
**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: 4. 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 |

# Vorwort

Danke und so!

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

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

# Zusammenfassung

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

// TODO: eng

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Codeverzeichnis

# Abkürzungsverzeichnis

**W3C** World Wide Web Consortium . . . 1
**WebRTC** Web Real-Time Communication . . . 1
**P2P** Peer-To-Peer . . . 1
**HTTP** Hypertext Transfer Protocol
**NPM** Node Package Manager . . . 13
**IETF** Internet Engineering Task Force
**RTP** Real-Time Transport Protocol . . . 6
**SRTP** Secure Real-Time Transport Protocol . . . 6
**JSEP** JavaScript Session Establishment Protocol . . . 5
**ICE** Interactive Connectivity Establishment . . . 6
**NAT** Network Address Translation
**STUN** Session Traversal Utilities for NAT . . . 11
**TURN** Traversal Using Relays around NAT
**SCTP** Stream Control Transport Protocol . . . 6
**TLS** Transport Layer Security . . . 12
**DTLS** Datagram Transport Layer Security . . . 12
**API** Application Programming Interface . . . 14
**TCP** Transmission Control Protocol . . . 14
**UDP** User Datagram Protocol . . . 12
**OSI** Open Systems Interconnection . . . 11
**IP** Internet Protocol . . . 11
**SIP** Session Initialization Protocol . . . 7
**SDP** Session Description Protocol . . . 7
**URL** Unique Resource Locator
**HTML** Hypertext Markup Language
**VM** Virtuelle Maschine . . . 24
**RTMFP** Real-Time Media Flow Protocol . . . 15
**RC4** Rivest Cipher 4 . . . 44
**HMAC** Hash-Based Message Authentication Code . . . 39
**SHA-1** Secure Hash Algorithm 1 . . . 39
**JSON** JavaScript Object Notation . . . 29

# 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 Web-Standard in seiner – bezüglich der zentralen Funktionalität – finalen Form. WebRTC ist eine Peer-To-Peer Web-Technologie, und wird primär für Webbrowserbasierte Echtzeit-Anwendungen wie Audio- und Videokommunikation verwendet.

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

In der Regel basieren Browserbasierte Mehrspieler-Spiele auf einer Client-Server-Architektur, wobei der Server die Rolle des bestimmenden Spielleiters übernimmt. Ein weiterer, gut definierter aber für Browserbasierte Spiele selten verwendeter Ansatz für die Vernetzung von Spielern ist die Peer-To-Peer (P2P)-Architektur. Bei dieser existiert kein zentraler Server, die Nutzer (Peers) sind gleichberechtigt und tauschen Daten direkt untereinander aus. Einer der großen Vorteile der Peer-To-Peer Architektur sind dabei die geringeren Datenmengen, welche über einen zentralen Server verwaltet werden müssen. Dies führt zu Kostenersparnissen in Form von weniger Bedarf an Hardware.

Beide dieser Netzwerkarchitekturen finden in der Spieleentwicklung Anwendung – jedoch ist die Nutzung von P2P-Netzwerken zum Datenaustausch bei Browserbasierten Mehrspieler-Spielen begrenzt. Dies ist nicht zuletzt auf das Lebensende des Adobe Flash Players im Dezember 2020 zurückzuführen, welcher über Peer-To-Peer Fähigkeiten, ermöglicht durch das Real-Time Media Flow Protocol von Adobe [Tho13], verfügte. Seitdem existiert – mit Ausnahme von WebRTC – keine alternative Möglichkeit um Nutzer von Webanwendungen, ohne die Nutzung von Plug-Ins oder Drittanbietersoftware, direkt untereinander zu vernetzen.

## 1.1 Zielsetzung

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

## 1.2 Aufbau der Arbeit

// TODO: logischerweise als letztes...

# 2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Implementierung benötigten theoretischen, sowie technischen Grundlagen beschrieben. Dabei werden primär die Funktionalität von WebRTC an sich, als auch die damit verbundenen Signal-Mechanismen und Infrastrukturen behandelt.

## 2.1 Echtzeitanwendungen

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

## 2.2 Netzwerkarchitekturen

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 der Regel kann ein Server dabei eine Vielzahl an Clients versorgen.

#### Authoritative-Server

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

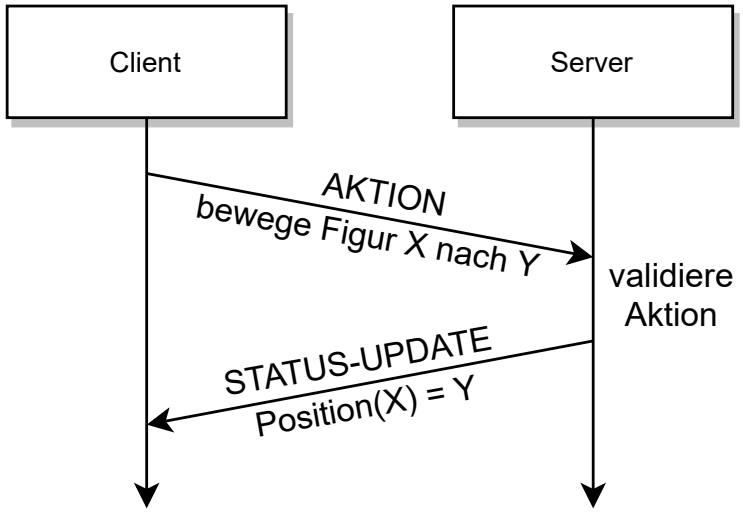


Abbildung 2.1: Beispielhafte Interaktion in einem Authoritative-Server-Modell.

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

### 2.2.2 Peer-To-Peer Netzwerke

## 2.3 Network Address Translation

## 2.4 Web Real-Time-Communication

Bei WebRTC handelt es sich um einen Quelloffenen Standard zur Echtzeitkommunikation zwischen Browsern. 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 Verlgeich 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 arbitraren Daten.

