



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: 18. Mai 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

Danksagung

Hiermit möchte ich mich insbesondere bei meiner Familie für die Unterstützung während meines Studiums, sowie dieser Arbeit bedanken. Ich danke Professor Godehardt, dass dieser sich trotz vollem Terminplan auf dieses Thema eingelassen hat, sowie für die Hilfestellung bei der Themenfindung und für die Betreuung dieser Arbeit. Auch möchte ich Professor Baun danken, dass er sich als Zweitprüfer bereit erklärt hat.

Besonderer Dank gilt natürlich meiner Kaffeemaschine, welche das Erstellen dieser Arbeit bedauerlicherweise nicht überstanden hat.

"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 genetzte Miteinander so sehr aus, dass das normale Miteinander nahezu komplett vernachlässigt wird (vgl. World of Warcraft)."

– Netzwerke – www.stupidedia.org

Zusammenfassung

Peer-To-Peer-Netzwerkinfrastrukturen sind bei der Entwicklung von Mehrspieler-Spielen – aufgrund von weniger Hardwarebedarf zur Datenübertragung zwischen Nutzern – in der Regel kosteneffizienter als Client-Server-Modelle. Mit dem Lebensende des Adobe Flash Players im Dezember 2020 ist WebRTC nun (Stand 2021) die einzige Browserbasierte Peer-To-Peer Technologie, welche ohne Plugins verwendbar ist. Diese ist jedoch im Bereich der Browserbasierten Mehrspieler-Spieleentwicklung noch nicht weit verbreitet. Mit der Veröffentlichung von WebRTC als offizieller Web-Standard durch das W3C ist WebRTC nun bereit für weit verbreitete Anwendung, auch in Bereichen außerhalb der Video- und Audio-kommunikation.

Diese Arbeit befasst sich mit der Anwendbarkeit von WebRTC zur Entwicklung von Browserbasierten Mehrspieler-Brettspielen im Vergleich zu „traditionellen“ Client-Server-Architekturen. Dabei wird eine WebRTC-Netzwerkinfrastruktur konzipiert. Auf dieser wird das Brettspiel „Mensch Ärgere Dich Nicht“ prototypisch entwickelt und aufgesetzt.

Abstract

Peer-To-Peer networks are widely used for the development of multiplayer video games. Peer-To-Peer networks tend to be more cost-efficient than client-server networks, due to less computing-resources required to forward data between players. With the end-of-life of the Adobe Flash Player in December 2020, WebRTC is now the only peer-to-peer technology, which can be used out-of-the-box in most browsers without requiring additional plugins. WebRTC is currently (Q1 2021) not widely used for the development of browser-based multiplayer video games. Especially with the release of the WebRTC-Recommendation by the W3C in January 2021, WebRTC is now a fully fledged web-standard, ready for wide-spread use in areas besides video- and audiocommunications.

This thesis evaluates the usability of WebRTC for the development of browser-based multiplayer board games in contrast to more “traditional” client-server network models. A WebRTC network infrastructure is deployed. Based on this, a prototype of the popular german board game “Mensch ärgere Dich nicht” is developed and deployed.

Inhaltsverzeichnis

Abbildungsverzeichnis	VI
Tabellenverzeichnis	VII
Quellcodeverzeichnis	VIII
Abkürzungsverzeichnis	IX
1 Einleitung	1
1.1 Zielsetzung	1
1.2 Aufbau der Arbeit	1
2 Grundlagen	3
2.1 Echtzeitanwendungen	3
2.2 Netzwerkarchitekturen	3
2.2.1 Client-Server-Modelle	3
2.2.2 Peer-To-Peer Netzwerke	4
2.3 Network Address Translation	4
2.4 Web Real-Time-Communication	5
2.4.1 Aufbau von WebRTC	5
2.4.2 Protokolle	7
2.5 Node.js	12
2.5.1 NPM: Node Package Manager	13
2.5.2 Verwendete Node-Packete	13
3 WebRTC in Mehrspielerspielen	15
4 Konzept	18
4.1 Anforderungen	18
4.1.1 Anforderungen – Netzwerkstruktur	18
4.1.2 Anforderungen – Clientseitige WebRTC-Verbindungen	19
4.1.3 Anforderungen – Brettspiel	19
4.2 Konzept – Serverinfrastruktur	21
4.3 Konzept – Peer-To-Peer Netzwerkarchitektur	22
4.4 Konzept – Brettspiel	22
4.4.1 Zufallszahlen	22
5 Implementation	24
5.1 Bereitstellungsplattform	25

5.2	Netzwerkinfrastruktur	25
5.2.1	Webserver	25
5.2.2	Peer-To-Peer Funktionalität	28
5.2.3	Raumverwaltung	32
5.2.4	Signalisierungskanal	35
5.2.5	Bereitstellung des Webservers	36
5.3	STUN- und TURN-Server	36
5.3.1	Installation	37
5.3.2	Konfiguration	37
5.3.3	Authentifizierung	38
5.3.4	Ports	40
5.3.5	Test	40
5.4	Implementation des Brettspiels	41
5.4.1	Spielablauf	42
5.4.2	Darstellung des Spielbretts	44
6	Evaluation	46
6.1	Probleme der Implementation	46
6.2	WebRTC für Browserbasierte Mehrspieler-Brettspiele	48
7	Zusammenfassung und Ausblick	50
8	Literaturverzeichnis	51
A	Anhang – Peer.js	54
B	Anhang – Server.js	58
C	Anhang – game.js	62

Abbildungsverzeichnis

2.1	Beispielhafte Interaktion in einem Authoritative-Server-Modell.	4
2.2	Diagramm der WebRTC-Architektur.	6
2.3	WebRTC Protokollstack.	7
2.4	JSEP-Verbindungsaufbau.	8
2.5	Diagramm der verschiedenen ICE-Kandidaten.	10
3.1	Einige Jocly-Spiele, mit WebRTC Telepräsenz.	15
3.2	Google CubeSlam, mit integriertem WebRTC Videostream.	16
4.1	Laufendes Spiel mit drei Spielern. Gelb ist am Zug. Die A-Felder sind die farbigen Felder am Rand des Spielbretts, die B-Felder sind die farbigen Felder in der Mitte des Spielbretts.	20
4.2	Konzept der Netzwerkinfrastruktur.	21
5.1	Port-Regeln der VM im Azure-Webportal.	40
5.2	Ablaufdiagramm des Brettspiels.	42
5.3	Würfel-Nachrichtenaustausch.	43
5.4	Ziehen-Nachrichtenaustausch.	44
5.5	Schemenhafte Darstellung des Spielbretts, mit Positionen und Farben.	45

Tabellenverzeichnis

2.1	Vergleich von TCP und UDP mit SCTP.	12
2.2	Verwendete Prozessverwaltungssysteme nach Betriebssystem.	14
3.1	Verwendung von WebRTC in Videospiel-Frameworks und Engines.	17
5.1	Struktur des Projektes.	24
5.2	Konfiguration der Datenkanäle.	31
5.3	Spielerfarben.	33
5.4	Weitere Raumspezifische Events.	35
5.5	Erläuterung der Konfigurationsparameter.	37
5.6	Portweiterleitung.	40
6.1	Vergleichskriterien	48

Codeverzeichnis

2.1	SDP-Datenstring eines Relais-ICE-Kandidaten	11
5.1	express Server – Server.js	27
5.2	Initialisierung des socket.io Servers – Server.js	27
5.3	Clientseitiger Verbindungsaufbau – game.js	28
5.4	Funktion zur Registrierung von Rückruffunktionen – Peer.js	28
5.5	Funktion zum Emittieren eines Events – Peer.js	29
5.6	Funktion bei Erhalt einer Nachricht – Peer.js	29
5.7	Format des Signalisierungsprotokolls	29
5.8	connect-Funktion – peer.js	30
5.9	Erstellen von Verbindungen und Datenkanälen – peer.js	30
5.10	Funktion zum Verarbeiten von Signalnachrichten – Peer.js	31
5.11	Event zum Erstellen eines Raums – Server.js	32
5.12	Event zum Betreten eines Raums – Server.js	33
5.13	Raumbeitritt auf der Client-Seite – game.js	34
5.14	Event zum Weiterleiten eines Signals – Server.js	35
5.15	Coturn-Konfigurationsdatei – turnserver.conf	38
5.16	Funktion zum Erstellen von Zugangsdaten – utils.js	39
5.17	Relais-Kandidat	40
5.18	Server-Reflexiver-Kandidat	41
5.19	Spielstandsdaten – Game.js	41
5.20	Senden des Spielstands – game.js	42

Abkürzungsverzeichnis

W3C	World Wide Web Consortium	1
WebRTC	Web Real-Time Communication	1
P2P	Peer-To-Peer	1
HTTP	Hypertext Transfer Protocol	12
NPM	Node Package Manager	13
IETF	Internet Engineering Task Force	5
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	4
STUN	Session Traversal Utilities for NAT	8
TURN	Traversal Using Relays around NAT	9
SCTP	Stream Control Transport Protocol	6
TLS	Transport Layer Security	12
DTLS	Datagram Transport Layer Security	12
API	Application Programming Interface	
TCP	Transmission Control Protocol	11
UDP	User Datagram Protocol	11
OSI	Open Systems Interconnection	11
IP	Internet Protocol	11
SIP	Session Initialization Protocol	7
SDP	Session Description Protocol	7
URL	Unique Resource Locator	14
HTML5	Hypertext Markup Language 5	41
VM	Virtuelle Maschine	25
RTMP	Real-Time Media Protocol	15
RTMFP	Real-Time Media Flow Protocol	15
RC4	Rivest Cipher 4	43
HMAC	Hash-Based Message Authentication Code	39
SHA-1	Secure Hash Algorithm 1	39
JSON	JavaScript Object Notation	29
DOM	Document Object Model	44

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 ein offizieller Webstandard in seiner – bezüglich der zentralen Funktionalität – finalen Form. WebRTC ist eine Peer-To-Peer Webtechnologie, und wird primär für browserbasierte Echtzeitanwendungen wie Audio- und Videokommunikation verwendet.

Die Beliebtheit von interaktiven Mehrspielerspielen, 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 [BGM20].

In der Regel basieren browserbasierte Mehrspielerspiele auf einer Client-Server-Architektur, wobei der Server die Rolle des bestimmenden Spielleiters übernimmt.

Bei browserbasierten Mehrspielerspielen ist die Nutzung von Peer-To-Peer (P2P)-Netzwerken zum Datenaustausch 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 Webanwendungen, ohne die Nutzung von Plug-Ins oder Drittanbietersoftware, direkt untereinander zu vernetzen.

1.1 Zielsetzung

In dieser Arbeit sollen 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

Die vorliegende Arbeit ist – neben diesem Einleitungskapitel – in insgesamt sechs weitere Kapitel unterteilt:

2. **Grundlagen:** Dieses Kapitel befasst sich mit den für die Implementation und Evaluation notwendigen technischen Grundlagen. Hier wird insbesondere auf die Struktur von WebRTC, sowie die verwendeten Protokolle eingegangen.
3. **WebRTC in Mehrspielerspielen:** Dieses Kapitel befasst sich mit dem Stand der Verwendung von WebRTC in Mehrspielerspielen, und warum diese Technik in diesem Themenbereich relevant ist.
4. **Konzept:** In diesem Kapitel werden die Anforderungen an die Netzwerkstruktur und das Brettspiel erläutert. Dabei handelt es sich nicht um eine formale Anforderungsanalyse, es soll lediglich ein Überblick über die zu implementierenden Komponenten gegeben werden. Zudem werden Konzepte, welche für die Implementation relevant sind, diskutiert.
5. **Implementation:** Dieses Kapitel behandelt die Implementation der WebRTC- und Serverinfrastruktur, sowie des Brettspiels.
6. **Evaluation:** In diesem Kapitel werden sowohl aufgetretene Probleme bei der Implementierung, als auch die Verwendung von WebRTC für browserbasierte Mehrspielerbrettspiele, diskutiert.
7. **Zusammenfassung und Ausblick:** Zuletzt wird der Inhalt der Arbeit reflektiert, und Ansätze für zukünftige, weiterführende Arbeiten gegeben.

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 Signalisierungsmechanismen 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 [r19]. 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 Gesprächspartner 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

In dieser Arbeit werden zwei Netzwerkarchitekturen behandelt: Client-Server und Peer-To-Peer 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 Browsern kann die WebSocket-API zur Erstellung von bidirektionalen Client-Server-Verbindungen verwendet werden.

Authoritative-Server

Bei der Entwicklung von Mehrspielerspielen 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).

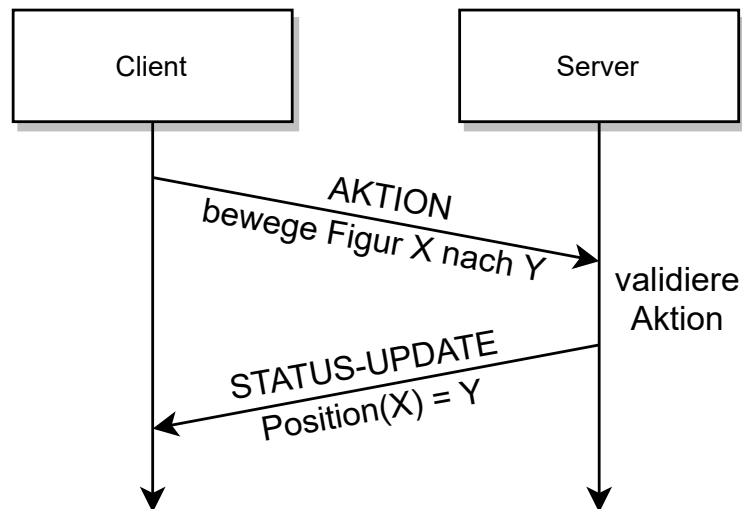


Abbildung 2.1: Beispielhafte Interaktion in einem Authoritative-Server-Modell.

Dieser Ansatz eignet sich besonders für Spiele, in welchen Spieler sich 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

Ein weiterer, gut definierter, aber für browserbasierte Spiele selten verwendeter Ansatz für die Vernetzung von Spielern ist die Peer-To-Peer (P2P)-Architektur. P2P-Netzwerke sind dezentralisiert. Es existiert kein zentraler Server, die Nutzer (Peers) sind gleichberechtigt und tauschen Daten direkt untereinander aus. Somit ist kein zentraler Server von Nöten, um Daten zwischen Clients weiterzuleiten. Dies führt zu Kostenersparnissen auf Seiten des Anbieters, in Form von weniger Bedarf an Hardware.

