Datenkanälen zur Entwicklung von Browserbasierten, Peer-To-Peer Mehrspieler Brettspielen untersucht. Dabei werden insbesondere die Vor- und Nachteile einer Nutzung von WebRTC im Vergleich zu traditionellen Client-Server Infrastrukturen betrachtet. Dabei wird ein prototypisches Brettspiel entworfen, wobei sämtlicher Spielrelevanter Datenverkehr über WebRTC Datenkanäle abgewickelt wird.

Abstract

This study examines the practicality of utilizing WebRTC data channels to develop browser-based multi-player board games, especially in contrast to more traditional client-server architectures. A prototype game is developed, where the entiriry of the game-relevant data is transmitted across WebRTC data channels.

Inhaltsverzeichnis

Abbildungsverzeichnis VIII

Tabellenverzeichnis IX

Abkürzungsverzeichnis X

1 Einleitung 1
1.1 WebRTC in Browserbasierten Mehrspieler-Spielen . . . 2
1.2 Zielsetzung . . . 3
1.3 Aufbau der Arbeit . . . 3

2 Grundlagen 4
2.1 Echtzeitanwendungen . . . 4
2.2 WebRTC . . . 4
2.2.1 Aufbau von WebRTC . . . 5
2.2.2 Protokolle und Frameworks . . . 6
2.3 Node.js . . . 10
2.4 socket.io . . . 10
2.5 CoTurn STUN / TURN Server . . . 11
2.6 AWS . . . 11

3 Design und Implementation der WebRTC–Infrastruktur 12
3.1 SVG Test . . . 12
3.2 Analyse . . . 12
3.3 Implementation der Peer-To-Peer Funktionalität . . . 12
3.4 Implementation des Signaling-Servers . . . 12
3.5 Aufsetzen und Konfiguration eines STUN und TURN Servers . . . 12

4 Mensch-Ärgere-Dich-Nicht 13
4.1 Spielablauf . . . 13
4.2 Analyse . . . 13
4.3 Implementation . . . 13
4.3.1 Darstellung des Spielbretts . . . 13
4.3.2 Implementation der Spiellogik . . . 13
4.3.3 Synchronisation des Spielstands . . . 13
4.3.4 Probleme der Implementation . . . 13

5 Evaluation 14

6 Zusammenfassung und Ausblick 15

7 Literaturverzeichnis 16

Abbildungsverzeichnis

1.1 Screenshots einiger Jocly-Spiele. 2

2.1 Diagramm der WebRTC-Architektur. 5
2.2 JSEP-Verbindungsaufbau. 8
2.3 Aufbau eines SCTP-Packets. 9

3.1 Ein toller SVG-Export Test. 12

Tabellenverzeichnis

2.1 Vergleich von TCP und UDP mit SCTP. 9

Abkürzungsverzeichnis

W3C World Wide Web Consortium

RTC Real-Time Communication

WebRTC Web Real-Time Communication

P2P Peer-To-Peer

HTTP Hypertext Transfer Protocol

NPM Node Package Manager

IETF Internet Engineering Task Force

RTP Real-Time Transport Protocol

SRTP Secure Real-Time Transport Protocol

JSEP JavaScript Session Establishment Protocol

ICE Interactive Connectivity Establishment

NAT Network Address Translation

STUN Session Traversal Utilities for NAT

TURN Traversal Using Relays around NAT

SCTP Stream Control Transport Protocol

DTLS Datagram Transport Layer Security

API Application Programming Interface

TCP Transmission Control Protocol

UDP User Datagram Protocol

OSI Open Systems Interconnection

IP Internet Protocol

SDK Source Development Kit

SIP Session Initialization Protocol

SDP Session Description Protocol

1 Einleitung

Am 26. Januar 2021 veröffentlichte das World Wide Web Consortium (W3C) die Web Real-Time Communication (WebRTC) Recommendation. Eine Recommendation des W3C ist dabei ein offizieller Web-Standard in seiner – bezüglich der zentralen Funktionalität – finalen Form. WebRTC ist eine Peer-To-Peer Web-Technologie, und wird primär für Echtzeit-Applikationen wie Audio- und Videochats verwendet.

Die Beliebtheit von interaktiven Mehrspieler-Brettspielen, welche durch das einfache Abrufen einer Webseite spielbar sind, steigt von Jahr zu Jahr, nicht zuletzt bedingt durch die seit März 2020 andauernde Covid-19 Pandemie. Die Auswahl an verschiedenen Spielarten fällt dabei divers aus – von Schach bis hin zu Karten- oder Gesellschaftsspielen, wie zum Beispiel Monopoly.