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

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

### 2.4.1 Aufbau von WebRTC

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

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:

- Die **RTCPeerConnection**-Schnittstelle repräsentiert eine WebRTC-Verbindung zwischen dem lokalen Browser (Local-Peer), und einem externen Browser (Remote-Peer) [o.A21]. Die Signalisierungsmechanismen folgen dabei dem JavaScript Session Establishment Protocol (JSEP), die Verbindung selbst wird via dem

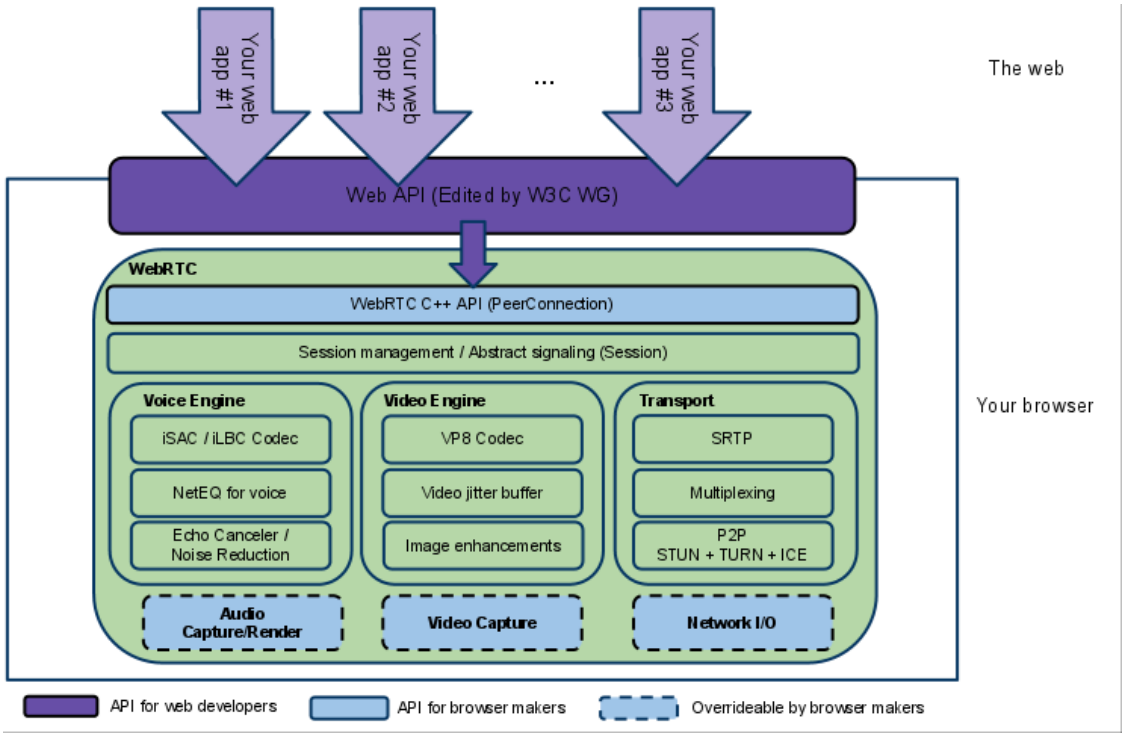


Abbildung 2.2: Diagramm der WebRTC-Architektur.
Quelle: `https://webrtc.github.io/webrtc-org/architecture/`

Interactive Connectivity Establishment (ICE) Framework hergestellt [LR14].

- Ein **RTCDataChannel** ist ein, von der RTCPeerConnection erstellter, bidirektionaler Datenkanal, welcher den Austausch von arbitraren Nachrichten zwischen Browsern ermöglicht. Eine RTCPeerConnection kann mehrere Datenkanäle besitzen. Zum Datenaustausch wird das Stream Control Transport Protocol (SCTP) verwendet [LR14, o.A21].

- Die **MediaStream**-API dient dazu, Audio- und Videosignale eines Gerätes abzurufen. 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. Dazu gehören kommen Protkolle wie STUN, TURN und das ICE-Framework, welche in den Folgenden Unterpunkten näher erläutert werden.

### 2.4.2 JSEP: JavaScript Session Establishment Protocol

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

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

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

Die JSEP-API stellt eine Reihe an asynchronen Funktionen zur Verfügung, welche das Erstellen und Setzen der Konfigurationen ermöglichen. Diese Funktionen sind in der WebRTC-API Teil der RTCPeerConnection.

Um eine Verbindung aufzubauen, ruft Alice erst *createOffer()* auf. Daraufhin wird ein SDP-Packet (*anfrage*) generiert, welches die lokalen Sitzungsparameter enthält. Alice setzt nun ihre lokale Konfiguration via *setLocalDescription()*. Das SDP-Packet wird über einen nicht vorgegebenen Signalkanal 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 seine lokale Konfiguration, und sendet die Antwort zurück zu Alice. Zuletzt setzt Alice ihre externe Konfiguration. Damit ist der anfängliche Austausch von Sitzungsparametern

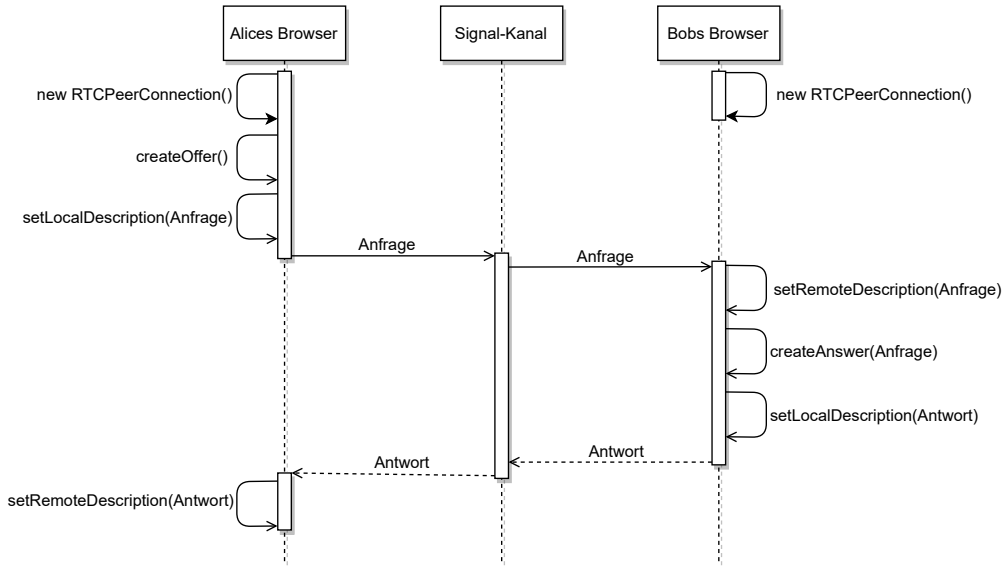


Abbildung 2.3: JSEP-Verbindungsaufbau.

abgeschlossen [Bis14]. JSEP regelt dabei nur den Austausch von Konfigurationen zwischen zwei Peers, Informationen über die Verbindung werden über das ICE-Framework ausgetauscht. Der vereinfachte Ablauf des Verbindungsaufbaus ist Abbildung 2.3 zu entnehmen.

