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: 15. 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.

// TODO: eng

Inhaltsverzeichnis

Abbildungsverzeichnis VI

Tabellenverzeichnis VII

Codeverzeichnis VIII

Abkürzungsverzeichnis IX

1 Einleitung 1
1.1 Zielsetzung . . . 2
1.2 Aufbau der Arbeit . . . 2

2 Grundlagen 3
2.1 Echtzeitanwendungen . . . 3
2.2 Netzwerkarchitekturen . . . 3
2.2.1 Client-Server und das Authoritative-Server-Modell . . . 3
2.2.2 Peer-To-Peer Netzwerke . . . 4
2.3 Network Address Translation . . . 4
2.4 Web Real-Time-Communication . . . 4
2.4.1 Aufbau von WebRTC . . . 5
2.4.2 JSEP: JavaScript Session Establishment Protocol . . . 7
2.4.3 ICE: Interactive Connectivity Establishment . . . 8
2.4.4 SCTP: Stream Control Transmission Protocol . . . 12
2.4.5 DTLS: Datagram Transport Layer Security . . . 13
2.5 Node.js . . . 13
2.5.1 NPM: Node Package Manager . . . 14
2.5.2 Verwendete Node-Packete . . . 14

3 WebRTC in Mehrspieler-Spielen 16

4 Design und Implementation 18
4.1 Implementation der WebRTC-Infrastruktur . . . 18
4.1.1 Implementation des Webservers . . . 19
4.1.2 Raum-Management . . . 21
4.1.3 Signalisierung . . . 23
4.2 Implementation der Peer-To-Peer Funktionalität . . . 24
4.3 Aufsetzen und Konfiguration eines STUN und TURN Servers . . . 25

5 Evaluation 26

6 Zusammenfassung und Ausblick 27

7 Literaturverzeichnis 28

Abbildungsverzeichnis

2.1 Client-Server Interaktion in einem Authoritative-Server-Modell. 4
2.2 Diagramm der WebRTC-Architektur. 5
2.3 JSEP-Verbindungsaufbau. 8
2.4 Diagramm der verschiedenen ICE-Kandidaten. 9

3.1 Einige Jocly-Spiele, mit WebRTC Videokommunikation. 16
3.2 Google CubeSlam, mit integriertem WebRTC Videostream. 17

Tabellenverzeichnis

2.1 Vergleich von TCP und UDP mit SCTP. 12

4.1 Vergleich von TCP und UDP mit SCTP. 18
4.2 Spielerfarben. 22

Codeverzeichnis

2.1 SDP-Datenstring eines Relais-ICE-Kandidaten 10

4.1 express Server – Server.js 19
4.2 Initialisierung des socket.io Servers – Server.js 20
4.3 Event zum Erstellen eines Raums – Server.js 21
4.4 Event zum Betreten eines Raums – Server.js 22
4.5 Event zum Weiterleiten eines Signals – Server.js 23
4.6 Javascript LstListing Test 24
4.7 Javascript LstListing Test 24
4.8 Javascript LstListing Test 25

Abkürzungsverzeichnis

W3C World Wide Web Consortium 1
WebRTC Web Real-Time Communication 1
P2P Peer-To-Peer 1
HTTP Hypertext Transfer Protocol
NPM Node Package Manager 14
IETF Internet Engineering Task Force
RTP Real-Time Transport Protocol 6
SRTP Secure Real-Time Transport Protocol 6
JSEP JavaScript Session Establishment Protocol 6
ICE Interactive Connectivity Establishment 6
NAT Network Address Translation
STUN Session Traversal Utilities for NAT 11
TURN Traversal Using Relays around NAT 11
SCTP Stream Control Transport Protocol 6
TLS Transport Layer Security 13
DTLS Datagram Transport Layer Security 13
API Application Programming Interface 14
TCP Transmission Control Protocol 14
UDP User Datagram Protocol 13
OSI Open Systems Interconnection 12
IP Internet Protocol 12
SDK Source Development Kit 7
SIP Session Initialization Protocol 7
SDP Session Description Protocol 7
URL Unique Resource Identifier
HTML Hypertext Markup Language

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 Peer-To-Peer (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 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.

1.2 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. Zudem werden kurz einige ausgewählte Spiele beschrieben, welche bereits WebRTC nutzen.

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.

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 Netzwerkarchitekturen

2.2.1 Client-Server und das Authoritative-Server-Modell

// Muss ich wirklich Client-Server erklären??

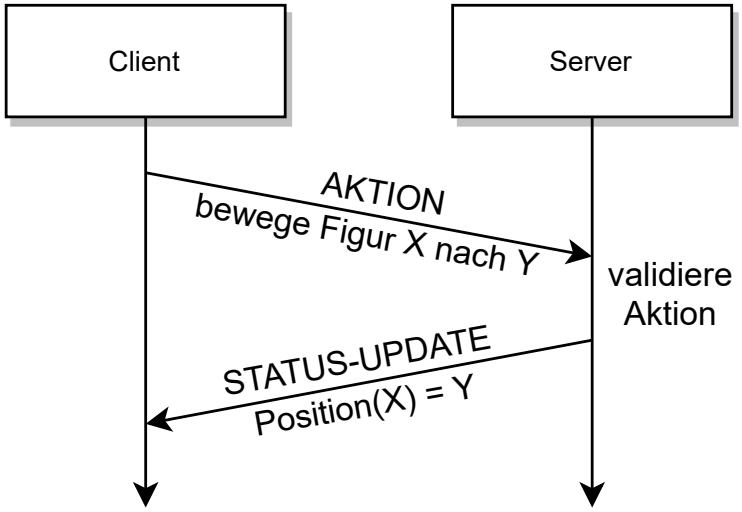

Abbildung 2.1: Client-Server Interaktion in einem Authoritative-Server-Modell.