In der Regel basieren diese Spiele auf einer Client-Server-Architektur, wobei der Server die Rolle des bestimmenden Spielleiters übernimmt. Dabei werden sämtliche Aktionen eines Spielers über den Server validiert, und mit anderen Spielern synchronisiert. Die Clients sind dabei lediglich für die Darstellung des vom Server verwalteten Spielstands zuständig.[Bur12]

Ein weiterer, gut definierter und häufig genutzter Ansatz für die Vernetzung von Spielern ist die P2P Architektur. Bei dieser existiert kein zentraler Server, die Nutzer (Peers) sind gleichberechtigt und tauschen Daten direkt untereinander aus. Das Spiel wird dabei entweder von einem der Peers (dem sogenannten 'Host') orchestriert, oder ist auf alle Peers verteilt. Einer der großen Vorteile der Peer-To-Peer Architektur sind dabei die geringeren Datenmengen, welche über einen zentralen Server verwaltet werden müssen. Dies führt zu Kostenersparnissen in Form von weniger Bedarf an Hardware.

Beide dieser Netzwerkarchitekturen finden in der Spieleentwicklung Anwendung – jedoch ist die Nutzung von Peer-To-Peer zum Datenaustausch bei Browserbasierten Mehrspieler-Spielen begrenzt. Dies ist nicht zuletzt auf das Lebensende des Adobe Flash Players im Dezember 2020 zurückzuführen, welcher über Peer-To-Peer Fähigkeiten, ermöglicht durch das Real-Time Media Flow Protocol von Adobe, verfügte. Seitdem existiert – mit Ausnahme von WebRTC – keine alternative Möglichkeit um Nutzer von Web-Applikationen, ohne die Nutzung von Plug-Ins oder Drittanbietersoftware, direkt untereinander zu vernetzen.

1.1 WebRTC in Browserbasierten Mehrspieler-Spielen

WebRTC wird bereits in einigen Mehrspieler-Spielen, sowie Networking- und Spiel-Frameworks verwendet. In der Regel wird WebRTC dabei jedoch lediglich für Sprach- und Videokommunikation eingesetzt. Die Strategie- und Brettspiel Plattform *Jocly* ist eine der ersten Plattformen, welche bereits seit 2013 WebRTC nutzt, damit Spieler sich in Echtzeit über ihre Webcams beim Spielen sehen, sowie miteinander kommunizieren können[LL13] (vgl. Abbildung 1). Ähnlich verhält es sich bei einigen weiteren Frameworks, wie zum Beispiel *Tabloro*[1].

Abbildung 1.1: Screenshots einiger Jocly-Spiele.

Es existieren zudem einige Spiele sowie Frameworks, welche WebRTC zum Austausch von Spielrelevanten Daten nutzen. Bei diesen handelt es sich jedoch in der Regel um Prototypen und Frameworks, welche auf Echtzeit-Applikationen ausgelegt sind. Zum Beispiel die Netzwerk-Bibiliothek *HumbleNet*. Bei HumbleNet handelt es sich um eine Peer-To-Peer Bibliothek zum portieren von Peer-To-Peer Spielen in Browserumgebungen[RV17].

Weiterhin existieren eine Reihe an prototypischen Spielen, welche WebRTC zum Austausch von Daten nutzen. Bei diesen handelt es sich überwiegend um Echtzeit-Spiele wie das 2013 von Google entwickelte *Cube Slam*[2], oder den von Mozilla entwickelten

[1] vgl. `https://github.com/fyyyyy/tabloro`
[2] vgl. `https://experiments.withgoogle.com/cube-slam`

First-Person-Shooter *Bananabread*[3], welche WebRTC zum direkten Austausch von Spielrelevanten Daten zwischen Spielern nutzen.
TODO: Falsch - CubeSlam nutzt WebRTC auch nur für integrierten Videochat

Im Bereich der Rundenbasierten Brettspieleentwicklung findet WebRTC hingegen nur begrenzt Anwendung. Diese beschränkt sich primär auf die zuvor beschriebene Nutzung für Audio- und Videokommunikation.

1.2 Zielsetzung

In dieser Arbeit soll ein Brettspiel, sowie sämtliche benötigten Komponenten zum Aufbau von Peer-To-Peer Netzwerken via WebRTC, prototypisch entworfen, implementiert und aufgesetzt werden. Ziel ist es, darauf basierend die Anwendbarkeit von WebRTC für die Entwicklung von Brettspielen im Browser unter Nutzung von Peer-To-Peer Netzwerken zu evaluieren. Dabei soll primär auf Vor- und Nachteile einer Nutzung von Peer-To-Peer Netzwerken via WebRTC im Vergleich zu Client-Server Modellen eingegangen werden.

1.3 Aufbau der Arbeit

Im Kapitel 'Grundlagen' werden zunächst WebRTC an sich, sowie weitere verwendete Web-Technologien beschrieben, um eine Theoretische Grundlage für die spätere Implementierung zu schaffen.

Daraufhin wird das Design, sowie die Implementation einer WebRTC-Infrastruktur im nächsten Kapitel beschrieben. Zudem werden die benötigten Server prototypisch in der Microsoft-Azure Cloud aufgesetzt.