### 2.4.3 ICE: Interactive Connectivity Establishment

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

Eine 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 können.

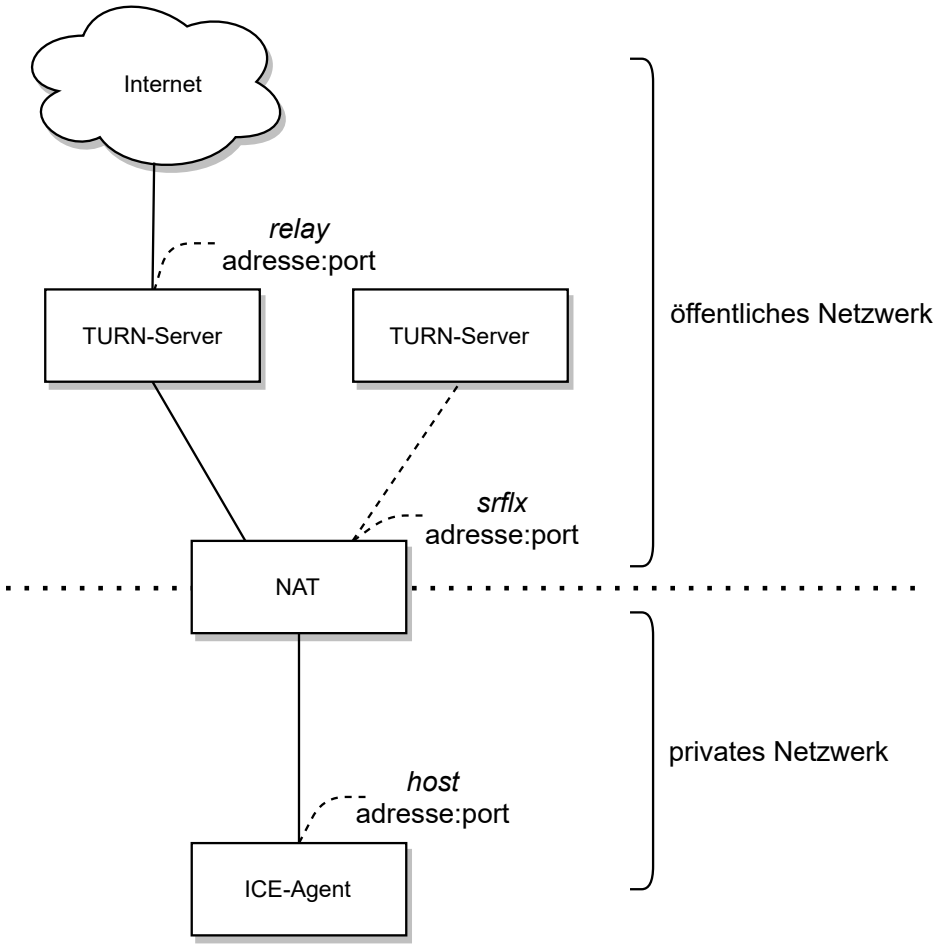


Abbildung 2.4: Diagramm der verschiedenen ICE-Kandidaten.
Quelle: Angepasst nach `https://tools.ietf.org/html/rfc5245`

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

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

### Trickle-ICE

Die STUN und TURN Server lassen sich beim Erstellen der RTCPeerConnection konfigurieren. WebRTC prüft die Verbindung zu allen möglichen ICE-Kandidaten eines Peers asynchron und parallel, sobald die lokale Beschreibung einer Verbindung gesetzt ist. Dieser Prozess – der sogenannte „ICE-Sammelprozess“ – setzt eine Kommunikation mit STUN- und TURN-Servern vorraus und kann daher, je nach Latenz und Antwortzeit der Server, einige Zeit beanspruchen.

Aus diesem Grund ermöglicht es WebRTC, ICE-Kandidaten zu „tricklen“, also jeweils einzelne Kandidaten bei Erhalt einer Serverantwort an den externen Peer zu schicken. Dies optimiert den Vorgang des Austauschs von ICE-Kandidaten, da der JSEP-Anfrage-Antwort-Prozess parallel zum ICE-Sammelprozess stattfinden kann.

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

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

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

#### STUN: Session Traversal Utilities for NAT

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

#### TURN: Traversal Using Relays around NAT

Im Gegensatz zu Client-Server-Verbindungen, welche nur vom Client eröffnet werden können, kann eine Peer-To-Peer Verbindung zudem sowohl vom lokalen Peer, als auch von einem Peer außerhalb des lokalen Subnetzes eröffnet werden. Hier besteht das Problem, dass nicht jede Art von NAT eine solche Interaktion erlaubt [HS01].

In solchen Fällen muss ein 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.

### 2.4.4 SCTP: Stream Control Transmission Protocol

Zur Übertragung von Daten via RTCDataChannels nutzt WebRTC das Stream Control Transport Protocol (SCTP). Der SCTP Standard wurde erstmals im Jahre 2000 von der IETF veröffentlicht, und seitdem weiterentwickelt und erweitert. SCTP ist ein nachrichtenorientiertes Transportprotokoll, welches im Open Systems Interconnection (OSI)-Referenzmodell, ähnlich wie UDP oder TCP, auf der Transportschicht liegt. Das Protokoll arbeitet dabei basierend auf verbindungslosen Netzwerkprotokollen, wie 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 parrallelen Datenströmen innerhalb einer Verbindung. Dazu ist der Datenteil eines SCTP-Packets in sognenannte „Chunks“ aufgeteilt, wobei Daten-Chunks jeweils einem Datenstrom zugeordnet werden können [Ste07].

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

Tabelle 2.1: Vergleich von TCP und UDP mit SCTP.

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

### 2.4.5 DTLS: Datagram Transport Layer Security

Das Datagram Transport Layer Security (DTLS)-Protokoll ist ein auf Transport Layer Security (TLS) aufbauendes Protokoll zur Verschlüsselung von Daten in auf Datengramm-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 zwischen den Verbindungspartnern ausgetauscht. Dieser Tunnel liegt auf dem User Datagram Protocol (UDP).