2.2.2 Peer-To-Peer Netzwerke

2.3 Network Address Translation

2.4 Web Real-Time-Communication

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. WebRTC ist primär auf Audio- und Videokommunikation ausgelegt, ermöglicht aber auch das Senden von arbitraren Daten.

Der Standard wurde zuerst von Global IP Solutions (GIPS) entwickelt. In 2011 erwarb Google GIPS, machte die WebRTC-Komponenten Open-Source, und ermöglichte die Integration der Technologie in Web-Browser durch die Entwicklung einer JavaScript-API. Seitdem arbeitet das W3C an der Standardisierung der Technologie.

Der Standard wird nun 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].

Am 26. Januar 2021 veröffentlichte das W3C die WebRTC-Recommendation – WebRTC ist damit ein offizieller, vom W3C befürworteter Web-Standard, welcher für eine weitverbreitete Verwendung bereit ist.

2.4.1 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.

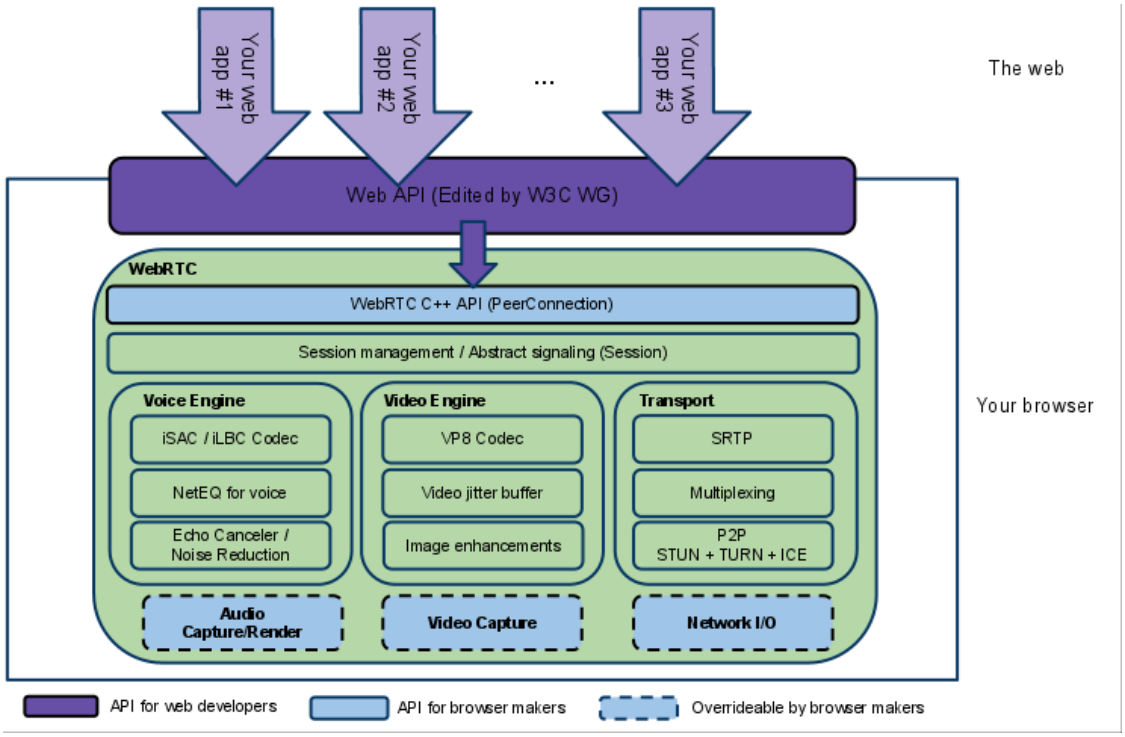

Abbildung 2.2: Diagramm der WebRTC-Architektur.
Quelle: `https://webrtc.github.io/webrtc-org/architecture/`

Wie dem Architekturdiagramm in Abbildung 2.2 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 in den Folgenden Unterpunkten näher erläutert werden.

