HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Bachelor-Thesis

in

Medieninformatik

Konzeption, Implementation und Evaluation von Netzwerkkomponenten für eine auf die Lehre spezialisierte Game-Engine

Referent:	Prof. Jirka Dell'Oro-Friedl
Korreferent:	Prof. Dr. Ruxandra Lasowski
Vorgelegt am:	02.09.2019

Vorgelegt von:	Falco Böhnke
	250100
	Im Großacker, 28
	79252, Stegen
	falco.boehnke@hs-furtwangen.de

Abstract

Im Rahmen dieser Bachelorarbeit wurden Komponenten für die Netzwerkkommunikation innerhalb des auf Electron basierenden Editors „Fudge“ konzipiert und entwickelt.

Als erster Schritt wurden die Einschränkungen ermittelt, die sich durch die Verwendung von Electron, TypeScript und Node.js für die Umsetzung der Komponenten ergeben. „Fudge“, als didaktische Game Engine und Spiel Editor, stellt dabei besondere Anforderungen an das Design der Netzwerkkomponenten. Darauf basierend wurden anschließend geeignete Webtechnologien für die Entwicklung ausgewählt.

Als Ergebnis dieser Arbeit stehen nun funktionsfähige und erweiterbare Netzwerkkomponenten zur Verfügung. Diese basieren auf WebSocket, WebRTC und agieren innerhalb einer Electron Applikation. Sie sind skalierbar und können um Funktionalitäten erweitert werden, wodurch die Netzwerkkomponenten einen soliden Startpunkt für die Ausbildung angehender Spieleentwickler bieten. Offen ist noch die Frage, wie die Komponenten in einer auf Fudge basierenden Applikation eingebunden werden können. Weitere Forschung ist nötig um Komplikationen zwischen Fudge und Electron zu beheben.

In the scope of this Thesis Networkcomponents for use in “Fudge”, a Game Engine and Editor based on Electron, have been designed and developed.

First, limitations imposed on the development by Electron ,TypeScript and Node.js been determined. Additionally, requirements by “Fudge” itself have been set. Based on this, Technologies for the development have been chosen.

As a result there now exist fully functioning Network Components, based on the Webtechnologies WebSocket and WebRTC, in Electron based Applications. They are scalable and expandable, which makes them a solid starting point for Game Developers new to Videogame Networking.

A question to be answered remains: How can the components be integrated into Fudge itself? Currently there exist incompatibilities brought on by TypeScript that have to be resolved by further research.

Inhaltsverzeichnis

Darstellungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
HTML	Hypertext Markup Language
ICE	Interactive Connectivity Establishment
IP	Internet Protocol
JS	JavaScript
JSEP	JavaScript Session Establishment Protocol
JSON	JavaScript Object Notation
NAT	Network Address Translation
SDP	Session Description Protocol
STUN	Simple Traversal of User Datagram Protocol Through Network Address Translators
TCP	Transmission Control Protocol
TS	TypeScript
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
XML	Extensible Markup Language

1 Einleitung

Die Videospielindustrie ist in den letzten Jahren massiv gewachsen. Allein in den Niederlanden wuchs der Markt für Videospiele um ca. 318%, von 518 Millionen Euro im Jahr 2012 auf 1.65 Milliarden Euro im Jahr 2019, an, mit vergleichbaren Tendenzen weltweit (vgl. Wijman 2018) (vgl. Kazimier 2017). Der starke Anstieg der Umsatzzahlen hat zu einem gesteigerten Konkurrenzkampf innerhalb des Videospiel Marktes geführt. Innovationen sind unerlässlich geworden und so sehen sich Entwickler mit immer komplexeren Projektanforderungen konfrontiert (siehe Abbildung 1). Insbesondere im Bereich der Netzwerkkommunikation sind die Anforderung an hoch performante Lösungen gestiegen.

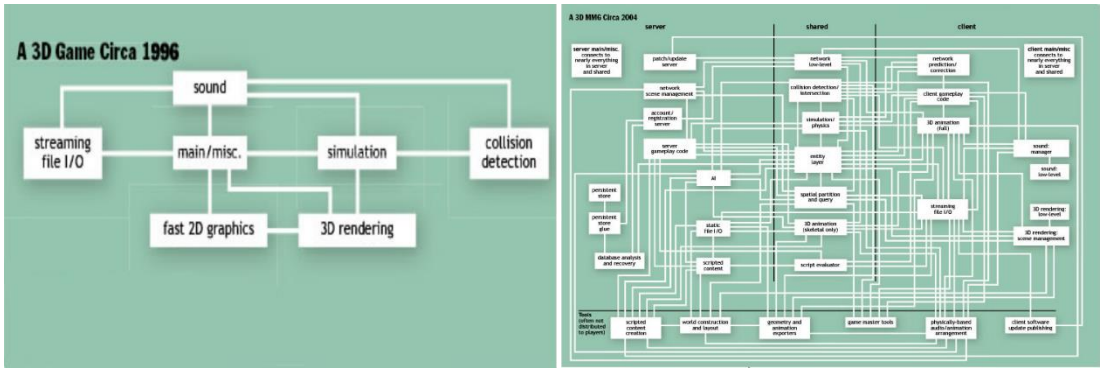


Abbildung 1: Vergleich benötigter Komponenten in einem digitalen Spiel, 1996 und 2004, Quelle: Blow (2004)

Spiel-Engines (auch Game-Engines) sind Entwicklungsumgebungen die für die Produktion von Videospielen optimiert wurden. Sie finden daher verstärkt Einsatz um die Arbeit der Entwickler zu erleichtern und komplexe Komponenten verwendungsfertig bereit zu stellen. Diese fertigen Komponenten sollen durch Automation und Abstraktion fundamentale Arbeiten übernehmen und so Kapazitäten für Optimierung und Kreativität freisetzen. Doch die Undurchsichtigkeit der abstrakten Systeme und die oftmals komplizierte Struktur der Spiel-Engines (siehe Abbildung 1). macht es Entwicklern in

Ausbildung schwer die grundlegende Funktionsweise der einzelnen Komponenten eines Videospiels zu erfassen und zu erlernen. So sind Entwickler gezwungen die spezifischen Funktionsweisen einer Entwicklungsumgebung zu erlernen, anstatt grundlegender Konzepte. Ist dann ein Wechsel auf eine andere Entwicklungsumgebung notwendig, müssen Entwickler sich erneut an die Umgebungsentwicklung anpassen, wobei dort ähnliche Komponenten mitunter andere Funktionsweisen haben. Besonders der Mangel einer von Grund auf für die Lehre konzipierten Game-Engine verschärft diese Problematik weiter.

Um diese Lücke zu füllen wurde das Projekt ‚Fudge' von Prof. Jirka Dell'Oro-Friedl ins Leben gerufen. Fudge ist eine Game-Engine und ein Spiel-Editor, der die Strukturen und Prozesse eines Videospiels offenlegt und Entwickler dennoch mit grundlegenden Funktionalitäten versorgt. Als Spiel-Editor bietet Fudge zusätzlich zu den Basiskomponenten, wie Animation und Physik, eine Oberfläche, mit der sich Spiele editieren und erstellen lassen.

In dieser Arbeit wurden Netzwerkkomponenten für Fudge entwickelt, die Entwicklern die Möglichkeit geben vernetzte Spiele zu entwickeln ohne die notwendigen Komponenten von Grund auf selbst zu schreiben. Der Aufbau der Arbeit enthält zunächst eine einführende Darstellung des Entwicklungsrahmen mit Fudge und Electron sowie der in modernen Spielen üblichen Strukturen zur Kommunikation zwischen Spielinstanzen. Dazu gehört ebenfalls ein Exkurs in modernen Netzwerkprotokollen, die in der Netzwerkkommunikation üblich sind und daher relevant für die Umsetzung der Netzwerkkommunikation in Fudge.

Aus diesen Informationen erschließen sich die Anforderungen an die Netzwerkkomponenten.

Unter Berücksichtigung der Rahmenbedingungen und Anforderungen werden dann die möglichen Technologien ermittelt, die zur Entwicklung verwendet werden können. Diese werden evaluiert und entsprechend ihrer Vorteile und Nachteile ausgewählt.

Darauffolgend wird die Umsetzung der einzelnen notwendigen Netzwerkkomponenten dargelegt und ihr Aufbau mithilfe von Unified Modeling Language Diagrammen dargestellt, sowie ihre Funktionsweise und Ablauf

beispielhaft mithilfe von Aktivitätsdiagrammen dargestellt und zusätzlich erläutert.

Abschließend werden die Ergebnisse der Evaluation und Entwicklung zusammengefasst und diskutiert. Ein Ausblick über mögliche Weiterentwicklungen der Komponenten schließt die Arbeit ab.

2 Rahmenbedingungen – Theoretische Grundlagen

In diesem Kapitel werden die Rahmenbedingungen für die Entwicklung der Netzwerkkomponenten dargelegt, welche Einschränkungen und Besonderheiten existieren und welche besonderen Anforderungen durch Fudge und Electron entstehen.

2.1 Fudge

Fudge ist eine Open Source Software, die Game-Engine und Spieleditor für die Entwicklung von zweidimensionalen und dreidimensionalen Spielen kombiniert und besonderes Augenmerkt auf didaktisch sinnvolle Strukturen sowie menschenlesbaren Quellcode legt. Der Name gründet sich dabei aus dem vollen Titel der Software: „**F**urtwangen **U**niversity **D**idactic **G**ame **E**ditor“. Als Game-Engine übernimmt Fudge dabei grundlegende Aufgaben wie die visuelle Repräsentation des Spielverlaufs sowie dem allgemeinen Ablauf, Audiowiedergabe und Animationen. Zudem bietet der Editor eine Oberfläche mithilfe derer Änderungen visualisiert, Spiele editiert und zeitgleich dargestellt werden können. Fudge ähnelt in dieser Hinsicht stark den gebräuchlicheren kommerziellen Game-Engines wie Unity und der Unreal Engine (vgl. Patel 2018) die ebenfalls Editor und Engine kombinieren. Gegenüber diesen kommerziellen Produkten ist Fudge von Grund auf auf die Anwendung im akademischen Rahmen spezialisiert und hat dadurch in diesem Bereich klare Vorteile. Datenformate sind allgemein gültig und menschenlesbar gestaltet - das bedeutet der Quellcode soll sich durch geschickte Namensgebung und klare Strukturen selbst erklären können. Außerdem wird so die Versionskontrolle vereinfacht, da eine Integration mit Anbietern wie GitHub oder GitLab standardmäßig möglich ist. Ein weiterer wichtiger Punkt ist, dass Fudge sich auf das Prinzip von „Composition Over Inheritance“(COI) stützt. Dem COI -Prinzip folgend werden Komponenten kreiert die wiederverwendbar sind und aus denen sich andere Klassen zusammensetzen lassen. Außerdem reduziert das COI-

Prinzip die Notwendigkeit Gemeinsamkeiten zwischen Klassen zu finden um sie in sinnvolle Familienbäume zusammenzufassen.

Fudge basiert in seiner grundlegenden Struktur auf Webtechnologien, namentlich HTML, CSS, JavaScript und TypeScript, Node.js und Electron in Kombination mit der Browserumgebung Chromium. Diese erlauben in Kombination mit Electron die Verwendung vom Fudge-Editor auf allen üblichen Desktopgeräten. Vollständig gepackte Applikationen in Electron sind autark und daher plattformübergreifend kompatibel. Digitale Software, die in Electron entstanden ist, kann daher auf vielen verschiedenen Endgeräten, Betriebssystemen und sogar direkt im Browser verwendet werden (Electron Dokumentation, 2019).

Besonderes Augenmerk liegt im Rahmen dieser Bachelorarbeit auf TypeScript, Node.js und Electron, da aus diesen Technologien Möglichkeiten und Limitationen während der Entwicklung entstehen können. Im Folgenden wird kurz erläutert, was die einzelnen Technologien sind, welche Möglichkeiten und Limitationen sie haben, und inwiefern sie die Entwicklung der Netzwerkkomponenten für Fudge beeinflussen.

2.2 JavaScript und TypeScript

JavaScript (JS) ist eine Skriptsprache Ursprünglich während des ersten „browser war“ entwickelt hat sich JavaScript als eine der Kerntechnologien des World Wide Web (WWW) etabliert (Brown, 2018). Eine Skriptsprache ist dabei explizit designed um auf ein existierendes System oder eine vorhandene Entität aufzubauen. Als solches folgt JavaScript dem international anerkannten ECMAScript Standard. Dieser Standard besagt welche Typen, Werte, Objekte, Eigenschaften, Funktionen und Programm Syntax sowie Semantik von JavaScript angeboten und unterstützt werden müssen.