Basierend auf der, im vorherigen Kapitel implementierten WebRTC Infrastruktur wird eine prototypische Web-Applikation erstellt, welche mehreren Spielern das Spielen des Brettspiels 'Mensch ärgere Dich Nicht' ermöglicht. Dieses Spiel wurde gewählt, da es mit mehr als zwei Spielern spielbar ist, und das Regelwerk des Spiels hinreichende Komplexität aufweist, um den Peer-To-Peer Ansatz auf die Probe zu stellen.

Daraufhin werden – basierend auf der Implementation des Spiels und der unterliegenden Netzwerkinfrastruktur – die Limitationen sowie Möglichkeiten, welche WebRTC im Umfeld der Spieleentwicklung im Browser eröffnet diskutiert. Diese wird primär mit Client-Server Architektur verglichen.

Am Ende dieser Arbeit folgt ein abschließendes Fazit, sowie ein Ausblick für weitere Arbeit in diesem Themenbereich.

[3] vgl. https://hacks.mozilla.org/2013/03/webrtc-data-channels-for-great-multiplayer/

2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Implementierung benötigten theoretischen, sowie technischen Grundlagen beschrieben. Dabei werden primär die Funktionalität von WebRTC an sich, als auch die damit verbundenen Signal-Mechanismen und Infrastrukturen behandelt. Zudem wird ein kurzer Überblick über die Cloud-Computing Plattform Microsoft Azure gegeben, da diese als prototypische Deployment-Plattform verwendet wird.

2.1 Echtzeitanwendungen

Unter den Begriff Echtzeitanwendung fällt prinzipiell jede Anwendung, deren von Nutzern ausgelöste Ereignisse nur gewisse, in der Regel für den Nutzer nicht wahrnehmbare Verzögerungen aufweisen dürfen. Ein Beispiel für Echtzeitanwendungen sind Audio- und Videokommunikationsprogramme. Die Audio- und Videodaten müssen schnellstmöglich zwischen den Teilnehmern eines Anrufs ausgetauscht werden, um den Eindruck zu vermitteln, dass die Gesprächsteilnehmer direkt miteinander sprechen. Spricht zum Beispiel eine Person, so muss der Ton zur nahezu gleichen Zeit bei allen anderen Personen, welche dem Anruf teilhaben, ankommen.

2.2 WebRTC

Bei WebRTC handelt es sich um einen Quelloffenen Standard zur Echtzeitkommunikation zwischen Browsern. Im Gegensatz zu weiteren Echtzeitstandards, wie zum Beispiel WebSockets, setzt WebRTC nicht auf ein Client-Server Modell. Stattdessen ermöglicht der Standard es Browsern, welche den WebRTC Standard unterstützen, sich ohne zusätzliche Software oder Plugins direkt miteinander zu verbinden. Der Datenaustausch findet somit direkt zwischen den Browsern – den sogenannten Peers – statt, ohne dass die Daten zusätzlich über einen Server weitergeleitet werden müssen. Dies führt in der Regel zu geringeren Latenzen, sowie Kostenersparnissen durch weniger Serverlast.

Der Standard wird von einer Arbeitsgruppe des W3C, der WebRTC Working Group (dt. WebRTC Arbeitsgruppe) entwickelt und erhalten. Insgesamt 18 Organisationen sind in der WebRTC Arbeitsgruppe vertreten, unter anderem Microsoft, Google, Mozilla, Cisco und Apple [web].

2.2.1 Aufbau von WebRTC

WebRTC ist kein proprietärer, einzelner und zusammenhängender Standard, sondern eine Ansammlung bereits existierender Protokolle, Technologien und Standards, welche unter anderem den Aufbau von Verbindungen, Audio- und Videoübertragung, sowie Datenübertragung regeln.

