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: 28. 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

Tabellenverzeichnis

Codeverzeichnis

Abkürzungsverzeichnis

W3C World Wide Web Consortium . . . 1
WebRTC Web Real-Time Communication . . . 1
P2P Peer-To-Peer . . . 1
HTTP Hypertext Transfer Protocol . . . 4
NPM Node Package Manager . . . 15
IETF Internet Engineering Task Force
RTP Real-Time Transport Protocol . . . 7
SRTP Secure Real-Time Transport Protocol . . . 7
JSEP JavaScript Session Establishment Protocol . . . 6
ICE Interactive Connectivity Establishment . . . 7
NAT Network Address Translation
STUN Session Traversal Utilities for NAT . . . 11
TURN Traversal Using Relays around NAT . . . 12
SCTP Stream Control Transport Protocol . . . 7
TLS Transport Layer Security . . . 14
DTLS Datagram Transport Layer Security . . . 14
API Application Programming Interface . . . 4
TCP Transmission Control Protocol . . . 16
UDP User Datagram Protocol . . . 14
OSI Open Systems Interconnection . . . 12
IP Internet Protocol . . . 12
SDK Source Development Kit . . . 7
SIP Session Initialization Protocol . . . 7
SDP Session Description Protocol . . . 8
URI Unique Resource Identifier . . . 4
URL Unique Resource Locator . . . 4
HTML Hypertext Markup Language
VM Virtuelle Maschine . . . 25
RTMFP Real-Time Media Flow Protocol . . . 17

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 Web-Browserbasierte Echtzeit-Anwendungen wie Audio- und Videokommunikationsanwendungen verwendet.

Die Beliebtheit von interaktiven Mehrspieler-Spielen, welche durch das einfache Abrufen einer Webseite spielbar sind, steigt von Jahr zu Jahr, und wird zudem durch die seit Beginn 2020 anhaltende Corona-Pandemie weiter gefördert [o.A20].

In der Regel basieren Browserbasierte Mehrspieler-Spiele auf einer Client-Server-Architektur, wobei der Server die Rolle des bestimmenden Spielleiters übernimmt. Ein weiterer, gut definierter aber für Browserbasierte Spiele selten verwendeter 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. 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 [Tho13], 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

// TODO: logischerweise als letztes...

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.

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-Modelle

Eine Client-Server Netzwerktopographie setzt sich aus primär zwei Arten an Knoten zusammen: Clients und Servern. Ein Server hat die Aufgabe, den Clients Daten und Services zur Verfügung zu stellen [Sil15]. In der Regel kann ein Server dabei eine Vielzahl an Clients versorgen.

Web-Architektur

Die Architektur von Webanwendungen basiert in der Regel auf auf einem Client-Server-Modell, wobei der Browser des Clients Hypertext Transfer Protocol (HTTP)-Anfragen an den Server sendet, um Inhalte vom Server abzurufen. Diese Inhalte lassen sich anhand eines Unique Resource Identifier (URI), beziehungsweise einer Unique Resource Locator (URL) abrufen. Die Webanwendung kann zudem durch JavaScipt-Code mit eingebetteten Application Programming Interface (API)s der Browser interagieren [**?**].

Authoritative-Server

Bei der Entwicklung von Mehrspieler-Spielen wird oftmals das sogenannte „Authoritative-Server"-Modell verwendet. Dabei ist ein Zentraler Server für die gesamte Verwaltung des Spielstandes zuständig. Die Spieler – die Clients – halten lediglich den minimal benötigten Status, um das Spiel darzustellen. Jede Aktion des Spielers, welche den Spielstand beeinflusst – zum Beispiel das Bewegen einer Spielfigur oder das Ziehen einer Karte – muss über den Server geregelt und autorisiert werden [Gam] (vgl. Abbildung 2.1).

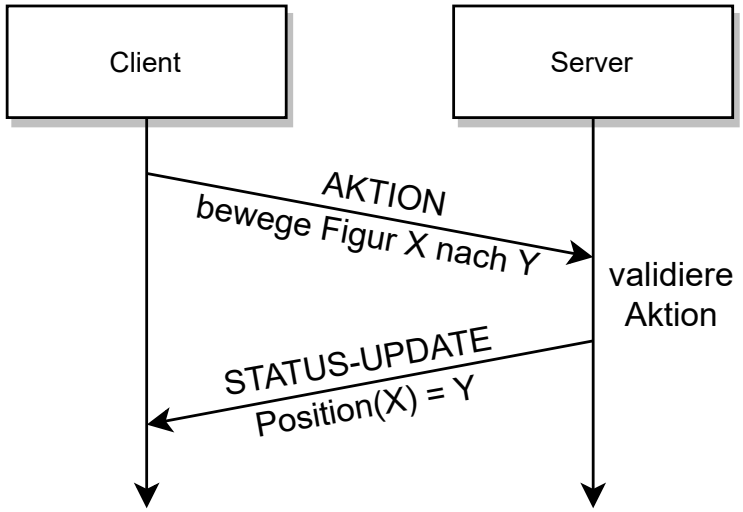

Abbildung 2.1: Beispielhafte Interaktion in einem Authoritative-Server-Modell.

Dieser Ansatz eignet sich besonders für Spiele, in welchen Spieler nicht durch unerlaubte Modifikation der Spieldateien Vorteile verschaffen dürfen. Da alle Aktionen über den Server geregelt werden, ist es einfach unerlaubte Änderungen am Spielstand zu verbieten [Gam].

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-Browsern 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 [o.Ac].

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.