JavaScript ist zudem eine interpretierte, keine kompilierte Programmiersprache Kompilation bedeutet, dass Quellcode von einer Software, dem Compiler, in Maschinen Code übersetzt wird. Das Ergebnis ist ein lauffähiges Programm aus Maschinen Code. Ein Interpreter dagegen liefert das Ergebnis eines Quellcodes

zurück, also das Ergebnis einer Berechnung. Allerdings muss beachtet werden, dass die Grenzen zwischen interpretierten und kompilierten Sprachen immer mehr verwischen, da mit der V8 JavaScript Engine auch JavaScript zu Maschinencode kompiliert werden kann.

Da JavaScript in jedem modernen Browser nutzbar ist, können angehende Entwickler schnell erste Erfolge erzielen. Zudem lassen sich komplexe JavaScript Abläufe Stück für Stück programmieren und zeitgleich im Browser auf ihre Funktionalität testen. Dies erleichtert den Einstieg erheblich da Experimente wenig Zeit kosten und schnell ausprobiert werden können.

Außerdem existieren viele Fallstricke anderer Programmiersprachen in JavaScript nicht, unter anderem Einschränkungen durch Typisierung und komplexe Strukturen durch multiple Vererbung, die Erfahrung und solide Planung voraussetzen.

Ein weiterer Vorteil von JavaScript ist die Codeausführung auf Seiten des Clients. Ein Client ist eine Partei, die an der Netzwerkkommunikation teilnimmt aber kein Server ist. Bei JavaScript kann ein Server lediglich den Quellcode des Programms an den Client liefern, das Programm wird jedoch im Browser des Clients ausgeführt. So lassen sich Daten bis zu einem gewissen Grad ohne zusätzliche Anfragen an den Server validieren. Dies geschieht auf Kosten der Sicherheit. Browser injizierte JavaScript Programme können Daten auslesen und verschicken und sogar Schäden an den ausführenden Geräten selbst verursachen, weswegen JavaScript in Browserumgebungen von der direkten Verwendung von Netzwerk-Sockets ausgeschlossen ist - ein Socket ist ein Zugang der auf einen bestimmten Port fokussiert ist – da aus Sicherheitsgründen keine API zur Verfügung steht. Informationen können über die Internet Protocol Adresse und einen Port eindeutig zugewiesen und einer Applikation zugeordnet werden.

Die fehlende Typisierung und der nicht existente Debugger - Software die Fehler im Quellcode erkennt und ausliest - erschweren die Wartung ungemein und führen häufig zu Laufzeitfehlern die nur durch zeitintensive Fehlersuche behoben werden können. Dazu kommt, dass die Verwendung von Objekt-

orientierten Prinzipien durch die fehlende Typisierung und ausschließlich einfache Vererbung viel Kreativität erfordert.
Dennoch erfreut sich JavaScript sehr großer Beliebtheit und ist im Jahr 2019 der defacto Standard für moderne Browser. Von über 1,3 Millarden bekannter Webseiten (July 2019 Web Server Survey (2018)) nutzen ca. 95% (Usage statistics of JavaScript as client-side programming language on websites(2019)) JavaScript.
Einige der Nachteile von JavaScript lassen sich jedoch mithilfe von TypeScript beheben und umgehen. TypeScript ist eine von Microsoft entwickeltes, typisiertes Super Set für JavaScript. Quellcode wird in TypeScript geschrieben und der TypeScript eigene Compiler wandelt den Code in funktionierendes JavaScript um. Durch die nun vorhandene Typisierung und einen TypeScript eigenen Debugger sind Fehler jedoch bereits im Quellcode leicht ersichtlich. Klassen, Objekte und ihre Funktionen lassen sich deklarieren und festlegen; sind eindeutig definiert und können während der Laufzeit nicht ungewollt verändert oder überschrieben werden (By Example, 2019). So lassen sich objektorientierte Programmierprinzipien erfolgreich umsetzen, der Quellcode ist übersichtlich strukturiert, die Wartbarkeit ist simpel und der Quellcode lässt sich mit allgemeingültigen Modellen, wie der Unified Modelling Language, darstellen und dokumentieren. Die Beliebtheit von TypeScript steigt daher stetig an und ersetzt mehr und mehr Web-Entwicklung mit reinem JavaScript.
Für die Entwicklung in Fudge wird ausschließlich TypeScript verwendet, da der resultierende, noch nicht kompilierte Quellcode sich menschenlesbar gestalten lässt, ein Debugger zur Verfügung steht und sich der Quellcode durch objektorientierte Programmierung in modulare, wiederverwendbare Komponenten aufspalten lässt. Damit kann das COI-Prinzip innerhalb von Fudge befriedigt werden.

2.3 Node.js

Node.js ist eine Laufzeitumgebung, die alle Komponenten für die Ausführung von JavaScript Code beinhaltet. Node.js basiert, wie JavaScript selbst, auf der V8 JavaScript Runtime Engine und wird von deren Compiler zu Maschinen Code übersetzt (Patel, 2018). Node.js ist durch seine autarke Natur geeignet um in Serverumgebungen Verwendung zu finden. Dies ist möglich, da Node.js eine eigene Laufzeitumgebung einpackt und stets zur Verfügung steht.

Außerdem bietet Node.js, im Gegensatz zu reinem JavaScript dessen Event-Schleife nur auf einem Thread arbeitet, die Möglichkeit asynchron auf mehreren Threads zu arbeiten und so nicht blockierend Anfragen zu verarbeiten - Ein Thread ist ein eine unabhängige Dateneinheit, die einem Kern eines Computerprozessors zugeordnet wird. So lassen sich viele Anfragen gleichzeitig statt nacheinander abarbeiten. Dies ist besonders kritisch in einer Serverumgebung da jede Anfrage nacheinander einzeln bearbeitet werden müsste und sich Anfragen dadurch gegenseitig blockieren. Durch die asynchrone Arbeitsweise kann Node.js nicht-blockierende Input/Output Requests anbieten. Node.js erfreut sich dabei steigender Beliebtheit (Usage statistics of Node.js (2019)), wodurch viele Bibliotheken und Module frei zur Verfügung gestellt werden. Über den node.js eigenen Package-Verwalter „npm“ können Pakete abgerufen, installiert und integriert werden. Dies macht Node.js kompatibel mit den meisten geläufigen Programmiersprachen, Application Programming Interfaces (APIs) und Programmen.

Node.js verpackt Netzwerkfunktionalitäten, auf die mittels TypeScript zugegriffen werden kann und Node.js ist daher für die Entwicklung von Netzwerkkomponenten in Fudge unerlässlich.

2.4 Electron

Electron basiert auf dem Projekt Atom Shell das im Jahr 2013 begann und im Jahr 2015 zu Electron umbenannt wurde (Warcholinski, 2019). Initial war es als plattformübergreifender Text Editor gedacht, in welchem der Nutzer mit Webtechnologien wie JavaScript, CSS und HTML arbeiten kann.

Electron kombiniert die Chromium Rendering Library (auch: Blink), die ein open-source Fundament für Google Chrome darstellt, mit Node.js und der V8 JavaScript Engine in einer einzigen Laufzeitumgebung. Dadurch können Programme die in JavaScript oder Node.js geschrieben wurden unabhängig vom Endgerät verwendet werden, vorausgesetzt das Electron-Paket ist für das Endgerät gedacht. Diese Programme sind plattformübergreifend kompatibel und können sowohl für Desktop, Mobile und Browserumgebungen genutzt werden. Webentwickler können mithilfe von Electron vollwertige Desktopsoftware und Applikationen für Smartphones erstellen, ohne von ihren bekannten Technologien abweichen zu müssen. Wichtig zu beachten ist, dass Electron auf einer Browserumgebung aufbaut und dementsprechend die gleichen Einschränkungen und Limitationen der handelsüblichen Browser aufweist, seien dies nun Sicherheitsbestimmungen über den Versand von Informationen oder Inkompatibilitäten mit manchen Webtechnologien.

2.5 Analyse gebräuchlicher Netzwerkprotokolle

2.5.1 ISO/OSI-Modell

Das ISO/OSI Modell (kurz: OSI-Modell) ist ein Produkt der International Organization for Standardization (ISO), dass Internettechnologien in Schichten aufteilt und standardisiert. Da die Gesamtheit der Funktionalitäten der Datenkommunikation zu komplex sind um verständlich zu sein, wurden „… gemäß dem Konzept der virtuellen Maschinen mehrere Schichten ausgedacht, um die Materie etwas übersichtlicher zu beschreiben“ (Mandl, 2018, S. 14). Das so entstandene OSI-Modell unterteilt dabei die Funktionalitäten der Datenkommunikation in Sieben solche Schichten, von denen jede Schicht mit der darunterliegenden über Schnittstellen kommunizieren kann.

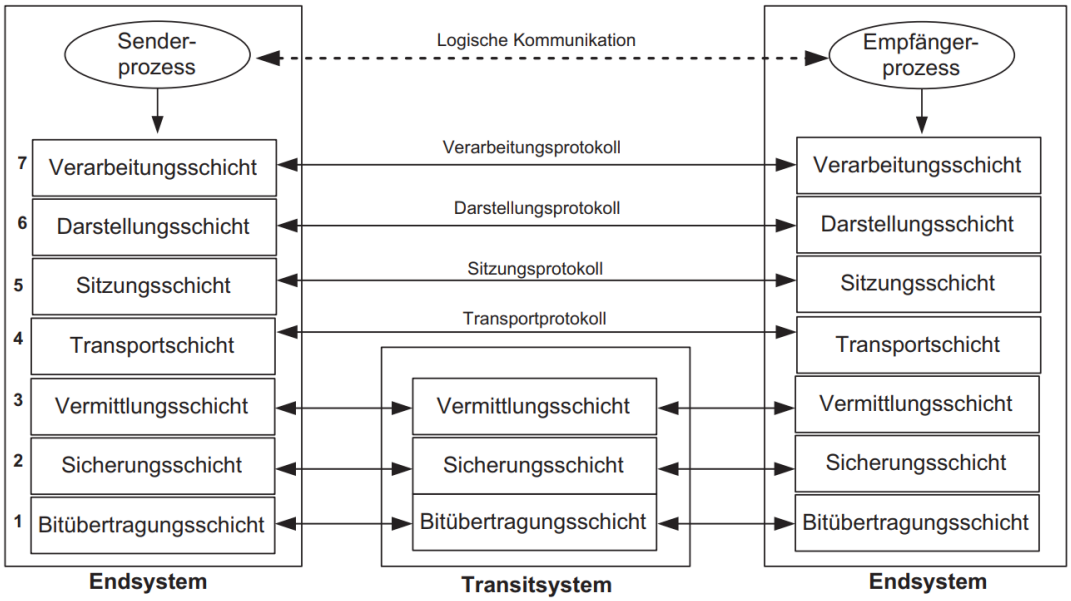


Abbildung 2: Das ISO/OSI Modell. Quelle: Mandl, 2017, S. 14

Schichten 1-4 (siehe Abbildung 2) werden „…gemeinsam als Transportsystem bezeichnet.“ (Mandl, 2018, S. 16) Wichtig hierbei ist, die Aufgaben der einzelnen Schichten im Transportsystem zu unterscheiden: Schicht 2 und Schicht 3 dienen dazu Verbindungen zwischen Rechnern herzustellen. Schicht 2 ermöglicht direkte Ende-zu-Ende Verbindungen zwischen Rechnern unter Verwendung von Protokollen wie dem Internet Protocol (IP). Schicht 3 fügt die Möglichkeit hinzu in einem Netzwerk, das heißt über mehrere Knoten, eine Ende-zu-Ende Verbindung zwischen Rechnern herzustellen. Für diese Arbeit relevant ist die vierte Schicht, die sogenannte Transportschicht. Diese „… kümmert sich um die Ende-zu-Ende-Kommunikation zwischen zwei Prozessen auf einem oder unterschiedlichen Rechnern.“ (Mandl, 2018, S. 17) Sie ist dafür zuständig Software, im Falle von Fudge sind es digitale Spiele oder interaktive Anwendungen, direkt miteinander kommunizieren zu lassen. Diese Kommunikation wird über sogenannte Packets bewerkstelligt. Packets bestehen aus einem Header mit Informationen wie Quelle, Ziel und Status und einem Körper, der die Nutzdaten beinhaltet. Der Transportschicht stehen zwei hier relevante Datenprotokolle zur Übertragung und Zuweisung dieser Packets zur Verfügung: das User-Datagram-Protocol (UDP) und das Transmission-Control-

Protocol(TCP). Transmission Control Protocol und Transmission Control Protocol/Internet Protocol (TCP/IP) sind zu unterscheiden: TCP/IP bezeichnet die gesamte Protokollfamilie der Kommunikation über Netzwerke und Internet. Der Name erschließt sich aus den zwei wichtigsten Protokollen, dem TCP und dem IP. Dementsprechend bezeichnet TCP ein spezifisches Protokoll innerhalb der TCP/IP Gruppe. Im weiteren Verlauf der Arbeit wird ausschließlich auf das spezifische TCP eingegangen. Im Folgenden wird dazu näher auf das TCP und das UDP eingegangen.