## 2.5 Node.js

Node.js ist eine kostenlose, plattformunabhängige JavaScript Laufzeitumgebung. Diese ermöglicht das Ausführen von JavaScript Programmen außerhalb eines Browsers, zum Beispiel auf einem Server. Node.js ist Open-Source und kann kostenlos verwendet werden. Programme setzen sich aus sogenannten *modules*, zu Deutsch Modulen, zusammen. Ein Modul kann dabei jegliche Funktionalität, wie zum Beispiel Klassen, Funktionen und Konstanten exportieren, welche dann wiederrum von weiteren Modulen oder Programmen verwendet werden können. Module können über das *require*-Stichwort geladen werden. Node.js bietet integrierte Webserver-Funktionalität via dem HTTP-Modul, welches das Erstellen eines Webservers ermöglicht [o.Ab]. Weiterhin bietet Node.js mit dem „crypto“-Modul kryptografische Funktionen, welche auf OpenSSL (Open Secure Socket Layer) basieren.

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

### 2.5.1 NPM: Node Package Manager

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

### 2.5.2 Verwendete Node-Packete

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

#### socket.io

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

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

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

Die Kommunikation zwischen Client und Server wird bei Socket.io über Events geregelt. Client und Server können Events – definiert durch einen String – mit angehängten Daten emittieren. Basierend auf dem Event-String wird dann auf der Empfängerseite eine Rückruffunktion aufgerufen, vorrausgesetzt diese ist definiert. Die Daten werden der Rückruffunktion als Parameter übergeben. Socket.io ermöglicht auf der Serverseite sowohl das Broadcasting an alle, beziehungsweise an ein Subset an Clients, als auch Unicasting an einen spezifischen Client [o.Ac].

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

#### express

„express“ ist ein Node-Web-Framework, welches bei der Erstellung von Webanwendungen zum Einsatz kommt. Express agiert als Middleware zwischen einem HTTP-Server und einer Webanwendung, und ermöglicht es, basierend auf HTTP-Methoden und URLs – den sogenannten „Pfaden“ – verschiedene Aktionen auszuführen [o.Aa]. 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 [o.Aa].

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

# 3 WebRTC in Mehrspieler-Spielen

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

WebRTC wird bereits in einigen Browserbasierten Mehrspieler-Spielen, sowie Networking- und Spiel-Frameworks verwendet. In der Regel wird WebRTC jedoch lediglich für Sprach- und Videokommunikation eingesetzt. Die Strategie- und Brettspiel Plattform *Jocly* ist eine der ersten Plattformen, welche bereits seit 2013 WebRTC nutzt, damit Spieler sich in Echtzeit über ihre Webcams beim Spielen sehen, sowie miteinander kommunizieren können (vgl. Abbildung 3.1) [LL13]. Ähnlich verhält es sich bei einigen weiteren Frameworks, wie zum Beispiel *Tabloro*[1], einem Browserbasierten „Tabletop“-Spielsimulator. Diese Spiele und Frameworks nutzen eine Client-Server-Architektur für den regulären (nicht Video und Audio) Datenaustausch.

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

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

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

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


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

# 4 Konzept

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

## 4.1 Anforderungen

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

### 4.1.1 Anforderungen an die Netzwerkstruktur

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

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

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

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

### 4.1.2 Anforderungen an die Clientseitigen WebRTC-Verbindungen

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

### 4.1.3 Anforderungen an das Brettspiel

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

#### Betrug

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

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

### Spielregeln

Am Spiel *Mensch ärgere Dich nicht* nehmen zwei bis vier Spieler teil. 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.

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 dessen 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.

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 Server-Infrastruktur

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

## 4.3 Peer-To-Peer Netzwerkarchitektur

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

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

Der Full-Mesh-Ansatz bildet das Gegenteil zum Authoritative-Peer-Modell. Hier verwaltet jeder Peer eine lokale Kopie des vollen Spielstands, und besitzt eine Verbindung zu jedem anderen Peer. 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 vorraussetzt.

## 4.4 Zufallszahlen

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

Peer-Netzwerk ist dies nicht ohne weiteres möglich [AS09].

### 4.4.1 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 Formel bei $n$ Peers eine Zahl, welche – vorrausgesetzt 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“, unter Nutzung von Commitment-Verfahren. Bei einem Commitment-Verfahren werden die zu sendenden Daten zuerst mit einem Passwort verschlüsselt und unter den Teilnehmern ausgetauscht. Sind alle verschlüsselten Daten ausgetauscht, so werden die Passwörter ausgetauscht. Es ist einem Peer somit nicht ohne weiteres möglich, auf alle weiteren Zahlen zu warten, ohne vorher eine eigene Zahl erstellt zu haben [AS09].

### 4.4.2 Seeded Random-Number-Generators

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

# 5 Design und Implementation

Das Projekt gliedert sich primär in zwei Teile: Das Brettspiel „Mensch Ärgere Dich Nicht“, und die unterliegende WebRTC- und Netzwerkinfrastruktur. Die Projektstruktur selbst ist in Abbildung 5.1 beschrieben. Auf die Spiel- und Netzwerkspezifischen Script-Dateien wird in deren jeweiligen Sektionen weiter eingegangen.

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

Tabelle 5.1: Struktur des Projektes.

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

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

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

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

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

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

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

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

## 5.2 Implementation der Netzwerkinfrastruktur

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

und die STUN- und TURN-Server.

### 5.2.1 Implementation des Webservers

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

#### Erstellen 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 Packete via

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

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

#### express Webserver

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

```
1 const express = require('express');
2 const config = require('./config.json');
3
4 const app = express();
5 const server = require('http').Server(app);
6
7 app.use(express.static(config.server.public));
8
9 app.get('/', (req, res) => {
10   res.status(200);
11   res.sendFile(`${__dirname}${config.server.indexPage}`);
12 });
13
14 app.get('/game/*/', (req, res) => {
15   res.status(200);
16   res.sendFile(`${__dirname}${config.server.gamePage}`);
17 });
18
19 server.listen(config.server.listeningPort);
```

Quellcode 5.1: express Server – Server.js

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

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

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

Zuletzt muss in Zeile 19 noch der Port definiert werden, über welchen der Server von außen ansprechbar sein soll. Dieser ist ebenfalls via der Konfigurationsdatei definierbar.

### 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.

```
[...]
const server = require('http').Server(app);
const io = require('socket.io')(server);
[...]
io.sockets.on('connection', (socket) => {
  [...] // Rueckruffunktionen
}
```

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.