2.4.2 JSEP: JavaScript Session Establishment Protocol

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 Konfiguration (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 dazu eine Reihe an asynchronen Funktionen zur Verfügung, welche das Erstellen und Setzen der Konfigurationen 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 Konfiguration via $setLocalDescription(anfrage)$. Das SDP-Packet wird über einen nicht vorgegebenen Signalkanal zu Bob gesendet. Dieser setzt daraufhin die externe Konfiguration seiner Verbindung via $setRemoteDescription(anfrage)$, und ruft daraufhin die Funktion $createAnswer(anfrage)$ auf, welche eine Antwort ($antwort$) generiert. Bob setzt seine lokale Konfiguration via $setLocalDescription(antwort)$, und sendet die Antwort

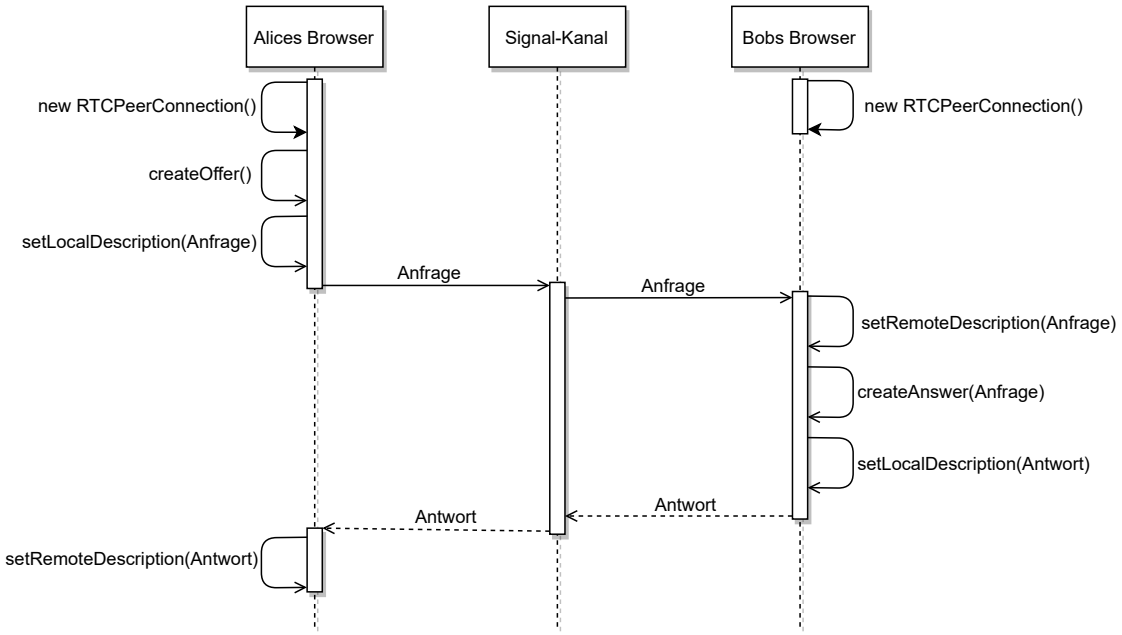

Abbildung 2.3: JSEP-Verbindungsaufbau.

zurück zu Alice. Alice setzt ihre externe Konfiguration via *setRemoteDescription*(*antwort*). Damit ist der anfängliche Austausch von Sitzungsparametern abgeschlossen [Bis14]. JSEP regelt dabei nur den Austausch von Konfigurationen zwischen zwei Peers, Informationen über die Verbindung werden über das ICE-Framework ausgetauscht. Der vereinfachte Ablauf des Verbindungsaufbaus ist Abbildung 2.3 zu entnehmen.

2.4.3 ICE: Interactive Connectivity Establishment

Das Interactive Connectivity Establishment (ICE)-Framework erlaubt es Browsern (Peers), Verbindungen untereinander aufzubauen. Es wird benötigt, da dies aufgrund der Tatsache, dass sich Peers in der Regel in einem lokalen Subnetz hinter einem NAT (Network Address Translation) befinden, nicht ohne weiteres möglich ist. Das ICE-Framework bietet in Kombination mit den Protokollen STUN und TURN Möglichkeiten, direkte Verbindungen zweier Peers durch NAT herzustellen, beziehungsweise Datenverkehr über ein Relais umzuleiten.

Eine WebRTC Verbindung (RTCPeerConnection) besitzt dabei immer einen sogenannten ICE-Agenten, sowohl auf der lokalen, als auch auf der externen Seite. Dabei agiert einer der Agenten einer Verbindung als der kontrollierende, der Andere als der kontrollierte Agent. Der kontrollierende Agent hat dabei die Aufgabe, das ICE-Kandidatenpaar, welches für die Verbindung genutzt werden soll, auszuwählen. Ein ICE-Kandidat beinhaltet

Informationen, im SDP-Format, über Transportadressen – Adress-Port-Tupel –, über welche ein Peer erreicht werden kann. Ein ICE-Kandidatenpaar ist ein Paar von zwei ICE-Kandidaten, welche zum gegenseitigen Verbindungsaufbau zweier Peers verwendet werden können.