2.3 Network Address Translation

Aufgrund der Form von IPv4-Adressen ist deren Anzahl limitiert. Daher befinden sich viele Geräte in der Regel in einem Subnetz – einem lokalen IPv4-Adressraum – und nutzen zusammen eine öffentliche IPv4-Adresse. Network Address Translation (NAT) ermöglicht es den Geräten im Subnetz, Geräte außerhalb des Subnetzes zu erreichen. Router verwalten dazu eine NAT-Übersetzungstabelle, mit welcher Nachrichten zwischen externen und inter-

nen Geräten weitergeleitet werden [Bau19]. NATs sind insbesondere bei der Erstellung von P2P-Verbindungen problematisch, da sich ein Gerät von außerhalb nicht direkt mit einem Gerät in einem Subnetz verbinden kann [HS01].

2.4 Web Real-Time-Communication

Bei WebRTC handelt es sich um einen quelloffenen Standard zur Echtzeitkommunikation zwischen Browsern. Der Standard ermöglicht es Browsern, welche den WebRTC Standard unterstützen, sich ohne zusätzliche Software oder Plugins direkt miteinander zu verbinden. Dies führt in der Regel zu – im Vergleich zu Datenaustausch über einen zentralen Server – geringeren Latenzen, sowie Kostenersparnissen durch weniger Serverlast. WebRTC ist primär auf Audio- und Videokommunikation ausgelegt, ermöglicht aber auch das Senden von arbiträren 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 Webbrowsern durch die Entwicklung einer JavaScript-API. Seitdem arbeiten das W3C und die Internet Engineering Task Force (IETF) an der Standardisierung der JavaScript-APIs [LR14].

Am 26. Januar 2021 veröffentlichte das W3C die WebRTC-Recommendation – WebRTC ist damit ein offizieller, vom W3C und der IETF 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. Diese ermöglichen, unter anderem, den Aufbau von Verbindungen, Audio- und Videoübertragung, sowie Datenübertragung.

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 setzt sich aus einer Reihe an JavaScript Schnittstellen zusammen. Diese Schnittstellen können auf das unterliegende Framework zugreifen. Primär werden dabei die folgenden Schnittstellen verwendet:

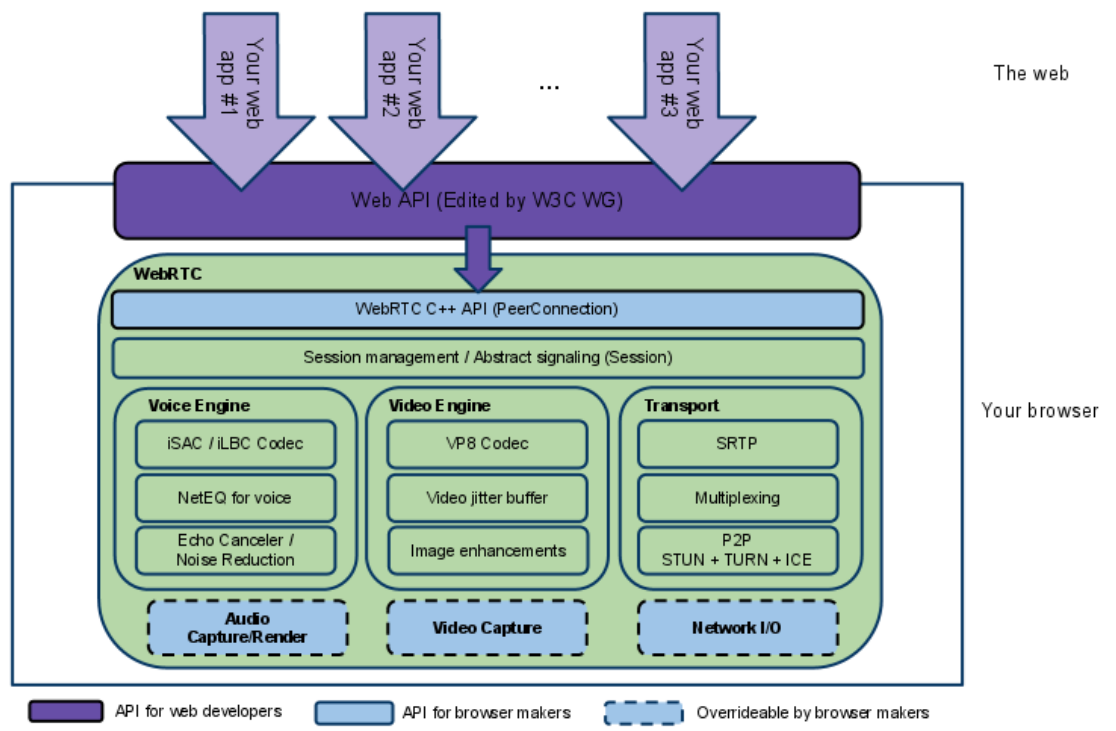


Abbildung 2.2: Diagramm der WebRTC-Architektur.

Quelle: <https://webrtc.github.io/webrtc-org/architecture/> (Stand 07.05.2021)

- Die **RTCPeerConnection**-Schnittstelle repräsentiert eine WebRTC-Verbindung zwischen dem lokalen Browser (Local-Peer), und einem externen Browser (Remote-Peer) [rtcc]. Die Signalisierungsmechanismen folgen dabei dem JavaScript Session Establishment Protocol (JSEP), die Verbindung selbst wird via dem Interactive Connectivity Establishment (ICE) Framework hergestellt [LR14, rtcc].
- Ein **RTCDataChannel** ist ein, von der RTCPeerConnection erstellter, bidirektionaler Datenkanal, welcher den Austausch von arbiträren Nachrichten zwischen Browsern ermöglicht. Eine RTCPeerConnection kann mehrere Datenkanäle besitzen. Zum Datenaustausch wird das Stream Control Transport Protocol (SCTP) verwendet [LR14, rtcb].
- Die **MediaStream**-API dient dazu, Audio- und Videosignale eines Gerätes abzurufen. Zur Übertragung dieser wird das Real-Time Transport Protocol (RTP), beziehungsweise das Secure Real-Time Transport Protocol (SRTP) verwendet [LR14].

WebRTC Framework

Das WebRTC-Framework gliedert sich in Audio-, Video-, und Übertragungssysteme. Die Audio- und Videosysteme befassen sich dabei unter Anderem mit der Abfrage von Audiodaten des Gerätemikrofons oder Videodaten über eine Kamera. Zudem sind diese Systeme für die En- und Decodierung von Audio- und Videodaten auf Basis verschiedener Audio-

und Videocodecs zuständig. Die Transportsysteme umfassen Protokolle und Systeme, um Sitzungen zwischen Peers aufzubauen, und Daten zwischen den Peers zu auszutauschen.

2.4.2 Protokolle

WebRTC nutzt eine Reihe an Protokollen und Frameworks, um den Sitzungsaufbau, die Verbindung, sowie den Datenaustausch zwischen Peers zu ermöglichen. Der (vereinfachte) WebRTC Protokollstack ist in Abbildung 2.3 dargestellt.

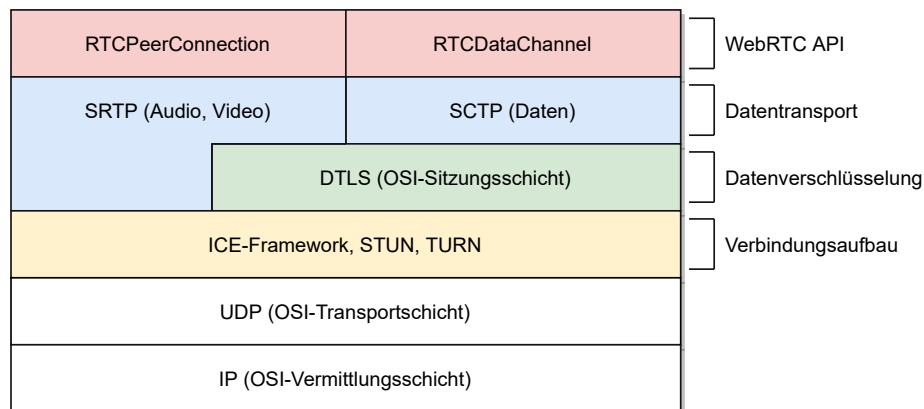


Abbildung 2.3: WebRTC Protokollstack.

Quelle: Nach <https://webrtc-security.github.io/> (Stand 07.05.2021), Abb. 2

JavaScript Session Establishment Protocol

Die Signalisierungsebene einer WebRTC-Anwendung ist nicht vom WebRTC-Standard definiert, damit verschiedene Applikationen mitunter verschiedene Signalisierungsprotokolle, wie zu Beispiel das Session Initialization Protocol (SIP), oder ein proprietäres Protokoll nutzen können [LR14].

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 [LR14]. Eine Sitzung wird immer zwischen zwei Endpunkten etabliert – einem initiiierendem Endpunkt, und einem empfangenden Endpunkt. In den folgenden Paragraphen werden die Synonyme „Alice“ und „Bob“ für diese Endpunkte verwendet.

Beide Endpunkte besitzen jeweils eine lokale, und eine externe Konfiguration (eng. „Description“). Diese definieren die Sitzungsparameter, zum Beispiel welche Daten auf der Senderseite versendet werden sollen, welche Daten auf der Empfängerseite zu erwarten sind, oder Informationen über verwendete Audio- und Videocodecs. Diese Informationen werden über das Session Description Protocol (SDP) in Form von Key-Value-Strings definiert [Bis14].

Die JSEP-API stellt 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` [rtcc].

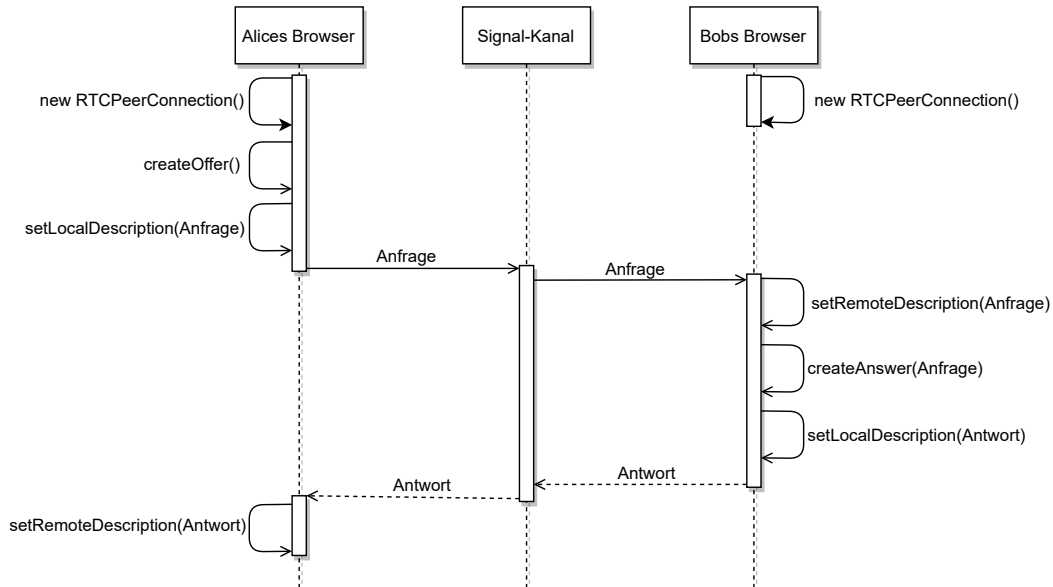


Abbildung 2.4: JSEP-Verbindungs Aufbau.

Um eine Verbindung aufzubauen ruft Alice erst `createOffer()` auf. Daraufhin wird ein SDP-Paket (*Anfrage*) generiert, welches die lokalen Sitzungsparameter enthält. Alice setzt nun ihre lokale Konfiguration via `setLocalDescription()`. Das SDP-Paket wird über einen nicht vorgegebenen Signalisierungskanal zu Bob gesendet. Dieser setzt daraufhin die externe Konfiguration seiner Verbindung via `setRemoteDescription()`, und ruft daraufhin die Funktion `createAnswer(Anfrage)` auf, welche eine Antwort (*Antwort*) generiert. Bob setzt diese als seine lokale Konfiguration, und sendet die Antwort zurück zu Alice. Zuletzt setzt Alice ihre externe Konfiguration. 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 [LR14, rtce]. Der vereinfachte Ablauf des Verbindungsaufbaus ist Abbildung 2.4 zu entnehmen.

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 diesen Peer zu schicken [PHRW⁺20].

TURN: Traversal Using Relays around NAT

Im Gegensatz zu Client-Server-Verbindungen, welche in der Regel nur vom Client eröffnet werden können, kann eine Peer-To-Peer Verbindung 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 eine solche Interaktion erlaubt [HS01]. In solchen Fällen muss ein Traversal Using Relays around NAT (TURN)-Server verwendet werden. Ein TURN-Server agiert als ein zwischen den Peers liegender Relais-Server, für Fälle, in denen eine direkte Verbindung zwischen Peers aufgrund von NAT-Einschränkungen nicht möglich ist.

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 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 [rtca].

Eine `RTCPeerConnection` besitzt immer einen ICE-Agenten. Einer der Agenten einer Verbindung agiert als der kontrollierende, der Andere als der kontrollierte Agent. Der kontrollierende Agent hat die Aufgabe, das ICE-Kandidatenpaar, welches für die Verbindung genutzt werden soll, auszuwählen. Ein ICE-Kandidat beinhaltet Informationen – im SDP-Format – über Transportadressen, 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 [rtce].

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 [rtce]. 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 [rtce]. Wie aus Abbildung 2.5 zu entnehmen, werden 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-Adresse besitzt, oder sich sowohl der lokale, als auch externe Peer im gleichen Subnetz, oder auf dem gleichen Gerät befinden [rtce].
- 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 [rtce].

- 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 [rtce].