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

Quellcode 5.3: Clientseitiger Verbindungsaufbau – game.js

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

### 5.2.2 Implementation der Peer-To-Peer Funktionalität

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

#### Eventbasiertes Nachrichtenprotokoll

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

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

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

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

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

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

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

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

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

### Signalisierungsprotokoll

Zur Signalisierung wird ein simples, proprietäres Protokoll verwendet. Eine 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. Hier ist wichtig, dass die Konfiguration des Kanals auf beiden Seiten exakt übereinstimmt. |
| id | 0 | Die numerische ID des Datenkanals, bei im Vorab vereinbarten Datenkanälen muss diese gegeben sein und zwischen beiden Peers übereinstimmen. |
| ordered | true | Das unterliegende SCTP-Protokoll soll die Nachrichten geordnet absenden und erhalten. |
| maxRetransmits | null | Ist dieser Wert nicht gesetzt, so versucht das SCTP-Protokoll so lange ein Packet abzuschicken, bis dieses beim Empfänger angekommen ist. |

Tabelle 5.2: Konfiguration der Datenkanäle.

Bei Erhalt eines ICE-Kandidaten in der *onicecandidate*-Rückruffunktion wird dieser über den Signalisierungskanal an den externen Peer gesendet.

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

```
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 Raum-Verwaltung

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

### Erstellen eines Raums

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

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

Erstellt ein Nutzer einen Raum, so sendet dieser das *game-room-create*-Event zum Server (vgl. Abbildung 5.11). Der Server generiert eine vierstellige Zeichenkette, welche als Raum-ID dient. 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 beitretbar ist.

### Beitritt eines Raums

Das Betreten eines Raums ist über das *game-room-join*-Event geregelt. Versucht ein Nutzer einem Raum beizutreten, so sendet dieser das Event an den Server (vgl. Abbildung 5.12). Falls der Raum noch keinen Host-Spieler besitzt, so wird der erste Beitretende Spieler zum Host ernannt. Bei erfolgreichem Beitritt erhält der Nutzer das *game-room-joined*-Event zurück, welches sämtliche, zum Verbindungsaufbau mit den weiteren Spielern, benötigten Daten enthält. Dazu gehören das Array der weiteren Spieler im Raum, die eigene Peer-ID und die Zugangsdaten zu den STUN- und TURN-Servern. Zusätzlich wird dem Nutzer mitgeteilt, ob dieser der Host des Raums ist. Zudem werden – in Zeile 22 – alle weiteren Spieler des Raums benachrichtigt, dass ein weiterer Spieler beigetreten ist.

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.

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

Tabelle 5.3: Spielerfarben.

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).

```
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

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

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

Tabelle 5.4: Weitere Raumspezifische Events.

#### Weitere Raumspezifische Events

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

### 5.2.4 Signalisierungskanal

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

```
io.sockets.on('connection', (socket) => {
  [...]
  socket.on('signal', (roomID, targetID, e) => {
    const room = rooms[roomID];

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

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

### 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. PM2 lässt sich über den Node Package Manager installieren. Eine Applikation kann über den Befehl

```
$ pm2 start Server.js
```

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

```
$ pm2 startup
```

Dieser Befehl generiert die Start-Konfiguration und aktiviert die Startup-Funktionalität. Nun muss die momentane 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 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 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 Packetmanager *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:

```
TURNSERVER_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.

```
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
fingerprint
```

Quellcode 5.15: Coturn-Konfigurationsdatei – turnserver.conf

| 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 nutzt 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. |
| fingerprint | Fingerprinting von STUN- und TURN-Nachrichten soll für WebRTC standardmäßig aktiviert sein. Dies ist in der Konfigurations-Hilfsdatei von coturn definiert. |

Tabelle 5.5: Erläuterung der Konfigurationsparameter.

### 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, ungeignet.

- **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 vorraussetzt. 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 Password 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 Webservers 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.

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

| 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.

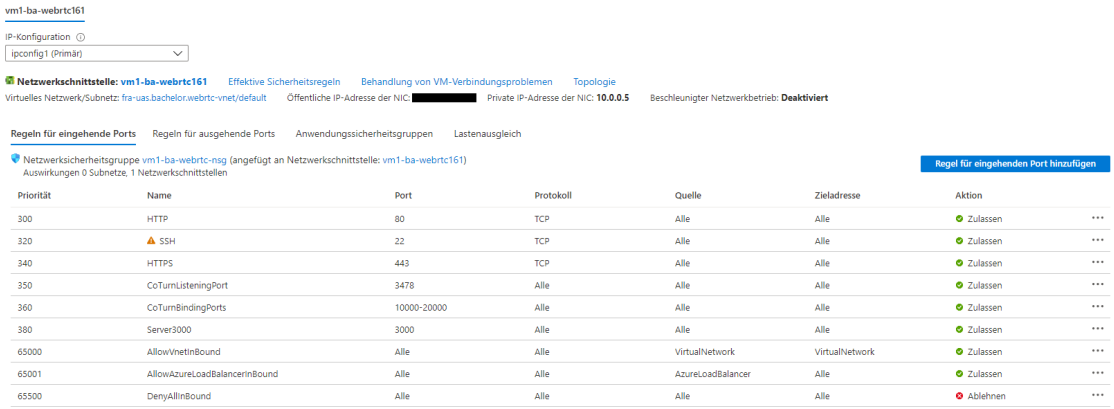


Abbildung 5.1: Port-Regeln der VM im Azure-Webportal.

Einstellungen eine Regel für eingehende Ports hinzugefügt werden (vgl. Abbildung 5.1).

### 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 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 Standartwerten 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 Standartwerten 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.

```
// aufgerufen wenn alle (neuen) Datenkanaele offen sind
peer.on('dataChannelsOpen', () => {
  if (isHost && game.gamestate.fsm !== 'started') {
    peer.emit(peerID, 'gamestate', game.gamestate);
  }
  [...]
});
```

Quellcode 5.20: Senden des Spielstands – game.js

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

### 5.4.1 Begriffserklärungen

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 dem Modifikation des Spielstands. Gewinnt der Spieler am Zug das Spiel, so wird das Spiel beendet - es folgen keine weiteren Züge und Runden. Falls nicht wird die Kontrolle an den nächsten Spieler übergeben, welcher dann wiederrum am Zug ist.

### 5.4.2 Spielablauf

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.