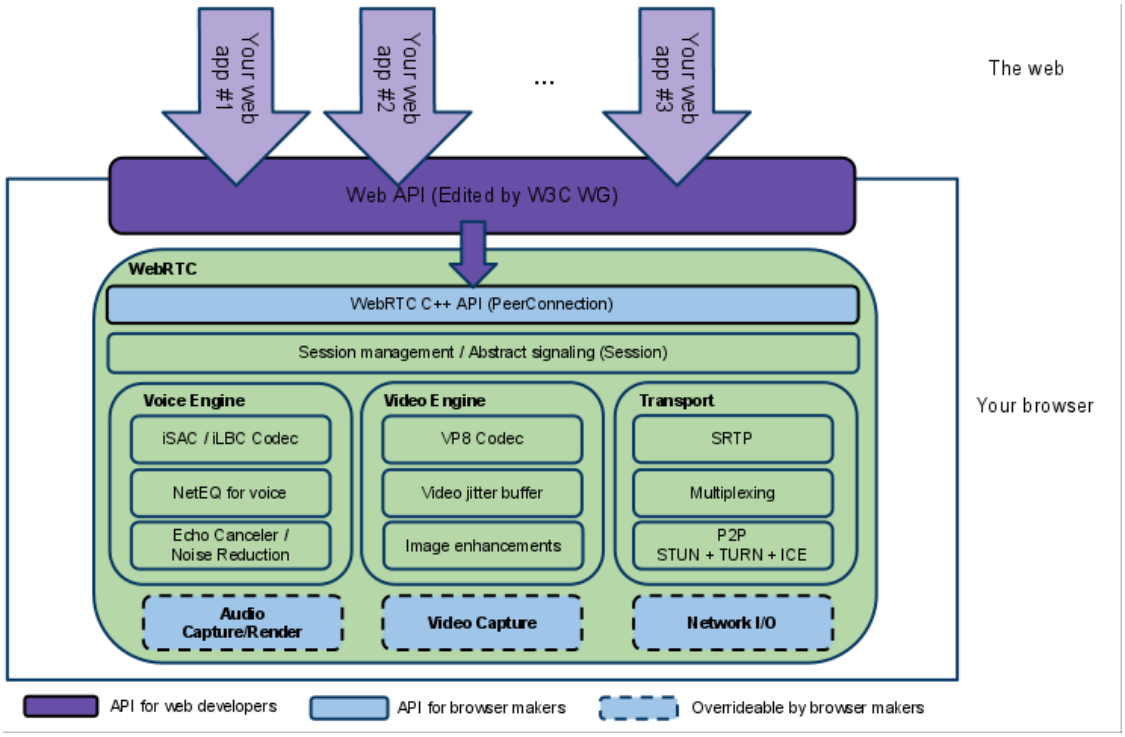

Abbildung 2.1: Diagramm der WebRTC-Architektur.
Quelle: `https://webrtc.github.io/webrtc-org/architecture/`

Wie dem Architekturdiagramm in Abbildung 2.1 zu entnehmen, gliedert sich WebRTC primär in eine Web-API und das WebRTC-Framework. Hinzu kommen Signalisierungsmechanismen, welche zum Aufbau einer Verbindung benötigt werden. Diese sind nicht durch den WebRTC-Standard vorgeschrieben. Es ist dem Entwickler überlassen, wie die Signalisierung letztendlich implementiert wird – es muss lediglich möglich sein, Daten zur Sitzunsinitialisierung zwischen jeweils zwei Peers auszutauschen.

Web-API

Die Web-API dient dabei zur Entwicklung von Browserbasierten Webanwendungen, und setzt sich aus einer Reihe an JavaScript Schnittstellen zusammen. Diese Schnittstellen können auf das unterliegende Framework zugreifen, und ermöglichen zum Beispiel das Erstellen von Verbindungen, Datenkanälen und Video-Streams. Primär werden dabei die folgenden Schnittstellen verwendet:

- Die **RTCPeerConnection**-Schnittstelle repräsentiert eine WebRTC-Verbindung zwischen dem lokalen Browser (Local-Peer), und einem externen Browser (Remote-Peer) [Con21]. Die Signalisierungsmechanismen folgen dabei dem JavaScript Session Establishment Protocol (JSEP), die Verbindung selbst wird via dem Interactive Connectivity Establishment (ICE) Framework hergestellt.
- Ein **RTCDataChannel** ist ein, von der RTCPeerConnection erstellter, bidirektionaler Datenkanal, welcher den Austausch von arbitraren Nachrichten zwischen Browsern ermöglicht. Eine RTCPeerConnection kann mehrere Datenkanäle besitzen. Zum Datenaustausch wird das Stream Control Transport Protocol (SCTP) verwendet [Con21].
- Die **MediaStream**-API dient dazu, Audio- und Videosignale eines Gerätes abzurufen. Dabei wird ein Datenstrom erzeugt, welcher in Echtzeit an externe Browser gesendet werden kann. Dazu wird das Real-Time Transport Protocol (RTP), beziehungsweise das Secure Real-Time Transport Protocol (SRTP) verwendet. Da Audio- und Videoübertragung für diese Arbeit nicht relevant sind, wird auf diese Protokolle nicht weiter eingegangen.

WebRTC Framework

Das WebRTC-Framework gliedert sich primär in Audio- Video- und Übertragungssysteme. Die Audio- und Videosysteme befassen sich dabei unter anderem mit der Abfrage von Audiodaten des Gerätemikrofons, sowie Videodaten über eine Kamera, welche an das Gerät angeschlossen ist. Zudem sind diese Systeme für die en- und decodierung von Audio- und Videodaten auf Basis verschiedener „Codecs“ zuständig. Auf diese wird hier nicht weiter eingegangen.

Die Transportsysteme umfassen Protokolle und Systeme, um Sitzungen zwischen Peers aufzubauen, und Daten zwischen den Peers zu versenden. Die Sitzungskomponenten basieren dabei auf „libjingle“, einem Quelloffenen C++ Source Development Kit (SDK), welches das Erstellen von Peer-To-Peer Sitzungen ermöglicht. Hinzu kommen Protkolle wie STUN, TURN und ICE, welche im Unterpunkt „Protokolle“ näher erläutert werden.

2.2.2 Protokolle und Frameworks

Die mit der Implementierung zusammenhängenden Protokolle werden im Fogenden näher erläutert – sowohl deren Funktionalität, als auch deren Zusammenspiel untereinander.