2.5.2 Transmission Control Protocol/Internet Protocol

Das Internet Protocol ist die erste vom Übertragungsmedium unabhängige Schicht des OSI-Modell. Es erlaubt Computer in logische Einheiten - genannt Subnetze - zu gruppieren und die Computer dann mit eindeutigen Adressen - den IPs - anzusprechen. Ein Beispiel für eine solche IP ist der localhost, der eigene Netzwerkanschluss, eines jeden Geräts der bei Desktop-Computern unter der IP 127.0.0.1 zu finden ist

Mit IP als zugrundeliegender Schicht im OSI-Modell ist TCP das fundamentale Protokoll des Internets. TCP ist ein zuverlässiges, verbindungsorientiertes Protokoll. Zuverlässig im Kontext der Netzwerkkommunikation bedeutet eine Garantie für die Ankunft der Packets, in der korrekten Reihenfolge, durch TCP. Sollte ein Packet verloren gehen, beispielsweise durch eine schlechte Internetverbindung, so werden alle Packets zurückgehalten bis das verlorene Packet nachgesendet und empfangen wurde. Dies macht TCP das Protokoll der Wahl, wenn Packets und ihre Daten nicht verändert werden, nicht verloren gehen und nicht dupliziert werden sollen, und zudem in der richtigen Reihenfolge eintreffen müssen.

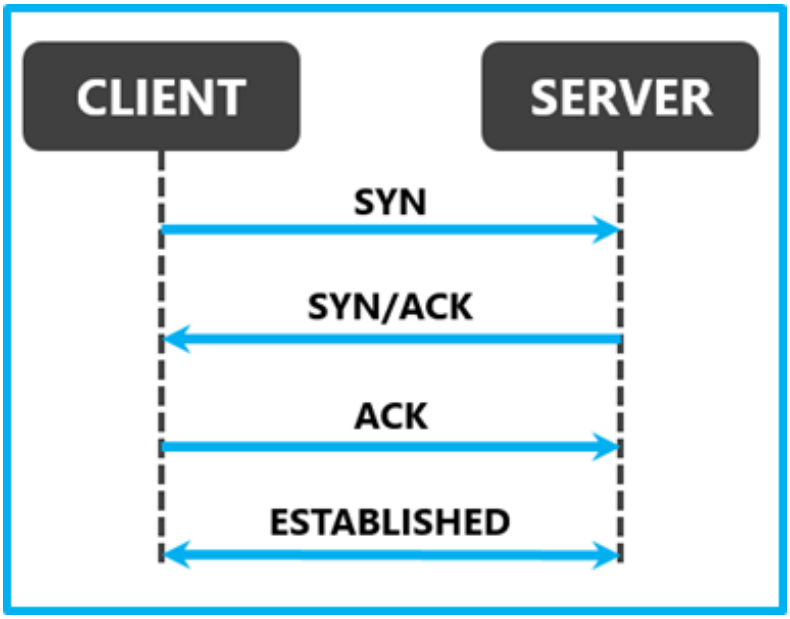


Abbildung 3: HTTP Hand Shake, Quelle: Panday (2013)

Verbindungsorientiert bedeutet zwingend: eine Verbindung zwischen Clients, per Drei Wege Handshake Verhandlung (siehe Abbildung 3) muss hergestellt werden, ehe Packets versendet werden können. Dies erfordert die Initiierung der Verbindung seitens eines Computers, eine Anerkennung der Verbindungsanfrage seitens des Empfängers und schlussendlich eine Bestätigung des Initiators. Sind diese drei Schritte erfolgreich, so wird eine Verbindung zwischen den Computern geöffnet.

Dadurch erlaubt TCP eine „… vollduplex-fähige, bidirektionale virtuelle Verbindung zwischen Anwendungsprozessen“ (Mandl, 2018, S. 56). Dies ermöglicht es allen beteiligten Kommunikationspartnern zu beliebigen Zeitpunkten und sogar zeitgleich Nachrichten zu senden und zu empfangen.

Für die Verwendung in Login-Servern oder Lobby/Matchmaking-Servern bietet sich TCP an, jedoch kostet der Verbindungsaufbau Zeit und der Datenaustausch erhält eine größere Latenz, Verzögerungen in Übertragung und Interpretation der Packets, durch die ausführlichen Header Daten.

TCP eignet sich dagegen nicht für den Datenaustausch innerhalb von Echtzeitanwendungen, die Daten zeitkritisch empfangen. Durch die zuverlässige Verbindung, die TCP erfordert, müssen Packets in der korrekten Reihenfolge

eintreffen. Geht ein Packet verloren wird der komplette Datenverkehr eingestellt bis das verlorene Packet eintrifft oder es durch Nachsenden empfangen wird. Dies nennt sich Netzwerkstau. Besonders deutlich wird dies in Echtzeitspielen oder Audioübertragungen: Geht ein Packet in einem Spiel verloren wird die Verbindung des Clients solange blockiert, bis das Packet eintrifft. In dieser Zeit findet keine Aktualisierung des Spielstandes statt und eine merkliche Verzögerung tritt ein. Ebenso ist dies bei der Übertragung von Audio oder Video der Fall. Bild und Ton werden solange angehalten bis das Packet eintrifft. So fehlt zwar keine Information die Verzögerung wird stattdessen als störendes Ruckeln wahrgenommen. Dadurch disqualifiziert sich TCP für die Kommunikation zwischen Echtzeitspielen, da keine gute Netzwerkverbindung der Clients gewährleistet werden kann. Allerdings kann TCP gut für den initialen Verbindungsaufbau zu einem Login-Server oder einer Spiel-Lobby verwendet werden, da dort zuverlässige Kommunikation wichtiger ist als zeitkritische Verarbeitung.

In der Funktionsweise ist TCP vergleichbar mit einer traditionellen Telefonverbindung: Der Anrufer äußert einen Kommunikationswunsch, der Angerufene erkennt diesen Wunsch an indem er den Anruf entgegennimmt und seinen Kommunikationsbereitschaft mit einem Gruß signalisiert. Diese erkennt der Anrufer mit einem Gegengruß an und die Kommunikation kann beginnen.

2.5.3 User Datagram Protocol

Im Gegensatz zum TCP steht das User Datagram Protocol. Es ist weder verbindungsorientiert, noch zuverlässig (Mandl, 2018, S. 106). Es eignet sich besonders für die Übertragung von Packets deren Verlust nicht systemkritisch ist, da der Empfang der Daten nicht garantiert wird. Gleichzeitig besteht keine Gefahr eine Verbindung bis zum Eintreffen eines Packets zu blockieren. Netzwerkstaus, eine Eigenheit von TCP, können so vermieden werden. Zudem ist UDP nicht Verbindungsabhängig, die Kommunikation erfolgt direkt über UDP-Kommunikationsendpunkte. Diese bestehen aus einem Tupel das zum einen die IP und zum anderen den UDP Port der Anwendung enthält. Dies erlaubt außerdem den Versand von Broadcast Nachrichten - Nachrichten die ungezielt

losgeschickt und von allen zuhörenden Endgeräten empfangen werden können. Außerdem ist der Header eines UDP Packets (auch: Datagram) wesentlich leichtgewichtiger, wodurch die Latenz bei Versand und Empfang der Nachrichten reduziert ist. Datagramme sind in der Nachrichtengröße ihrer Packets, im Gegensatz zu TCP Packets, eingeschränkt. UDP ist also besonders für kleine, rasant zu verschickenden Nachrichten geeignet.

Die Unzuverlässigkeit von UDP fordert von Entwicklern Nachrichten zu validieren. Verlorene Packets oder Packets die in der falschen Reihenfolge eintreffen müssen validiert, verworfen, neu angefordert oder geordnet werden. So können beispielsweise Zeitstempel mitgeschickt werden. Packets die später, aber mit einem früheren Zeitstempel, eintreffen werden verworfen. Diese Eigenheit kann zusätzliche Arbeit seitens der Entwickler erfordern, im Rahmen von Fudge wird eine umfassende Validierung der Packets voraussichtlich nicht nötig sein.

Das UDP ist vergleichbar mit einem Radiosender: Unabhängig davon ob es Empfänger gibt werden Informationen verschickt. Clients die ihre Radios auf den Empfang der Funkwellen justieren werden die Nachrichten empfangen, andere nicht. Gehen Teile der Übertragung verloren, so setzt die Übertragung aus und beginnt erneut sobald neue Informationen empfangen werden.

Durch diese Eigenschaften wird die Übertragung von Spielständen oder Tastendrücken in digitalen Spielen heutzutage größtenteils per UDP gehandhabt. Verlorene Packets haben selten kritische Auswirkungen auf die Systeme des Spiels, störendes Ruckeln durch Netzwerkstau werden vermieden und durch die Broadcast-Eigenschaften von UDP lassen sich mehr Clients mit weniger Aufwand als bei TCP ansprechen.

2.5.4 Evaluation UDP und TCP

Eigenschaften	UDP	TCP
Verbindungsorientiert	Nein, dadurch ist Broadcasting möglich	Ja, garantierte Packet Ankunft möglich
Flusskontrolle (auch Ordering)	Nein, dadurch kein Netzwerkstau bei verlorenen Packets	Ja, dadurch keine weitere Validierung der Packets notwendig
Staukontrolle	Nein, nicht notwendig	Ja, zusätzlich erforderlich um Netzwerkstaus sinnvoll zu verarbeiten
Headergröße	Statisch, 8 Byte	Dynamisch, 20-60 Byte
Eignung	Games, Broadcasts, Streaming, Anwendungen bei denen einzelne Packets nicht systemkritisch sind aber Geschwindigkeit nötig ist	Webbrowsing, Login, Uservalidierung, überall wo Daten unveränderlich, geordnet und lückenlos benötigt werden

Abbildung 4: Eigenschaften und Eignung von TCP und UDP. Quelle: Eigene Darstellung

Wie in Abbildung 4 zu sehen, haben beide Technologien ihre Besonderheiten, die sie in ihrem Nutzen voneinander trennen. Für die Kommunikation in Echtzeitspielen ist dabei UDP aufgrund der fehlenden Ordnung und inhärenten Unzuverlässigkeit zu bevorzugen.

TCP dagegen bietet sich besonders für Lobbyserver, Login Server und Dateitransfers an, da die Daten garantiert und in der korrekten Reihenfolge gesendet werden. Dies erspart auch die Notwendigkeit die Packets zusätzlich validieren zu müssen, da TCP diese Aufgabe selbstständig übernimmt.

2.6 Netzwerkstrukturen in digitalen Spielen

2.6.1 Client-Server

Eine der ältesten und meist genutzten Varianten der Netzwerkkommunikation in Spielen stellt die Client-Server Struktur dar (Vogl, 2018). Sie leitet sich direkt aus der modernen Internetwelt ab, in der jegliche Kommunikation über Server-Client Verbindungen stattfindet. Dabei gibt es einen dedizierten Server der als Dreh und Angelpunkt für alle Clientinteraktionen zwischen Spiel und Clients dient. Der Server arbeitet Anfragen ab, synchronisiert Spielstände (auch: Gamestates), validiert Eingaben der User und leitet relevante Informationen an die Clients weiter. Dies hat sich insbesondere in rasanten Spielen wie Multiplayer-Online-Shootern, beispielsweise Apex Legends, Overwatch und Fortnite, durchgesetzt, da die Server in einer stabilen Infrastruktur aufgebaut werden und ganztägig betrieben werden können. Qualitativ hochwertige Spielerlebnisse sind so ermöglicht. Latenz und Verbindungsabbrüche sind hierbei üblicherweise vom Client, nicht dem Server, ausgelöst. Zudem erlaubt eine Client-Server-Struktur die Kontrolle des Spielerlebnisses was bei der heutigen Beliebtheit von E-Sport fähigen Spielen an Wichtigkeit gewonnen hat. Gleichzeitig erfordert diese Art der Netzwerkkommunikation eine solide Serverinfrastruktur da ansonsten das Spielerlebnis für alle Spieler gleichermaßen leidet.

Die Client-Server-Struktur ist die häufigste Struktur der Netzwerkkommunikation, die in digitalen Spielen verwendet wird. Sie bietet, adäquate Serverinfrastruktur vorausgesetzt, die geringste Latenz bei der Verarbeitung von Daten, erlaubt es viele dutzend oder sogar hunderte von Spielern in das Spiel einzubinden und erlaubt volle Kontrolle über das Spielgeschehen, einschließlich der Kontrolle über die Clients die den Service nutzen dürfen und wie lange der Service zur Verfügung steht. Hohe initiale Kosten, Betriebskosten und Wartungskosten seitens der Betreiber, sowie das nötige Fachwissen für den Aufbau einer sogenannten Server-Farm begrenzen die Verwendung jedoch auf Studios und Publisher mit ausreichend Ressourcen. Client-Server-Strukturen finden bei unabhängig entwickelten Spielen selten Verwendung.