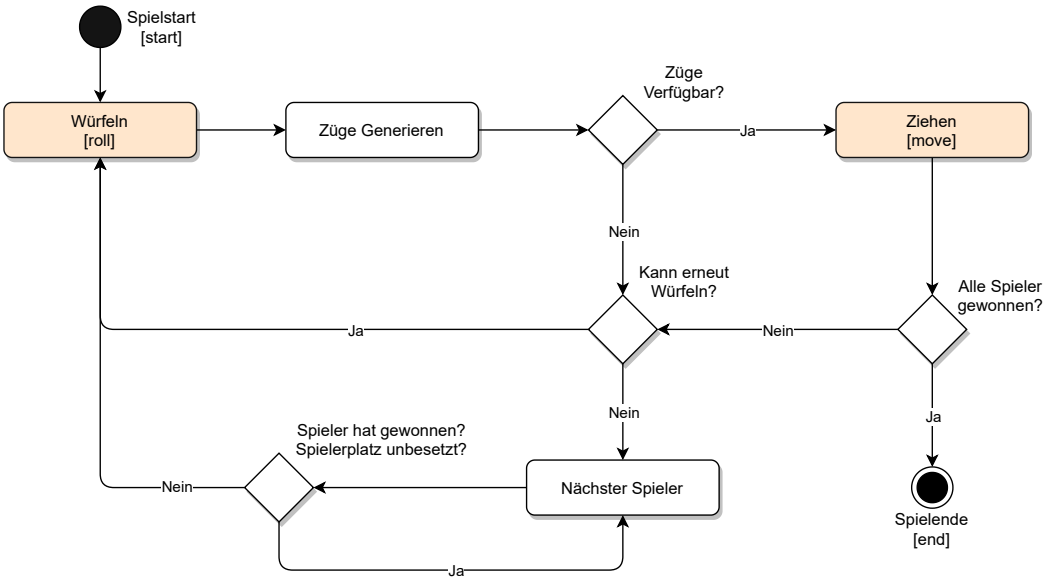


Abbildung 5.2: Ablaufdiagramm des Brettspiels.

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 standartmäßig keine Möglichkeit, einen *Seeded-Random-Number-Generator* zu erstellen. Aus diesem Grund wird die „seedrandom“-Bibliothek eingebunden[1]. Die Bibliothek stellt die *Math.seedrandom*-Funktion bereit, welche einen Pseudozufallszahl-Generator erstellt. Standartmäßig nutzt dieser die Stromverschlüsselung Rivest Cipher 4 (RC4) zum Generieren von Zufallszahlen [Bau10].

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

[1] seedrandom: `https://github.com/davidbau/seedrandom`

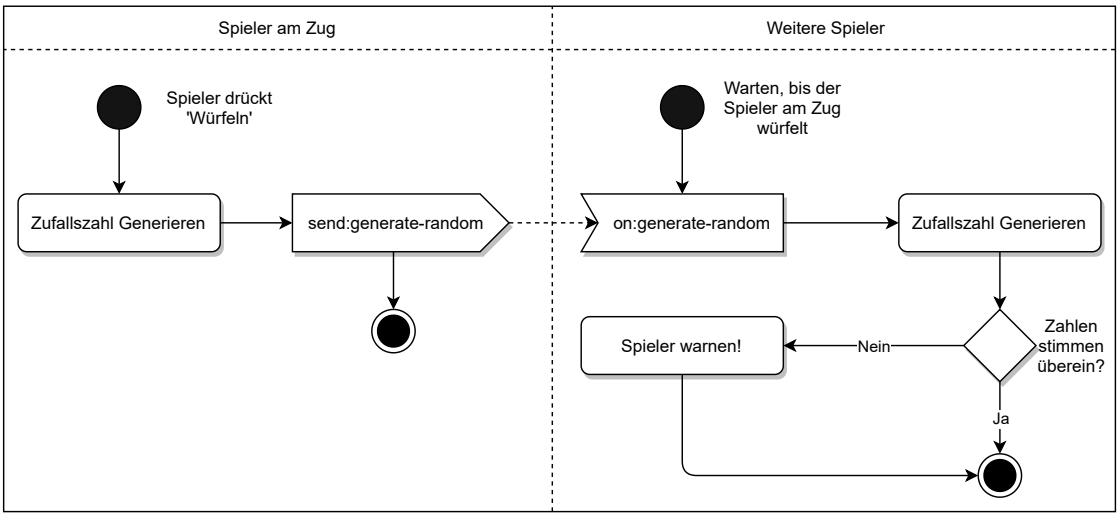


Abbildung 5.3: Würfel-Nachrichtenaustausch.

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.

### 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).

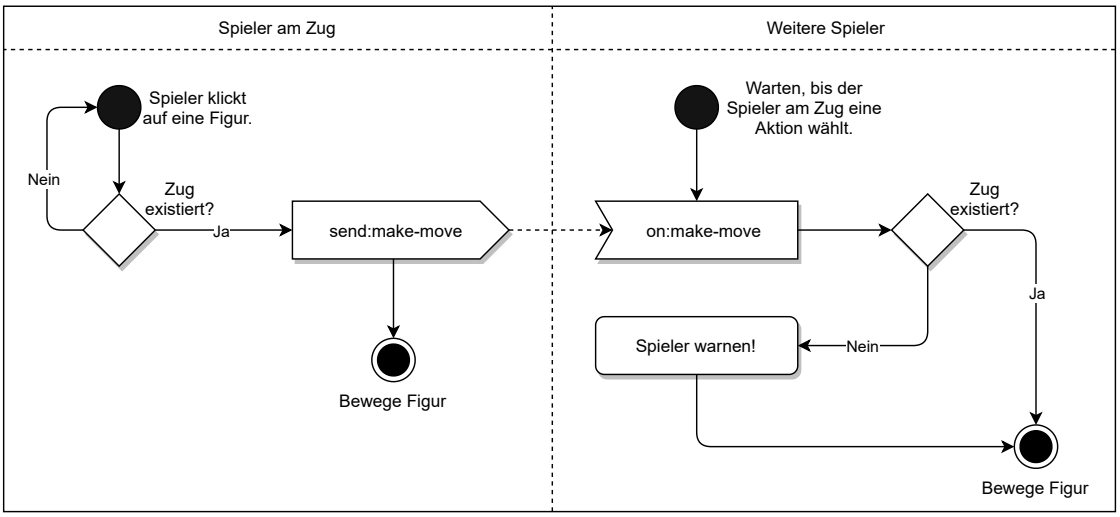


Abbildung 5.4: Ziehen-Nachrichtenaustausch.

### 5.4.3 Test

Bei Abruf der Webanwendung wird die Seite zur Erstellung eines Raumes aufgerufen. Klickt ein Nutzer auf *Raum Erstellen*, so wird dieser nach Erstellen des Raums auf die *game*-Seite weitergeleitet. Weitere Spieler können dieser Seite über den Direktlink beitreten. Nachdem die Datenkanäle aller Spieler offen sind, kann der Host – der Ersteller des Raums – das Spiel starten. Spieler können während dem Spielverlauf das Spiel verlassen oder, vorrausgesetzt ein Spielerplatz ist frei, beitreten.