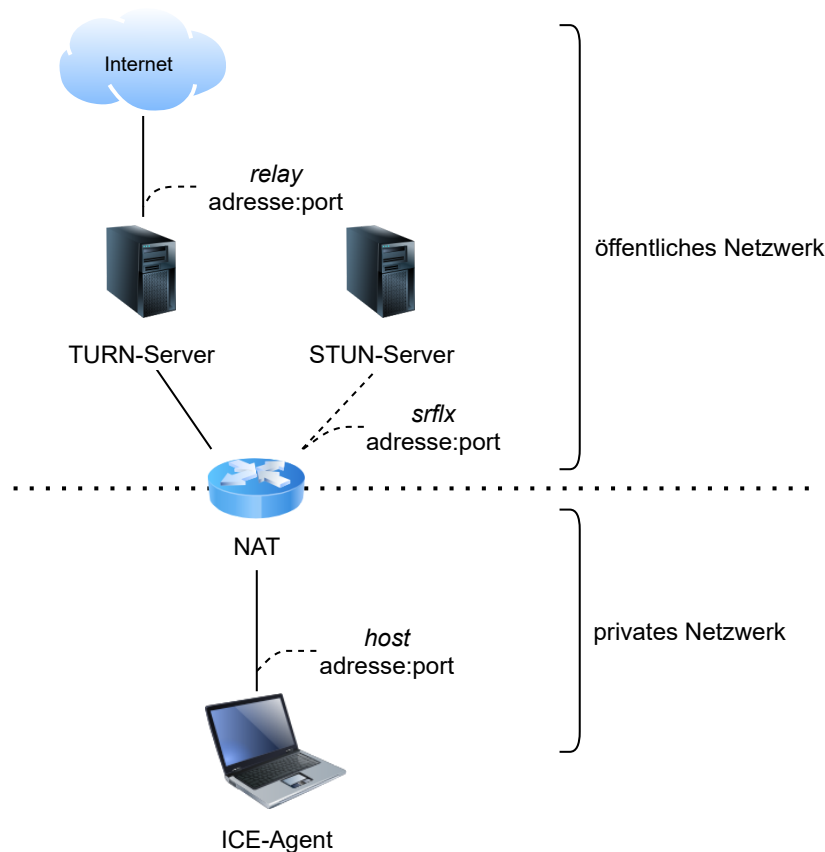


Abbildung 2.5: Diagramm der verschiedenen ICE-Kandidaten.
Quelle: Nach RFC5245, Abb. 2

Die STUN und TURN Server lassen sich beim Erstellen der `RTCPeerConnection` konfigurieren. WebRTC prüft die Verbindung zu allen möglichen ICE-Kandidaten eines Peers, sobald die lokale Beschreibung einer Verbindung gesetzt ist. Dieser Prozess – der sogenannte „ICE-Sammelprozess“ – setzt eine Kommunikation mit STUN- und TURN-Servern voraus 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 parallel zum ICE-Sammelprozess stattfinden kann [rtcd].

Jede `RTCPeerConnection` besitzt daher die Rückruffunktion „*onicecandidate*“, welche bei Erhalt eines neuen ICE-Kandidaten aufgerufen wird. Die Parameter dieser Rückruffunktion enthalten die SDP-Daten des Kandidaten. 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. Nachdem diese Daten an den externen Peer gesendet wurden, muss der Kandidat auf der Empfängerseite über die *addIceCandidate*-Funktion der *RTCPeerConnection* hinzugefügt werden [rtce].

```
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

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 nachrichtenorientiertes Transportprotokoll, welches im Open Systems Interconnection (OSI)-Referenzmodell, ähnlich dem User Datagram Protocol (UDP) und dem Transmission Control Protocol (TCP), auf der Transportschicht liegt. Das Protokoll arbeitet dabei basierend auf verbindungslosen Netzwerkprotokollen, wie dem Internet Protocol (IP) [Ste07].

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

- **Reihenfolge:** SCTP ermöglicht es, sowohl geordnete, als auch ungeordnete Datenströme aufzubauen [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 parallelen Datenströmen innerhalb einer Verbindung. Dazu ist der Datenteil eines SCTP-Packets in sogenannte „Chunks“ aufgeteilt, wobei Daten-Chunks jeweils einem Datenstrom zugeordnet werden können [Ste07].

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

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

Tabelle 2.1: Vergleich von TCP und UDP mit SCTP.

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 auf Datagramm-basierenden Verbindungen. Im Gegensatz zu TLS, welches für die Nutzung mit TCP konzipiert wurde, kann DTLS also über UDP übertragen werden.

Die SCTP-Verbindung eines Datenkanals läuft nicht direkt auf dem Internet Protocol (IP). 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 – welcher wiederum über UDP liegt – zwischen den Verbindungspartnern ausgetauscht [Res19]. Im Gegensatz zu SCTP bietet SRTP eigene Verschlüsselung, und kann auch ohne DTLS-Tunnel verwendet werden. Wann welche Konfiguration verwendet wird ist dabei je Implementation in Browser oder Anwendung unterschiedlich.

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. Bei Node.js handelt es sich um eine quelloffene Software, diese 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 wiederum von weiteren Modulen oder Programmen verwendet werden können. Module können über das *require*-Stichwort geladen werden. Node.js bietet integrierte Webserverfunktionalität via dem Hypertext Transfer Protocol (HTTP)-Modul, welches das Erstellen eines Webserver ermöglicht [nod]. 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 enthalten¹, und kann mit den entsprechenden Paketmanagern, beziehungsweise über die Website² heruntergeladen und installiert werden.

¹Weitere Informationen: <https://nodejs.org/en/download/package-manager/>

²Node.js Downloads: <https://nodejs.org/en/download/>

2.5.1 NPM: Node Package Manager

Zur Verwaltung und zum Teilen von Paketen nutzt Node.js den Node Package Manager (NPM). Ein Paket sind in diesem Kontext ein oder mehrere Module, gekoppelt mit allen Dateien, welche diese benötigen. Pakete werden auf *npmjs.com* gehostet. Die Liste der von einem Projekt verwendeten Pakete wird in der Datei *package.json* – im Wurzelverzeichnis der Node.js-Anwendung – gespeichert.

2.5.2 Verwendete Node-Pakete

Neben den standardmäßig in Node.js enthaltenen „http“- und „crypto“-Modulen werden drei weitere Pakete genutzt: die „socket.io“ Bibliothek, das „express“-Framework, und der Prozessmanager „PM2“.

socket.io

„socket.io“ ist eine Bibliothek, welche bidirektionale Echtzeitkommunikation zwischen einem Client und einem Server ermöglicht. Dazu nutzt Socket.io intern WebSockets[sio]. Die WebSocket-API ist in allen modernen Browsern standardmäßig implementiert³, und ermöglicht Kommunikation zwischen einem Browser und einem Server. Der Datenaustausch findet dabei über TCP statt [FM11]. Socket.io läuft serverseitig auf einem Node.js Server [sio].

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, vorausgesetzt 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 [sio].

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. Diese werden beim Eingang von Daten je nach Event-Typ aufgerufen. Der Server kann ebenfalls Events an Clients emittieren.

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

express

„express“ ist ein Node-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 Unique Resource Locator (URL)s – den sogenannten „Pfaden“ – verschiedene Aktionen auszuführen [exp]. 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 [exp].

Process Manager 2 (PM2)

„PM2“ ist ein Prozessmanager für Node.js-Anwendungen. PM2 ermöglicht das Starten, Stoppen, Überwachen und Löschen von Prozessen. Zudem automatisiert PM2 das Erstellen von Startup-Skripten, mit welchen bei (Neu-)Start einer Maschine Node.js-Anwendungen automatisch als System-Daemon-Prozesse gestartet und konfiguriert werden. Je nach verwendetem Betriebssystem nutzt PM2 dafür verschiedene Prozessverwaltungssysteme (vgl. Tabelle 2.2). Die Nutzung von PM2 basiert auf Kommandozeilenbefehlen [pm2].

System	Betriebssystem
systemd	Ubuntu 16+, Debian 7+, CentOS 7
launchd	MacOSx
openrc	Gentoo Linux, Arch Linux
rcd	FreeBSD
systemv	CentOS 6, Amazon Linux

Tabelle 2.2: Verwendete Prozessverwaltungssysteme nach Betriebssystem.
Quelle: <https://pm2.keymetrics.io/docs/usage/startup/> (Stand 16.05.21)

3 WebRTC in Mehrspielerspielen

Vor dem Lebensende des Adobe Flash-Players im Dezember 2020 wurde dieser häufig zur Entwicklung von browserbasierten Mehrspielerspielen verwendet. Diese Spiele wurden auf verschiedenen Plattformen wie zum Beispiel *Newgrounds.com* oder *Itch.io* angeboten. Der Flash-Player verfügte über Peer-To-Peer Fähigkeiten via Adobes Real-Time Media Protocol (RTMP), beziehungsweise dem Real-Time Media Flow Protocol (RTMFP). RTMP basiert auf TCP, RTMFP auf UDP-Verbindungen. Für die Signalisierungsmechanismen nutzte der Flash-Player *Adobe Cirrus* (non-kommerziell) oder den *LiveCycle Collaboration Service* (kommerziell) [Krc09]. Damit bot der Flash-Player als Plattform für Mehrspielerbrettspiele eine kostengünstige Alternative zur Nutzung von 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.



Abbildung 3.1: Einige Jocly-Spiele, mit WebRTC Telepräsenz.

Quelle: <https://bloggeek.me/jocly-webrtc-interview/> (Stand 07.05.2021)

WebRTC wird bereits in einigen browserbasierten Mehrspielerspielen, sowie Networking- und Spiel-Frameworks verwendet. In der Regel wird WebRTC jedoch lediglich für Sprach-

und Videokommunikation eingesetzt. Die Strategie- und Brettspielplattform *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].



Abbildung 3.2: Google CubeSlam, mit integriertem WebRTC Videostream.
Quelle: <https://experiments.withgoogle.com/cube-slam> (Stand 07.05.2021)

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 eine Bildschirmaufnahme 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*¹ nutzt WebRTC zum Austausch von Spieldaten.

Die Nutzung von WebRTC in (kommerziellen) Mehrspielerspielen, Frameworks und Plattformen lässt sich in drei grobe Kategorien aufteilen: Telepräsenz, Datenübertragung, und Render-Streaming. Beispiele für weitere Spiele und Frameworks, welche in diese Kategorien fallen, sind in Tabelle 3.1 gelistet.

Mehrspielerspiele, welche auf den Videospiel-Engines Godot und Construct2 oder Construct3 basieren, können auch unter Nutzung von WebRTC browserbasiert aufgesetzt werden. Bei diesen Spielen handelt es sich in der Regel aber um Echtzeitspiele, oder diese sind – im Falle von Brettspielen – auf maximal zwei Spieler begrenzt².

¹vgl. <https://hacks.mozilla.org/2013/03/webrtc-data-channels-for-great-multiplayer/>, Stand 12.05.2021

²vgl. <https://www.construct.net/en/tutorials/building-basic-board-game-390> und Construct-Mehrspielerspiele: <https://www.construct.net/en/free-online-games/multiplayer-7>, Stand: 12.05.2021

Software	Nutzung	Beschreibung
Roll20	Telepräsenz	„Roll20“ ist ein online-Tabletop zum Spielen des Rollenspiels „Dungeons and Dragons“. Roll20 nutzt WebRTC für Telepräsenz in Form von Audio- und Videochat [rol20].
Tabloro	Telepräsenz	„Tabloro“ ist eine Virtuelle Tabletop-Plattform im Browser. Diese nutzt WebRTC für Telepräsenz in Form von Audio- und Videochat ¹ .
Construct	Datenübertragung	Die „Construct“ Videospielengine (Construct2 und Construct3) integriert WebRTC zur Datenübertragung zwischen Spielern. Falls WebRTC nicht verwendet werden kann, werden WebSockets verwendet. Spiele, welche auf dieser Engine basieren können auch im Browser gespielt werden [con16] ² . Die Engine wird primär für 2D-Plattformerspiele verwendet.
Godot	Datenübertragung	Die Godot-Engine unterstützt seit 2019 WebRTC zur Datenübertragung [god] ³ .
Unity	Render-Streaming	In der Unity-Engine kann WebRTC verwendet werden, um Bilder, welche auf Hardwarestarken Geräten gerendert wurden, auf Hardwareschwache Geräte zu streamen [San19].

Tabelle 3.1: Verwendung von WebRTC in Videospiel-Frameworks und Engines.

Im Bereich der Rundenbasierten Brettspieleentwicklung im Browser findet WebRTC nur begrenzt Anwendung. Diese beschränkt sich primär auf die zuvor beschriebene Nutzung für Telepräsenz zwischen Spielern.

¹vgl. <https://github.com/fyyyyy/tabloro>, Stand: 12.05.2021

²vgl. auch <https://www.construct.net/en/tutorials/multiplayer-tutorial-concepts-579/connectivity-2>, Stand 12.05.2021

³vgl. auch <https://godotengine.org/article/godot-webrtc-report3>, Stand 12.05.2021

4 Konzept

Dieses Kapitel befasst sich mit dem Konzept hinter 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 – 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 voraus. 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 Teilmenngen unterteilt werden. Zwischen den Spielern in einer solchen Teilmenge müssen Signalisierungsdaten zum Verbindungsaufbau austauschbar sein. Spieler müssen diesen Teilmenngen beitreten, und die Teilmenngen 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 beitrifft oder betrügt.

4.1.2 Anforderungen – Clientseitige 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. Die Parameter dieser Rückruffunktionen sollen dabei die Daten des Events beinhalten.

4.1.3 Anforderungen – Brettspiel

Um den Ablauf von Brettspielen zu beschreiben, und diesen als Zustandsmaschine darzustellen, werden hier einige Begriffe definiert, welche im Weiteren verwendet werden:

- Ein *Spieler* ist eine agierende Person, welche am Spiel teilnimmt. An einem Spiel nehmen stets N ($1 \leq N \leq M$) Spieler teil. Im Falle des Spiels *Mensch ärgere Dich nicht* gilt $M = 4 \geq N$.
- Eine *Runde* ist ein Ablauf von N *Zügen*, wobei jeder *Spieler* je einmal *am Zug* ist. War jeder Spieler je einmal *am Zug*, so wird eine neue Runde gestartet.
- Ein *Zug* ist ein Ablauf an Aktionen eines Spielers. Der *Zug* beginnt, wenn ein Spieler die Kontrolle über das Spiel zugewiesen bekommt. Der *Zug* endet entweder, falls der Spieler aufgrund der Spielregeln keine valide Modifikation am Spielstand vornehmen darf, oder mit der Modifikation des Spielstands. Gewinnt der Spieler *am Zug* das Spiel, so wird dieser in den folgenden Runden übersprungen. Haben alle Spieler gewonnen, so wird das Spiel beendet. Falls nicht wird die Kontrolle an den nächsten Spieler übergeben, welcher dann wiederum *am Zug* ist.

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

Jeder Spieler besitzt vier Spielfiguren. Das Spiel wird auf einem Spielbrett gespielt. Auf diesem existieren jeweils 16 A-Felder, 16 B-Felder, sowie 40 weiße Felder. Dabei ist das erste Feld das gelbe, das zehnte Feld das grüne, das zwanzigste Feld das rote, und das dreißigste Feld das schwarze Startfeld (vgl. Abbildung 4.1).

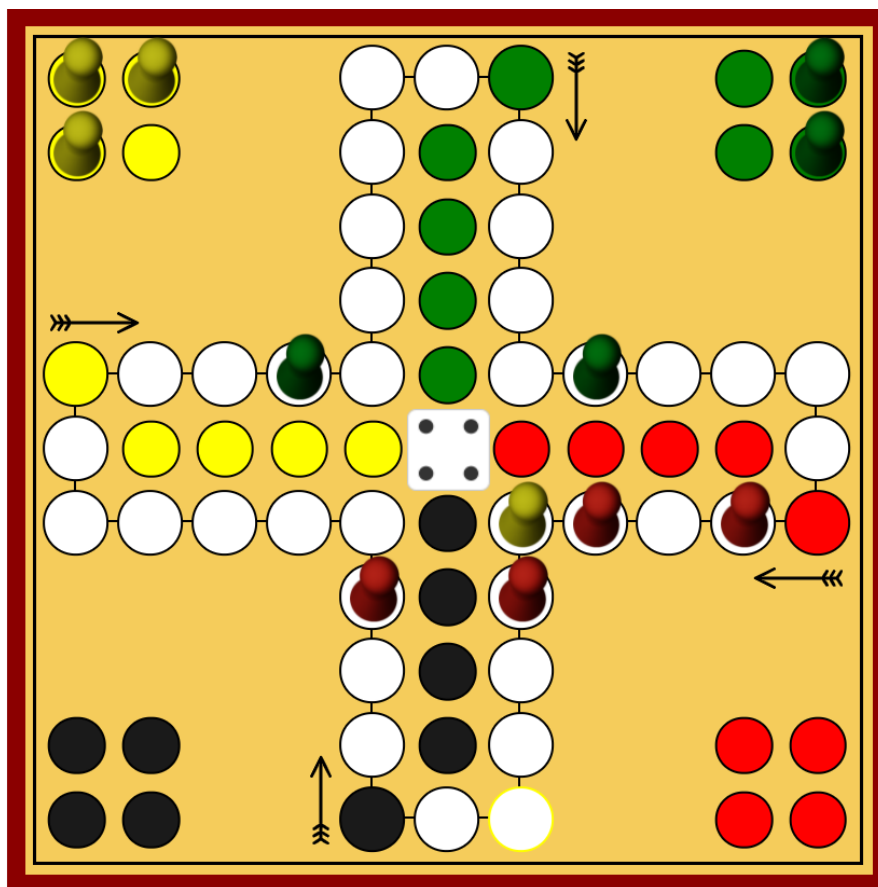


Abbildung 4.1: Laufendes Spiel mit drei Spielern. Gelb ist am Zug. Die A-Felder sind die farbigen Felder am Rand des Spielbretts, die B-Felder sind die farbigen Felder in der Mitte des Spielbretts.

Bei Spielstart stehen alle Figuren auf den A-Feldern der jeweiligen Spieler. Ein Spieler hat gewonnen, wenn alle Figuren des Spielers auf dessen B-Feldern stehen. Das Spiel ist in Runden aufgeteilt, in jeder Runde ist jeder Spieler einmal am Zug. Jeder Spielzug beginnt mit dem Würfeln einer Zahl im Intervall $[1,6]$. Falls der Spieler eine sechs würfelt, so muss dieser, falls mindestens eine Figur in den A-Feldern ist, und das Startfeld des Spielers frei von eigenen Figuren ist, mit einer Figur die A-Felder verlassen. Die verlassende Figur wird auf

das Startfeld des jeweiligen Spielers gestellt. Stehen alle Figuren eines Spielers bei Zugbeginn in den A-Feldern, so darf dieser bis zu drei mal würfeln, bis eine Sechs gewürfelt wurde. Würfelt ein Spieler eine Sechs, so darf dieser nach Wahl seines Spielzugs erneut würfeln. Nachdem ein Spieler seinen Zug beendet, ist der nächste Spieler im Uhrzeigersinn am Zug.

Landet eine Figur nach einem Spielzug auf einem Feld, wo bereits eine Figur eines anderen Spielers steht, so wird diese Figur „geworfen“, und muss in die A-Felder des Spielers zurück gesetzt werden.

Nachdem eine Figur – relativ zum Startfeld eines Spielers – 40 Felder weit bewegt wurde, kann diese die B-Felder eines Spielers betreten. In den B-Feldern dürfen Figuren nicht übereinander hinweg springen, und Würfe, welche die Figur über das letzte B-Feld hinaus bewegen würden, sind invalide. Nur der Spieler, welchem die B-Felder gehören, kann diese betreten.

Hat ein Spieler alle Figuren in dessen B-Felder bewegt, so können die weiteren Spieler das Spiel weiter zu Ende spielen. Gewonnene Spieler werden in Spielrunden ignoriert.

4.2 Konzept – Serverinfrastruktur

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. Die konzipierte Netzwerkstruktur ist in Abbildung 4.2 dargestellt. Neben der beschriebenen Signalisierungsebene müssen STUN- und TURN-Server gegeben sein. Beide diese Server sollen prototypisch in einem Server zusammengefasst sein. Dazu soll der „coturn“ STUN- und TURN-Server verwendet werden.

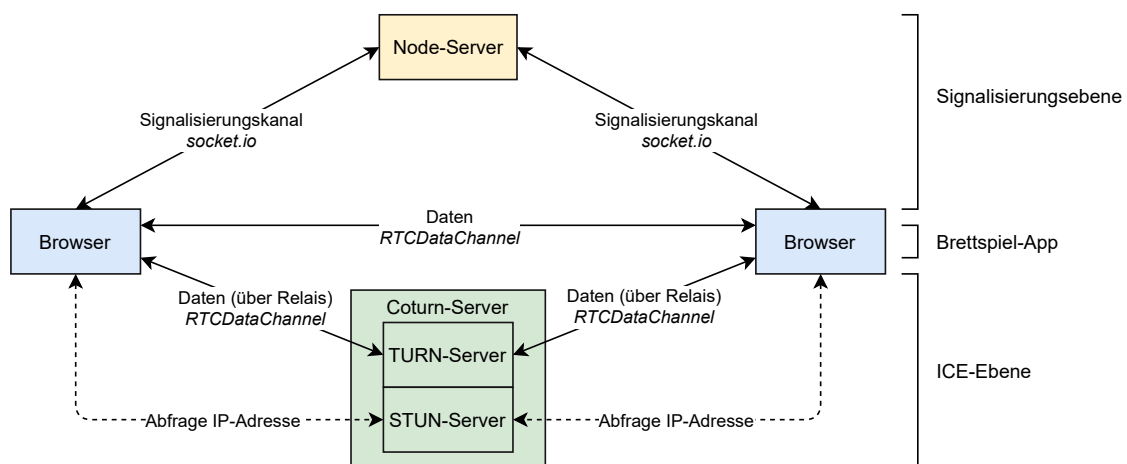


Abbildung 4.2: Konzept der Netzwerkinfrastruktur.

4.3 Konzept – Peer-To-Peer Netzwerkarchitektur

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

Das Authoritative-Peer-Modell folgt dem Authoritative-Server-Modell. Die Funktionen des Servers werden dabei von einem der Peers übernommen. Es wird eine Stern-Netzwerktopographie erzeugt, wobei der sogenannte „Host-Peer“ den zentralen Knoten bildet. Bei n Peers existieren insgesamt $n-1$ Verbindungen, der Host muss diese zu allen weiteren Peers aufrecht erhalten. Sämtlicher Datenaustausch läuft über den Host-Peer, welcher zudem die volle Autorität 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. Insgesamt existieren bei n Peers also $n(n-1)$ Verbindungen im Netzwerk. 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 voraussetzt.

4.4 Konzept – Brettspiel

Das Brettspiel soll via HTML5 und JavaScript im Browser implementiert werden. Die Darstellung des Spiels erfolgt mit stilisierten DOM-Elementen. Die Qualität der Darstellungs des Spiels ist hier nicht relevant, weshalb kein Frontend-Framework verwendet wird.

4.4.1 Zufallszahlen

Ist die Netzwerktopographie eines Brettspiels ein Authoritative-Server-Modell, so ist es leicht, faire und 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 [AS09].

Verteiltes Generieren von Zufallszahlen

Ein naiver Ansatz zur Generation von Zufallszahlen ist, jeden Peer n eine Zahl z_n bis zu einem Wert max generieren zu lassen, welche zwischen den Peers ausgetauscht werden. Anschließend ergibt die folgende Summe bei n Peers eine Zahl, welche – vorausgesetzt mindestens einer der Peers hat eine zufällige Zahl generiert und nicht betrogen [AS09] – zufällig ist:

$$z = \sum_{n=1}^n z_n \mod n, \quad z, z_n \in [0, max]$$

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“ [NPVS07], unter Nutzung von Commitment-Verfahren [AS09]. 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 ohne weiteres möglich, auf alle weiteren Zahlen zu warten, ohne vorher eine eigene Zahl erstellt zu haben [AS09].

Seeded Random-Number-Generators

Eine andere Methode zur Generierung von fairen Zufallszahlen in verteilten Systemen wie Peer-Netzwerken sind „Seeded-Random-Number-Generators“ [AS09]¹. Dabei lässt sich ein Zufallszahlengenerator mit einem Startwert, dem sogenannten „Seed“, initialisieren. Der Generator erzeugt basierend auf diesem Seed eine Folge an Pseudozufallszahlen, welche unter 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. Aus diesem Grund wurde dieser Ansatz für die Implementierung gewählt.

¹vgl. Sektion 1.2 und verwandte Werke

5 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.

In den folgenden Kapiteln werden Ausschnitte des Quellcodes, insbesondere der Dateien Peer.js und Server.js gelistet. Um diese besser in das Gesamtbild einordnen zu können, ist der volle Quellcode dieser Dateien in Anhang A und B zu finden.

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-, Plattform- und Infrastructure-As-A-Service (SaaS, PaaS, IaaS) anbietet.

Azure ermöglicht das Erstellen von Rechenressourcen 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 genutzten 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 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.

5.2.1 Webserver

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 der Node-Webanwendung

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 Pakete via