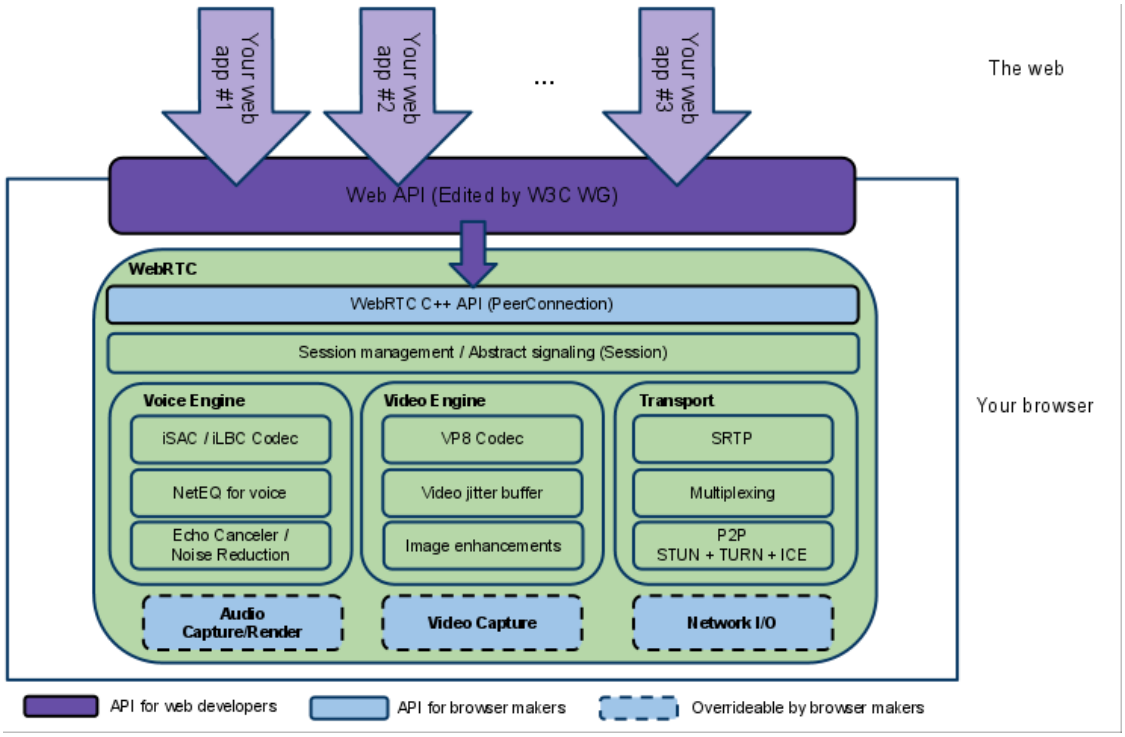

Abbildung 2.2: Diagramm der WebRTC-Architektur.
Quelle: `https://webrtc.github.io/webrtc-org/architecture/`

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) [o.A21]. Die Signalisierungsmechanismen folgen dabei dem JavaScript

Session Establishment Protocol (JSEP), die Verbindung selbst wird via dem Interactive Connectivity Establishment (ICE) Framework hergestellt [LR14].

- 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 [LR14, o.A21].
- 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 [LR14]. 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.

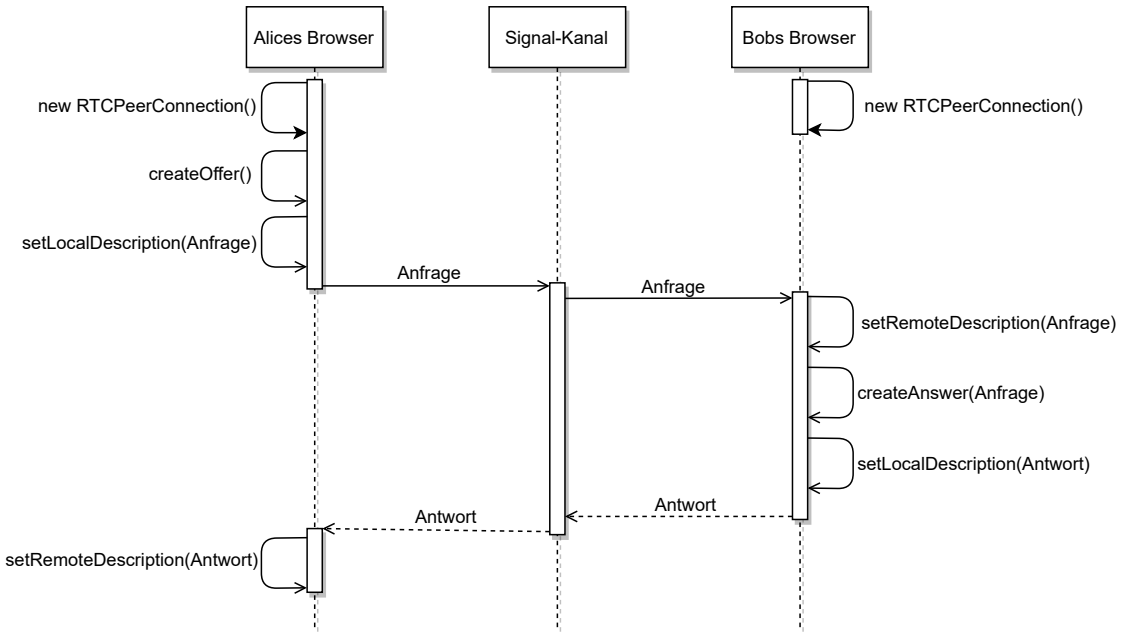

Abbildung 2.3: JSEP-Verbindungsaufbau.

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 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.

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 (*typ host*) ist der tatsächliche Adress-Port-Tupel eines Peers. Host-Kandidaten können nur dann verwendet werden, wenn der Peer eine öffentliche IP-