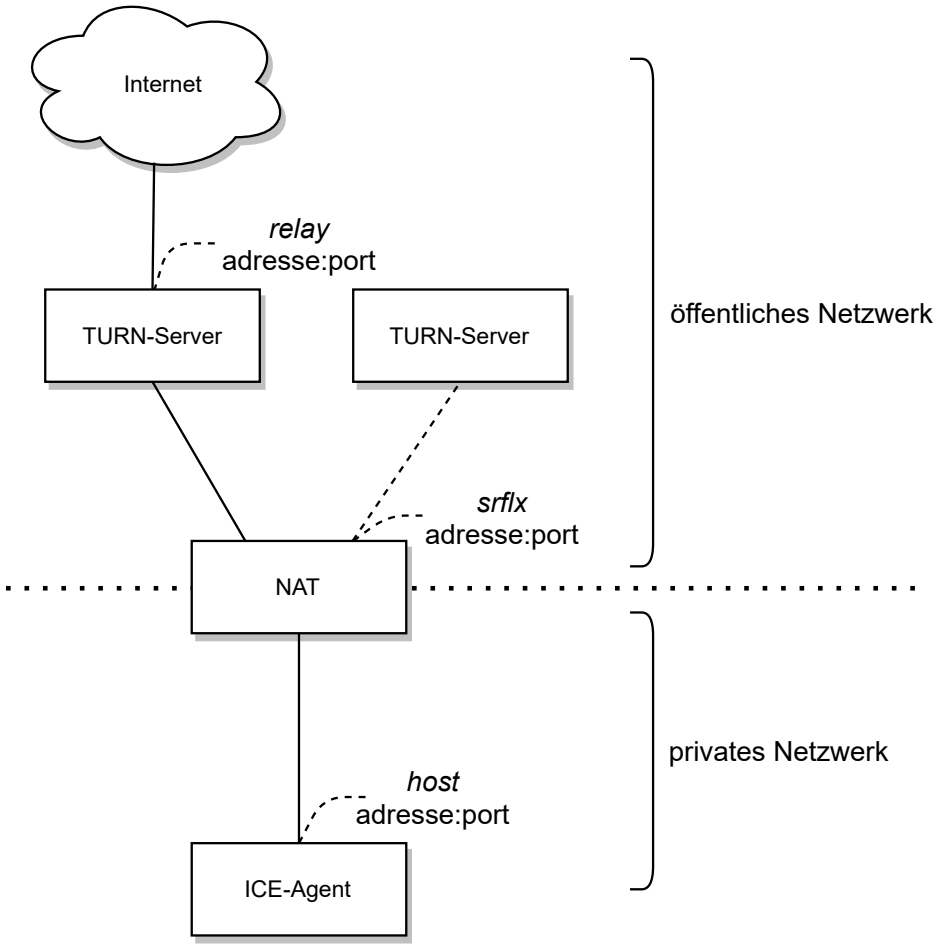

Abbildung 2.4: Diagramm der verschiedenen ICE-Kandidaten.
Quelle: Angepasst nach `https://tools.ietf.org/html/rfc5245`

Ein Gerät hinter einem NAT besitzt keine eigene, öffentliche IP-Adresse. Ein Peer muss jedoch seine eigene, öffentliche IP-Adresse kennen, um dem Verbindungspartner mitteilen zu können, an welche Adresse dieser die Daten schicken soll. In der Regel handelt es sich bei ICE-Kandidaten um UDP-Transportadressen. Es existieren zwar auch TCP-Kandidaten, diese werden allerdings nicht von allen Browsern unterstützt und in der Regel nicht verwendet. Wie aus Abbildung 2.4 zu entnehmen, werden dabei primär drei Arten an UDP-ICE-Kandidaten verwendet:

- Ein **Host**-Kandidat (*host*) ist der tatsächliche Adress-Port-Tupel eines Peers.

Host-Kandidaten können nur dann verwendet werden, wenn der Peer eine öffentliche IP-Adresse besitzt, oder sich sowohl der lokale, als auch externe Peer im gleichen Subnetz, oder auf dem gleichen Gerät befinden.

- Ein **Server-Reflexiver**-Kandidat (*srflx*) repräsentiert die öffentliche IP-Adresse eines Peers, also die öffentliche IP-Adresse des NATs, hinter welchem sich der Peer befindet. Befindet sich der Peer nicht hinter einem NAT, so ist die Adresse gleich der Host-Adresse, und der Kandidat wird verworfen.

- Ein **Relais**-Kandidat (*relay*) ist ein Adress-Port-Tupel, welcher dem Peer von einem TURN-Server zugeordnet wurde. Diese Adresse ist dabei die Adresse, welche der TURN-Server nutzt, um eingehende Daten an den Peer, und ausgehende Daten von dem Peer weiterzuleiten.

Trickle-ICE

Die STUN und TURN Server lassen sich beim Erstellen der RTCPeerConnection konfigurieren. Das Sammeln der ICE-Kandidaten beginnt mit dem Setzen der lokalen Konfiguration (*localDescription*) einer RTCPeerconnection. Die gefundenen ICE-Kandidaten müssen über den Signalisierungs-Kanal an den externen Peer gesendet werden, damit dieser die Kandidaten zum Verbindungsaufbau nutzen kann. WebRTC prüft die Verbindung zu allen möglichen ICE-Kandidaten eines Peers asynchron in Parallel, sobald die lokale Beschreibung einer Verbindung gesetzt ist. Dieser Prozess setzt eine Kommunikation mit STUN- und TURN-Servern vorraus, und kann daher, je nach Latenz und Antwortzeit der Server, einige Zeit beanspruchen.