```
$ npm install socket.io  
$ npm install express  
$ sudo npm install -g pm2
```

heruntergeladen werden. Daraufhin können die *socket.io* und *express.io* Pakete über die *require*-Funktion in ein Script eingebunden werden. PM2 muss mit dem Parameter *-g* global installiert werden, damit die Kommandozeilenfunktionalität des Prozessmanagers verwendbar ist. Dazu sind Superuser-Rechte notwendig.

express

Zur Erstellung des Webserver wird das „express“-Framework verwendet. Der Quellcode befindet sich in der „Server.js“-Datei, und ist in Abbildung 5.1 abgebildet.

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 notwendigen 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 der Inhalt des „public“-Ordners statisch zur Verfügung gestellt.

```
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

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.

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   [...] // Rückruffunktionen
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 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 Datenaustausch 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 Signalnachricht besitzt immer eine Quell-Peer-ID, eine Ziel-Peer-ID, einen Signaltypen, und die eigentlichen SDP-Daten des Signals. Dabei wird zwischen den Signal-Typen *offer*, *answer* und *ice-candidate* unterschieden. Die Signalnachrichten werden in JavaScript Object Notation (JSON) encodiert und über den Signalisierungskanal weitergeleitet (siehe: 5.2.4 Signalisierungskanal).

```

1 {
2   source : <Peer-ID>,
3   target : <Peer-ID>,
4   type   : 'offer' | 'answer' | 'ice-candidate',
5   data   : <SDP-Daten>
6 }

```

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.
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.

```

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       connection.setRemoteDescription(e.data).then(() => {
7         return connection.createAnswer();
8       }).then((answer) => {
9         this.signal(this._createSignal('answer', answer, e.src));
10        connection.setLocalDescription(answer);
11      });
12      break;
13     case 'answer':
14       this.connections[e.src].setRemoteDescription(e.data);
15       break;
16     case 'ice-candidate':
17       this.connections[e.src].addIceCandidate(e.data);
18       break;
19   }
20 }

```

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

5.2.3 Raumverwaltung

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. Ein Raum-Objekt wird erzeugt, welches die zuvor beschriebenen Daten enthält. Ist der Raum erstellt, so wird ein Event an den erstellenden Client zurückgeschickt, um diesem mitzuteilen, dass der Raum nun beitreterbar 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 zum Host ernannt. Bei erfolgreichem Beitritt erhält der Nutzer das *game-room-joined*-Event zurück, welches sämtliche, zum Verbindungsaufbau mit den weiteren Spielern, benötigten Da-

ten 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 22 – alle weiteren Spieler des Raums benachrichtigt, dass ein weiterer Spieler beigetreten ist.

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

Tabelle 5.3: Spielerfarben.

In den 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 6 wird dieses Array durchlaufen. Ist ein Array-Index nicht definiert, so wird der neu beigetretene Spieler an diese Stelle des Spieler-Arrays gesetzt.

```

1 io.sockets.on('connection', (socket) => {
2   const PLAYER_SLOT_PRIORITY = [0, 2, 1, 3];
3   socket.on('game-room-join', (roomID) => {
4     const room = rooms[roomID];
5     [...]
6     for (let i = 0; i < 4; i++) {
7       if (!room.players[PLAYER_SLOT_PRIORITY[i]]) {
8         const peerID = utils.uuid4();
9         const color = PLAYER_SLOT_PRIORITY[i];
10
11         socket.join(roomID);
12         sockets[peerID] = socket.id;
13         room.players[color] = {peerID : peerID, color : color};
14         room.seed = Math.random();
15
16         if (!room.host) {
17           room.host = socket.id;
18         }
19
20         socket.emit('game-room-joined', room.players, room.started, room.seed, peerID,
21           utils.generateTURNCredentials(socket.id), room.host === socket.id);
22         socket.to(roomID).emit('game-room-client-joining', room.seed, peerID, color);
23         break;
24       }
25     }
26   });
27   [...];
28 });

```

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

Dazu wird in Zeile 8 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).

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. 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 6–9 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 – vorausgesetzt 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

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.

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.
game-room-kick-player	Kann vom Host ausgelöst werden, entfernt einen Spieler aus dem Raum. Wird an alle verbleibenden Spieler weitergeleitet.

Tabelle 5.4: Weitere Raumspezifische Events.

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. 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

Zur Signalisierung muss immer die ID des Raums angegeben werden, in welchem die Signalmeldung 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.

5.2.5 Bereitstellung des Webservers

Der Webserver kann über den *node*-Befehl gestartet werden. Dies ist jedoch nicht für permanente Bereitstellung geeignet, da der Server bei Neustart oder Absturz der VM nicht automatisch neu gestartet wird. Daher wird der Prozess-Daemon-Manager „PM2“ (Process Manager 2) verwendet. Eine Applikation kann über den Befehl

```
$ pm2 start Server.js --watch
```

gestartet werden. Der *watch* Parameter lädt das Script erneut, falls eine Änderung an der Datei vorgenommen wurde. Dies ist insbesondere bei der Entwicklung hilfreich.

Auf Ubuntu 16+ nutzt PM2 *systemd* zur internen Verwaltung der Prozesse. Um das Programm bei Neustart des Servers auszuführen, muss ein entsprechendes „Startup-Script“ erstellt werden. PM2 bietet dazu den *startup*-Befehl:

```
$ pm2 startup
```

Dieser Befehl generiert einen Kommandozeilenbefehl zum Erstellen einer Start-Konfiguration für das genutzte Betriebssystem, in diesem Fall:

```
$ sudo env PATH=$PATH:/usr/bin /usr/local/lib/node_modules/pm2/bin/pm2 startup  
systemd -u robin --hp /home/robin
```

Nachdem dieser Befehl ausgeführt wurde, muss die momentane Prozess-Konfiguration in die Start-Konfiguration geschrieben werden. Der Befehl

```
$ pm2 save
```

schreibt die Konfiguration aller momentan laufenden PM2-Services in das Start-Script. Bei Neustart des Servers, oder Absturz der Anwendung, wird die Webanwendung nun automatisch als System-Daemon neu gestartet.