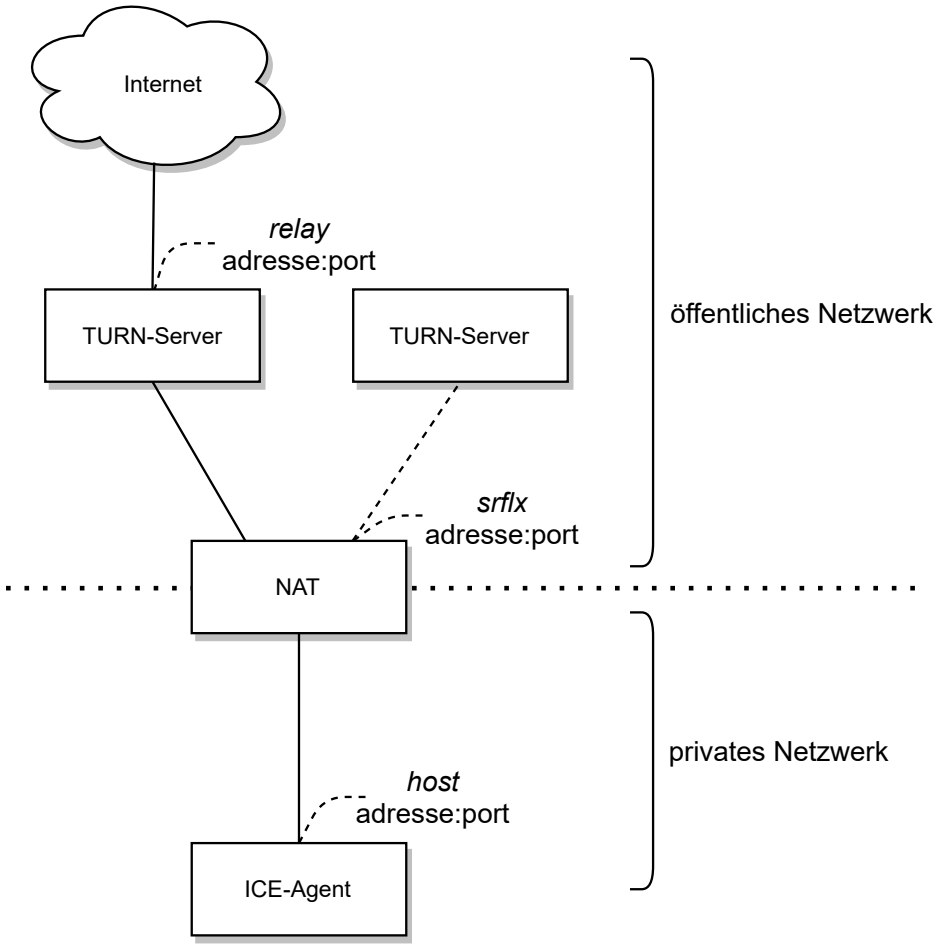

Abbildung 2.4: Diagramm der verschiedenen ICE-Kandidaten.
Quelle: Angepasst nach `https://tools.ietf.org/html/rfc5245`

Adresse besitzt, oder sich sowohl der lokale, als auch externe Peer im gleichen Subnetz, oder auf dem gleichen Gerät befinden.