Aus diesem Grund ermöglicht es WebRTC, ICE-Kandidaten zu „tricklen“, also nach und nach, bei Erhalt einer Antwort eines Servers, an den externen Peer zu schicken. Dies optimiert den Vorgang des Austauschs von ICE-Kandidaten, da nun nicht mehr darauf gewartet werden muss, dass alle Kandidaten ermittelt wurden, bevor der JSEP-Anfrage-Antwort-Prozess stattfinden kann.

```
1 candidate:1411127089 1 udp 33562367 20.56.95.156 12926 typ relay raddr 0.0.0.0 rport 0
      generation 0 ufrag uVtu network-cost 999
```

Quellcode 2.1: SDP-Datenstring eines Relais-ICE-Kandidaten

Jede RTCPeerConnection besitzt daher die Rückruffunktion „*onicecandidate*“, welche bei Erhalt eines neuen ICE-Kandidaten aufgerufen wird. Die Parameter dieser Rückruffunktion enthalten die SDP-Daten, welche den Kandidaten beschreiben. Die

Struktur dieser Daten ist Beispielhaft Abbildung 4.8 zu entnehmen. Dabei handelt es sich um einen UDP-Relais-Kandidaten, zu erkennen am „typ relay“, hervorgehoben in Rot. Die Adresse und der Port sind in Blau hervorgehoben. Nachdem die Daten an den externen Peer gesendet wurden, muss der Kandidat auf der Empfängerseite über die *addIceCandidate*-Funktion der RTCPeerConnection hinzugefügt werden.

STUN: Session Traversal Utilities for NAT

Das Session Traversal Utilities for NAT (STUN)-Protokoll wird verwendet, um die öffentliche IP-Adresse eines Peers zu ermitteln. Auf Anfrage an einen STUN-Server erhält ein Peer seine Server-Reflexive Transportadresse zurück, welche von externen Peers verwendet werden kann, um Daten an den lokalen Peer zu schicken. Durch die Anfrage an den STUN-Server wird dabei die benötigte Port- und Adresszuordnung in die NAT-Übersetzungstabelle des Routers geschrieben. Eine Server-Reflexive Adresse kann auch von einem TURN-Server abgefragt werden, vorrausgesetzt dieser unterstützt zusätzlich das STUN-Protokoll.

TURN: Traversal Using Relays around NAT

Im Gegensatz zu Client-Server-Verbindungen, welche nur vom Client eröffnet werden können, kann eine Peer-To-Peer Verbindung zudem sowohl vom lokalen Peer, als auch von eimem Peer außerhalb des lokalen Subnetzes eröffnet werden. Hier besteht das Problem, dass nicht jede Art von NAT das Eröffnen einer Verbindung von Außerhalb erlaubt [HS01]. Genauer, handelt es sich bei dem NAT um ein sogenanntes „Symmetrisches NAT“, so sind nur Client-Server Verbindungen, welche der im NAT befindliche Peer eröffnet, möglich.

An diesem Punkt setzt das Traversal Using Relays around NAT (TURN)-Protokoll an. Ein TURN-Server ermöglicht es, Daten, welche von einem externen Peer an den TURN-Server geschickt wurden, an einen Peer weiterzuleiten. Zudem kann der Peer Daten an den Turn-Server schicken, welche wiederrum an externe Peers weitergeleitet werden. Ein TURN-Server agiert somit als ein zwischen den Peers liegender Relais-Server, für Fälle, in denen eine direkte Verbindung zwischen Peers aufgrund von NAT-Einschränkungen nicht möglich ist.

2.4.4 SCTP: Stream Control Transmission Protocol

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 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].

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].

	TCP	UDP	SCTP
Nachrichtenordnung	Geordnet	Ungeordnet	Konfigurierbar
Zuverlässigkeit	Zuverlässig	Unzuverlässig	Konfigurierbar
Flusskontrolle	Ja	Nein	Ja
Überlastkontrolle	Ja	Nein	Ja
Multihoming	Nein	Nein	Ja
Mehrere Datenströme	Nein	Nein	Ja

Tabelle 2.1: Vergleich von TCP und UDP mit SCTP.

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

2.4.5 DTLS: Datagram Transport Layer Security

Das Datagram Transport Layer Security (DTLS)-Protokoll ist ein auf Transport Layer Security (TLS) aufbauendes Protokoll zur Verschlüsselung von Daten in Datagram-basierenden Verbindungen. Im Gegensatz zu TLS, welches für die Nutzung mit TCP konzipiert wurde, kann DTLS also auch über UDP übertragen werden.

Das WebRTC-Framework nutzt DTLS zur Verschlüsselung von Datenkanälen (RTCDataChannel). Da das von Datenkanälen genutzte SCTP-Protokoll über keine eigene Verschlüsselung verfügt, und Verschlüsselung aller Daten eine zentrale Anforderung von WebRTC darstellt, werden Daten über einen DTLS-Tunnel zwischen den Verbindungspartnern ausgetauscht. Dieser Tunnel liegt auf dem User Datagram Protocol (UDP). Die SCTP-Verbindung eines Datenkanals läuft also nicht direkt auf dem Internet Protocol (IP), sonder über einen DTLS-Tunnel, welcher wiederrum auf UDP liegt.

2.5 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.

2.5.1 NPM: Node Package Manager

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.

2.5.2 Verwendete Node-Packete

// TODO: 2 Zeilen Text hier, dass das nicht so scheiße aussieht...

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 emittieren. Basierend auf dem Event-String wird dann auf der Empfängerseite eine

[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

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.

3 WebRTC in Mehrspieler-Spielen

WebRTC wird bereits in einigen Browserbasierten 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 (vgl. Abbildung 3.1) [LL13]. Ähnlich verhält es sich bei einigen weiteren Frameworks, wie zum Beispiel *Tabloro*[1], einem Browserbasierten „Tabletop"-Spielsimulator. Diese Spiele und Frameworks nutzen eine Client-Server-Architektur für den regulären (nicht Video und Audio) Datenaustausch.