2.6.2 Peer To Peer – Mesh Verbindungen

Seltener genutzt wird das Peer-To-Peer Verbindungsverfahren. Hierbei gibt es keinen dedizierten Server. Stattdessen verbinden sich alle Clients miteinander und erzeugen ein sogenanntes Mesh (auch: Netz).

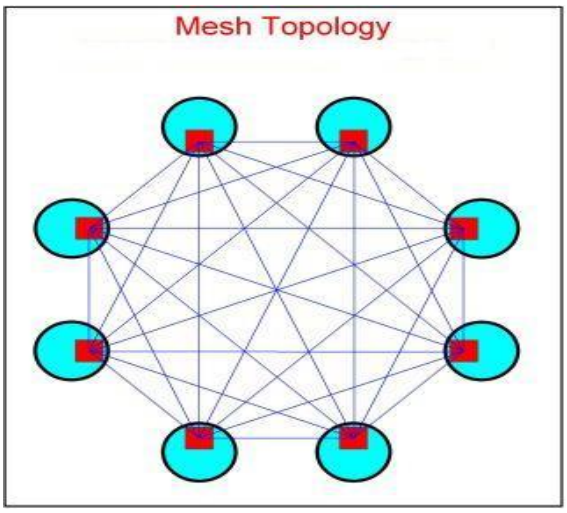


Abbildung 5: Darstellung einer Mesh Verbindung mit 8 Clients. Quelle: Mesh Topology (o.D.)

Spielstände werden zwischen den Clients synchronisiert, Eingaben gehen von den initiierenden Clients in das Netzwerk und werden von den empfangenden Clients verarbeitet. Dies erlaubt es serverlose Multiplayerspiele zu erschaffen und so die Kosten für die Bereitstellung und Wartung einer Serverinfrastruktur zu umgehen. Besonders bei kleineren Studios und einzelnen Entwicklern kann dies ein entscheidendes Kriterium sein. Peer To Peer Verbindungen sind potentiell gefährlich für das Spielerlebnis: Oftmals leiden Clients mit schwacher Computerhardware unter den großen Leistungsanforderungen einer Peer To Peer Verbindung und das Spielerlebnis leidet. Es lassen sich dadurch kaum Multiplayerspiele mit vielen Spielern realisieren. Die Organisation eines Peer To Peer Meshes erzeugt eine Vielzahl von zu verwaltenden Verbindungen für jeden Client: In jedem Mesh muss jeder Client eine Verbindung zu jedem Client außer sich selbst aufrechterhalten und verwalten. Außerdem gibt es keine zentrale Validierung der ins Netzwerk gesendeten Daten. Dadurch ist ein Peer To Peer Mesh anfällig für Dateninjektionen.

Gut aufgebaute Mesh-Netzwerke bieten allerdings die Möglichkeit eine Verbindung zu überbrücken (siehe Abbildung 6). So kann die Kommunikation trotz des Verlustes einer oder mehrerer Verbindungen durch Umleitung des Datenverkehrs aufrechterhalten werden.

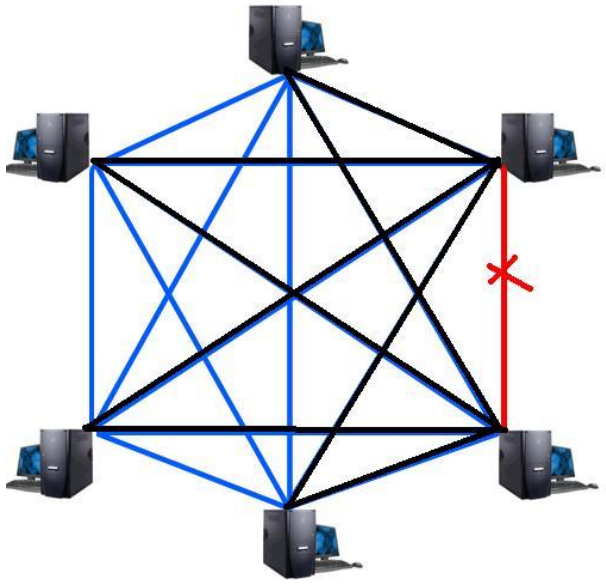


Abbildung 6: Mesh Verbindung mit einfachem Verbindungsausfall. Quelle: Mesh Topology (o.D.)

Diese Art der Netzwerkkommunikation findet hauptsächlich in kleineren, unabhängig entwickelten digitalen Spielen Anwendung, deren Spielerzahl häufig im Rahmen des Spieldesigns auf wenige Spieler beschränkt ist. Die Einfachheit der Entwicklung und der vollständige Verzicht auf die Bereitstellung einer Serverinfrastruktur sind klare Vorteile dieser Verbindungsmethode.

2.6.3 „Unechtes“ Peer To Peer

Bei unechtem Peer To Peer handelt es sich eigentlich um eine Client-Server Variante. Hierbei werden Clients üblicherweise per Matchmaking oder Lobbysystem zusammengeführt. Der Matchmaking/Lobbyserver organisiert dann die Verbindungen indem er einen Client als „Host“ auswählt - Client und Server in Kombination - und die Rolle des Servers für alle anderen Clients übernimmt. Anschließend verbinden sich alle Clients innerhalb der Lobby mit

diesem Peer-Server (übersetzt: Gleichgestellten-Server). In der Funktionsweise agiert diese Verbindungsart dann gleich wie eine normale Client-Server Verbindung. Dies erlaubt es dem Anbieter nur den Server für das Matchmaking bereit zu stellen. Die Belastung für den Matchmaking-Server ist wesentlich geringer als bei einer reinen Client-Server Struktur und Latenzen zu diesem Server spielen keine Rolle. So können selbst schwache Server-Infrastrukturen diese Aufgaben adäquat erfüllen. Auf der Seite des Spiels hat der Host, der sich selbst als Spieler mit dem lokalen Server verbindet, stets eine Verzögerung von 0ms. Dies bietet einen gewaltigen Vorteil gegenüber den anderen Clients deren Kommunikationslatenz substantiell höher ausfällt. Außerdem ist diese Verbindungsart, ähnlich wie reines Peer To Peer, anfällig für Hosts mit schwacher Hardware: Nicht nur das digitale Spiel muss dargestellt, sondern auch die Spielverwaltung im Serverteil übernommen werden. Ein unerwarteter Verbindungsabbruch seitens des gewählten Hosts erfordert zusätzliche Ausnahmebehandlungen und Vorausplanung der Entwickler. Im Falle von Verbindungsabbrüchen ist eine solche Ausnahmebehandlung üblicherweise als „Host Migration“ bekannt. Dabei wird ein neuer Host bestimmt der die Aufgaben des verlorenen Hosts übernimmt. Anschließend werden die Spielstände erneut synchronisiert und das Spiel wird fortgesetzt. Dies führt zu teils sekundenlangen Unterbrechungen. Unechtes Peer To Peer findet aufgrund dieser Eigenschaften nur noch sehr selten in größeren Spielen Verwendung. Ein namentliches Beispiel ist „For Honor“ von Ubisoft Montreal, dessen Verwendung dieser Verbindungsart immer wieder für Kontroversen gesorgt hat.[1,2]

3 Methodik

Im letzten Kapitel wurden Anforderungen und Voraussetzungen an Webtechnologien gestellt, die für die Entwicklung von Netzwerk Komponenten in Fudge erfüllt werden müssen. Im Folgenden werden mögliche Kandidaten vorgestellt und auf ihre Eignung als zugrundeliegende Technologie für die Netzwerkkomponenten geprüft.

3.1 Webtechnologien für TCP-Kommunikation

3.1.1 HTTPS/2

HTTPS/2 ist eine Weiterentwicklung des für das Internet fundamentalen HTTP/1.1 Protokolls (vgl. Ludin & Garza, 2017). HTTP dient als einfaches Anfrage-Antwort Protokoll, mithilfe dessen Anfragen an einen Server geschickt und von diesem mit den gewünschten Daten beantwortet werden können (Gorski, Lo Iacono, Nguyen, 2015, S.1) . HTTP stellt die Grundlage für die Kommunikation im Internet dar, erlaubt den Austausch von Hypertext Formaten und die Lokalisierung von Ressourcen mithilfe von URLs. HTTP liegt im OSI-Modell (siehe Abbildung 2) auf der siebten, der Anwendungsschicht. HTTP erwartet eine zuverlässige Verbindung und baut als solches auf dem TCP der vierten OSI-Model Schicht auf (siehe Abbildung 2).

HTTP/2 basiert auf dem von Google entwickelten SPDY-Protokoll und wurde im Jahr 2015 veröffentlich. Ziel dieser Weiterentwicklung ist es jedoch nicht, das alte Protokoll HTTP/1.1 abzulösen. Vielmehr soll HTTP/2 die Möglichkeiten der Internetkommunikation erweitern, sofern alle Parteien der Kommunikation mit HTTP/2 kompatibel sind.

HTTP/2 erreicht im Gegensatz zu HTTP/1.1 wesentlich bessere Datenraten und Latenzzeiten. Dies wird durch den Wechsel von textbasierten Übertragungen auf Binär Code basierte Übertragungen erreicht. Dies vereinfacht das parsen, umwandeln in unterschiedliche Datenformate, der erhaltenen Daten. Zusätzlich komprimiert das HTTP/2 Protokoll die Headerbereiche, wodurch mehrere Bytes pro Anfrage eingespart werden können und die Kombination multipler Header

zu nur einem einzelnen möglich ist. Dies spart zusätzlich Datenvolumen ein und erlaubt für schnellere Kommunikation.

Eine weitere, und womöglich die größte, Änderung gegenüber HTTP/1.1 ist die Möglichkeit des Multiplexing.

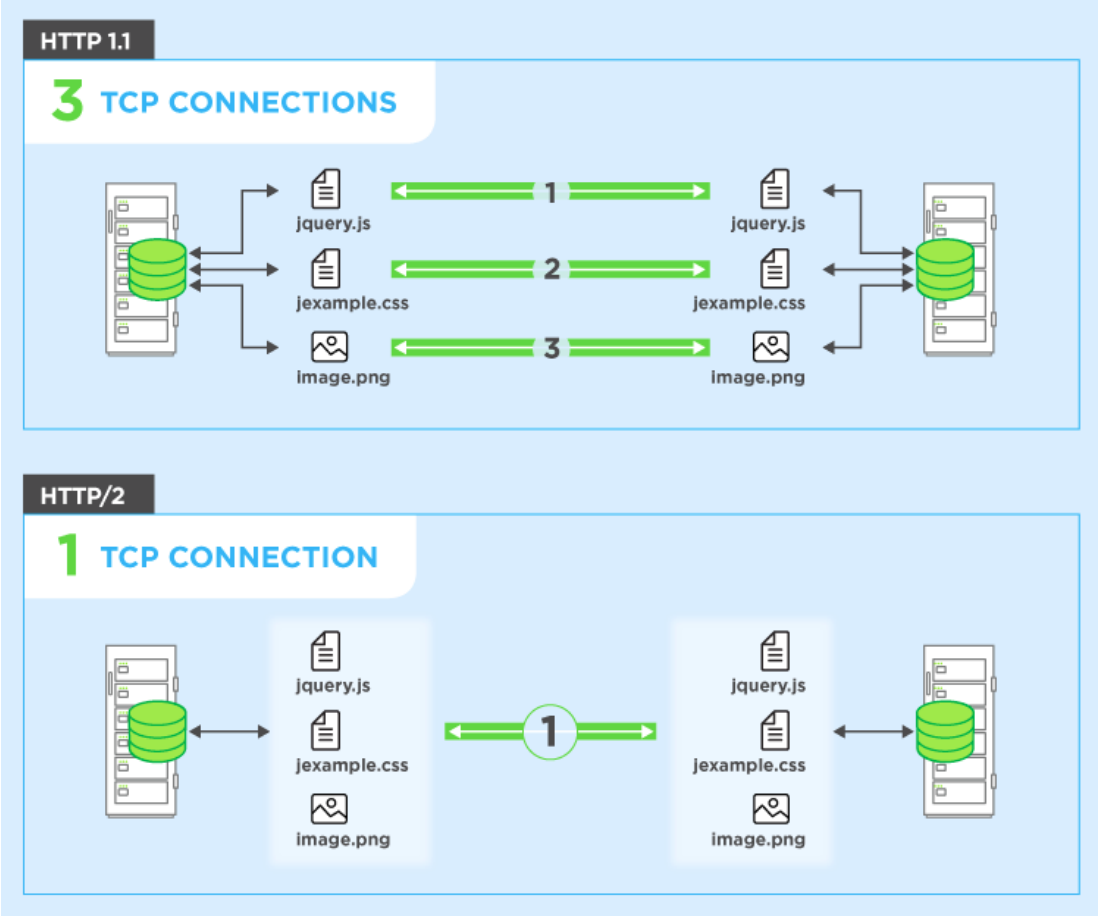


Abbildung 7: Menge der TCP Verbindungen bei drei Nachrichten über HTTP 1.1 und HTTP/2. Quelle: Factory.hr (2018)

Multiplexing bedeutet, dass über eine einzelne Verbindung, meistens TCP, mehrere Anfragen geschickt werden können ohne für jede einzelne Anfrage eine neue Verbindung herstellen zu müssen (siehe Abbildung 7). So ist nur einmal eine Verbindungsverhandlung notwendig wodurch die Anzahl der gesendeten Header von fünf, zwei seitens des Clients für den Drei Wege Handshake und drei weitere für die jeweiligen Ressourcenanfragen, auf drei reduziert wird. Dies stellt eine Dateneinsparung von 40% dar.