5.3 STUN- und TURN-Server

Es existieren verschiedene Quelloffene STUN- und TURN-Server. Ein bekannter, häufig verwendeter Server ist „coturn“. Coturn unterstützt sowohl die wichtigsten STUN- (RFC3489, RFC5389), als auch TURN-Spezifikationen (RFC5766). Zudem unterstützt Coturn die Authentifizierungsmaßnahmen der „TURN-REST-API“, welche das Erstellen von zeitlich limitierten Login-Daten zur Authentifizierung von Clients – ohne die Notwendigkeit von Nutzer-Datenbanken – ermöglicht [Ube13]. Zur Relais-Datenübertragung unterstützt Coturn sowohl TCP, als auch UDP. Coturn ist zudem in den Repositories der verwendeten Linux-Distribution enthalten, was eine Installation erheblich vereinfacht.

5.3.1 Installation

Auf Ubuntu Version 18.04+ lässt sich der Server einfach über den Paketmanager *apt* via der Kommandozeile installieren:

```
$ apt-get install coturn
```

Um den Server nach einem Systemneustart automatisch als System-Daemon zu starten, muss in der Datei */etc/default/coturn* die folgende Zeile geschrieben werden:

```
TURN_SERVER_ENABLED=1
```

Der Server kann nun über den System- und Servicemanager *systemd* gestartet, gestoppt oder neu gestartet werden:

```
$ systemd <start|stop|restart> coturn
```

5.3.2 Konfiguration

Die Server lässt sich über die Konfigurationsdatei */etc/turnserver.conf* konfigurieren. Die Konfiguration folgt einem Schlüssel-Wert Schema, getrennt durch ein Gleichheitszeichen. Der (gekürzte) Inhalt der Konfigurationsdatei ist in Abbildung 5.15 gelistet. Die Einstellungen werden in Tabelle 5.5 näher erläutert.

Einstellung	Beschreibung
listening-port	Der Port, auf welchem der Server auf Anfragen reagieren soll.
listening-ip	Ist dieser Wert gleich 0.0.0.0, so werden alle im System definierten IP-Adressen als Listening-Adressen verwendet.
external-ip	Die Externe IP-Adresse des Servers. Der Server befindet sich in einem Subnetz, daher wird auch die interne IP-Adresse benötigt.
min-port / max-port	Die Port-Reichweite, welche für Relais-Adressen verwendet werden kann.
use-auth-secret	Der Server soll die zeitlimitierte Authorisierungsmethode mit statischem Schlüsselwort verwenden (siehe: Authentifizierung).
static-auth-secret	Das statische Schlüsselwort zur Generierung von Passwörtern zur zeitlich limitierten Nutzung des Servers.
realm	Ein TURN-Server unterstützt verschiedene „Realms“, um Anfragen verschiedener Ursprünge untereinander zu isolieren. In diesem Fall wird nur ein „Realm“ benötigt. Der Wert dieser Konfiguration ist bei Verwendung der TURN-REST-API jedoch nicht relevant, und kann einen beliebigen Wert enthalten [Ube13].
stale-nonce	Ein Wert von 0 bewirkt, dass die Authentifizierung einer Sitzung nicht automatisch nach einiger Zeit ausläuft.

Tabelle 5.5: Erläuterung der Konfigurationsparameter.

```
listening-port=3478
listening-ip=0.0.0.0
external-ip=<external-ip>/<internal-ip>

min-port=10000
max-port=20000

use-auth-secret
static-auth-secret=<my-auth-secret>
realm=ba-webrtc.westeurope.cloudapp.azure.com

stale-nonce=0
```

Quellcode 5.15: Coturn-Konfigurationsdatei – turnserver.conf

5.3.3 Authentifizierung

Coturn bietet verschiedene Möglichkeiten, Nutzer zu authentifizieren, um eine Nutzung des Servers von Dritten, außerhalb des vorgesehenen Einsatzzwecks, zu limitieren:

- **Statische Nutzerdaten:** In der Konfigurationsdatei können Nutzer-Passwort-Tupel statisch definiert werden. Diese Art der Authentifizierung ist einfach zu implementieren, allerdings in Situationen, in welchen die Nutzer die Zugangsdaten einfach auslesen können, ungeeignet.
- **Dynamische Nutzerdaten:** Coturn kann Nutzerdaten aus einer Datenbank auslesen. In diese Datenbank können Nutzerdaten dynamisch eingefügt, oder entfernt werden.
- **TURN-REST-API:** Die TURN-REST-API verwendet geheime Schlüssel, welche zwischen einem Webserver und dem TURN-Server geteilt werden müssen. Der Webserver kann, basierend auf dem geheimen Schlüssel, dynamische, zeitlich limitierte Zugangsdaten generieren [Ube13]. Der Schlüssel kann statisch in der Konfigurationsdatei (*static-auth-secret*) angegeben werden. Zudem ist es möglich, eine Reihe an Schlüsseln dynamisch in einer Datenbank zu definieren.

Für die Implementierung wird die TURN-REST-API verwendet, da diese kein Aufsetzen einer Datenbank voraussetzt. Zudem eignen sich zeitlich limitierte Zugangsdaten hier gut, da die Anwendung einen praktisch öffentlichen Service bereitstellt, welcher von jedem Nutzer, welcher die Website aufruft, verwendbar sein soll. Da dafür kein Log-In notwendig sein soll, ist es nicht notwendig Nutzerdatenbanken mit personalisierten Zugangsdaten für jeden Nutzer zu erstellen.

Zugangsdaten

Zugangsdaten bestehen immer aus einem Nutzernamen und einem Passwort. Der Nutzername setzt sich aus einer Nutzer-ID und einem Timestamp zusammen [Ube13]. Die Form des Nutzernamen ist wie folgt vorgegeben:

```
timestamp:nutzerid
```

Der Timestamp ist der Ablaufzeitpunkt der Zugangsdaten, die Nutzer-ID kann beliebig gewählt werden, oder leer sein. Das Passwort ist der Hash-Based Message Authentication Code (HMAC) aus dem Nutzernamen in der oben beschriebenen Form und dem geheimen Schlüssel. Als Hash-Funktion wird der Secure Hash Algorithm 1 (SHA-1) verwendet. Zuletzt wird das Passwort in *base64* encodiert. Ein Passwort lässt sich somit folgendermaßen generieren:

```
base_64(hash_hmac('sha1', nutzername, schluessel));
```

Implementierung auf dem Webserver

Die im vorherigen Unterpunkt beschriebene Generierung von Zugangsdaten wird vom Webserver übernommen. Tritt ein Spieler einem Raum bei und möchte sich unter Nutzung des STUN- und TURN-Servers mit weiteren Peers verbinden, so werden für diesen Nutzer spezifische, zeitlimitierte Zugangsdaten generiert.

```
1 const config = require('./config.json');
2 const crypto = require('crypto');
3 [...]
4 generateTURNCredentials : (user) => {
5   const timestamp = Date.now() + config.ice.staticAuthCredentialLifetime;
6   const username = `${timestamp}:${user}`;
7   const hmac = crypto.createHmac('sha1', config.ice.staticAuthSecret);
8   hmac.setEncoding('base64');
9   hmac.write(username);
10  hmac.end();
11  const password = hmac.read();
12
13  return [
14    { urls: config.ice.stunServerAddress },
15    { urls: config.ice.turnServerAddress, username: username, credential: password }
16  ];
17 },
18 }
```

Quellcode 5.16: Funktion zum Erstellen von Zugangsdaten – utils.js

Die in Abbildung 5.16 dargestellte Funktion generiert direkt ein Array an ICE-Servern mit zugehörigen Zugangsdaten, welche an den Client übertragen werden können. Der Nutzername (*user*) ist dabei die ID des Client-Sockets, die Lebenszeit der Zugangsdaten ist in der Konfigurationsdatei des Webserver definiert. Der geheime Schlüssel (*config.ice.staticAuthSecret*) muss mit dem in der Coturn-Konfigurationsdatei gesetzten Wert übereinstimmen.

5.3.4 Ports

Zuletzt müssen noch alle benötigten Ports freigegeben werden. Dazu kann das *ufw* (*Uncomplicated Firewall*)-Werkzeug verwendet werden:

```
$ sudo ufw enable
$ sudo ufw allow <port>/<protocol>
```

Alle zu öffnenden Ports sind in Tabelle 5.6 gelistet.

Port	Protokoll	Beschreibung
3000	TCP	Port des Webservers.
3478	TCP/UDP	Listening-Port des STUN- und TURN-Servers.
10000-20000	UDP	Ports für Relais-Adressen des TURN-Servers.

Tabelle 5.6: Portweiterleitung.

Die Ports müssen zudem für die VM über das Azure-Webportal freigegeben werden. Dazu muss für jeden der Ports für die Ressource der VM im Unterpunkt *Netzwerk* der Einstellungen eine Regel für eingehende Ports hinzugefügt werden (vgl. Abbildung 5.1).

Priorität	Name	Port	Protokoll	Quelle	Zieladresse	Aktion
300	HTTP	80	TCP	Alle	Alle	Zulassen
320	SSH	22	TCP	Alle	Alle	Zulassen
340	HTTPS	443	TCP	Alle	Alle	Zulassen
350	CoTurnListeningPort	3478	Alle	Alle	Alle	Zulassen
360	CoTurnBindingPorts	10000-20000	Alle	Alle	Alle	Zulassen
380	Server3000	3000	Alle	Alle	Alle	Zulassen
65000	AllowVnetInBound	Alle	Alle	VirtualNetwork	VirtualNetwork	Zulassen
65001	AllowAzureLoadBalancerInBound	Alle	Alle	AzureLoadBalancer	Alle	Zulassen
65500	DenyAllInBound	Alle	Alle	Alle	Alle	Ablehnen

Abbildung 5.1: Port-Regeln der VM im Azure-Webportal.

5.3.5 Test

Um zu überprüfen, ob die STUN- und TURN-Server wie gewünscht funktionieren, müssen die vom ICE-Agenten einer *RTCPeerConnection* gefundenen ICE-Kandidaten geprüft werden. Wird eine neu erstellte *RTCPeerConnection* mit dem ICE-Server-Array konfiguriert, so werden bei Verbindungsaufbau die Folgenden ICE-Kandidaten (mit Ausnahme des *host*-Kandidaten) gefunden:

```
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 5.17: Relais-Kandidat

```
1 candidate:842163049 1 udp 1677729535 (oeffentliche IP-Adresse) 55948 typ srflx raddr  
0.0.0.0 rport 0 generation 0 ufrag QzUf network-cost 999
```

Quellcode 5.18: Server-Reflexiver-Kandidat

Die Transportadresse der Kandidaten ist jeweils in Blau, der Typ in Rot markiert. Beide Kandidaten werden gefunden. Der Port des Relais-Kandidaten liegt zwischen 10000 und 20000. Dies bestätigt, dass sowohl der STUN-, als auch der TURN-Server ansprechbar sind, und die Zugangsdaten korrekt sind.

5.4 Implementation des Brettspiels

Das Brettspiel ist unter Verwendung von Hypertext Markup Language 5 (HTML5) und JavaScript-Modulen entwickelt. Zum Datenaustausch wird der vom Peer-Objekt bereitgestellte Datenkanal verwendet. Jeder Spieler besitzt eine eigene Kopie des Spielstands, welcher zwischen den Peers synchron gehalten werden muss. Zu diesem Spielstand gehören die in Abbildung 5.19 aufgelisteten Parameter, mit zugehörigen Standardwerten bei Initialisierung des Spiels.

```
1 export default function Game([...]) {  
2   [...]  
3   this.gamestate = {  
4     fsm : 'start',           // Zustand des Spiels (start|roll|move|end)  
5     pieces : [],             // Positionen der Spielfiguren  
6     current : -1,            // Farbe des Spielers am Zug  
7     lastRoll : -1,           // Zuletzt geworfene Zahl  
8     currentRolls : -1,       // Anzahl an Wuerfen im momentanen Zug  
9     canRollAgain : false     // Kann der Spieler am Zug erneut wuerfeln?  
10  };  
11  [...]  
12 }
```

Quellcode 5.19: Spielstandsdaten – Game.js

Synchronisation des Spielstands bei Spielbeitritt

Ist das Spiel gestartet, so weicht der Spielstand von den in Abbildung 5.19 gelisteten Standardwerten ab. In diesem Fall muss ein neu beitretender Spieler den gesamten Spielstand von einem weiteren Spieler zugeschickt bekommen. Dies wird immer vom Host des Spielraums übernommen, wenn der Datenkanal zum neuen Spieler geöffnet ist (vgl. Quellcode 5.20). Dazu wird das *gamestate*-Event verwendet.