JSEP

Die Signalisierungsebene einer WebRTC-Anwendung ist nicht vom WebRTC-Standard definiert, damit verschiedene Applikationen mitunter verschiedene

Signalisierungsprotkolle, wie zu Beispiel das Session Initialization Protocol (SIP), oder ein proprietäres Protokoll nutzen können.

Das JavaScript Session Establishment Protocol (JSEP) erlaubt es einem Entwickler, die volle Kontrolle über die unterliegende Zustandsmaschine des Signalisierungsprozesses zu haben, welche die Initialisierung einer Sitzung kontrolliert. Damit werden die Signalisierungs- und Datenübertragungsebene effektiv voneinander getrennt. Eine Sitzung wird immer zwischen zwei Endpunkten etabliert, einem initiierendem Endpunkt, und einem empfangenden Endpunkt. In den folgenden Paragraphen werden die Synonyme „Alice“ und „Bob“ für diese Endpunkte verwendet.

Beide Endpunkte besitzen dabei jeweils eine lokale, und eine externe Beschreibung (eng. „localDescription“ und „remoteDescription“). Diese definieren die Sitzungsparameter, zum Beispiel welche Daten auf der Senderseite versendet werden sollen, beziehungsweise welche Daten auf der Empfängerseite zu erwarten sind, oder Informationen über verwendete Audio- und Videocodecs. Diese Informationen werden über das Session Description Protocol (SDP) definiert.

Die JSEP-API stellt dabei Funktionen zur verfügung, welche das Erstellen der Beschreibungen ermöglichen. Diese Funktionen sind in der WebRTC-API teil der RTCPeerConnection.

Um eine Verbindung aufzubauen, ruft Alice erst $createOffer()$ auf. Daraufhin wird ein SDP-Packet ($anfrage$) generiert, welches die lokalen Sitzungsparameter enthält. Alice setzt nun ihre lokale Beschreibung via $setLocalDescription(anfrage)$. Das SDP-Packet wird über einen nicht vorgegebenen Signalkanal zu Bob gesendet. Dieser setzt daraufhin die externe Beschreibung der Verbindung via $setRemoteDescription(anfrage)$, und ruft daraufhin die Funktion $createAnswer(anfrage)$ auf, welche eine Antwort ($antwort$) generiert. Bob setzt seine lokale Beschreibung via $setLocalDescription(antwort)$, und sendet die Antwort zurück zu Alice. Alice setzt ihre externe Beschreibung via $setRemoteDescription(antwort)$. Damit ist der anfängliche Austausch von Sitzungsparametern abgeschlossen [**?**]. Der vereinfachte Ablauf des Verbindungsaufbaus ist Abbildung 2.2 zu entnehmen.

ICE

STUN und TURN

SCTP

Zur Übertragung von Daten via RTCDataChannels nutzt WebRTC das Stream Control Transport Protocol (SCTP). Der SCTP Standard wurde erstmals im Jahre 2000 von der IETF veröffentlicht, und seitdem weiterentwickelt und erweitert. SCTP ist ein zuverlässiges, Nachrichtenorientiertes Transportprotokoll, welches im Open Systems Interconnection (OSI)-Referenzmodell, ähnlich wie UDP oder TCP, auf der Transportschicht liegt. Das Protokoll arbeitet dabei basierend auf verbindungslosen