Trotz all dieser Einsparungen ist HTTP/2 dennoch nicht für die Verwendung in Echtzeitspielen geeignet, da auch hier eine zuverlässige und geordnete Abfolge an Packets erwartet wird: Netzwerkstau ist also immer noch ein Problem. Außerdem müssen Verbindungsabbrüche und Wiederverbindungen von Clients explizit gehandhabt werden, da HTTP/2 diese Funktionalitäten nicht nativ mitbringt.

3.1.2 WebSocket

Das WebSocket Protokoll baut auf HTTP/1.1 auf und erweitert es um die Fähigkeit bidirektionale Echtzeitdatenverbindungen zu erstellen und aufrecht zu erhalten. Das WebSocket Protokoll folgt dem HTML5 Paradigma der Einfachheit und Standardisierung und wird vom World Wide Web Consortium (W3C) offiziell unterstützt. Die Spezifikationen des Protokolls sind zudem in RFC6455 (Fette, 2011) festgelegt worden und können so jederzeit eingesehen werden.

WebSocket etabliert eine Verbindung mithilfe eines HTTP Upgrade Requests (vergleiche Abbildung 8) in einer HandShake Verhandlung (Gorski, Lo Iacono, Nguyen, 2015, S.36).

Wird die Request seitens des Empfängers bestätigt wird das verwendete Netzwerkprotokoll auf WebSocket aufgerüstet und die Kommunikation kann beginnen. Grundsätzlich kann so jedem Gerät ein Upgrade Request gesendet werden, da HTTP die Grundlage der Internet Kommunikation darstellt. Ist die Verbindung aufgestellt wird die TCP Verbindung, im Gegensatz zu HTTP/1.1, aufrecht- und offen gehalten. So bricht WebSocket aus dem HTTP üblichen Request-Response Zyklus aus und ermöglicht asynchronen und zeitunabhängigen Nachrichtenaustausch.

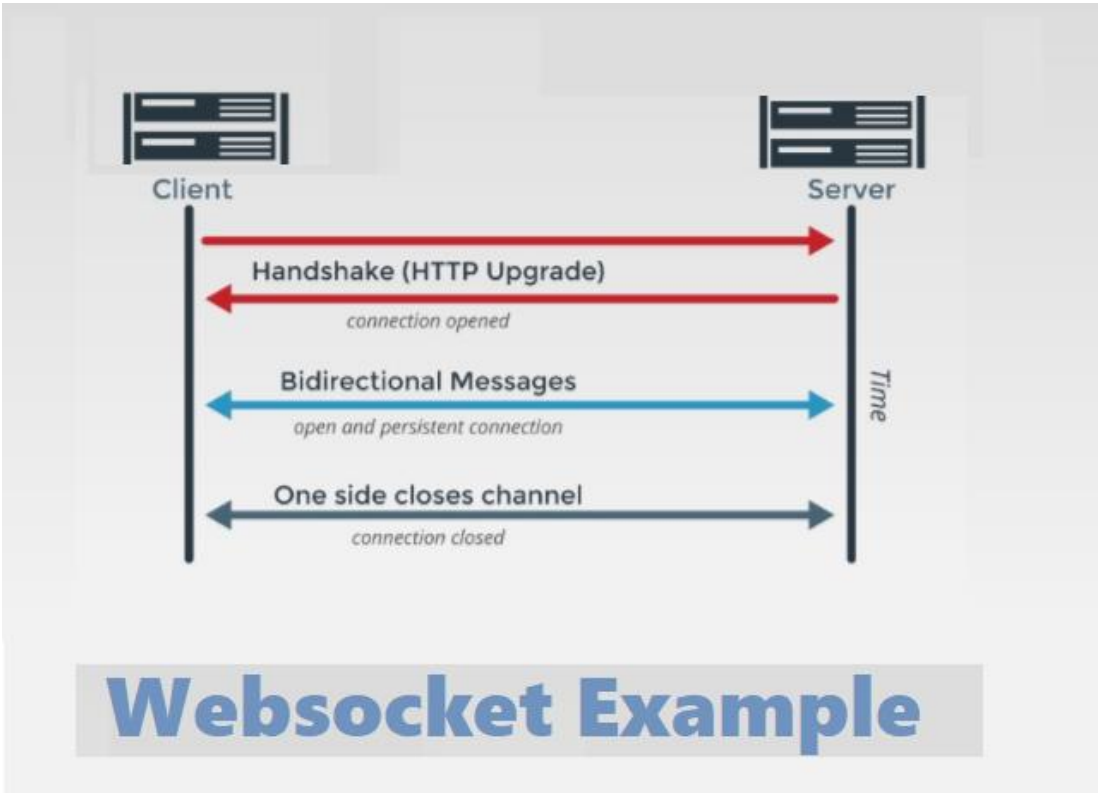


Abbildung 8: Eröffnung einer WebSocket Verbindung. Quelle: JsTutorials Team (2019)

In vielerlei Hinsicht verhält sich WebSocket wie HTTP/2, jedoch auf einem höheren Level. Das bedeutet es gibt eine Schicht der Abstraktion zwischen Socket Programmierung und dem Entwickler. So übernimmt die WebSocket API die Verwaltung des WebSocket Protokolls, stellt ereignisgetriebene Nachrichtenbehandlung zur Verfügung wie auch eine automatische Neuverbindung von Clients die einen Verbindungsabbruch erlitten haben. Somit vereinfacht die WebSocket API es eine solide Serverstruktur aufzubauen, ohne Entwicklern dabei die Kontrolle und Durchsicht zu nehmen. Zwei Eigenschaften die für die Entwicklung von Fudge wichtig sind.

WebSocket und seine API eignet sich daher gut für die Verbindung zu einem Server, da es effektiv arbeitet und die Serverstruktur nur schwach belastet, gleichzeitig aber für eine zuverlässige Verbindung sorgt durch die wichtigen Daten sicher übertragen werden können. Da WebSocket auf HTTP aufbaut und HTTP selbst von TCP abhängig ist, eignet sich WebSocket nur bedingt für die Verwendung in digitalen Spielen: Rundenbasierte Spiele können von der

Websocket API verwaltet werden und zuverlässiger Datenaustausch ist gewährleistet, für große Datenmengen die rasant verschickt und verarbeitet werden müssen ist es allerdings nicht geeignet. Zudem leidet es ebenfalls unter potentiellen Netzwerkstaus.

3.1.3 Socket.io

Socket.io ist eine JavaScript Bibliothek. Es nutzt WebSocket als zugrundeliegende Technologie und baut darauf auf. Die Bibliothek ist dabei in zwei Teile getrennt, eine API für Clients, die auf JavaScript basiert und eine API für den Server die auf Node.js basiert.

Socket.io hat die gleichen Stärken und Schwächen die WebSocket mit sich bringt, allerdings fügt Socket.io eine weitere Automations- und Abstraktionsschicht hinzu. Dadurch werden grundlegende Funktionalitäten die in der Netzwerkkommunikation üblich sind versteckt. Für erfahre Entwickler ist Socket.io ein mächtiges Tool für schnelle Prototypen und übernimmt viele der zeitaufwändigeren Arbeiten, behindert aber den Lernfluss wenn Prinzipien der Netzwerkkommunikation vermittelt werden müssen.

Da Socket.io viele grundlegende Aufgaben übernimmt, ist es für Fudge nicht geeignet. Fudge soll dazu dienen angehenden Entwicklern eben jene grundlegenden Aufgaben zu übertragen und so Lösungsansätze und Funktionsweisen zu ersinnen und zu verstehen.

3.2 UDP Kommunikation

3.2.1 Datagram-Sockets

Datagram-Sockets sind eine Art von Netzwerksocket die ursprünglich aus der Programmiersprache Java stammt. Sie basieren auf UDP und erlauben daher unzuverlässige und verbindungslose Kommunikation zwischen Server und Client. Datagram Sockets sind auf einem sehr niedrigen Level, das heißt mit sehr geringer Abstraktion und Automation, angesiedelt. Dadurch bieten sie sich für erfahrene Entwickler an die so volle Kontrolle über die Funktion ihres Quellcodes behalten. So ist es dem Entwickler überlassen sämtliche Notwendigkeiten zu

berücksichtigen und nach Bedarf zu implementieren. Durch diese Anforderung bieten sich sogenannte low-level Technologien hauptsächlich für erfahrene Programmierer an, die häufige Fallstricke vermeiden und soliden Quellcode produzieren können. Datagram-Sockets sind dementsprechend schwer zu erlernen und selbst einfache Funktionen, wie die automatische Wiederaufnahme einer Verbindung zu einem Client, können zu einer zeitintensiven Arbeit ausarten, die im Lernprozess nur bedingt zielführend ist.
Zudem kann JavaScript in einer Browserumgebung nicht direkt mit Datagram Sockets arbeiten. Dies ist auf Sicherheitsentscheidungen seitens der Browserentwickler zurückzuführen, die zuvor bereits erwähnt wurden.
Zu guter Letzt sind Datagram-Sockets zwar standardisiert, aber für die Verwendung in selbständigen Applikationen vorgesehen. Üblicherweise werden sie in Java Programmen Verwendung finden, nicht aber in Browserumgebungen. Dementsprechend bietet sich die direkte Verwendung von Datagram-Sockets nicht für Fudge an, da Electron auf einer Browserumgebung basiert und mit Fudge produzierte digitale Spiele und Anwendungen auch in einer reinen Browserumgebung verwendbar sein müssen.

3.2.2 WebRTC

Ursprünglich von der Firma Global IP Solutions entwickelt ist WebRTC, der volle Name lautet Web Real-Time Communication, im Jahre 2010 in den Besitz von Google übergegangen und wird seit 2011 von einer Arbeitsgruppe der W3C standardisiert. Die Standardisierung findet mit Unterstützung von Google inc, der Mozilla Foundation und Opera Software ASA statt. Dies zeigt eindeutig den Bedarf nach einer Lösung für die UDP Kommunikation im Browser.
WebRTC ist ein offener Standard und eine Sammlung von Protokollen und APIs. Diese Sammlung soll die Netzwerk Kommunikation in Echtzeit zwischen Client-Client Verbindungen in Browserumgebungen ermöglichen. Es basiert unter anderem auf HTML5 und JavaScript. Die Datenübertragung erfolgt über sogenannte DataChannels (Datenkanäle) die mithilfe des neu eingeführten JavaScript Session Establishment Protocol (JSEP) ausgehandelt und etabliert werden.

Für die Verhandlung nutzt WebRTC sogenannte STUN und TURN Server, die entweder bereitgestellt werden müssen oder im kleinen Rahmen von Google verwendet werden können. Ein STUN Server liefert einem verhandelnden Client seine globale IP-Adresse zurück, der TURN Server dient als Rückfallmöglichkeit, sollte eine direkte Peer To Peer Verbindung nicht möglich sein. Beide Server müssen vor der Verhandlung festgelegt werden.

Die Verhandlung selbst wird über einen Webserver abgewickelt. Dieser leitet die notwendigen Informationen an den gewünschten Peer weiter.

Zuerst steht das RTCOffer (auch: Offer): Es bereitet den Peer auf die Verhandlung vor und liefert gleichzeitig das Session Description Protocol (SDP) – Informationen über die verwendeten Datenströme – mit. Diese kann der „Angerufene“ Client um eine gültige Antwort zu generieren, die dem „Anrufer“ zugesendet wird. Der „Anrufer“ prüft diese Antwort und wenn sie gültig ist, beginnt der Austausch der Interactive Connectivity Establishment (ICE) Candidates – Verbindungsmöglichkeiten, die den Clients zur Verfügung stehen. Wurde ein gültiger ICE-Candidate etabliert, so ist die Verhandlung abgeschlossen und die Verbindung steht (siehe Abbildung 9).

WebRTC erlaubt dabei nicht nur den Austausch von beliebigen Daten über Datenkanäle, sondern erlaubt es auch über die gleiche Verbindung mithilfe weiterer Kanäle Video- und Audiodaten zu übertragen. Zudem ist WebRTC auf einem höheren Level angesiedelt als Datagram-Sockets, wodurch einige der grundlegenden Aufgaben von der API selbst übernommen werden, ohne jedoch die Grundfunktionen zu verstecken.