Abbildung 3.1: Einige Jocly-Spiele, mit WebRTC Videokommunikation.
Quelle: `https://bloggeek.me/jocly-webrtc-interview/`

Weiterhin existieren eine Reihe an prototypischen Spielen und Frameworks, welche

[1] vgl. `https://github.com/fyyyyy/tabloro`

WebRTC zum Austausch von Daten nutzen. Bei diesen handelt es sich überwiegend um Echtzeitspiele wie das 2013 von Google entwickelte *CubeSlam*. Abbildung 3.2 zeigt dabei eine Bildschirmaufname eines CubeSlam-Spiels. CubeSlam nutzt WebRTC Medienstreams, um Videodaten zu übertragen, und Datenkanäle, um die Spieldaten zwischen den Spielern zu synchronisieren. Auch der 2012 von Mozilla entwickelte First-Person-Shooter *Bananabread*[2] nutzt WebRTC zum Austausch von Spieldaten.

Abbildung 3.2: Google CubeSlam, mit integriertem WebRTC Videostream.
Quelle: `https://experiments.withgoogle.com/cube-slam`

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

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

4 Design und Implementation

Das Projekt gliedert sich primär in zwei Teile: Das Brettspiel „Mensch Ärgere Dich Nicht“, und die unterliegende WebRTC- und Netzwerkinfrastruktur. Die Projektstruktur selbst ist in Abbildung 4.1 beschrieben. Auf die Spiel- und Netzwerkspezifischen Script-Dateien wird in deren jeweiligen Sektionen weiter eingegangen.

Dateipfad	Beschreibung
/*	Grundverzeichnis des Servers
/Server.js	Diese Datei ist die ausführbare Script-Datei des Webservers. Hier wird der express-Webserver erstellt, sowie die WebSocket Verbindungen via socket.io, und die Spielräume verwaltet.
/utils.js	Hilfsfunktionen zum Generieren von TURN-Passdaten, Raum-IDs und Peer-IDs.
/config.json	Server-Konfiguration.
/package.json	NPM-Konfiguration.
/public/*	Öffentliche Dateien, welche über den Webserver abgerufen werden können.
/public/index.html	HTML-Datei der Seite, auf welcher ein Spieler einen Raum erstellen oder beitreten kann.
/public/game.html	HTML-Datei der Spiel-Seite.
/public/resources/*	CSS, Bild- und Scriptressourcen.
/public/resources/script/index.js	JS-Datei der Index-HTML-Seite.
/public/resources/script/game.js	JS-Datei der Spiel-HTML-Seite.
/public/resources/script/network/*	Netzwerkspezifische Script-Dateien
/public/resources/script/game/*	Spielspezifische Script-Dateien

Tabelle 4.1: Vergleich von TCP und UDP mit SCTP.

4.1 Implementation der WebRTC-Infrastruktur

Die Netzwerkinfrastruktur ist in drei Teile geteilt: Der Webserver, welcher die Seiten bereitstellt und als Signalisierungskanal dient, die Clientseitige WebRTC-Implementation

und die STUN- und TURN-Server. Zusammen bilden diese drei Komponenten das „WebRTC-Dreieck“.

4.1.1 Implementation des Webservers

Um das Verbinden von Peers via WebRTC zu ermöglichen, muss vorerst ein Signalisierungskanal existieren, welcher Nachrichten zwischen Peers weiterleiten kann. Die Art des Signal-Kanals ist dabei nicht vom WebRTC-Standard vorgegeben. Für die Implementierung wurde daher ein Signal-Server geschaffen, welcher WebSockets zur Datenübertragung nutzt. Der Server dient zusätzlich als Webserver, welcher das Brettspiel als Webanwendung bereitstellt.

Erstellen des Node-Servers

Um einen Node-Server zu erstellen, muss der Node-Package-Manager über den Kommandozeilenbefehl

```
$ npm init
```

initialisiert werden. Dieser Befehl erstellt die „package.json“-Datei im momentanen Arbeitsordner. Zudem müssen die benötigten Packete via

```
$ npm install socket.io
$ npm install express
```

heruntergeladen werden. Daraufhin können diese Packete über die *require*-Funktion in ein Script eingebunden werden.

express Web-Server

Zur Erstellung des Webservers wird das „express“-Packet verwendet. Der Quellcode befindet sich in der „Server.js“-Datei, und ist in Abbildung 4.1 abgebildet.

```
1 const express = require('express');
2 const config = require('./config.json');
3
4 const app = express();
5 const server = require('http').Server(app);
6
7 app.use(express.static('public'));
8
```

```
9  app.get('/', (req, res) => {
10   res.status(200);
11   res.sendFile(`${__dirname}${config.server.indexPage}`);
12 });
13
14 app.get('/game/*/', (req, res) => {
15   res.status(200);
16   res.sendFile(`${__dirname}${config.server.gamePage}`);
17 });
18
19 server.listen(config.server.listeningPort);
```

Quellcode 4.1: express Server – Server.js

Zuerst muss eine express-Anwendung über die *express*()-Funktion erstellt werden. Da für die Nutzung von socket.io allerdings ein HTTP-Serverobjekt nötig ist, wird dieses in Zeile 5 erstellt. Dabei wird die express-Anwendung als Parameter bei der Servererstellung mitgegeben. Die express-Anwendung nutzt nun diesen HTTP-Server für die Webserver-Funktionalität.