- Ein **Server-Reflexiver**-Kandidat (*typ 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 (*typ 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. WebRTC prüft die Verbindung zu allen möglichen ICE-Kandidaten eines Peers asynchron in Parallel, sobald die lokale Beschreibung einer Verbindung gesetzt ist. Dieser Prozess – der sogenannte „ICE-Sammelprozess“ – 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 jeweils einzelne Kandidaten bei Erhalt einer Serverantwort an den externen Peer zu schicken. Dies optimiert den Vorgang des Austauschs von ICE-Kandidaten, da der JSEP-Anfrage-Antwort-Prozess nun parallel zum ICE-Sammelprozess stattfinden kann. Unterstützt ein älterer Browser kein „Trickle-ICE“, so müssen vor dem Abschicken der JSEP-Anfrage oder Antwort alle ICE-Kandidaten gesammelt, und den SDP-Daten der jeweiligen Nachricht beigefügt sein.

```
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 2.1 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. Der Kandidat wird automatisch zur lokalen Beschreibung des Peers, welcher den Kandidaten fand, hinzugefügt. Nachdem die Daten daraufhin an den externen Peer gesendet wurden, muss der Kandidat auf der Empfängerseite über die *addIceCandidate*-Funktion der externen Konfiguration 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 einem 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].

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.

// TODO: vllt. diese Abschnitte weglassen, nicht zwingend notwending...

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

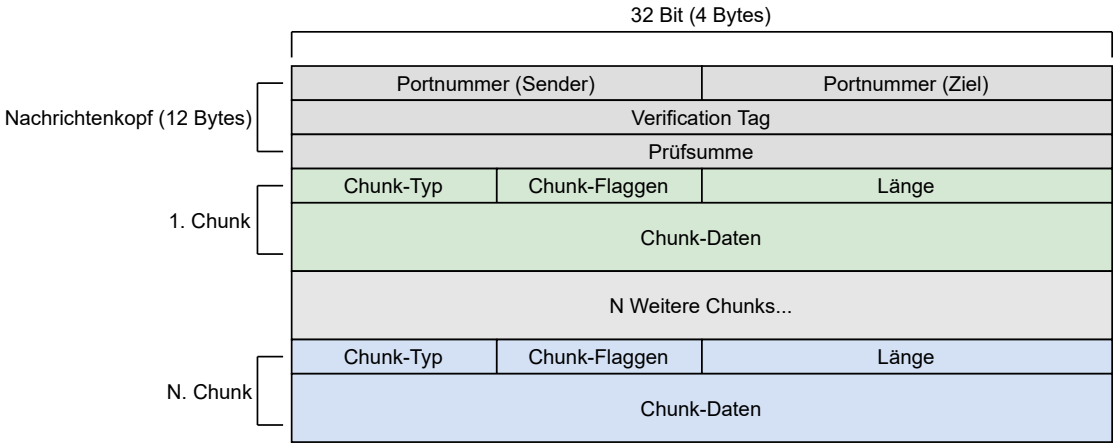

Abbildung 2.5: Aufbau eines SCTP-Packets.

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 Kontrolle der Verbindung 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 2.5 zu entnehmen. Die maximale größe eines „Chunks“ ist dabei, aufgrund des zwei-Byte langen Längenfelds auf 65,535 Kilobyte limitiert.

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.

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 [o.Aa]. Weiterhin bietet Node.js mit dem „crypto“-Modul kryptografische Funktionen, welche auf OpenSSL (Open Secure Socket Layer) basieren.

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.

2.5.2 Verwendete Node-Packete

Neben den standartmäßig in Node.js enthaltenen „http“- und „crypto“-Modulen werden zwei weitere Packete genutzt: die „socket.io“ Bibliothek und das „express“-Framework.

socket.io

„socket.io“ ist eine Bibliothek, welche bidirektionale Echtzeitkommunikation zwischen einem Client und einem Server ermöglicht. Dazu nutzt Socket.io intern WebSockets[o.Ab].

[1] Weitere Informationen: `https://nodejs.org/en/download/package-manager/`
[2] Node.js Downloads: `https://nodejs.org/en/download/`

Ein WebSocket ermöglicht Kommunikation zwischen einem Client und einem Server. Der Datenaustausch findet dabei über das Transmission Control Protocol (TCP) statt [FM11]. Das WebSocket API wird von allen aktuellen Browsern unterstützt[3]. Socket.io läuft auf einem Node.js Server[o.Ab].

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 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 [o.Ab].

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.

express

„express“ ist ein Node-Web-Framework, welches bei der Erstellung von Webanwendungen zum Einsatz kommt. Express agiert als Middleware zwischen einem HTTP-Server und einer Webanwendung, und ermöglicht es, basierend auf HTTP-Methoden und URLs – den sogenannten „Pfaden“ – verschiedene Aktionen auszuführen [**?**]. Zudem bietet express weitere Funktionalitäten, wie das Bereitstellen von statischen Website-Daten, oder die Integration von „Rendering-Engines“, welche die Daten einer Website je nach Pfad dynamisch anpassen [**?**].

[3] vgl. `https://caniuse.com/mdn-api_websocket`, Stand: 08.04.2021

3 WebRTC in Mehrspieler-Spielen

Vor dem Lebensende des Adobe Flash-Players im Dezember 2020 wurde dieser häufig zur Entwicklung von Browserbasierten Mehrspieler-Spielen verwendet. Diese Spiele wurden auf verschiedenen Plattformen wie zum Beispiel *Newgrounds.com* angeboten. Der Flash-Player verfügte dabei über Peer-To-Peer-Fähigkeiten via Adobes Real-Time Media Flow Protocol (RTMFP). Somit bot der Flash-Player als Plattform eine kostengünstige Alternative zu Client-Server-Netzwerkarchitekturen. Mit dem Wegfall des Flash-Players ist WebRTC nun die einzige Technologie, welche ohne zusätzliche Software oder Plugins zum direkten Vernetzen von Browsern genutzt werden kann.

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.

Weiterhin existieren eine Reihe an prototypischen Spielen und Frameworks, welche 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.

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.

[1] vgl. https://github.com/fyyyyy/tabloro
[2] vgl. https://hacks.mozilla.org/2013/03/webrtc-data-channels-for-great-multiplayer/

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

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

4 Konzept

Dieses Kapitel befasst sich mit dem Verwendeten Konzept der Implementierung, Dabei wird primär auf die Anforderungen der Netzwerkstruktur eingegangen.

4.1 Anforderungen

Es soll ein System geschaffen werden, welches das Spielen des Brettspiels „Mensch Ärgere Dich Nicht“ für mindestens vier Spieler ermöglicht. Die Anforderungen an das System gliedern sich in zwei Teile: Anforderungen an das Spiel an sich, und Anforderungen an die unterliegende Netzwerkstruktur.

4.1.1 Anforderungen an die Netzwerkstruktur

Die Spieler sollen sich an verschiedenen Geräten, sowie in verschiedenen Subnetzen befinden können. Der zum Spielablauf notwendige Datenaustausch soll über ein WebRTC-Peer-To-Peer Netzwerk zwischen den Spielern stattfinden. Dies setzt die Verwendung von STUN- und TURN-Servern vorraus. Es muss mindestens ein STUN- und ein TURN-Server existieren. Der Datenaustausch zwischen Peers muss dabei zuverlässig und geordnet sein.

Dritte sollen die STUN- und TURN-Server nur zu deren vorgesehenem Zweck – zum Spielen des Brettspiels – in Verbindung mit der Webanwendung nutzen können. Dazu müssen für jeden Nutzer dynamische, spezifische, zeitlich begrenzte Anmeldedaten erstellt werden, mit welchen eine Verbindung zu den Servern möglich ist. Diese Daten müssen beim Betreten eines Spiels an den jeweiligen Spieler vergeben werden.

Damit mehr als ein Spiel zur gleichen Zeit stattfinden kann, müssen die Spieler in Teilmengen unterteilt werden. Zwischen den Spielern in einer solchen Teilmenge müssen Signalisierungsdaten zum Verbindungsaufbau austauschbar sein. Spieler müssen diesen Teilmengen beitreten, und die Teilmengen verlassen können. Eine Teilmenge repräsentiert

dabei einen virtuellen „Tisch“ oder „Raum“. Ein „Raum“ muss zudem über einen Spieler verfügen, welcher Berechtigungen zum Entfernen von Spielern besitzt, falls ein unerwünschter Spieler beitritt oder betrügt.

4.1.2 Anforderungen an die Clientseitigen WebRTC-Verbindungen

Für jeden Peer muss ein Objekt existieren, welches sämtliche Verbindungen zu weiteren Peers verwaltet. Für jede Verbindung muss jeweils ein geordneter, zuverlässiger Datenkanal existieren. Es soll ein Event-Basiertes Nachrichtenprotokoll verwendet werden. Für verschiedene Events sollen – ähnlich dem Syntax von socket.io – Rückruffunktionen registrierbar sein können, die Parameter dieser Rückruffunktionen sollen dabei die Daten des Events beinhalten.

4.1.3 Anforderungen an das Brettspiel

Das Brettspiel soll durch das einfache Aufrufen einer Website spielbar sein. Die Webanwendung soll das Spielen einer virtuellen Nachbildung des Gesellschaftsspiels „Mensch Ärgere Dich Nicht“ ermöglichen. Das Spiel muss von bis zu vier Spielern gleichzeitig spielbar sein. Spieler sollen dem Spiel beitreten und es während das Spiel noch läuft wieder verlassen können. Falls beim Verlassen eines Spielers noch Spieler vorhanden sind, so sollen diese weiterspielen können.