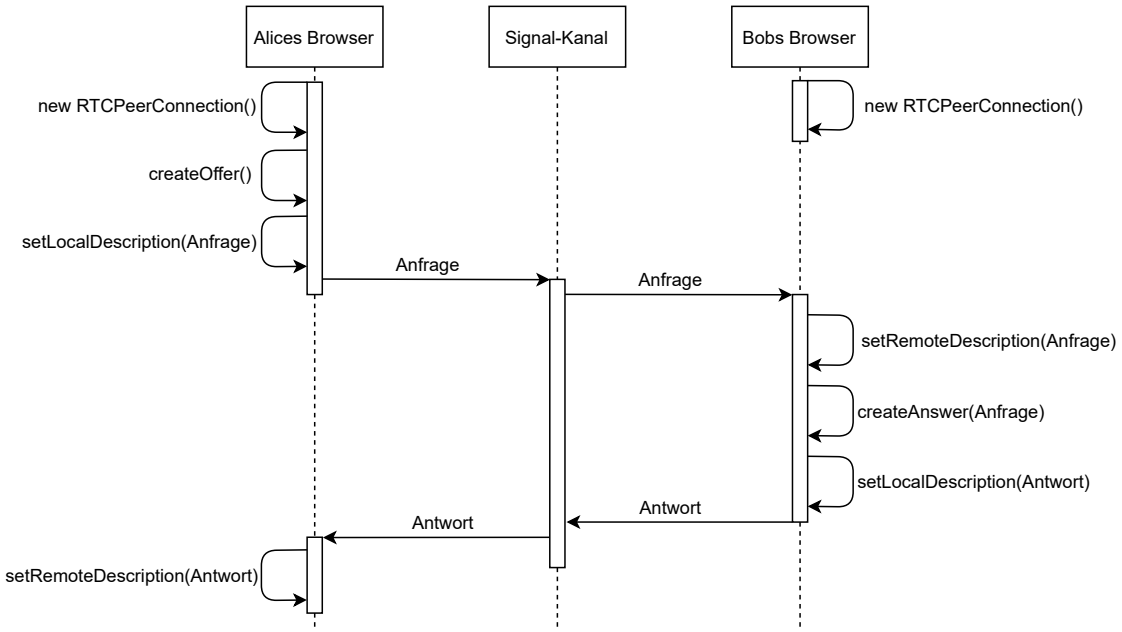

Abbildung 2.2: JSEP-Verbindungsaufbau.

Netzwerkprotokollen wie zum Beispiel dem Internet Protocol (IP) [Ste07].

Im Gegensatz zu TCP und UDP lassen sich bei SCTP, je nach gewünschter Verbindungsart, die folgenden Aspekte unabhängig voneinander frei konfigurieren:

- **Reihenfolge**: SCTP ermöglicht es, sowohl geordnete, als auch ungeordnete Datenströme aufzubauen. Falls ein Datenstrom geordnet ist, so müssen die Datenpakete in der Reihenfolge beim Empfänger ankommen, wie sie vom Sender losgeschickt wurden. In ungeordneten Datenströmen ist die Reihenfolge der Pakete, wie sie beim Empfänger ankommen, nicht relevant [Ste07].

- **Zuverlässigkeit**: Die Zuverlässigkeit der Paketlieferungen ist auf zwei Arten konfigurierbar. Es ist möglich, eine maximale Anzahl an Versuchen festzulegen, mit welcher versucht wird, ein Datenpaket zu versenden. Zudem kann eine maximale Lebenszeit für Pakete angegeben werden. Ist diese Lebenszeit, das sogenannte 'Retransmission Timeout' für eine Paketsendung abgelaufen, so wird kein weiterer Versuch unternommen, das Paket abzuschicken [Ste07].

- **Datenströme**: Im Gegensatz zu TCP und UDP ermöglicht SCTP Multiplexing auf Basis von mehreren, separaten sowie parrallelen Datenströmen innerhalb einer Verbindung. Dazu ist der Datenteil eines SCTP-Packets in sognenannte „Chunks“ aufgeteilt, wobei Daten-Chunks jeweils einem Datenstrom zugeordnet werden können [Ste07].

Ein direkter Vergleich der drei Protokolle lässt sich aus Tabelle 2.1 entnehmen. Im Gegensatz zu UDP bietet SCTP außerdem Datenfluss- und Stausteuerung. Damit gestaltet sich SCTP weitaus flexibler als die beiden gängigsten Transportprotokolle. Zudem

	TCP	UDP	SCTP
Nachrichtenordnung	Geordnet	Ungeordnet	Konfigurierbar
Zuverlässigkeit	Zuverlässig	Unzuverlässig	Konfigurierbar
Datenflusssteuerung	Ja	Nein	Ja
Stausteuerung	Ja	Nein	Ja
Multihoming	Nein	Nein	Ja
Mehrere Datenströme	Nein	Nein	Ja

Tabelle 2.1: Vergleich von TCP und UDP mit SCTP.

ermöglicht SCTP Multihoming. Existiert mehr als eine Transportadresse, unter der ein Endpunkt erreicht werden kann, so ist es möglich, im Falle eines Ausfalls des Pfades zum primär verwendeten Endpunkt, Daten über einen weiteren Netzwerkpfad umzuleiten, und an einen weiteren Endpunkt zu verschicken [Ste07, Dim16]. Dies erhöht die Zuverlässigkeit der Datenübertragung, und ermöglicht es selbst bei Ausfall eines Netzwerkpfades, Daten weiterhin auszutauschen.

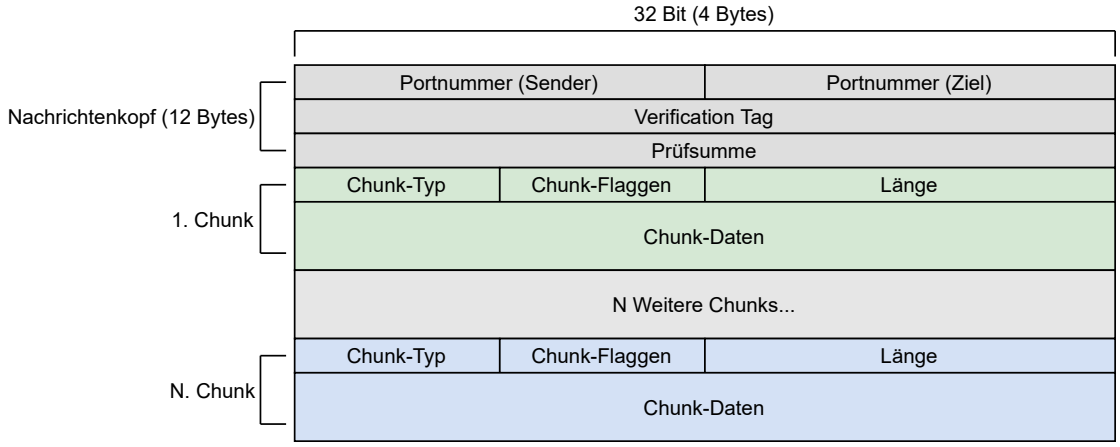

Abbildung 2.3: Aufbau eines SCTP-Packets.