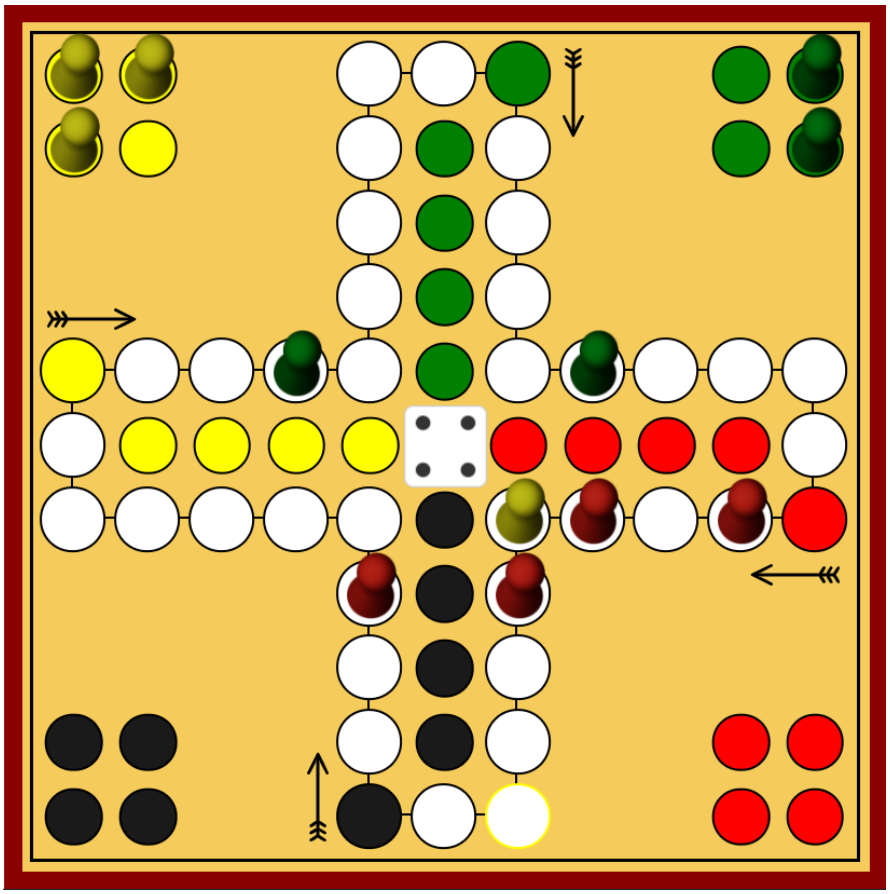

Abbildung 5.5: Laufendes Spiel mit drei Spielern. Gelb ist am Zug.

# 6 Evaluation

Dieses Kapitel befasst sich mit der Evaluation der Nutzung von WebRTC für Browserbasierte Mehrspieler-Brettspiele. Dabei werden sowohl technische Aspekte betrachtet, als auch auf Möglichkeiten und Einschränkungen, welche WebRTC mit sich bringt, eingegangen.

## 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 Implementierung 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 RTCPeerConnection 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 desruptiven Spielerverhaltens wird von der Implementierung in keinster Weise verhindert. Lösungsansätze sind hier zum Beispiel Zeitlimits für Spieleraktionen. Insbesondere in Full-Mesh Architekturen ist ein einheitlicher Timer jedoch schwer implementierbar.

## 6.2 WebRTC im Vergleich Client-Server Architekturen für Browserbasierte Brettspiele

Die technischen Vorraussetzungen, welche zur Entwicklung eines Mehrspieler-Brettspiels gegeben sein müssen, werden von WebRTC vollends erfüllt. WebRTC ermöglicht das problemlose Erstellen von zuverlässigen, geordneten Datenkanälen, welche in jedem Fall benötigt werden.

### 6.2.1 Technische Aspekte

### 6.2.2 Strukturelle Aspekte

### 6.2.3 Komplexität

# 7 Zusammenfassung und Ausblick

# 8 Literaturverzeichnis


[AS09] Awerbuch, Baruch ; Scheideler, Christian: Robust random number generation for peer-to-peer systems. In: *Theoretical Computer Science* 410 (2009), February, 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

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

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

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

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

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

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

[NPVS07] Neumann, Cristoph ; Prigent, Nicolas ; Varvello, Matteo ; Suh, Kyoungwon: Challenges in Peer-to-Peer Gaming. In: *ACM SIGCOMM Computer Communication Review* (2007), Januar. http://dx.doi.org/https://doi.org/10.1145/1198255.1198269. – DOI https://doi.org/10.1145/1198255.1198269

[o.Aa] o.A: *Express/Node introduction*. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction, . – Abgerufen 28.04.2021

[o.Ab] o.A: *Node.js Introduction*. https://www.w3schools.com/nodejs/nodejs_intro.

asp, . – Abgerufen: 07.04.2021

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

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

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

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

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

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

[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

```
export default function Peer(id, iceServers) {
  this.id = id;
  this.callbacks = {};

  this.rtcConfiguration = { iceServers : iceServers };
  this.connections = {};

  this.signal = null;   // callback to send a signaling message to a remote peer
}

Peer.prototype.closeConnections = function() {
  Object.values(this.connections).forEach((connection) => connection.close());
}

Peer.prototype.closeConnection = function(remotePeerId) {
  if (this.connections[remotePeerId]) {
    this.connections[remotePeerId].close();
    delete this.connections[remotePeerId];
  }
}

Peer.prototype.setICETransportPolicy = function(policy) {
  if (policy !== 'relay' && policy !== 'all') {
    console.error('ICE transport policy must be of values [relay, all]');
    return;
  }

  this.rtcConfiguration.iceTransportPolicy = policy;
}

Peer.prototype.setICEServers = function(servers) {
  this.rtcConfiguration.iceServers = servers;
}

Peer.prototype.setSignalingCallback = function(cb) {
  this.signal = cb;
}

Peer.prototype._receiveMessage = function(e) {
  const message = JSON.parse(e.data);

  if (message.event && this.callbacks[message.event]) {
    this.callbacks[message.event](...message.data, message.src);
  }
}