Betrug

In Browserbasierten Webanwendungen ist es leicht, den JavaScript-Quellcode bei Laufzeit zu modifizieren. So können Spieler zum Beispiel unfaire Würfelergebnisse generieren, oder unerlaubte Spielzüge machen. Insbesondere die Würfelergebnisse lassen sich leicht manipulieren, ohne dass andere Spieler den Betrug überhaupt mitbekommen.

Spieler sollen daher nicht durch Modifikation von Script-Dateien betrügen können, ohne dass dies den anderen, nicht betrügenden Spielern angezeigt wird. Bei Benachrichtigung über den Betrug eines Spielers soll dieser Spieler vom, in Unterpunkt 4.1.1 beschriebenen, „Host“-Spieler aus dem Spiel entfernt werden können.

Spielregeln

Oh je das wird elend zu erklären, MÄDN ist leider etwas komplexer algorithmisch zu beschreiben als gedacht... evtl. nur Verweis auf die offiziellen Spielregeln?

4.2 Server-Infrastruktur

Für die Raum-Funktionalität, Signalisierungsmechanismen, sowie die Generierung von STUN- und TURN-Anmeldedaten müssen entsprechende Server existieren. Für die Implementierung werden alle diese Funktionen prototypisch in einem Server zusammengefasst.

4.3 Peer-To-Peer Netzwerkarchitektur

Bei der Netzwerkarchitektur des Peer-To-Peer Netzwerks kommen primär zwei Ansätze infrage: Das Authoritative-Peer-Modell, oder ein volles Peer-Netz.

Das Authoritative-Peer-Modell folgt dem Authoritative-Server-Modell, mit dem Unterschied, dass die Funktionen des Servers dabei von einem der Peers übernommen werden. Es wird eine Stern-Netzwerktopographie erzeugt, wobei der sogenannte „Host-Peer“ den zentralen Knoten bildet. Sämtlicher Datenaustausch läuft über den Host-Peer, welcher zudem die volle Authorität über den Spielstand besitzt. Die Client-Peers besitzen dabei nur den zur Darstellung des Spiels notwendigen, minimalen Spielstand. Der Vorteil dieses Modells ist die geringere Anzahl an Verbindungen im Netzwerk, sowie eine einfachere Entwicklung, da diese an das Authoritative-Server-Modell angelehnt werden kann. Problematisch ist jedoch der Single-Point-Of-Failure – verliert der Host-Peer die Verbindung, so gehen alle Verbindungen sowie der Spielstand verloren. Für diese Fälle muss ein „Host-Peer-Migrationsprozess“ existieren, um alle Verbindungen zu einem neu gewählten Host-Peer erneut zu erstellen. Existiert dieser nicht, so beendet das Verlassen des Host-Peers zwangsweise das Spiel.

Der Full-Mesh-Ansatz bildet das Gegenteil zum Authoritative-Peer-Modell. Hier verwaltet jeder Peer eine lokale Kopie des vollen Spielstands, und besitzt eine Verbindung zu jedem anderen Peer. Insbesondere bei Hohen Datenraten und einer hohen Anzahl an Peers skaliert diese Netzwerktopologie schlecht, was jedoch in diesem Fall aufgrund der geringen Spieleranzahl, sowie des geringen Datenvolumens praktisch irrelevant ist. Letztendlich

wurde dieser Ansatz für die Implementierung der Netzwerkarchitektur gewählt, da dieser keine komplexen Host-Migrationsprozesse vorraussetzt.

4.4 Zufallszahlen

Ist die Netzwerktopographie eines Brettspiels ein Authoritative-Server-Modell, so ist es einfach, faire, unabhängige Zufallszahlen zu erstellen. Ein Client kann einfach eine Anfrage an den Server schicken, welcher eine Zahl generiert und an den Client zurückschickt. Macht der Spieler darauf einen Spielzug, so kann dieser vom Server mit Hinblick auf die zuvor generierte Zufallszahl validiert werden. In einem Full-Mesh Peer-Netzwerk ist dies nicht ohne weiteres möglich.

4.4.1 Verteiltes Generieren von Zufallszahlen

Ein naiver Ansatz zur Generation von Zufallszahlen ist daher, jeden Peer eine Zahl z bis zu einem Wert max generieren zu lassen, welche ausgetauscht und addiert werden. Anschließend ergibt $z \mod max$ eine Zahl, welche – vorrausgesetzt mindestens einer der Peers hat eine zufällige Zahl generiert und nicht betrogen – zufällig ist. Hier existiert jedoch das sogenannte „Look-Ahead-Problem“, wobei ein Peer einfach auf die Zufallszahlen aller anderen Peers warten kann, bevor dieser die eigene Zufallszahl abgeschickt hat. Basierend auf den Zufallszahlen der weiteren Peers kann der böswillige Peer nun eine Zahl abschicken, welche diesem ein gewünschtes Ergebnis – verbunden mit einem Spielerischen Vorteil – bringt. Für dieses Problem existieren Lösungsansätze wie das „Lockstep-Protokoll“, unter Nutzung von Commitment-Verfahren. Bei einem Commitment-Verfahren werden die zu sendenden Daten zuerst mit einem Passwort verschlüsselt und unter den Teilnehmern ausgetauscht. Sind alle verschlüsselten Daten ausgetauscht, so werden die Passwörter ausgetauscht. Es ist einem Peer somit nicht möglich, auf alle weiteren Zahlen zu warten, ohne vorher eine eigene Zahl erstellt zu haben.

4.4.2 Seeded Random-Number-Generators

Eine einfachere Methode zur Generierung von fairen Zufallszahlen in verteilten Systemen wie Peer-Netzwerken sind daher „Seeded-Random-Number-Generators“. Dabei lässt sich ein Zufallszahlengenerator mit einem Startwert, dem sogenannten „Seed“ initialisieren. Der Generator erzeugt basierend auf diesem Seed eine Folge an Pseudozufallszahlen, welche

bei der Nutzung des gleichen Seeds stets gleich ist. Um eine hinreichende Zufälligkeit der Zahlenfolge zu garantieren, ist der Seed in der Regel eine Zufallszahl, welche zum Beispiel einmalig von einem Server generiert werden kann. Besitzt jeder Peer den gleichen Seed, so muss nur die Aktion des Generierens an sich synchronisiert werden – daraufhin generiert jeder Peer eine faire, unabhängige Pseudozufallszahl. Wichtig ist hierbei, dass der Spielstand zwischen den Peers unbedingt synchron gehalten werden muss. Generiert ein Teilnehmer eine Zahl zu viel oder zu wenig, so sind die Generatoren nicht mehr synchron. Dieser Ansatz bietet sich bei Brettspielen besonders an, da diese – im Gegensatz zu Echtzeitspielen – in der Regel in klar definierte „Züge“ und „Runden“ eingeteilt sind. Ein synchroner Spielstand zwischen den Spielern ist somit einfach beizubehalten.

5 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 5.1 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 5.1: Struktur des Projektes.

5.1 Bereitstellungsplattform

Zur Bereitstellung der in diesem Kapitel beschriebenen Server wird eine Virtuelle Maschine auf der Azure-Plattform genutzt. Azure ist eine Cloud-Computing-Plattform von

Microsoft, welche sowohl Software-, Platform- und Infrastructure-As-A-Service (SaaS, PaaS, IaaS) anbietet.

Azure ermöglicht das Erstellen von Rechnenressourcen in der Cloud, in der Form von Virtuellen Maschinen. Dabei existieren verschiedene Ausführungen von Virtuellen Maschinen, welche sich beim dem eingesetzten Betriebssystem, sowie der „Größe“ anpassen lassen. Die Größe gibt dabei den Arbeitsspeicher, die Anzahl der (virtuellen) Prozessorkerne, sowie die Art und Anzahl der Datenträger vor.

Für die prototypische Bereitstellung der Server wurde eine Virtuelle Maschine (VM) der Größe „B1s“ erstellt. Die B-Reihe ist dabei auf geringe Arbeitsbelastung ausgelegt. Eine B1s-VM verfügt über einen virtuellen Prozessorkern, ein Gigabyte Arbeitsspeicher und zwei Datenträger mit maximal 4 Gigabyte an temporärem Speicher. Für eine prototypische Bereitstellung, bei welcher nicht viel Serverlast zu erwarten ist, ist eine B1s-VM ausreichend.

Als Betriebssystem eignet sich praktisch jedes Betriebssystem, welches das bereitstellen von Servern ermöglicht. Azure bietet dabei viele verschiedene Linux-Distributionen und Windows-Server-Images. Als Betriebssystem wurde hier Ubuntu 18.04 LTS (Long-Term-Support, dt. Langzeitsupport) gewählt, primär da das Packet des genutzen STUN- und TURN-Servers in den Repositories von Ubuntu 18.04+ enthalten ist, und so einfach über den Packetmanager *apt* installiert werden kann.

Der VM muss zusätzlich eine öffentliche IP-Adresse zugewiesen werden, damit die auf der Maschine laufenden Server von Außen erreichbar sind. Für diese IP-Adresse lässt sich zudem ein Domain-Name in der Form