Ein SCTP-Packet besteht aus einem Kopf- und einem Datenteil. Der Kopfteil beinhaltet neben dem Quell- und Zielport, sowie einer Prüfsumme noch ein „Verification Tag“, welches verwendet wird, um auf der Empfängerseite den Absender des Packets zu verifizieren, und eingehende Packete von denen früherer Verbindungen zu unterscheiden [Ste07]. Der Datenteil des Packets ist in sogenannte „Chunks“ aufgeteilt. Jedes „Chunk“ besitzt dabei Informationen wie den Chunk-Typ, die Länge in Bytes, oder die Zugehörigkeit zu Datenströmen. Insgesamt existieren über 14 Chunk-Typen, von einfachen Daten-Chunks zu Chunks, welche Daten zur Sitzungskontrolle beinhalten [Ste07]. Je nach verwendeten Erweiterungen des SCTP-Protokolls kann die Anzahl der Chunk-Typen variieren. Alle Arten dieser „Chunks“ zu beschreiben würde den Rahmen dieser Arbeit überschreiten. Die generelle Struktur eines Datenpackets ist Abbildung 2.3 zu entnehmen. Die maximale größe eines „Chunks“ ist dabei, aufgrund des zwei-Byte langen Längenfelds auf 65,535 Kilobyte limitiert.

DTLS

2.3 Node.js

Node.js ist eine kostenlose, plattformunabhängige JavaScript Laufzeitumgebung. Diese ermöglicht das Ausführen von JavaScript Programmen außerhalb eines Browsers, zum Beispiel auf einem Server. Node.js ist Open-Source und kann kostenlos verwendet werden. Programme setzen sich aus sogenannten *modules*, zu Deutsch Modulen, zusammen. Ein Modul kann dabei jegliche Funktionalität, wie zum Beispiel Klassen, Funktionen und Konstanten exportieren, welche dann wiederrum von weiteren Modulen oder Programmen verwendet werden können. Module können über das *require*-Stichwort geladen werden. Node.js bietet integrierte Webserver-Funktionalität via dem HTTP-Modul, welches das Erstellen eines Webservers ermöglicht [nod]. Eine JavaScript-Datei kann über den Befehl

```
$ node <Pfad zur Skript-Datei>
```

ausgeführt werden.

Node.js ist in den Repositories aller aktuellen Linux-Distributionen enthalten[1], und kann mit den entsprechenden Packetmanagern, beziehungsweise über die Website[2] heruntergeladen und installiert werden.

Zur Verwaltung und zum Teilen von Packeten nutzt Node.js den Node Package Manager (NPM). Ein Packet sind in diesem Kontext ein oder mehrere Module, gekoppelt mit allen Dateien, welche diese benötigen. Packete werden auf *npmjs.com* gehostet. Die Liste der von einem Projekt verwendeten Module wird in der Datei *package.json* gespeichert. Ein Packet kann via NPM über den Befehl

```
$ npm install <Packetname>
```

installiert werden. Ist kein Packetname angegeben, so werden alle Packete, welche im Gleichen Ordner in der *package.json*-Datei eingetragen sind, installiert.

2.4 socket.io

Socket.io ist eine Bibliothek, welche bidirektionale Echtzeitkommunikation zwischen einem Client und einem Server ermöglicht. Dazu nutzt Socket.io intern WebSockets[soc]. Ein WebSocket ermöglicht Kommunikation zwischen einem Client und einem Server. Der Datenaustausch findet dabei über das Transmission Control Protocol (TCP) statt [FM11]. Das WebSocket Application Programming Interface (API) wird von allen aktuellen Browsern unterstützt[3]. Socket.io läuft auf einem Node.js Server[soc].

Die Kommunikation zwischen Client und Server wird bei Socket.io über Events geregelt. Client und Server können Events – definiert durch einen String – mit angehängten Daten

[1] Weitere Informationen: https://nodejs.org/en/download/package-manager/
[2] Node.js Downloads: https://nodejs.org/en/download/
[3] vgl. https://caniuse.com/mdn-api_websocket, Stand: 08.04.2021

emittieren. Basierend auf dem Event-String wird dann auf der Empfängerseite eine Rückruffunktion aufgerufen, vorrausgesetzt diese ist definiert. Die Daten werden der Rückruffunktion als Parameter übergeben. Socket.io ermöglicht auf der Serverseite sowohl das Broadcasting an alle, beziehungsweise an ein Subset an Clients, als auch Unicasting an einen spezifischen Client.