Da die HTML-Dateien die notwendingen Scripts aus anderen Dateien importieren, müssen diese via URL vom Webserver abrufbar sein. Die Funktion *app.use* erlaubt es, Dateien via URL öffentlich bereitzustellen. In Zeile 7 wird daher der Inhalt des „public“-Ordners statisch zur Verfügung gestellt.

Des Weiteren werden in Zeile 9 und 14 die Pfade definiert, auf welchen die HTML-Dateien abrufbar sind. Ruft ein Nutzer die Basis-URL des Servers auf, so sendet der Server die „index.html“-Datei. Ruft ein Nutzer den Pfad “<Basis-Server-URL>/game/<Raum-ID>/“ auf, so wird dieser auf die Raum-Seite („room.html“) verwiesen. Dies bewirkt, dass ein Nutzer direkt über eine URL einem Spiel beitreten kann, und nicht über die Index-Seite beitreten muss. Die Dateipfade sind in der Server-Konfigurationsdatei definiert.

Zuletzt muss in Zeile 19 noch der Port definiert werden, über welchen der Server von außen ansprechbar sein soll. Dieser ist ebenfalls via der Konfigurationsdatei definierbar. Wichtig ist, dass dieser Port des Servers, auf welchem der Webserver läuft, weitergeleitet ist [siehe: Aufsetzen der Server (oder so, // todo!!!)].

socket.io

```
1 [...]
2 const server = require('http').Server(app);
```

```
3 const io = require('socket.io')(server);
4 [...]
5 io.sockets.on('connection', (socket) => {
6   [...]
7 }
```

Quellcode 4.2: Initialisierung des socket.io Servers – Server.js

4.1.2 Raum-Management

Das Spiel „Mensch Ärgere Dich Nicht“ kann maximal von vier Spielern gespielt werden. Damit mehr als ein Spiel gleichzeitig gespielt werden kann, müssen Spieler in „Räume“ unterteilt werden. Ein Raum besitzt dabei eine Liste an bis zu vier Sets an Spielerdaten, eine Raum-ID und eine Host-ID. Bei der Host-ID handelt es sich um die Socket-ID des Spielers, welcher den Raum zuerst betritt. Dieser hat als einziger Spieler die Befugnis, weitere Spieler aus dem Raum zu entfernen. Spielerdaten enthalten jeweils die Peer-ID eines Spielers, einen Namen und die Spielfarbe des Spielers.

```
1  const utils = require('./utils.js');
2  [...]
3
4  const rooms = {}
5  const playerSockets = {}
6
7  io.sockets.on('connection', (socket) => {
8    socket.on('game-room-create', () => {
9      const id = utils.generateRoomID(rooms);
10     rooms[id] = {
11       id : id, // easier to identify room by player
12       players : [],
13       started : false,
14       host : null
15     };
16     [...]
17     socket.emit('game-room-created', id);
18   });
19   [...]
20 }
```

Quellcode 4.3: Event zum Erstellen eines Raums – Server.js

Erstellt ein Nutzer einen Raum, so sendet dieser das *game-room-create*-Event zum Server (vgl. Abbildung 4.5). Der Server generiert eine vierstellige Zeichenkette, welche als

Raum-ID dient. Mit dieser wird in Zeile 10 ein Raum-Objekt erzeugt, welches die zuvor beschriebenen Daten enthält. Zudem speichert das Raum-Objekt, ob das Spiel in diesem Raum bereits gestartet ist. Ist der Raum erstellt, so wird ein Event an den erstellenden Client zurückgeschickt, um diesem mitzuteilen, dass der Raum nun beitretbar ist.

Das Betreten eines Raums ist über das *game-room-join*-Event geregelt. Versucht ein Nutzer einem Raum beizutreten, so sendet dieser, das Event an den Server. Falls der Raum noch keinen Host-Spieler besitzt, so wird der erste Beitretende Spieler in Zeile 17 zum Host ernannt. Bei erfolgreichem Beitritt erhält der Nutzer, wie in Abbildung 4.4 in Zeile 20 zu sehen, das *game-room-joined*-Event zurück, welches sämtliche, zum Verbindungsaufbau mit den anderen Spielern des Raums, benötigten Daten enthält. Dazu gehören das Array der weiteren Spieler im Raum, die eigene Peer-ID und die Zugangsdaten zu den STUN- und TURN-Servern. Zusätzlich wird dem Nutzer mitgeteilt, ob dieser der Host des Raums ist. Zudem werden in Zeile 27–29 alle weiteren Spieler des Raums benachrichtigt, dass ein weiterer Spieler beigetreten ist.

Die Spielregeln von „Mensch Ärgere Dich Nicht“ besagen, dass beim Spielen mit zwei Spielern, die Farben Gelb und Rot gewählt werden sollen, damit die Spieler gegenüberstehende Startfelder haben. Daher werden die Spieler beim Betreten eines Raums nicht nach aufsteigender Reihenfolge in das Spieler-Array eingefügt, sondern nach der in Zeile 1 definierten Reihenfolge [0, 2, 1, 3]. Ab Zeile 8 wird dieses Array durchlaufen. Ist ein Array-Index nicht definiert, so wird der neu beigetretene Spieler an diese Stelle des Spieler-Arrays gesetzt.