```
<DNS-Name>.<Region der VM>.cloudapp.azure.com
```

festlegen. Der für diese VM gewählte Domain-Name lautet somit:

```
ba-webrtc.westeurope.cloudapp.azure.com
```

5.2 Implementation der Netzwerkinfrastruktur

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“.

5.2.1 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, und die Nutzer in „Räume“ unterteilt, damit mehr als ein Spiel zur gleichen Zeit stattfinden kann.

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“-Framework verwendet. Der Quellcode befindet sich in der „Server.js“-Datei, und ist in Abbildung 5.1 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(config.server.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 5.1: 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

Auf der Serverseite muss eine socket.io-Instanz erstellt werden. Diese nutzt den gleichen HTTP-Server wie die express-Anwendung. Baut ein Client über die Clientseitige socket.io-Bibliothek eine Verbindung auf, so wird auf der Serverseite das *connection*-Event ausgelöst, welches den Socket des Clients als Parameter enthält. Innerhalb der Rückruffunktion dieses Events müssen alle Rückruffunktionen für diesen Socket registriert werden. Der dazu notwendige Quellcode ist in Abbildung 5.2 abgebildet.

```
1 [...]
2 const server = require('http').Server(app);
3 const io = require('socket.io')(server);
4 [...]
5 io.sockets.on('connection', (socket) => {
6   [...] // Rueckruffunktionen
7 }
```

Quellcode 5.2: Initialisierung des socket.io Servers – Server.js

Auf der Client-Seite muss zuerst die Clientseitige socket.io-Bibliothek eingebunden werden. Läuft socket.io auf dem gleichen Server wie der express-Webserver, so stellt socket.io die Clientseitige Script-Datei auf dem Pfad */socket.io/socket.io.js* statisch bereit. Die Datei wird im Header der jeweiligen HTML-Datei über ein *script*-Tag eingefügt:

```
<script src="/socket.io/socket.io.js"></script>
```

Clientseitig stellt socket.io die *io*-Funktion zur Verfügung, welche die Adresse des zugehörigen socket.io-Servers als Parameter nimmt. Die Funktion gibt die Socket-Instanz des Clients zurück. Das *connect*-Event wird bei erfolgreicher Verbindung zum Server ausgelöst. Der hierzu notwendige Quellcode ist in Abbildung 5.3 abgebildet.

```
1 $(document).on('DOMContentLoaded', () => {
2   const socket = io('http://ba-webrtc.westeurope.cloudapp.azure.com:1234');
3   const roomID = window.location.pathname.split('/')[2];
4   socket.on('connect', () => {
5     socket.emit('game-room-join', roomID);
6   );
7   [...] // Rueckruffunktionen
8 )
```

Quellcode 5.3: Clientseitiger Verbindungsaufbau – game.js

Das *connect*-Event wird nicht nur beim ersten Verbindungsaufbau, sondern auch bei jeder Neuverbindung aufgerufen. Um zu vermeiden, dass Rückruffunktionen mehrfach registriert werden, werden diese außerhalb der *connect*-Rückruffunktion erstellt.

5.2.2 Implementation der Peer-To-Peer Funktionalität

Zur Verwaltung der Peer-To-Peer-Funktionalität werden „Peer“-Objekte verwendet. Jeder Client besitzt dabei ein Peer-Objekt. Der Peer verwaltet die RTCPeerConnections, sowie die zugehörigen Datenkanäle. Es ist dem Peer möglich, ähnlich dem socket.io-Syntax Rückruffunktionen zu erstellen, welche bei Erhalt von Daten je nach Event aufgerufen werden.

Eventbasiertes Nachrichtenprotokoll

Der von den Peer-Objekten zum Datenaustasuch verwendete Syntax ist an den Syntax von socket.io angelehnt. An einem Peer lassen sich über die *on*-Funktion Rückruffunktionen registrieren (vgl. Abbildung 5.4). Diese registriert die Funktion in einem Map-Objekt.

```
1 Peer.prototype.on = function(e, callback) {
2   this.callbacks[e] = callback;
3 }
```

Quellcode 5.4: Funktion zur Registrierung von Rückruffunktionen – Peer.js

Die *broadcast*-Funktion emittiert ein Event an alle Peers, zu welchen der Datenkanal offen ist (vgl. Abbildung 5.5). Zudem existiert die *emit*-Funktion, welche ein Event nur an einen Peer emittiert. Das Format des Nachrichtenprotokolls ist Zeile 3 zu entnehmen: Jede Nachricht besitzt jeweils einen Event-String und ein Array an Daten.

```
1 Peer.prototype.broadcast = function(e, ...args) {
2   Object.values(this.connections).forEach((connection) => {
3     connection.dc.send(JSON.stringify({event: e, data: args}));
4   });
5 }
```

Quellcode 5.5: Funktion zum Emittieren eines Events – Peer.js

Bei Empfang einer Nachricht wird die Rückruffunktion, welche für das Event registriert ist, ausgeführt (vgl. Abbildung 5.6). Zudem wird die Peer-ID des Peers, welcher das Event emittierte als weiterer Funktionsparameter angehängt.

```
1 Peer.prototype._receiveMessage = function(e, remotePeerId) {
2   const message = JSON.parse(e.data);
3   if (message.event && this.callbacks[message.event]) {
4     this.callbacks[message.event](...message.data, remotePeerId);
5   }
6 }
```

Quellcode 5.6: Funktion bei Erhalt einer Nachricht – Peer.js

Signalisierungsprotokoll

Zur Signalisierung wird ein simples, proprietäres Protokoll verwendet. Eine Signal-Nachricht besitzt immer eine Quell-Peer-ID, eine Ziel-Peer-ID, eine Nachrichtenart, und die eigentlichen SDP-Daten des Signals. Dabei wird zwischen den Signal-Typen *offer*, *answer* und *ice-candidate* unterschieden. Die Signalnachrichten werden als **JSON!** encodiert und über den Signalisierungskanal weitergeleitet (siehe: 5.2.4 Signalisierungskanal).