```

1 // aufgerufen wenn alle (neuen) Datenkanäle offen sind
2 peer.on('dataChannelsOpen', () => {
3   if (isHost && game.gamestate.fsm !== 'start') {
4     peer.emit(peerID, 'gamestate', game.gamestate);
5   }
6   [...]
7 });

```

Quellcode 5.20: Senden des Spielstands – game.js

Auf der Empfängerseite muss der Spielstand entsprechend aktualisiert, und die Figuren an deren jeweiligen Positionen bewegt werden.

5.4.1 Spielablauf

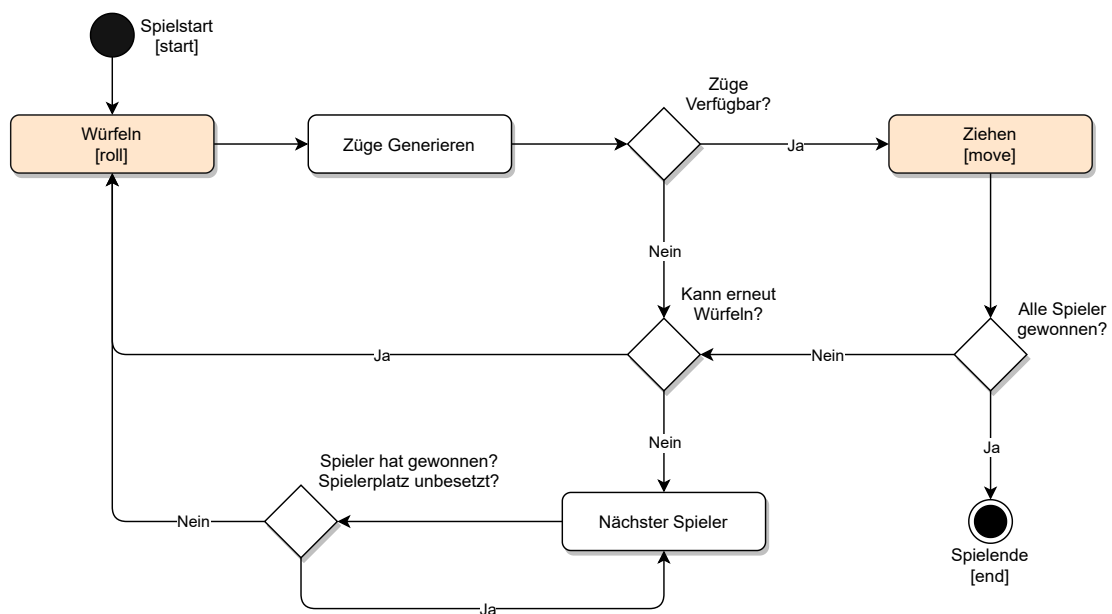


Abbildung 5.2: Ablaufdiagramm des Brettspiels.

Das Spiel lässt sich als Zustandsmaschine modellieren. Ein vereinfachtes Ablaufdiagramm des Spielverlaufs ist in Abbildung 5.2 dargestellt. Die „Zustände“ sind dabei in eckigen Klammern definiert. *Kann erneut Würfeln* ist keine einzelne Kondition, sondern eine Ansammlung verschiedener Konditionen: Ein Spieler kann erneut würfeln, falls sich alle dessen Figuren in den A-Feldern befinden, und dieser in diesem Zug unter drei mal gewürfelt hat. Ein Spieler darf zudem immer neu würfeln, falls dieser eine sechs geworfen hat.

Die in orange hervorgehobenen Zustände „Würfeln“ und „Ziehen“ sind von besonderer Bedeutung, da an diesen Punkten ein Datenaustausch zwischen Spielern notwendig ist. Die ausgeführte Logik unterscheidet sich hier basierend darauf, ob ein Spieler am Zug ist oder nicht.

Würfeln

JavaScript bietet standardmäßig keine Möglichkeit, einen *Seeded-Random-Number-Generator* zu erstellen. Aus diesem Grund wird die „seedrandom“-Bibliothek eingebunden¹. Die Bibliothek stellt die *Math.seedrandom*-Funktion bereit, welche einen Pseudozufallszahl-Generator erstellt. Standardmäßig nutzt dieser die Stromverschlüsselung Rivest Cipher 4 (RC4) zum Generieren von Zufallszahlen [Bau10].

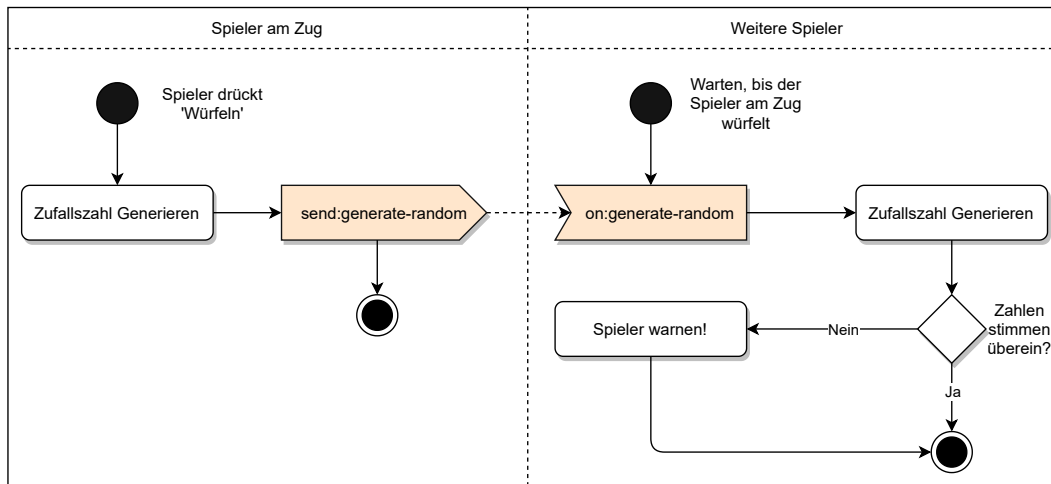


Abbildung 5.3: Würfel-Nachrichtenaustausch.

Würfelt ein Spieler, so wird über den *Seeded-Random-Number-Generator* eine Pseudozufallszahl generiert. Damit die Zufallszahlgeneratoren der weiteren Spieler synchron sind, muss diese Aktion den weiteren Spielern mitgeteilt werden. Dazu wird das *generate-random*-Event an alle weiteren Peers gesendet. Die Parameter dieses Events enthalten die Zufallszahl des Spielers am Zug. Erhält ein weiterer Spieler das Event, so generiert dieser eine eigene Zufallszahl, welche – falls der Spieler am Zug nicht betrügt, und die Zufallszahlgeneratoren der Spieler synchron sind – gleich der vom Spieler am Zug generierten Zufallszahl sein sollte (vgl. Abbildung 5.3).

Mit der gewürfelten Zahl wird daraufhin ein Array mit allen möglichen Spielzügen generiert, welche der Spieler am Zug basierend auf der gewürfelten Zahl ausführen kann. Ein Zug setzt sich aus einer Start- und Zielposition zusammen:

```

move = {
  from : <relative-position>,
  to   : <relative-position>
}
  
```

Die Start- und Zielposition eines Zugs ist dabei immer relativ zum Start-Feld des Spielers am Zug. Sowohl der Spieler am Zug, als auch alle weiteren Spieler generieren das Array an Spielzügen.

¹seedrandom: <https://github.com/davidbau/seedrandom>, Stand 12.05.2021

Ziehen

Der Spieler am Zug kann durch Klicken auf eine Spielfigur einen Zug wählen. Existiert dieser im Zug-Array, so wird der gewählte Zug den weiteren Spielern per Event über die P2P-Verbindung mitgeteilt. Die weiteren Spieler prüfen, ob dieser Zug in deren eigenen Zug-Arrays existiert. Existiert der Zug nicht, so wird davon ausgegangen, dass der Spieler am Zug betrügt, und eine Warnung wird ausgegeben. Hier unterscheidet sich die Logik also wieder zwischen dem Spieler am Zug und weiteren Spielern (vgl. Abbildung 5.4).

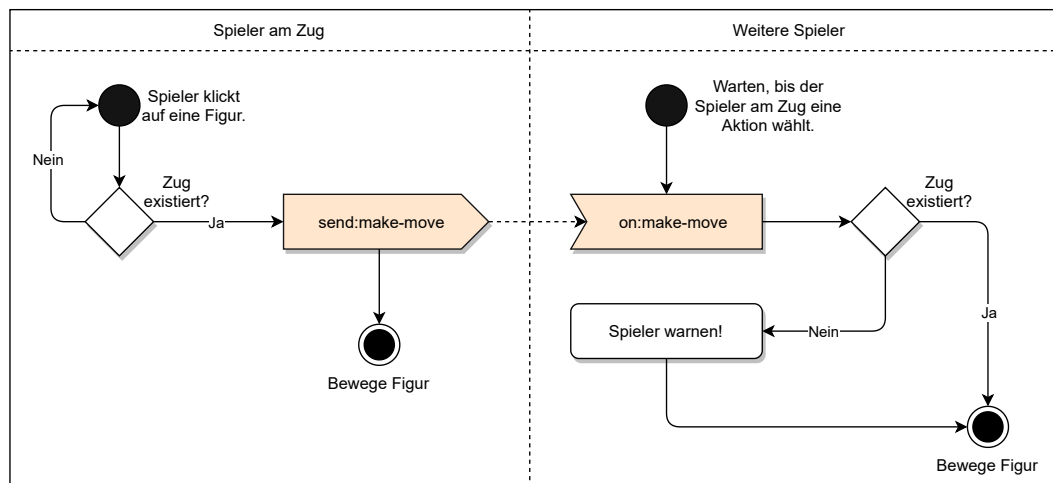


Abbildung 5.4: Ziehen-Nachrichtenaustausch.

5.4.2 Darstellung des Spielbretts

Das Spielbrett wird aus stilisierten Document Object Model (DOM)-Elementen generiert. Diese Elemente werden auf einem 11x11-Raster positioniert (vgl. Abbildung 5.5). Bei den Spielfiguren handelt es sich ebenfalls um DOM-Elemente, welche dynamisch, basierend auf dem Spielstand, an verschiedene Feld-Elemente angefügt werden. Um die Spiellogik universell für jeden Spieler anwenden zu können, besitzen die Spielerspezifischen A- und B-Felder „relative“ Positionen ($\langle \text{Farb-ID} \rangle : \langle \text{Position} \rangle$). Diese Positionen sind relativ zu dem Start-Feld der jeweiligen Spielerfarbe. Die „weißen“ Felder (0-39) besitzen lediglich eine „absolute“ Position. Das letztendlich generierte Spielbrett ist in Abbildung 4.1 dargestellt.

$\begin{smallmatrix} x \\ y \end{smallmatrix}$	0	1	2	3	4	5	6	7	8	9	10
0	0:-1	0:-2			8	9	10			1:-1	1:-2
1	0:-3	0:-4			7	1:40	11			1:-3	1:-4
2					6	1:41	12				
3					5	1:42	13				
4	0	1	2	3	4	1:43	14	15	16	17	18
5	39	0:40	0:41	0:42	0:43		2:43	2:42	2:41	2:40	19
6	38	37	36	35	34	3:43	24	23	22	21	20
7					33	3:42	25				
8					32	3:41	26				
9	3:-1	3:-2			31	3:40	27			2:-1	2:-2
10	3:-3	3:-4			30	29	28			2:-3	2:-4

Abbildung 5.5: Schemenhafte Darstellung des Spielbretts, mit Positionen und Farben.

6 Evaluation

Im Folgenden werden die aufgetretenen Probleme bei der Implementierung des Spiels diskutiert, und wie diese (nicht) gelöst wurden.

6.1 Probleme der Implementation

Die meisten Probleme der Implementation beziehen sich auf den potentiellen Betrug von Spielern. Dies ist dem Umstand geschuldet, dass das Spiel auf einem Webbrowser basiert. Im Gegensatz zu „traditionellen“ Applikationen, welche kompilierten Quellcode nutzen, ist es in einem Webbrowser leicht, die JavaScript-Dateien bei Laufzeit zu editieren. Zudem kann ein Nutzer über die Browserkonsole leicht eigene Scripts in die Webseite – und somit auch das Spiel – einspeisen.

Dabei lässt sich zwischen drei möglichen Betrugsaspekten unterscheiden: Vertraulichkeit, Integrität und Verfügbarkeit [NPVS07].

- **Vertraulichkeit:** Der Verteilte Spielstand macht es einem Spieler leicht, Spielinformationen auszulesen, welche dieser nicht auslesen sollte. Ein Beispiel für Betrug dieser Art ist das Auslesen von Karten weiterer Spieler bei einem Pokerspiel. Insbesondere in Browserbasierten Spielen kann ein Nutzer einfach den Quellcode ändern, und sich alle Informationen des Spiels ausgeben lassen. Beim Spiel *Mensch ärgere Dich nicht* ist dies jedoch, aufgrund von Mangel solcher Vertraulichen Informationen, nicht relevant.
- **Integrität:** Der Umstand, dass das Spiel in einem Webbrowser läuft, macht es Nutzern leicht, den Quellcode des Spiels zu ändern, und unerlaubte Änderungen am Spielstand vorzunehmen. Dem kann partiell durch feste Regeln, sowie einem festen Spielablauf vorgebeugt werden. In der Implementation generiert zum Beispiel jeder Spieler, nachdem der Spieler am Zug würfelte, alle möglichen Züge. Wählt der Spieler am Zug seine Aktion, so wird diese von allen Peers nochmal eigens gegen deren Zug-Array geprüft. Zudem wird eine Aktion (Würfeln, Ziehen) eines Spielers nur akzeptiert, falls sich das Spiel im dafür richtigen Zustand befindet, und die Nachricht über die RTC-PeerConnection des Spielers am Zug gesendet wurde.
- **Verfügbarkeit:** Ein Spieler kann das Vorranschreiten des Spiels verhindern, indem dieser zum Beispiel keine Zug-Nachricht schickt, oder nicht würfelt. Diese Art des disruptiven Spielerverhaltens wird von der Implementation nicht verhindert. Ein Lösungsansatz für dieses Problem ist ein Zeitlimit für Spieleraktionen, welches basierend auf Timestamps der eingehenden Nachrichten synchronisiert wird.

Weiterhin war es aus zeitlichen Gründen nicht möglich, die Funktionalität des Brettspiels ausgiebig zu testen. Hier existieren eine Reihe an Problemen mit dem Beitritt und Verlassen von Räumen, während das Spiel bereits läuft. So kann es zum Beispiel passieren, dass der Spielstand, welchen ein beitretender Spieler erhält, bereits bei Erhalt veraltet ist. Hier müssen Systeme existieren, welche bei Beitritt eines Spielers das Spiel effektiv pausieren, oder Aktionen des Spielers am Zug speichern, und erst mit allen weiteren Spielern zu synchronisieren, wenn der neu beigetretene Spieler den Spielstand erhalten hat, und alle Verbindungen zu diesem offen sind.

Ein weiterer Verbesserungsansatz ist die Fehlererkennung und -Behebung. Fehler der Signalisierung, des Verbindungsaufbaus oder des Spiels an sich werden nur in der Konsole angezeigt. Hier müssen robuste Systeme existieren, welche bei Verbindungsfehlern einen erneuten Verbindungsversuch unternehmen. Auch sollte die Implementierung auf Datenübertragung via einem Server zurückfallen können, für den Fall, dass WebRTC-Verbindungen nicht verwendbar sind (zum Beispiel in veralteten Browsern). Diese Fallbacks können – aufgrund des Eventbasierten Nachrichtenprotokolls – direkt in das Peer-Objekt integriert werden. Die Nachrichten können in diesen Fällen zum Beispiel auch über die socket.io Verbindung ausgetauscht werden.

Des Weiteren ist die starke Kopplung zwischen Node-Server und Brettspiel problematisch. Da der Node-Server verwendet wird, um Spielern für das Spiel „Mensch Ärgere Dich Nicht“ spezifische Farben zuzuweisen, und den Seed der Zufallsfunktion zu generieren, ist der Server nicht ohne weitere Modifikationen für weitere Spiele verwendbar. Hier sollte zwischen der Signalisierung und spielspezifischer Serverfunktionalität getrennt werden, um die Signalisierungs- und Raummechanismen auch für andere Spiele verwenden zu können.

6.2 WebRTC für Browserbasierte Mehrspieler-Brettspiele

Im Folgenden werden die Verwendung von Client-Server-Modellen und WebRTC-Netzwerkarchitekturen für die Entwicklung von Browserbasierten Mehrspieler-Brettspielen verglichen. Dies geschieht auf Basis der folgenden Kriterien:

Kriterium	Beschreibung
Flexibilität	Wie flexibel sind die Netzwerkarchitekturen? Dabei ist vor allem die Flexibilität der Datenübertragung von Bedeutung, da diese den Anwendungsbereich der Technologie stark einschränken kann.
Aufwand	Wie aufwendig ist die Verwendung der Technologie? Welche Voraussetzungen müssen gegeben sein, damit diese verwendet werden kann?
Komplexität	Wie komplex fällt eine Netzwerkarchitektur aus? Wie viele Verbindungen müssen verwaltet werden, und wie viele Geräte/Server sind im Netzwerk?
Sicherheit	Auf welche Art kann ein Nutzer die Nutzererfahrung von anderen Nutzern negativ beeinflussen? Wie leicht fällt dies?
Skalierbarkeit	Inwiefern schränkt die Netzwerkstruktur die Skalierbarkeit des Spiels – bezogen auf die Anzahl der Spieler – ein?

Tabelle 6.1: Vergleichskriterien

Flexibilität

Aufgrund des SCTP-Protokolls gestaltet sich die Datenübertragung über RTCDataChannels sehr viel flexibler als bei Client-Server-Verbindungen. Diese sind meist auf entweder TCP, oder UDP beschränkt – WebSockets nutzen zum Beispiel ausschließlich TCP [FM11]. Die Möglichkeit, gleichzeitig sowohl UDP-artige, als auch TCP-artige Datenströme zu verwenden eröffnet zudem Möglichkeiten zur Integration von Echtzeit-Features in Rundenbasierten Brettspielen – zum Beispiel das Anheben und Verschieben von Figuren per Drag-and-Drop, wobei die Position der Spielfigur in Echtzeit mit weiteren Spielen über einen UDP-artigen Datenkanal synchronisiert wird.

Aufwand

Der Aufwand, welcher zum Aufbau einer WebRTC-Verbindung notwendig ist, ist verglichen mit Client-Server-Architekturen hoch. Während bei Client-Server-Architekturen lediglich ein Server existiert, zu welchem eine Verbindung eröffnet werden muss – wie zum Beispiel bei Verwendung von socket.io – so muss zum Sitzungsaufbau via WebRTC eine zusätzliche Signalisierungsebene geschaffen werden.

Zudem müssen STUN- und TURN-Server existieren, damit das ICE-Framework die eigentliche Verbindung aufbauen kann. Zudem macht der Umstand, dass jeder Spieler eine eige-

ne Kopie des Spielstands verwalten muss, die Implementation eines P2P-Spiels prinzipiell komplexer als bei einem Spiel, welches auf dem Authoritative-Server-Prinzip basiert.

Komplexität der Netzwerkarchitektur

Die Notwendigkeit der Signalisierungsebene bringt erhöhte Komplexität der Netzwerkarchitektur im Vergleich zu Client-Server-Modellen mit sich: Die zu verwaltende Anzahl an Verbindungen im Netzwerk ist höher als in Client-Server-Modellen, da neben der Verbindung zum Signalisierungskanal auch Verbindungen zwischen den Browsern untereinander verwaltet werden müssen. Bei n Clients müssen insgesamt $n + n(n-1)$ (P2P Mesh-Topologie), beziehungsweise $2n-1$ (P2P Stern-Topologie) Verbindungen verwaltet werden. Die Komplexität der Netzwerkarchitektur wird zudem durch die Notwendigkeit von STUN- und TURN-Servern erhöht. Diese müssen entweder proprietär aufgesetzt, oder via einer Infrastruktur-As-A-Service-Lösung wie *Xirsys*¹ bereitgestellt werden.

Sicherheit

Es ist leichter, in P2P-Spielen zu betrügen, als in Client-Server-Modellen, da keine zentrale, vertrauliche Instanz in Form eines Authoritative-Servers existiert. Im Falle von Browserbasierten Spielen ist dies jedoch von besonderem Nachteil, da der Quellcode des Spiels leicht zugänglich im Browser abrufbar ist. Dieser kann während der Laufzeit ausgelesen und editiert werden. Zwar kann dem teilweise mit Quellcodeminimierung und -Verschleierung vorgebeugt werden, jedoch ist es nicht möglich, den Quellcode und Spielstand komplett zu verbergen. Dies führt zu erhöhter Komplexität der Anwendung, welche diese Nachteile in Betracht ziehen muss. Insbesondere dieses Kriterium stellte eine große Hürde bei der Implementierung dar (vgl. 6.1 Probleme der Implementierung).

Skalierbarkeit

Peer-To-Peer Systeme sind in der Regel skalierbarer als Client-Server-Modelle [NPVS07]. WebRTC ist jedoch bei erhöhter Anzahl an Peers durch Upload-Bandbreite und Rechenressourcen der Peers limitiert. Insbesondere im Bereich der Videoübertragung ist dies bei WebRTC ein Problem, da bei hoher Teilnehmeranzahl in einer Mesh-Topologie jeder Teilnehmer gleichzeitig mehrere Videostreams encodieren, decodieren und versenden muss.

Bei Brettspielen ist dies jedoch weniger relevant, da weder CPU-intensive En- und Decodierung, noch hohe Datenraten benötigt werden. Bei der Implementierung dieser Arbeit werden lediglich zwei Nachrichten per Zug benötigt. Zwar kann die Nachrichtenlatenz bei erhöhter Teilnehmeranzahl steigen, bei Brettspielen spielt Latenz – aufgrund deren Rundenbasierten Art – nur eine untergeordnete Rolle. Die Implementierung hat gezeigt, dass Spiele mit bis zu vier Spielern problemlos möglich sind. Je nach Spiel kann die maximale Anzahl an Spielern jedoch variieren.

¹vgl. <https://xirsys.com/>, Stand 07.05.2021

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war das Aufsetzen einer prototypischen WebRTC-Netzwerkinfrastruktur, welche das Spielen eines Brettspiels ermöglicht. Zudem wurde die Anwendbarkeit von WebRTC für browserbasierte Spiele mit standardmäßig verwendeten Client-Server-Architekturen verglichen.

Rückblickend wurde mit der Implementierung das Ziel, ein Brettspiel mithilfe einer WebRTC-Netzwerkinfrastruktur zu implementieren, erreicht. Zudem wurden Ansätze zur Vermeidung von Betrug diskutiert. Diese sind jedoch in diesem Fall von geringerer Bedeutung als anfänglich geschätzt, da Spieler bei diesem Gesellschaftsspiel – welches in der Regel von untereinander bereits bekannten Personen, und nicht Fremden gespielt wird – keinen Anreiz haben, zu betrügen.

Insbesondere dieser Betrugsaspekt bietet einen Ansatz für zukünftige Arbeiten. Peer-To-Peer-Spiele sind generell anfälliger für Betrug als Spiele, welche einen zentralen Server zur Synchronisation der Spieldaten nutzen. Betrug wird weiterhin durch den – durch die Implementierung im Browser bedingten – offenliegenden Quellcode erleichtert. Hier müssen existierende, weit verbreitete Anti-Betrugsmaßnahmen für P2P-Spiele – wie zum Beispiel das Lockstep-Protokoll oder Commitment-Verfahren – mitunter erneut evaluiert werden. Ein weiterer Ansatz ist die Konzeption, das Design und die Implementierung eines vollends betrugssicheren P2P-Spiels im Browser, welches auf der in dieser Arbeit implementierten WebRTC-Netzwerkstruktur basiert.

Ein weiterer Ansatz für zukünftige Arbeit ist die Verwendung von hybriden Netzwerkinfrastrukturen. Neben Telepräsenz kann WebRTC für ergänzende Echtzeit-Features in Brettspielen – zum Beispiel das Zeichnen auf einem Spielbrett oder Drag-and-Drop von Spielfiguren, wobei die Position der Spielfigur zwischen Spielern geteilt wird – verwendet werden.

Letztendlich wurde deutlich, dass WebRTC eine flexible, wenn auch komplexere Alternative zu Client-Server-Modellen darstellt, welche im Bereich der Browserbasierten Brettspielentwicklung Potential hat. Der größte Nachteil der Anwendung von WebRTC ist, wie bei allen P2P-Technologien auch, der Aspekt des Betrugs. Ist dieser nicht relevant, so bietet WebRTC eine kostengünstige Alternative zu Client-Server-Architekturen.

Mit dem Lebensende des Adobe Flash Players, und der Veröffentlichung als Web-Standard, ist WebRTC jetzt vollkommen bereit für weitverbreitete Verwendung – auch im Bereich der Browserbasierten Spieleentwicklung.

8 Literaturverzeichnis

- [AS09] AWERBUCH, Baruch ; SCHEIDELER, Christian: Robust random number generation for peer-to-peer systems. In: *Theoretical Computer Science* 410 (2009), Februar, Nr. 6–7, S. 453–466
- [Bau10] BAU, David: *Random Seeds, Coded Hints, and Quintillions*. http://davidbau.com/archives/2010/01/30/random_seeds_coded_hints_and_quintillions.html#more, Januar 2010. – Abgerufen: 29.04.2021
- [Bau19] BAUN, Christian: *Computernetze kompakt*. 5. Springer Vieweg, 2019. – ISBN 978–3–662–59896–2
- [BGM20] *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
- [Bis14] BISHT, Altanai: *WebRTC Integrator's Guide*. Packt Publishing, 2014. – ISBN 978–1–78398–126–7
- [con16] Multiplayer Game Design: Performance Enhancement with Employment of Novel Technology. In: *International Journal of Computer Applications* 145 (2016), Juli, Nr. 1, S. 27–32
- [exp] *Express/Node Introduction*. – https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction, Abgerufen 28.04.2021
- [FM11] FETTE, Ian ; MELNIKOV, Alexey: The WebSocket Protocol / Internet Engineering Task Force. Version: Dezember 2011. <https://tools.ietf.org/html/rfc6455>. Internet Engineering Task Force, Dezember 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
- [god] *WebRTC*. <https://docs.godotengine.org/en/stable/tutorials/networking/webrtc.html>, . – Abgerufen: 15.05.2021
- [HS01] HOLDREGE, Matt ; SRISURESH, Pyda: Protocol Complications with the IP Net-

- work Address Translator / The Internet Society. Version: Januar 2001. <https://tools.ietf.org/html/rfc3027>. Internet Engineering Task Force, Januar 2001 (3027). – RFC. – Abgerufen: 11.04.2021
- [Krc09] KRCHA, Tom: *Using peer-to-peer applications on the Adobe Flash Platform*. https://www.adobe.com/devnet/adobe-media-server/articles/p2p_apps_cirrus_lccs.html, November 2009. – Abgerufen: 07.05.2021
- [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
- [nod] *Node.js Introduction*. – https://www.w3schools.com/nodejs/nodejs_intro.asp, Abgerufen: 07.04.2021
- [NPVS07] NEUMANN, Cristoph ; PRIGENT, Nicolas ; VARVELLO, Matteo ; SUH, Kyoungwon: Challenges in Peer-to-Peer Gaming. In: *ACM SIGCOMM Computer Communication Review* (2007), Januar
- [PHRW⁺20] PETIT-HUGUENIN, Marc ; ROSENBERG, Jonathan ; WING, Dan ; MAHY, Rohan ; MATTHEWS, Philip: Session Traversal Utilities for NAT (STUN). Version: Februar 2020. <https://tools.ietf.org/html/rfc8489>. Internet Engineering Task Force, Februar 2020. – RFC. – ISSN 2070-1721. – Abgerufen: 06.05.2021
- [pm2] *PM2 Documentation*. – <https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/>, Abgerufen 14.05.2021
- [r19] R, Ilan_r_: Definition Echtzeitanwendung. (2019), Februar. – Abgerufen 10.04.2021
- [Res19] RESCORLA, Eric: WebRTC Security Architecture. Version: Juli 2019. <https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-20>. Internet Engineering Task Force, Juli 2019. – Internet-Draft. – Abgerufen: 06.05.2021
- [rol20] *Video and Voice Chat*. <https://help.roll20.net/hc/en-us/articles/360041544734-Video-and-Voice-Chat>, Mai 2020. – Abgerufen: 16.05.2021
- [rtca] *Introduction to WebRTC protocols*. – https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols, Abgerufen: 16.05.2021
- [rtcb] *RTCDATAChannel*. <https://developer.mozilla.org/en-US/docs/Web/API/RTCDATAChannel>, . – Abgerufen: 10.04.2021
- [rtcc] *RTCPeerConnection*. <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection>, . – Abgerufen: 10.04.2021

- [rtcd] *RTCPeerConnection.canTrickleIceCandidates*. – <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/canTrickleIceCandidates>, Abgerufen: 16.05.2021
- [rtce] *WebRTC connectivity*. – https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity, Abgerufen: 16.05.2021
- [San19] SANTORE, Laetitia: *Stream high-quality real-time graphics through your browser with our new WebRTC framework*. <https://blogs.unity3d.com/2019/09/17/stream-high-quality-real-time-graphics-through-your-browser-with-our-new-webrtc-framework/>, September 2019. – Abgerufen: 15.05.2021
- [Sil15] SILVEIRA, Rodrigo: *Multiplayer Game Development with HTML5*. Packt, 2015. – ISBN 978-1-78528-310-9
- [sio] *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: Dezember 2007. <https://tools.ietf.org/html/rfc4960>. Internet Engineering Task Force, Dezember 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
- [Ube13] UBERTI, Justin: *A REST API For Access To TURN Services / Network Working Group*. Version: Juli 2013. <https://tools.ietf.org/html/draft-uberti-behave-turn-rest-00>. Internet Engineering Task Force, Juli 2013. – Internet-Draft. – Abgerufen: 28.04.2021

A Anhang – Peer.js

```
1 export default function Peer(id, iceServers) {
2   this.id = id;
3   this.callbacks = {};
4
5   this.rtcConfiguration = { iceServers : iceServers };
6   this.connections = {};
7
8   this.signal = null; // callback to send a signaling message to a remote peer
9 }
10
11 Peer.prototype.closeConnections = function() {
12   Object.values(this.connections).forEach((connection) => connection.close());
13 }
14
15 Peer.prototype.closeConnection = function(remotePeerId) {
16   if (this.connections[remotePeerId]) {
17     this.connections[remotePeerId].close();
18     delete this.connections[remotePeerId];
19   }
20 }
21
22 Peer.prototype.setICETransportPolicy = function(policy) {
23   if (policy !== 'relay' && policy !== 'all') {
24     console.error('ICE transport policy must be of values [relay, all]');
25     return;
26   }
27
28   this.rtcConfiguration.iceTransportPolicy = policy;
29 }
30
31 Peer.prototype.setICEServers = function(servers) {
32   this.rtcConfiguration.iceServers = servers;
33 }
34
35 Peer.prototype.setSignalingCallback = function(cb) {
36   this.signal = cb;
37 }
38
39 Peer.prototype._receiveMessage = function(e) {
40   const message = JSON.parse(e.data);
41
42   if (message.event && this.callbacks[message.event]) {
43     this.callbacks[message.event](...message.data, message.src);
44   }
45 }
46
47 Peer.prototype._dataChannelOpen = function(remotePeerId) {
48   console.log(`[${remotePeerId}] data channel open`);
```