Spieler-Index	Farbe im Spiel
0	Gelb
1	Grün
2	Rot
3	Schwarz

Tabelle 4.2: Spielerfarben.

Dazu wird in Zeile 10 zuerst eine Peer-ID generiert, welche zu Signalisierungszwecken genutzt wird. Die „Farbe“ des Spielers ist dabei der Index des Spielers im Spieler-Array (vgl. Tabelle 4.2).

```
1 const PLAYER_SLOT_PRIORITY = [0, 2, 1, 3];
2 [...]
3 io.sockets.on('connection', (socket) => {
4   [...]
5   socket.on('game-room-join', (roomID) => {
6     const room = rooms[roomID];
```

```
7     [...]
8     for (let i = 0; i < 4; i++) {
9       if (!room.players[PLAYER_SLOT_PRIORITY[i]]) {
10        const peerID = utils.uuid4();
11        const color = PLAYER_SLOT_PRIORITY[i];
12
13        playerSockets[peerID] = socket.id;
14        room.players[color] = {peerID : peerID, color : color};
15
16        if (!room.host) {
17          room.host = socket.id;
18        }
19
20        socket.emit('game-room-joined',
21          room.players,
22          peerID,
23          utils.generateTURNCredentials(socket.id),
24          room.host === socket.id
25        );
26
27        room.players.forEach((player) => {
28          socket.to(playerSockets[player.peerID]).emit('game-room-client-joining',
      peerID, color)
29        });
30        break;
31      }
32    }
33  });
34  [...]
35 }
```

Quellcode 4.4: Event zum Betreten eines Raums – Server.js

4.1.3 Signalisierung

Blabla blalala bla Blabla blalala bla. Ich sollte mir echt mal das lipsum package runterladen...

```
1 [...]
2 io.sockets.on('connection', (socket) => {
3   [...]
4   socket.on('signal', (roomID, target, e) => {
```

```
    const room = rooms[roomID];

    if (room) {
      const target = room.players.find((players) => players.peerID === e.target)
      if (target) {
        socket.to(playerSockets[target.peerID]).emit('signal', e);
      }
    }
  });
  [...]
}
```

Quellcode 4.5: Event zum Weiterleiten eines Signals – Server.js

4.2 Implementation der Peer-To-Peer Funktionalität

```
Peer.prototype.connect = function(remotePeerId) {
  const connection = this._createConnection(remotePeerId, true);
  this.connections[remotePeerId] = connection;

  connection.createOffer().then((offer) => {
    connection.setLocalDescription(offer).then(() => this.sendSignal(
      this._createSignal('offer', connection.localDescription, remotePeerId)
    ));
  });
}
```

Quellcode 4.6: Javascript LstListing Test

```
Peer.prototype._createConnection = function(remotePeerId) {
  const connection = new RTCPeerConnection(this.rtcConfiguration);
  connection.dc = {}; // list of data channels belonging to this connection

  this.channels.forEach((options, i) => {
    const channel = connection.createDataChannel(options.label, {
      negotiated : true,
      id : i,
      ordered : options.ordered || true,
      maxRetransmits : options.maxRetransmits || null,
      maxPacketLifeTime : options.maxPacketLifeTime || null,
    });
    channel.onmessage = (e) => this._receiveMessage(e);

    connection.dc[channel.label] = channel;
  });
```

```
17
18   connection.onicecandidate = (e) => {
19     this.sendSignal(this._createSignal('ice-candidate', e.candidate, remotePeerId));
20   }
21
22   return connection;
23 }
```

Quellcode 4.7: Javascript LstListing Test

```
1  Peer.prototype.onsignal = function(e) {
2    switch(e.type) {
3      case 'offer':
4        const connection = this._createConnection(e.src);
5        this.connections[e.src] = connection;
6
7        connection.setRemoteDescription(e.data).then(() => {
8          connection.createAnswer().then((answer) => {
9            connection.setLocalDescription(answer).then(() => {
10             this.sendSignal(this._createSignal('answer', connection.localDescription,
       e.src));
11           })
12         });
13       });
14       break;
15     case 'answer':
16       this.connections[e.src].setRemoteDescription(e.data).then();
17       break;
18     case 'ice-candidate':
19       this.connections[e.src].addIceCandidate(e.data).then();
20       break;
21   }
22 }
```

Quellcode 4.8: Javascript LstListing Test

4.3 Aufsetzen und Konfiguration eines STUN und TURN Servers

5 Evaluation

6 Zusammenfassung und Ausblick

7 Literaturverzeichnis

[Bis14] Bisht, Altanai: *WebRTC Integrator's Guide*. Packt Publishing, 2014. – ISBN 978–1–78398–126–7

[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

[HS01] Holdrege, Matt ; Srisuresh, Pyda: Protocol Complications with the IP Network Address Translator / The Internet Society. Version: January 2001. https://tools.ietf.org/html/rfc3027. Internet Engineering Task Force, January 2001 (3027). – RFC. – Abgerufen: 11.04.2021

[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