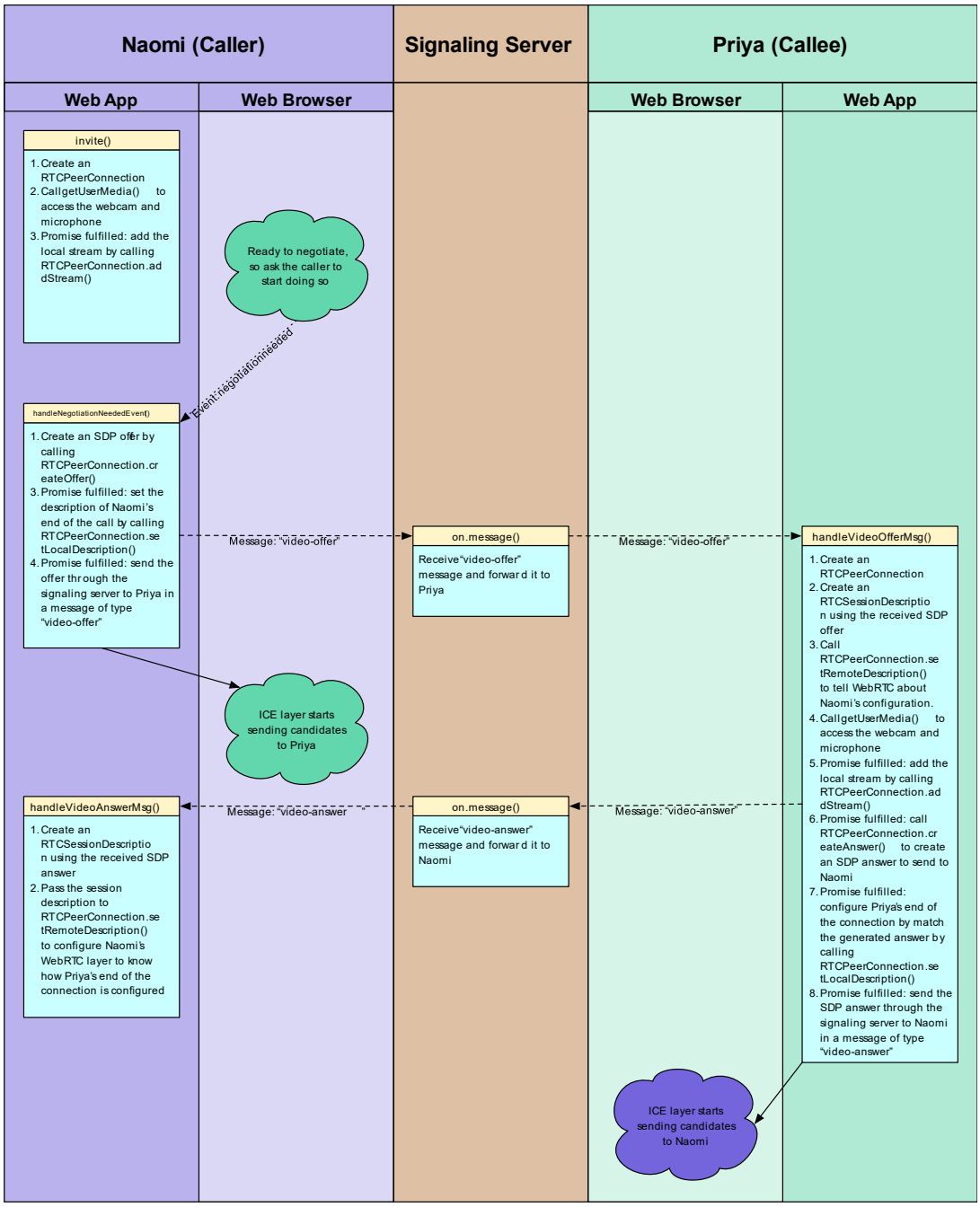


Abbildung 9: Verhandlungsablauf zur Etablierung einer WebRTC Peer Verbindung. Quelle. Signaling and video calling (o.D.)

Verbindungen sind reine Peer-To-Peer Verbindungen, das heißt jeder Kandidat muss sich mit jedem anderen Kandidaten zu einem zuvor erwähnten Mesh-Netzwerk verbinden (siehe Abbildung 5). Durch die Kombination mit Electron lässt sich ein WebRTC Client-Server-Modell realisieren: Der Server läuft innerhalb einer Electron Umgebung und jeder Client erstellt eine Peer-To-Peer

Verbindung mit dem Server selbst, nicht aber mit anderen Clients, die mit dem Server verbunden sind. So kann der Server sämtliche Client-Verbindungen bei sich zusammenführen und verwalten.
WebRTC sticht daher als bester Kandidat für die UDP-Kommunikation in digitalen Spielen hervor. Nicht nur können beliebige Datenformate verschickt werden, beispielsweise JavaScript Object Notation (JSON), Extensible Markup Language (XML) oder Binärdaten, zudem ist es sehr einfach Videostreams und Audiostreams zu versenden und zu empfangen, wodurch spielinterne Sprach- und Bildkommunikationssysteme entwickelt werden können. Außerdem ermöglicht die Kombination von Electron und WebRTC es sämtliche zuvor erwähnte Verbindungsstrukturen von digitalen Spielen abzudecken: Peer-To-Peer, unechtes Peer-To-Peer und sogar Client-Server Strukturen.

3.3 Evaluation und Auswahl Webtechnologien

Bei der UDP-Kommunikation fällt die Auswahl der zu wählenden Technologie sehr leicht: Außer WebRTC gibt es kein standardisiertes und universell unterstütztes Format für die UDP-Kommunikation in Browserumgebungen. Durch Sicherheitsbedenken ist JavaScript ohne einen Mittelmann nicht in der Lage UDP-Packets zu versenden, daher bietet es sich auch nicht an ein eigenes Protokoll zu entwickeln Außerdem ist WebRTC zukunftssicher und von den großen Browseranbietern standardisiert.
Im Bereich der TCP-Kommunikation gibt es mehrere Kandidaten zur Auswahl, namentlich WebSocket und HTTP/2. WebSocket besticht dabei durch Popularität, eine sehr ausführliche Dokumentation und einfache Verwendung. Es ist dadurch besonders geeignet für unerfahrene Entwickler. HTTP/2 bietet ähnliche Funktionalitäten wie WebSocket an, namentlich die bi-direktionalität und Multiplexing, ist aber auf einem niedrigeren Level angesiedelt und dadurch in der Anwendung aufwändiger.
Beide Technologien sind als zukunftssicher zu betrachten, HTTP/2 als Neuentwicklung, die von den Browseranbietern vorangetrieben wird,

WebSocket als stabiles und etabliertes Protokoll für das viele Bibliotheken existieren.

Die Entscheidung für die TCP-Kommunikation ist daher Ermessenssache und fällt aufgrund der ausführlichen Dokumentation und dem geringen, aber hilfreichen Level, an Automation auf WebSocket.

Netzwerkteilnehmer	**Baut auf, auf:**
Server	Electron + Node.js + TypeScript
Client	Electron + TypeScript
TransportMethode	**Gewählte Technologie**
TCP	WebSocket
UDP	WebRTC

Abbildung 10: Gewählte Technologien für die Entwicklung der Fudge Komponenten. Quelle: Eigene Darstellung

4 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Entwicklung vorgestellt. Dazu gehört die Festlegung von Interfaces zur allgemeingültigen Strukturierung der Komponenten, die Implementierung von unterstützenden Klassen, sowie eine beispielhafte und funktionstüchtige Implementation jeder zuvor vorgestellten Netzwerkstruktur die in einem Spiel Anwendung finden könnte.

Der Zugriff auf die einzelnen Komponenten untereinander erfolgt über das TypeScript eigene „Barrel“ System: Jeder Ordner der Projektstruktur erhält ein eigenes „Barrel“ – eine TypeScript Datei mit Namen index.ts die alle gewollten Module reexportiert. So können alle in der index.ts definierten Module mit nur einem einzigen Import-Statement eingebunden werden. Um alle Network Module einzubinden liegt im Root Ordner des Projekts eine „ModuleCollector.ts“, die sämtliche index.ts Dateien aus den Ordner reexportiert. Das Barrel-System erlaubt außerdem eine moderne Art von Namespacing, da die Barrelgruppe mit einem selbstgewählten Namen importiert werden kann. Es wurde „FudgeNetwork“ gewählt.

Die Ordnerstruktur des Projekts ist im Anhang zu finden.

4.1 Client Manager Interfaces

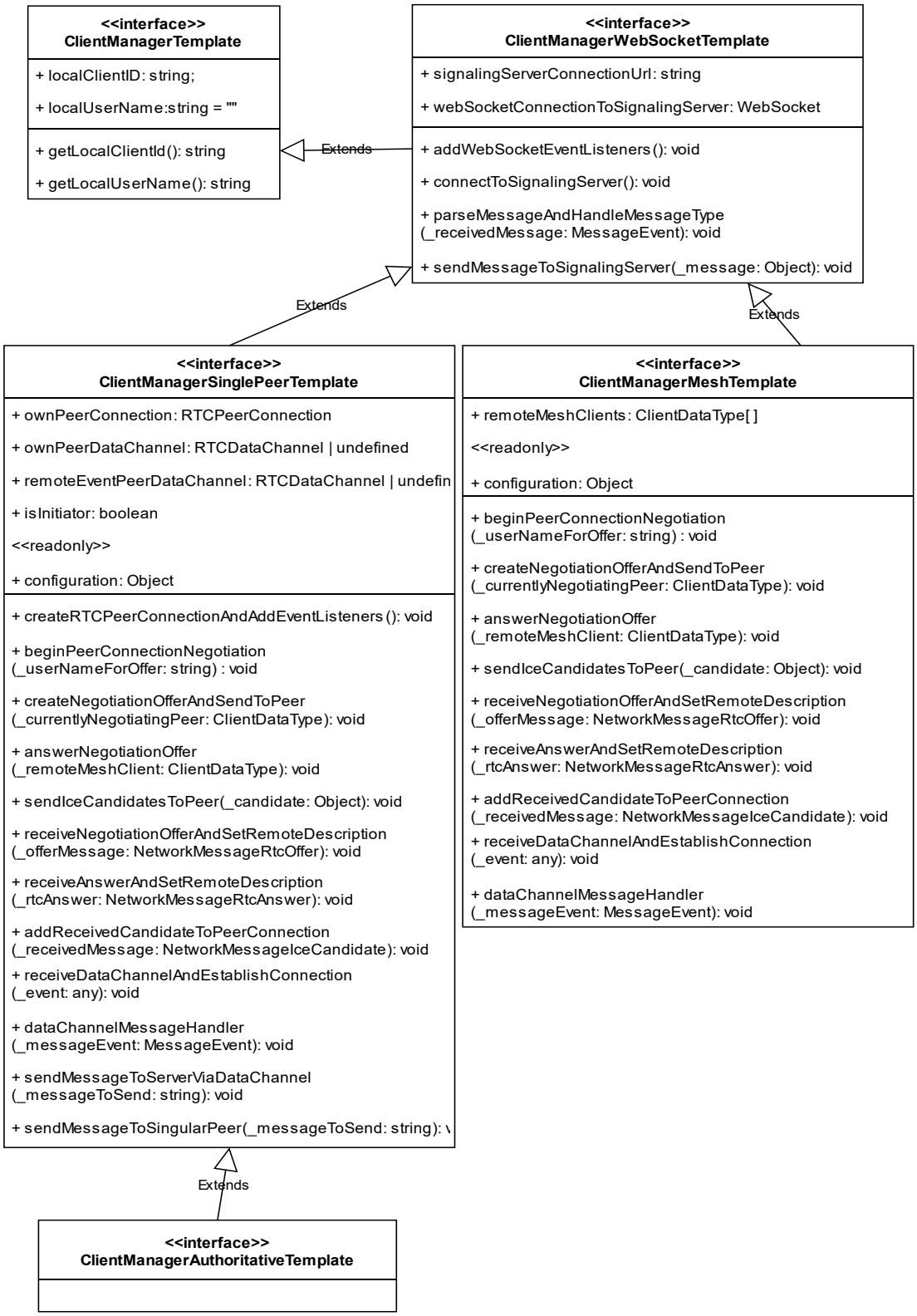


Abbildung 11: UML-Diagramm der Client Manager Interfaces. Quelle: Eigene Darstellung

Im Rahmen der Netzwerkkommunikation fungieren die Client Manager Klassen als Dreh und Angelpunkt der Clients - derer Netzwerkteilnehmer, die sich in irgendeiner Form im Netzwerk beteiligen und keine Server sind. Sie lassen sich in zwei Kategorien unterteilen: Diejenigen Client Manager, die ausschließlich über WebSocket-Verbindungen kommunizieren, und diejenigen Client Manager die zusätzlich zu ihrer WebSocket Verbindung eine WebRTC Verbindung etablieren können und sollen.
Für jede Netzwerkstruktur steht ein eigenes Interface zur Verfügung:

- Client Manager WebSocket - reine WebSocket Verbindungen
- Client Manager Mesh Network - Mesh Netzwerke über WebRTC
- Client Manager Single Peer - Peer To Peer Verbindungen zwischen zwei Clients
- Client Manager Authoritative – Peer To Peer zwischen Client und dediziertem Server

Klassen, die diese Interfaces implementieren sind für die Repräsentation eines Clients im Netzwerk zuständig. Muss eine netzwerkgetriebene Funktionalität umgesetzt werden, muss diese über einen Client Manager abgewickelt werden. Ihr Aufbau ist als UML-Diagramm dokumentiert (siehe Abbildung 11) und wird nachfolgend erläutert.