```

```
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       // STUN errors are in the range 300-699. See RFC 5389, section 15.6
86       // for a list of codes. TURN adds a few more error codes; see
87       // RFC 5766, section 15 for details.
88       console.warn('STUN-Server could not be reached or threw error.', e)
89     } else if (e.errorCode >= 700 && e.errorCode <= 799) {
90       // Server could not be reached; a specific error number is
91       // provided but these are not yet specified.
92       console.warn('TURN-Server could not be reached.', e)
93     }
94   }
95
96   return connection;
97 }
98
99 Peer.prototype._createSignal = function(type, data, target) {
100   return {type : type, src : this.id, target : target, data : data}
101 }
102
103 Peer.prototype.connect = function(remotePeerId) {
```

```
104   const connection = this._createConnection(remotePeerId, true);
105   this.connections[remotePeerId] = connection;
106
107   connection.createOffer().then((offer) => {
108     this.signal(this._createSignal('offer', offer, remotePeerId));
109     return connection.setLocalDescription(offer);
110   }).catch((e) => console.error(e));
111 }
112
113 Peer.prototype.onsignal = function(e) {
114   if (e.target && e.target === this.id) {
115     switch(e.type) {
116       case 'offer': // on offer, create our answer, set our local description and
      signal the remote peer
117         const connection = this._createConnection(e.src);
118         this.connections[e.src] = connection;
119
120         connection.setRemoteDescription(e.data).then(() => {
121           console.log('remote description set');
122           return connection.createAnswer();
123         }).then((answer) => {
124           this.signal(this._createSignal('answer', answer, e.src));
125           return connection.setLocalDescription(answer);
126         }).catch((e) => console.error(e));
127         break;
128       case 'answer': // on answer, set our remote description
129         this.connections[e.src].setRemoteDescription(e.data).then(() => console.log('
      remote description set')).catch(e => console.error(e));
130         break;
131       case 'ice-candidate': // on ice candidate, add the ice candidate to the
      corresponding connection
132         console.log('ice-candidate', e.data);
133         this.connections[e.src].addIceCandidate(e.data).then();
134         break;
135     }
136   }
137 }
138
139 Peer.prototype.on = function(e, cb) {
140   this.callbacks[e] = cb;
141 }
142
143 Peer.prototype.emit = function(target, e, ...args) { // function for the master peer
      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 Peer.prototype.broadcast = function(e, ...args) { // function for any peer to
      broadcast a message
151   Object.values(this.connections).forEach((connection) => {
152     const data = JSON.stringify({src : this.id, event: e, data: args});
153     connection.dc.send(data);
154   });
155 }
```

```
156
157 });
```

# 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   console.log(__dirname);
20   res.sendFile(`${__dirname}${config.server.game}`);
21 });
22
23 server.listen(config.server.listeningPort);
24
25 // room = {
26 //   id = <four-letter room ID>
27 //   started = true | false
28 //   host = <socket-ID of the host>
29 //   players = []
30 // }
31 const rooms = {};
32
33 // we don't want to store the socket-ids of players inside the player array of the
       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
       when signaling, so this object
35 // mapping peerID -> socketID of every player exists.
36 const sockets = {};
37
38 io.sockets.on('connection', (socket) => {
39   socket.on('game-room-create', () => {
40     const id = utils.generateRoomID(rooms);
41     rooms[id] = {
42       id : id, // easier to identify room by player
43       players : [],
44       started : false,
```

```
45        host : null
46      };
47
48      // if no client joins within a minute, destroy the room again (something went
        wrong on the clients' side, as the redirect didn't go through)
49      setTimeout(() => {
50        if (rooms[id] && rooms[id].players.length === 0) {
51          delete rooms[id];
52        }
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
        fuellen
60    socket.on('game-room-join', (roomID) => {
61      const room = rooms[roomID];
62
63      if (!room) return socket.emit('game-room-join-failed', 'no such room');
64      if (Object.keys(room.players).length >= 4) return socket.emit('game-room-join-
        failed', 'room full');
65
66      for (let i = 0; i < 4; i++) {
67        if (!room.players[PLAYER_SLOT_PRIORITY[i]]) {
68          const peerID = utils.uuid4();
69          const color = PLAYER_SLOT_PRIORITY[i];
70
71          socket.join(roomID); // broadcasting for a subset of sockets is done via
        socket.io rooms
72          sockets[peerID] = socket.id; // sockets mapped to peer-id for signaling (can't
         be in player object, because we don't want to send that)
73          room.players[color] = {peerID : peerID, color : color};
74          room.seed = Math.random();
75
76          if (!room.host) {
77            room.host = socket.id;
78          }
79
80          socket.emit('game-room-joined', room.players, room.started, room.seed, peerID,
         utils.generateTURNCredentials(socket.id), room.host === socket.id);
81          socket.to(roomID).emit('game-room-client-joining', room.seed, peerID, color);
82          break;
83        }
84      }
85    });
86
87    socket.on('start-game', (roomId) => {
88      const room = rooms[roomId];
89
90      if (room && !room.started && room.host === socket.id) {
91        room.started = true;
92
93        // Seed fuer die Zufallsfunktion
94        room.seed = Math.random();
95
```

```
96        // Seed an alle Spieler des Raums senden
97        socket.to(roomId).emit('game-start', room.seed);
98        // auch an den Host-Spieler (quirk mit socket.io, socket.to(socket) funktioniert
        nicht bei gleichem socket)!
99        socket.emit('game-start', room.seed);
100     }
101   });
102
103   socket.on('signal', (roomID, targetID, e) => {
104     const room = rooms[roomID];
105
106     if (room) {
107       const target = room.players.find((player) => player && player.peerID ===
      targetID);
108       if (target) {
109         socket.to(sockets[target.peerID]).emit('signal', e);
110       }
111     }
112   });
113
114   socket.on('disconnecting', () => {
115     // the following search operations aren't super efficient when the player count
      gets large,
116     // but it's for the sake of having less LOC to clutter up the document.
117     // if this was an actual production application, there should likely be a map of
118     // socketID -> {peerID, roomID} for each socket connected, to have O(1) lookup
      times
119     const peerID = Object.keys(sockets).find((peerID) => sockets[peerID] === socket.id
      );
120     const room = Object.values(rooms).find((room) => room.players.find((player) =>
      player && player.peerID === peerID));
121
122     if (room) {
123       const leaver = room.players.find((player) => player && player.peerID === peerID)
      ;
124       if (leaver) {
125         room.players = room.players.filter((player) => player && player.peerID !==
      peerID); // remove from room
126         if (room.players.length > 0) {
127           socket.to(room.id).emit('game-room-client-leaving', leaver.peerID, leaver.
      color); // notify remaining
128         } else {
129           delete rooms[room.id]; // remove the room when the last player left the room
130         }
131       }
132     }
133   });
134 });
```