Die Socket.io Bibliothek ist in eine Client-, und eine Serverseitige Bibliothek aufgeteilt. Die Clientseitige Bibliothek ermöglicht das Verbinden mit einem Node.js Server. Ein Client kann sowohl Events mit Daten emittieren, als auch Rückruffunktionen registrieren, mit welchen der Client Daten vom Server empfangen kann. Auf der Serverseite ist es möglich, bei Verbindungsaufbau Rückruffunktionen für einen neu verbundenen Client zu registrieren, welche bei dem Eingang von Daten je nach Event-Typ aufgerufen werden. Der Server kann ebenfalls Events an Clients emittieren.

2.5 CoTurn STUN / TURN Server

2.6 AWS

3 Design und Implementation der WebRTC–Infrastruktur

3.1 SVG Test

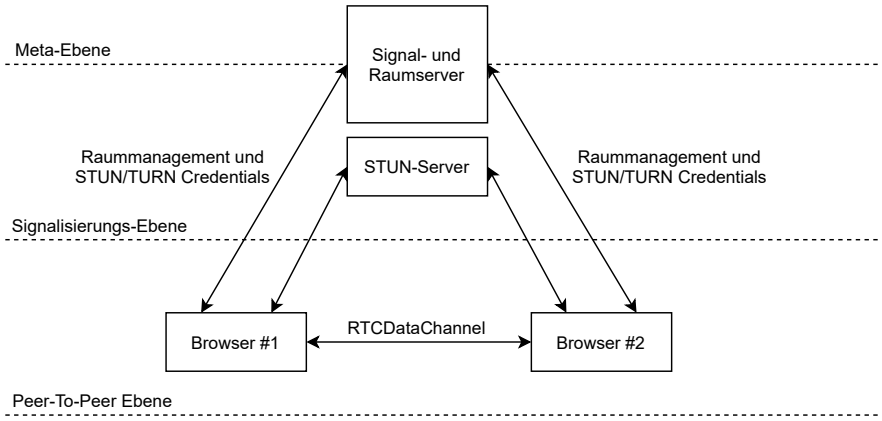

Abbildung 3.1: Ein toller SVG-Export Test.

3.2 Analyse

3.3 Implementation der Peer-To-Peer Funktionalität

3.4 Implementation des Signaling-Servers

3.5 Aufsetzen und Konfiguration eines STUN und TURN Servers

4 Mensch-Ärgere-Dich-Nicht

4.1 Spielablauf

4.2 Analyse

4.3 Implementation

4.3.1 Darstellung des Spielbretts

4.3.2 Implementation der Spiellogik

4.3.3 Synchronisation des Spielstands

4.3.4 Probleme der Implementation

Look-Ahead

Faire Zufallszahlen

5 Evaluation

6 Zusammenfassung und Ausblick

7 Literaturverzeichnis

[Bur12] Bura, Juriy: *Pro Android Web Game Apps*. Apress, 2012. – 477–478 S. – ISBN 978–1–4302–3820–1

[Con21] Contributors, MDN: RTCPeerConnection. Version: January 2021. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection. Mozilla Development Network, January 2021. – Forschungsbericht. – Abgerufen: 10.04.2021

[Dim16] Dimitrov, Tsvetomir: *Multi-homing in SCTP*. https://dzone.com/articles/multi-homing-in-sctp, April 2016. – Abgerufen 10.04.2021

[FM11] Fette, Ian ; Melnikov, Alexey: The WebSocket Protocol / Internet Engineering Task Force. Version: December 2011. https://tools.ietf.org/html/rfc6455. Internet Engineering Task Force, December 2011 (6455). – RFC. – ISSN 2070–1721. – Abgerufen: 08.04.2021

[LL13] Levent-Levi, Tsahi: *Jocly and WebRTC: An Interview With Michel Gutierrez*. https://bloggeek.me/jocly-webrtc-interview/, 2013. – Abgerufen: 01.04.2021

[nod] *Node.js Introduction*. https://www.w3schools.com/nodejs/nodejs_intro.asp, . – Abgerufen: 07.04.2021

[RV17] Rudd, Edward ; Vrignaud, Andre: *Introducing HumbleNet: a cross-platform networking library that works in the browser*. https://hacks.mozilla.org/2017/06/introducing-humblenet-a-cross-platform-networking-library-that-works-in-the-browser/, 2017. – Abgerufen: 01.04.2021

[soc] *Socket.io Documentation*. https://socket.io/docs/v4, . – Abgerufen: 08.04.2021

[Ste07] Stewart, Randall R.: Stream Control Transmission Protocol / Internet Engineering Task Force. Version: December 2007. https://tools.ietf.org/html/rfc4960. Internet Engineering Task Force, December 2007 (4960). – RFC. – Abgerufen: 11.04.2021

[web] *Web Real-Time Communications Working Group - Participants*. https://www.w3.org/groups/wg/webrtc/participants, . – Abgerufen: 10.04.2021