4.1.1 Client Manager

Interfaces, im Deutschen “Schnittstellen“, definieren was eine Klasse können muss, aber nicht wie sie es tut. Als solches dienen Schnittstellen als eine Art Bauplan, nach denen sich Klassen die diese Interfaces implementieren richten müssen.
Das *ClientManagerTemplate* ist ein solches Interface. Es definiert auf dem untersten Level welche Variablen und Funktionen ein Client im Netzwerk zur Verfügung stellen muss. Vom *ClientManagerTemplate* werden weitere Interfaces für die verschiedenen Netzwerkstrukturen abgeleitet (siehe beispielhaft Abbildung 23).

Die Variable, die ein Client Manager in jedem Fall bereitstellen muss, ist die *localClientID* – eine einzigartige Folge von Zeichen die den Client im Netzwerk eindeutig identifiziert. Optional kann der Client Manager einen *localUserName* bereitstellen. Zu Beidem gehören jeweilige Getter Methoden.

4.1.2 Client Manager WebSocket

Das *ClientManagerWebSocketTemplate* legt die benötigten Variablen und Funktionen für einen reinen WebSocket Client fest. Es legt fest welche Funktionen ein Client ohne Abhängigkeit von WebRTC zur Verfügung stellen muss. Es leitet sich aus dem *ClientManagerTemplate* ab (siehe Abbildung 11).
Die Implementation dieses Interfaces ist notwendig in Applikationen für die zuverlässige Datenübertragung via TCP gewünscht ist. Durch die Verwendung von WebSocket ist sie außerdem für schnelle Prototypen geeignet, beispielsweise um ein Konzept zu testen (Proof of Concept). Die Verhandlung (vergleiche Abbildung 8) erfolgt via HTTP-Handshake in Folge dessen eine TCP-Verbindung etabliert und aufrechterhalten wird, über die anschließend WebSocket Nachrichten verschickt werden können.

Erforderlich für einen WebSocket Client ist es EventListener – Funktionen, die auf die Ausführung eines Events warten und von diesem aufgerufen werden – festzulegen. Für eine WebSocket Verbindung sind das die Events „open“ – Eine Verbindung wurde eröffnet - „close“ – eine Verbindung wurde geschlossen - und „message“- eine Nachricht wurde über die Verbindung erhalten. Zusätzlich muss der Client sich mit dem WebSocket Server verbinden können, sowie eine Möglichkeit haben erhaltene Nachrichten – üblicherweise als JSON-String verschickt – in NetworkMessage Objekte zu parsen. Als letzte erforderliche Funktion muss der Client eine Nachricht an den WebSocket Server schicken können.

4.1.3 Client Manager Mesh

Das *ClientManagerMeshTemplate* legt fest, wie ein Client Manager für die Verwendung in einem Mesh Network aufgebaut sein muss.

Zusätzlich zu den Daten und Funktionalitäten die ein normaler WebSocket Client Manager benötigt, kommen nun auch WebRTC Funktionen hinzu. Ein *ClientManagerMesh* muss in der Lage sein, die Verhandlung für die Etablierung einer WebRTC Verbindung zu beginnen (siehe Abbildung 12), ein Angebot zu versenden, eine Antwort zu empfangen und abschließend seine ICE-Candidates an den Empfänger zu senden und die des Empfängers anzunehmen.

Auf der anderen Seite muss der Client Manager in der Lage sein ein Angebot zu erhalten, es zu verarbeiten, eine Antwort vorzubereiten, den Austausch von ICE-Candidates zu verwalten und den etablierten Datachannel mithilfe des ‚datachannel' Events zu erhalten (siehe Abbildung 12).

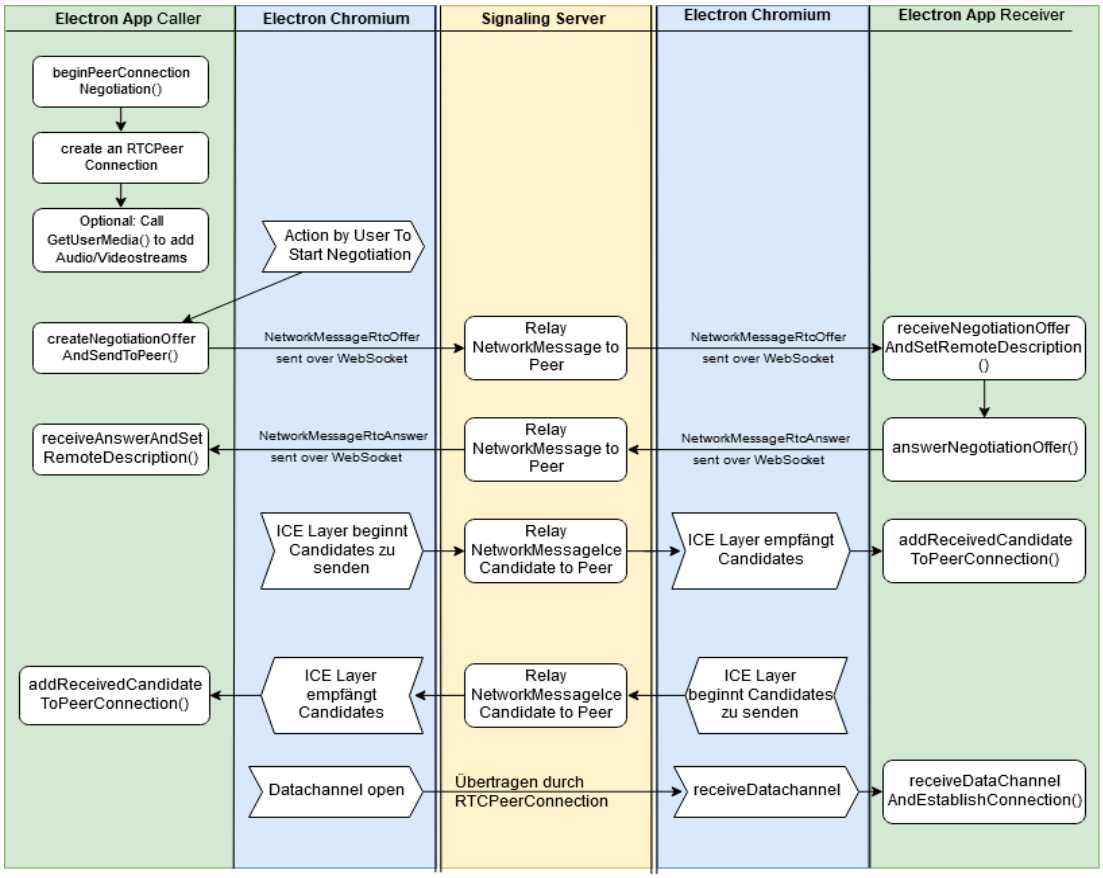


Abbildung 12: Ablauf einer WebRTC Verhandlung in Fudge. In Anlehnung an: Signaling and video calling (o.D.)

4.1.4 Client Manager Single Peer

Zu einem großen Teil überlappen sich die Interfaces des *MeshTemplate* und des *SinglePeerTemplate*, allerdings fallen sämtliche Listen weg und werden stattdessen durch einzelne Instanzen von RTCPeerConnections ersetzt. Außerdem muss dem Client Manager ein boolean ‚isInitiator' zur Verfügung stehen. Es muss also bei der Implementation zwischen Caller und Receiver unterschieden werden, um die passenden Eventhandler auf die RTCConnection zu setzen. Der Caller kann auf das „datachannel" Event verzichten (siehe Abbildung 12), der Receiver dagegen nicht. Zusätzlich muss unterschieden werden welcher RTCPeerConnection und RTCDataChannel zur Kommunikation angesprochen werden muss.

4.1.5 Client Manager Authoritative Peer

Für den Client in einer Client-Server Struktur besteht kein Unterschied zwischen einem Peer – einem gleichgestellten Netzwerclient – und einem Authoritative Server. Der Peer verbindet mit einem einzigen Netzwerkclient, dem Server. Wird die Komponente weiter ausgebaut sollte allerdings zwischen Single Peer und Authoritative Peer unterschieden werden können.

4.2 Fudge Server Interfaces

In der Netzwerkkommunikation von Fudge stellen die Fudge Server Interfaces das Gegenstück zu den Client Manager Interfaces dar: Sie definieren die notwendigen Variablen und Funktionen die verschiedene Serverstrukturen zur Verfügung stellen müssen. Namentlich wird unterschieden zwischen:

- WebSocket Server
- WebRTC Signaling Server
- Mesh Network Signaling Server
- Authoritative Signaling Server

Das WSServer Interface dient als Grundlage aller Server, da immer wenigstens eine WebSocket Verbindung ermöglicht werden muss. Andernfalls ist keinerlei Kommunikation über das Netzwerk möglich.

Komponenten die als Netzwerkserver für Client Manager agieren sollen, sollten ein Interface implementieren, welches sich vom WSServer Interface ableitet.

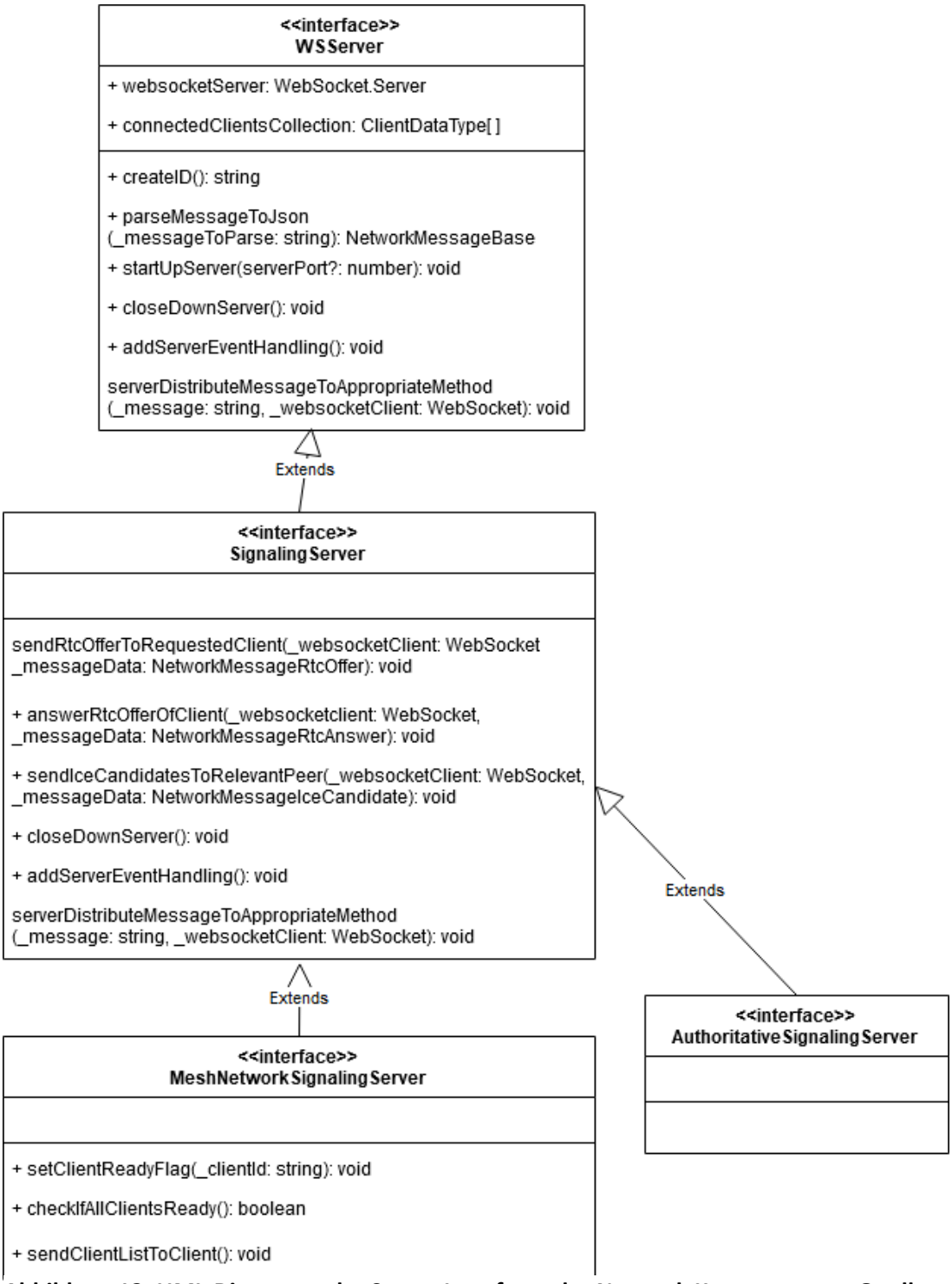


Abbildung 13: UML-Diagramm der Server Interfaces der Network Komponenten. Quelle: Eigene Darstellung

4.2.1 WSServer Interface

Das *WSServer* Interface definiert die notwendigen Funktionen für einen WebSocket Server (Basisserver) im Kontext Fudge.
Es muss ermöglicht werden verbundene Clients in einer Liste zusammenzufassen und ihnen eine einzigartige ID zuweisen zu können. Außerdem muss ein Basisserver Start- und Stoppbar sein, sowie Nachrichten zu den korrekten NetworkMessage Objekten zu parsen und diese anschließend korrekt zu verarbeiten.