```
{
    source : <Peer-ID>,
    target : <Peer-ID>,
    type : 'offer' | 'answer' | 'ice-candidate',
    data : <SDP-Daten>
}
```

Quellcode 5.7: Format des Signalisierungsprotokolls

Für ausgehende Signalnachrichten wird die *signal*-Funktion des Peers verwendet. Diese muss bei Erstellen des Peer-Objekts registriert werden und Daten an das Signalisierungsmedium schicken. Werden Daten über den Signalisierungskanal erhalten, so muss die *onsignal*-Funktion aufgerufen werden (vgl. Abbildung 5.13).

Verbindungen und Datenkanäle

Zum Verbindungsaufbau wird die *connect*-Funktion (Abbildung 5.8) verwendet. Als Argument nimmt diese die Peer-ID des Peers, zu welchem eine Verbindung aufgebaut werden soll. Die RTCPeerConnection wird in der Hilfsmethode *createConnection* des Peers erstellt, da diese Funktionalität auch auf der Empfängerseite bei Erhalt einer JSEP-Anfrage verwendet wird.

```
1 Peer.prototype.connect = function(remotePeerId) {
2   const connection = this._createConnection(remotePeerId);
3   this.connections[remotePeerId] = connection;
4
5   connection.createOffer().then((offer) => {
6     this.signal(this._createSignal('offer', offer, remotePeerId));
7     connection.setLocalDescription(offer);
8   }).catch((e) => console.error(e));
9 }
```

Quellcode 5.8: *connect*-Funktion – peer.js

Nach dem Erstellen der RTCPeerConnection wird der JSEP-Anfrage-Antwort Prozess zwischen den Peers eingeleitet. In Zeile 5 wird die Anfrage erstellt, in Zeile 6 wird diese dem Signalisierungsprotokoll entsprechend verpackt und über den Signalisierungskanal

weitergeleitet. Zuletzt muss in Zeile 7 die lokale Konfiguration des Peers – die SDP-Daten der Anfrage – gesetzt werden.

```
1  Peer.prototype._createConnection = function(remotePeerId) {
2    const connection = new RTCPeerConnection(this.rtcConfiguration);
3
4    const channel = connection.createDataChannel('game', {
5      negotiated : true,
6      id : i,
7      ordered : true,
8      maxRetransmits : null
9    });
10   channel.onmessage = (e) => this._receiveMessage(e);
11   // [...]
12   connection.onicecandidate = (e) => {
13     this.signal(this._createSignal('ice-candidate', e.candidate, remotePeerId));
14   }
15
16   connection.dc = channel;
17   return connection;
18 }
```

Quellcode 5.9: Erstellen von Verbindungen und Datenkanälen – peer.js

Der Quellcode zur Erstellung der RTCPeerConnection und des Datenkanals ist Abbildung 5.9 zu entnehmen. Die Konfiguration der RTCPeerConnection beinhaltet die vom Webserver bei Raumbeitritt erhaltenen STUN- und TURN-Serveradressen, sowie deren Nutzerdaten zur Authentifizierung. In den Zeilen 4 bis 10 wird ein Datenkanal mit dem Namen *game* erstellt. Die einzelnen Einstellungen werden in Tabelle 5.2 beschrieben.

Konfiguration	Wert	Beschreibung
negotiated	true	Da die Anwendung symmetrisch ist – beide Peers wissen, dass exakt ein geordneter, zuverlässiger Datenkanal erstellt werden soll – wird ein im vorab vereinbarter Datenkanal verwendet. Hier ist wichtig, dass die Konfiguration des Kanals auf beiden Seiten exakt übereinstimmt.
id	0	Die numerische ID des Datenkanals, bei im Vorab vereinbarten Datenkanälen muss diese gegeben sein und zwischen beiden Peers übereinstimmen.
ordered	true	Das unterliegende SCTP-Protokoll soll die Nachrichten geordnet absenden und erhalten.
maxRetransmits	null	Ist dieser Wert nicht gesetzt, so versucht das SCTP-Protokoll so lange ein Packet abzuschicken, bis dieses beim Empfänger angekommen ist.

Tabelle 5.2: Konfiguration der Datenkanäle.

Bei Erhalt eines ICE-Kandidaten in der *onicecandidate*-Rückruffunktion wird dieser über den

Signalisierungskanal an den externen Peer gesendet.

Bei Erhalt einer Signalnachricht wird diese in der *onsignal*-Funktion verarbeitet. Der Quellcode ist in Abbildung 5.10 abgebildet. Bei Erhalt einer Anfrage wird die RTCPeerConnection erstellt und die externe Konfiguration gesetzt. Daraufhin wird die Antwort erstellt und über die *signal*-Funktion an den externen Peer weitergeleitet. Zuletzt wird die Antwort als lokale Konfiguration der Verbindung gesetzt. Bei Erhalt einer Antwort wird diese als die externe Konfiguration der RTCPeerConnection gesetzt. Bei Erhalt eines ICE-Kandidaten wird dieser über die *addICECandidate*-Funktion der RTCPeerConnection hinzugefügt.

```
Peer.prototype.onsignal = function(e) {
  switch(e.type) {
    case 'offer':
      const connection = this._createConnection(e.src);
      this.connections[e.src] = connection;
      connection.setRemoteDescription(e.data).then(() => {
        return connection.createAnswer();
      }).then((answer) => {
        this.signal(this._createSignal('answer', answer, e.src));
        connection.setLocalDescription(answer);
      });
      break;
    case 'answer':
      this.connections[e.src].setRemoteDescription(e.data);
      break;
    case 'ice-candidate':
      this.connections[e.src].addIceCandidate(e.data);
      break;
  }
}
```

Quellcode 5.10: Funktion zum Verarbeiten von Signalnachrichten – Peer.js

5.2.3 Raum-Verwaltung

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.

Erstellen eines Raums

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

Quellcode 5.11: Event zum Erstellen eines Raums – Server.js

Erstellt ein Nutzer einen Raum, so sendet dieser das *game-room-create*-Event zum Server (vgl. Abbildung 5.11). 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.

Beitritt eines Raums

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 (vgl. Abbildung 5.12). 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 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.

In deb Spielregeln von „Mensch Ärgere Dich Nicht“ ist geregelt, 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 5.3: 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 5.3).