```

49
50 // check if all channels of all connections are open
51 if (Object.values(this.connections).every((connection) => {
52   return connection.dc.readyState === 'open'
53 }))) {
54   // if a callback for this event exists, run it
55   if (this.callbacks['onDataChannelsOpen']) this.callbacks['onDataChannelsOpen']();
56 }
57 }
58
59 Peer.prototype._createConnection = function(remotePeerId) {
60   const connection = new RTCPeerConnection(this.rtcConfiguration);
61   connection.dc = {}; // list of data channels belonging to this connection
62
63   // setup a reliable and ordered data channel, store in connection object
64   const channel = connection.createDataChannel('game', {
65     negotiated : true,
66     id : 0,
67     maxRetransmits : null, // no maximum number of retransmits
68     ordered : true // force ordered package retrieval
69   });
70   channel.onmessage = (e) => this._receiveMessage(e);
71   channel.onopen = () => this._dataChannelOpen(channel.label, remotePeerId);
72
73   connection.dc = channel;
74
75   connection.onsignalingstatechange = () => {
76     console.log(`[${remotePeerId}] signaling state '${connection.signalingState}'`);
77   }
78
79   connection.onicecandidate = (e) => {
80     this.signal(this._createSignal('ice-candidate', e.candidate, remotePeerId));
81   }
82
83   connection.onicecandidateerror = (e) => {
84     if (e.errorCode >= 300 && e.errorCode <= 699) {
85       console.warn('STUN-Server could not be reached or threw error.', e)
86     } else if (e.errorCode >= 700 && e.errorCode <= 799) {
87       console.warn('TURN-Server could not be reached.', e)
88     }
89   }
90
91   return connection;
92 }
93
94 Peer.prototype._createSignal = function(type, data, target) {
95   return {type : type, src : this.id, target : target, data : data}
96 }
97
98 Peer.prototype.connect = function(remotePeerId) {
99   const connection = this._createConnection(remotePeerId, true);
100   this.connections[remotePeerId] = connection;
101
102   connection.createOffer().then((offer) => {
103     this.signal(this._createSignal('offer', offer, remotePeerId));
104     return connection.setLocalDescription(offer);
105   }).catch((e) => console.error(e));

```

```

106 }
107
108 Peer.prototype.onsignal = function(e) {
109   if (e.target && e.target === this.id) {
110     switch(e.type) {
111       case 'offer': // on offer, create our answer, set our local description and
112                     // signal the remote peer
113         const connection = this._createConnection(e.src);
114         this.connections[e.src] = connection;
115
116         connection.setRemoteDescription(e.data).then(() => {
117           console.log('remote description set');
118           return connection.createAnswer();
119         }).then((answer) => {
120           this.signal(this._createSignal('answer', answer, e.src));
121           return connection.setLocalDescription(answer);
122         }).catch((e) => console.error(e));
123         break;
124       case 'answer': // on answer, set our remote description
125         this.connections[e.src].setRemoteDescription(e.data).then(() => console.log('
126 remote description set')).catch(e => console.error(e));
127         break;
128       case 'ice-candidate': // on ice candidate, add the ice candidate to the
129                             // corresponding connection
130         console.log('ice-candidate', e.data);
131         this.connections[e.src].addIceCandidate(e.data).then();
132         break;
133     }
134   }
135 }
136
137 Peer.prototype.on = function(e, cb) {
138   this.callbacks[e] = cb;
139 }
140
141 // TODO: not needed for mesh-architecture - maybe remove?
142 Peer.prototype.emit = function(target, e, ...args) { // function for the master peer
143   // to relay a targeted message
144   if (this.connections[target]) {
145     const data = JSON.stringify({src: this.id, event: e, data: args});
146     this.connections[target].dc.send(data);
147   }
148 }
149
150 // TODO: not needed for mesh-architecture - maybe remove?
151 Peer.prototype.relay = function(exclude, packet) { // function for the master peer to
152   // relay a message
153   Object.keys(this.connections).forEach((key) => {
154     if (key !== exclude) {
155       this.connections[key].dc.send(packet);
156     }
157   });
158 }
159
160 Peer.prototype.broadcast = function(e, ...args) { // function for any peer to
161   // broadcast a message
162   Object.values(this.connections).forEach((connection) => {

```

```
157     const data = JSON.stringify({src : this.id, event: e, data: args});  
158     connection.dc.send(data);  
159   });  
160 }
```

B Anhang – Server.js

```
1  const express = require('express');
2  const config = require('./config.json');
3  const utils = require('./utils.js');
4
5  const app = express();
6
7  const server = require('http').Server(app);
8  const io = require('socket.io')(server);
9  app.use(express.static(config.server.static));
10
11 // serve files
12 app.get('/', (req, res) => {
13   res.status(200);
14   res.sendFile(`${__dirname}${config.server.index}`);
15 });
16
17 app.get(`/game/*`, (req, res) => {
18   res.status(200);
19   res.sendFile(`${__dirname}${config.server.game}`);
20 });
21
22 server.listen(config.server.listeningPort);
23
24 // room = {
25 //   id = <four-letter room ID>
26 //   started = true | false
27 //   host = <socket-ID of the host>
28 //   players = []
29 // }
30 const rooms = {};
31
32 // we don't want to store the socket-ids of players inside the player array of the
33 // room, because we don't want to send
34 // them to the players when they connect. But we need to find the socket-id of a peer
35 // when signaling, so this object
36 // mapping peerID -> socketID of every player exists.
37 const sockets = {};
38
39 io.sockets.on('connection', (socket) => {
40   socket.on('game-room-create', () => {
41     const id = utils.generateRoomID(rooms);
42     rooms[id] = {
43       id : id, // easier to identify room by player
44       players : [null, null, null, null],
45       started : false,
46       host : null
47     };
48   });
49 });
```

```

47 // if no client joins within a minute, destroy the room again (something went
48 wrong on the clients' side, as the redirect didn't go through)
49 setTimeout(() => {
50   if (rooms[id] && rooms[id].players.length === 0) {
51     console.log('DELETED ROOM DUE TO INACTIVITY TIMEOUT');
52     delete rooms[id];
53   }, 60000);
54
55 // const room = manager.createRoom(app, 'Test');
56 socket.emit('game-room-created', id);
57 });
58
59 const PLAYER_SLOT_PRIORITY = [0, 2, 1, 3]; // Als erstes gegenueberliegende Farben
60   fuellen
61 socket.on('game-room-join', (roomId) => {
62   const room = rooms[roomId];
63
64   if (!room) return socket.emit('game-room-join-failed', 'no such room');
65   if (room.players.every((p) => p !== null)) return socket.emit('game-room-join-
66     failed', 'room full');
67
68   for (let i = 0; i < 4; i++) {
69     if (!room.players[PLAYER_SLOT_PRIORITY[i]]) {
70       const peerID = utils.uuid4();
71       const color = PLAYER_SLOT_PRIORITY[i];
72       console.log('NEW PLAYER COLOR INDEX', color);
73
74       socket.join(roomID); // broadcasting for a subset of sockets is done via
75       socket.io rooms
76       sockets[peerID] = socket.id; // sockets mapped to peer-id for signaling (can't
77       be in player object, because we don't want to send that)
78       room.players[color] = {peerID : peerID, color : color};
79       room.seed = Math.random();
80
81       if (!room.host) {
82         room.host = socket.id;
83       }
84
85       socket.emit('game-room-joined', room.players, room.started, room.seed, peerID,
86       utils.generateTURNCredentials(socket.id), room.host === socket.id);
87       socket.to(roomID).emit('game-room-client-joining', room.seed, peerID, color);
88       break;
89     }
90   }
91 });
92
93 socket.on('start-game', (roomId) => {
94   const room = rooms[roomId];
95
96   if (room && !room.started && room.host === socket.id) {
97     room.started = true;
98
99     // Seed fuer die Zufallsfunktion
100    room.seed = Math.random();
101
102    // Seed an alle Spieler des Raums senden

```

```

98     io.to(roomId).emit('game-start', room.seed);
99   }
100 });
101
102 socket.on('game-room-kick-player', (roomId, index) => {
103   const room = rooms[roomId];
104
105   if (room && socket.id === room.host && room.players[index]) {
106     io.to(roomId).emit('game-room-kick-player', room.players[index].peerID, index);
107     room.players[index] = null;
108   }
109 });
110
111 socket.on('signal', (roomId, targetID, e) => {
112   const room = rooms[roomId];
113
114   if (room) {
115     const target = room.players.find((player) => player && player.peerID ===
116       targetID);
117     if (target) {
118       socket.to(sockets[target.peerID]).emit('signal', e);
119     }
120   }
121 });
122
123 socket.on('disconnecting', () => {
124   // the following search operations aren't super efficient when the player count
125   // gets large,
126   // but it's for the sake of having less LOC to clutter up the document.
127   // if this was an actual production application, there should likely be a map of
128   // socketID -> {peerID, roomId} for each socket connected, to have O(1) lookup
129   // times
130   const peerID = Object.keys(sockets).find((peerID) => sockets[peerID] === socket.id
131     );
132   const room = Object.values(rooms).find((room) => room.players.find((player) =>
133     player && player.peerID === peerID));
134
135   if (room) {
136     const leaver = room.players.find((player) => player && player.peerID === peerID)
137     ;
138     console.log('LEAVER PLAYER COLOR INDEX', leaver.color)
139     if (leaver) {
140       room.players[leaver.color] = null;
141
142       // Host migration
143       if (socket.id === room.host && room.players.length > 0) {
144         const newHost = room.players.find((p) => p !== null);
145         if (newHost) {
146           room.host = sockets[newHost.peerID];
147           io.to(room.id).emit('game-room-host-migration', newHost.peerID);
148         }
149       }
150
151       if (room.players.length > 0) {
152         socket.to(room.id).emit('game-room-client-leaving', leaver.peerID, leaver.
153           color); // notify remaining
154       } else {

```

```
148         delete rooms[room.id]; // remove the room when the last player left the room
149     }
150 }
151 }
152 });
153 });
```

C Anhang – game.js

```
1 import Peer from './network/Peer.js';
2 import {Game} from './game/Game.js';
3 import Player from './game/Player.js';
4 import * as utils from './game/Utils.js';
5
6 const GAME_SERVER_CONNECTION_DEV = 'http://localhost:3000';
7 const GAME_SERVER_CONNECTION_PROD = 'http://20.56.95.156:3000'; // this should
  probably be injected by express middleware...
8
9 const DATA_CHANNELS = [{ // reliable, ordered data channel
10   label : 'game',
11   rtcDataChannelConfig : {
12     maxRetransmits : null, // no maximum number of retransmits
13     ordered : true // force ordered package retrieval
14   }
15 }];
16
17 $(window).on('load', () => {
18   const socket = io(GAME_SERVER_CONNECTION_DEV);
19   const roomId = window.location.pathname.split('/')[2];
20
21   socket.on('connect', () => {
22     socket.emit('game-room-join', roomId); // Raum beitreten
23   });
24
25   socket.on('game-room-joined', (players, started, seed, peerID, iceServers, isHost)
26     => {
27     let localPlayerIsHost = isHost;
28
29     const peer = new Peer(peerID, iceServers, DATA_CHANNELS); // erstellen des Peers
30     peer.setSignalingCallback((data) => socket.emit('signal', roomId, data.target,
31       data)); // Signalisierung VOM Peer weiterleiten
32     socket.on('signal', (e) => peer.onSignal(e)); // Signalisierung ZUM Peer
33       weiterleiten
34
35     const game = new Game($('#maedn'), peer, DATA_CHANNELS[0].label);
36     socket.on('game-start', (seed) => game.start(seed));
37     if (started) {
38       game.seed(seed);
39       game.paused = true;
40     }
41     game.render();
42
43     socket.on('game-room-client-joining', (seed, peerID, color) => {
44       game.paused = true;
45       game.seed(seed);
46       game.addPlayer(new Player(color, 'TestPlayer', false));
47     });
48   });
49 }
```



```

44     utils.uiEnablePlayerKickButtons(game.players, localPlayerIsHost, roomID, socket)
45     });
46
47     socket.on('game-room-client-leaving', (peerID, color) => {
48         peer.closeConnection(peerID);
49         game.removePlayer(color);
50         utils.uiEnablePlayerKickButtons(game.players, localPlayerIsHost, roomID, socket)
51     });
52
53     socket.on('game-room-host-migration', (newHostPeerID) => {
54         console.log('HOST MIGRATION', newHostPeerID === peerID, newHostPeerID, peerID);
55         localPlayerIsHost = newHostPeerID === peerID;
56         $('#start-game').prop('disabled', newHostPeerID !== peerID).click(() => socket.
emit('start-game', roomID));
57         $('#end-game').prop('disabled', newHostPeerID !== peerID).click(() => socket.
emit('end-game', roomID));
58     });
59
60     socket.on('game-room-kick-player', (peerID, color) => {
61         if (game.players[color]) {
62             if (game.players[color].isLocalPlayer) {
63                 // the server removes us when we are kicked, NOT when we disconnect
64                 // so leave this page normally
65                 window.location.href = '/';
66             } else {
67                 peer.closeConnection(peerID);
68                 game.removePlayer(color);
69                 utils.uiEnablePlayerKickButtons(game.players, localPlayerIsHost, roomID,
socket);
70             }
71         }
72     });
73
74     peer.on('onDataChannelsOpen', () => {
75         if (localPlayerIsHost) {
76             peer.broadcast('gamestate', game.gamestate);
77         }
78         // unpause whenever ALL dcs to other peers are open (meaning also new ones)
79         game.paused = false;
80     });
81
82     peer.on('gamestate', (gamestate) => {
83         game.setGamestate(gamestate);
84         game.paused = false; // unpause when receiving game state
85     });
86
87     players.forEach((player) => { // connect to other peers
88         if (!player) return;
89         player.peerID === peerID ? game.localPlayerColor = player.color : peer.connect(
player.peerID);
90         game.addPlayer(new Player(player.color, 'TestPlayer', game.localPlayerColor ===
player.color));
91     });
92
93     utils.uiEnablePlayerKickButtons(game.players, localPlayerIsHost, roomID, socket);

```

```
94
95 // utility stuff, don't go over this in document
96 $('#start-game').prop('disabled', !localPlayerIsHost).click(() => socket.emit('
start-game', roomId, socket.id));
97 $('#end-game').prop('disabled', !localPlayerIsHost).click(() => socket.emit('end-
game', roomId, socket.id));
98 });
99 });
```