```
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 5.12: Event zum Betreten eines Raums – Server.js

Der Zugehörige Clientseitige Quellcode ist Abbildung 5.13 zu entnehmen. Bei Beitritt eines Raums wird zuerst das Peer-Objekt erstellt, welches die WebRTC Verbindungen und Datenkanäle verwaltet (siehe [TODO: REF]). Zudem wird in Zeile 3 die Funktion für ausgehende Signale gesetzt. Diese nutzt die socket.io-Verbindung zur Datenübertragung. In Zeile 4 wird die Rückruffunktion für das *signal*-Event erstellt, welche die Signaldaten an *onsignal*-Funktion des Peer-Objekts weitergibt. In Zeile 6 bis 9 wird das Spiel an sich erstellt und initialisiert. Tritt ein weiterer Spieler dem Spiel bei, so wird die in Zeile 11 definierte Rückruffunktion ausgeführt. Wichtig beim Beitritt eines neuen Spielers ist, dass die Zufallsfunktion des Spiels für alle Spieler mit einem neuen Seed initialisiert wird. Ansonsten ist der Zufallszahlen-Generator des neu beigetretenen Spielers nicht synchron mit denen der weiteren Spieler.

```
1  socket.on('game-room-joined', (players, started, seed, peerID, servers, isHost) => {
2    const peer = new Peer(peerID, servers, /* [...] */);
3    peer.setSignalFunction((data) => socket.emit('signal', roomID, data.target, data));
4    socket.on('signal', (e) => peer.onsignal(e));
5
6    const game = new Game(/* [...] */);
7    socket.on('game-start', (seed) => game.start(seed));
8    started ? game.start(seed) : game.seed(seed);
9    game.render();
10
11   socket.on('game-room-client-joining', (seed, peerID, color) => {
12   // [...]
13     game.seed(seed);
14     game.addPlayer(new Player(color, false));
15   });
16   // [...]
17   players.forEach((player) => {
18     player.peerID === peerID
19       ? game.localPlayerColor = player.color
20       : peer.connect(player.peerID);
21
22     game.addPlayer(new Player(player.color, game.localPlayerColor === player.color));
23   });
24 });
```

Quellcode 5.13: Raumbeitritt auf der Client-Seite – game.js

Um Signalisierungsbedingte „Race-Conditions“ beim Verbindungsaufbau zwischen Peers zu vermeiden, ist der zuletzt dem Raum beigetretene Peer stets der Peer, welcher die

Verbindung zu anderen Peers initiiert. In Zeile 20 wird – vorrausgesetzt die Peer-ID des Spielers ist nicht gleich der Peer-ID des lokalen Peers – eine Verbindung über die *connect*-Funktion zu jedem anderen Spieler im Raum erzeugt. Zuletzt müssen die Spieler dem Spieler-Array des Spiel-Objektes hinzugefügt werden.

Weitere Raumspezifische Events

Event	Beschreibung
game-room-join-failed	Wird vom Server zurückgegeben, wenn der Raum, dem der Client beitreten will, nicht verfügbar oder voll ist.
game-room-client-leaving	Wird vom Server an alle Spieler eines Raums gesendet, wenn ein Spieler den Raum verlässt.
game-room-host-migration	Ausgelöst vom Server, wenn der Host eines Raums diesen verlässt. Die Socket-ID des neuen als Host agierenden Spielers wird als Parameter dieses Events mitgegeben.

Tabelle 5.4: Weitere Raumspezifische Events.

Des weiteren existieren Events, welche das Verlassen von Räumen, entweder durch den Nutzer manuell ausgelöst, oder bedingt durch Verbindungsverlust, behandeln. Verlässt ein Spieler einen Raum, so erhalten alle verbliebenen Spieler ein Event darüber. Verlässt der Host-Spieler den Raum, so geht der Host-Status auf den nächsten Spieler im Spieler-Array über. Verlässt der letzte verbleibende Spieler den Raum, so wird dieser entfernt. Diese Events sind in Tabelle 5.4 zusammengefasst.

5.2.4 Signalisierungskanal

Das Weiterleiten von Signalisierungsnachrichten zwischen Peers wird über die gleiche socket.io-Verbindung geregelt, wie die Raumverwaltung. Dazu wird das *signal*-Event verwendet. Zur Signalisierung muss dabei immer die ID des Raums angegeben werden, in welchem die Signal-Nachricht weitergeleitet werden soll. Zudem muss die Peer-ID des Ziel-Peers angegeben werden. Existieren Raum und Peer, so wird das *signal*-Event an diesen weitergeleitet. Abbildung 5.14 zeigt die Serverseitige Rückruffunktion des *signal*-Events.

```
1 io.sockets.on('connection', (socket) => {
2   [...]
3   socket.on('signal', (roomID, targetID, e) => {
4     const room = rooms[roomID];
5
6     if (room) {
7       const target = room.players.find((player) => && player.peerID === targetID);
```

```
8         if (target) {
9           socket.to(playerSockets[target.peerID]).emit('signal', e);
10        }
11      }
12    });
13    [...]
14  }
```

Quellcode 5.14: Event zum Weiterleiten eines Signals – Server.js

5.3 Aufsetzen und Konfiguration eines STUN und TURN Servers

6 Evaluation

7 Zusammenfassung und Ausblick

8 Literaturverzeichnis

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

[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

[Gam] Gambetta, Gabriel: *Fast-Paced Multiplayer (Part I): Client-Server Game Architecture*. https://www.gabrielgambetta.com/client-server-game-architecture.html, . – Abgerufen 21.04.2021

[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

[LR14] Loreto, Salvatore ; Romano, Simon P.: *Real-Time Communication with WebRTC*. O'Reilly Media, 2014. – ISBN 978–1–449–37187–6

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

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

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

[o.A20] o.A: *Global Browser Games Market 2020-2030: COVID-19 Implications and Growth - ResearchAndMarkets.com*. https://www.businesswire.com/news/home/20200513005313/en/Global-Browser-Games-Market-2020-2030-COVID-19-Implications-and-Growth---ResearchAndMarkets.com, Mai 2020. – Abgerufen: 17.04.2021

[o.A21] o.A: RTCPeerConnection. Version: January 2021. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection. Mozilla Development Network, January 2021. – Forschungsbericht. – Abgerufen: 10.04.2021

[Sil15] Silveira, Rodrigo: *Multiplayer Game Development with HTML5*. Packt, 2015. – ISBN 978–1–78528–310–9

[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

[Tho13] Thornburgh, Michael C.: Adobe's Secure Real-Time Media Flow Protocol / Internet Engineering Task Force. Version: November 2013. https://tools.ietf.org/html/rfc7016. Internet Engineering Task Force, November 2013 (7016). – RFC. – ISSN 2070–1721. – Abgerufen: 08.04.2021