4.2.2 Signaling Server Interface

Signaling Server ist ein Ausdruck, der sich speziell für WebRTC etabliert hat. Es handelt sich dabei um einen Server, der die WebRTC Verhandlung (siehe Abbildung 12) ermöglichen kann, also RTCOffer, RTCAnswer und ICECandidates an die gewünschten Clients weiterleiten kann. Der Signaling Server arbeitet zu diesem Zweck mit WebSocket, da diese Nachrichten zuverlässig ankommen müssen. Datenverluste können zum Fehlschlagen der Verhandlung führen. Ist die Verhandlung abgeschlossen kann die Verbindung des Servers zu den Clients beendet werden.

4.2.3 Authoritative Signaling Interface

Dieses Interface unterscheidet sich zum Zeitpunkt dieser Arbeit nur durch die schlussendliche Implementation im Quellcode. Es muss die gleichen Funktionalitäten liefern wie ein normaler Signaling Server, muss aber die Verbindungen an einen Fudge Server Authoritative Manager weiterleiten.

4.2.3 Mesh Network Interface

Das *MeshNetworkInterface* leitet sich aus dem Signaling Server Interface ab (siehe Abbildung 13) und garantiert so die Signaling Funktionalitäten für WebRTC. Dazu kommt die Fähigkeit den Bereitschaftsstatus von MeshClients zu unterscheiden und zu setzen. Dies ist wichtig, da ein Mesh Netzwerk zwischen allen partizipierenden Clients aufgebaut werden muss. Es ist also an den Clients ihre Bereitschaft dafür zu signalisieren.

Haben alle Clients ihre Verbindungsbereitschaft deklariert, beginnt der Verbindungsaufbau der beispielhaft mit fünf Clients in Abbildung 14 dargestellt wird. Diese Verbindungsmethode ist nur durch die maximal verwaltbare Anzahl an RTCPeerConnections limitiert und skaliert linear.

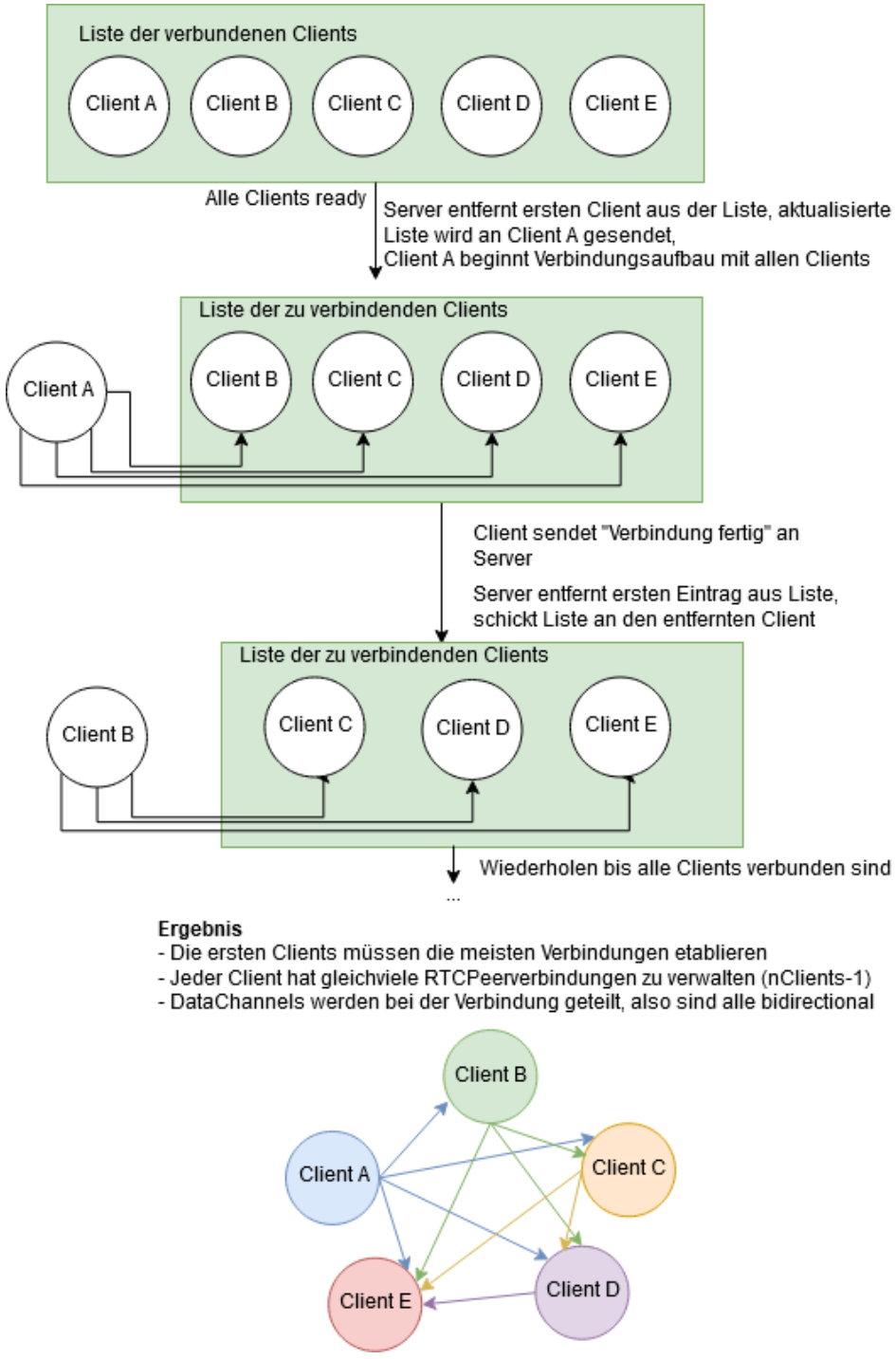


Abbildung 14: Ablauf des Aufbaus eines Mesh Network in Fudge. Quelle: Eigene Darstellung

4.3 Network Message Interfaces

Die Network Message Interfaces sollen Informationseinheitlichkeit sicherstellen. Besonders bei den Network Messages ist dies von entscheidender Bedeutung, da nur so eine standardisierte Weiterverarbeitung durch Server oder Clients gewährleistet werden kann. Netzwerknachrichten sind eine Datenklasse, die keine eigenen Funktionen besitzt.

Zwei Arten von Nachrichten werden unterschieden:

- Network Messages - Nachrichten für WebSocket Verbindungen
- PeerMessages – Nachrichten für WebRTC Datachannel

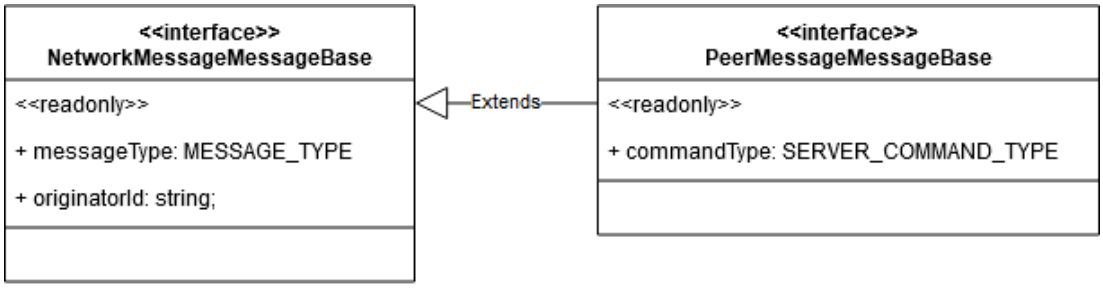


Abbildung 15: UML-Diagramm des NetworkMessage und PeerMessage Interface. Quelle: Eigene Darstellung

Somit kann zwischen Nachrichten für die Etablierung von Verbindungen und Nachrichten für den Austausch von Applikationsrelevanten Informationen unterschieden werden. Sie unterscheiden sich dabei durch einen zusätzlichen Enumerator *SERVER_COMMAND_TYPE* (siehe Abbildung 17) der Peer Messages eigen ist und gesondertes Serververhalten erlaubt.

4.3.1 Network Message Message Base

Entscheidendes Merkmal einer Netzwerknachricht ist es, einen Absenderidentifikator zu haben und einen Type zu besitzen, der die Art der Nachricht festlegt. So können Nachrichten eindeutig zugeordnet und die Verarbeitung gezielt kontrolliert werden.

4.3.2 Peer Message Message Base

Zusätzlich zu dem Identifikator und dem Type kommt bei einer Peer Message ein Enumerator für den *SERVER_COMMAND_TYPE* hinzu. Dies erlaubt die gesonderte Behandlung von Clientbefehlen die dem Server zugedacht sind. Ein Beispiel dafür ist das Kommando ein Objekt auf dem Server zu erschaffen, dem Objekt eine eigene ID zuzuweisen und es dem Absender des Befehls zuzuordnen.

4.4 Datenklassen

Bei den Datenklassen handelt es sich um Klassen die Daten entweder strukturieren oder für die Verwendung sammeln. Sie sind für die Struktur wichtig, könnten aber auch ersetzt oder anders implementiert werden.

4.4.1 ClientDataType

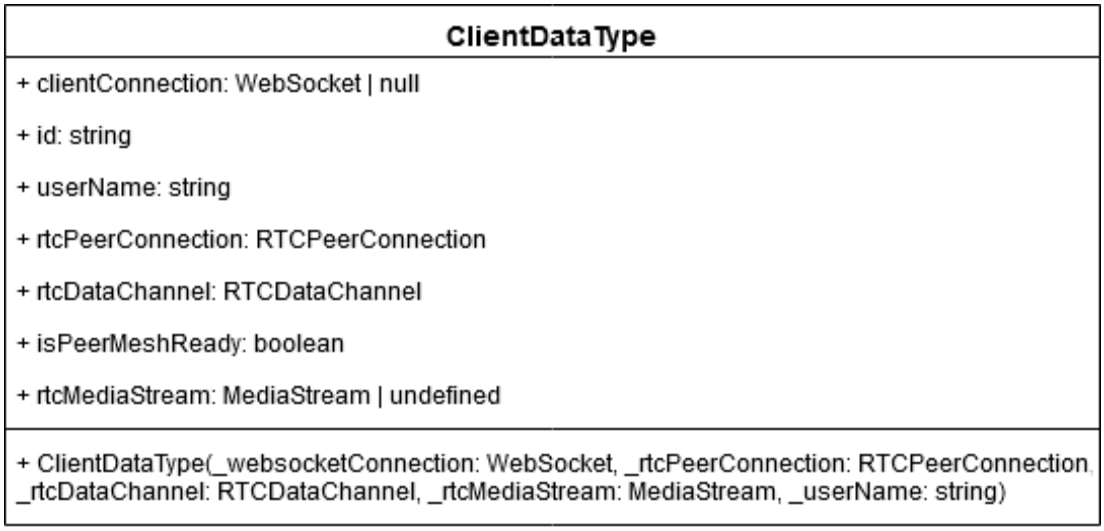


Abbildung 16: UML-Diagramm des ClientDataType. Quelle: Eigene Darstellung

Die in Abbildung 16 dargestellte Klasse erlaubt es Clients im Netzwerk zu definieren und alle relevanten Informationen zentral zu speichern, beispielsweise auf dem Server. So können Connections, Id und UserName in einem einzigen Objekt bereitgestellt werden.

4.4.2 Enumerators

Zum Abschlusszeitpunkt der Arbeit gibt es zwei Enumerator Arten:

- *MESSAGE_TYPE* – Denunziert die Art und Struktur einer Nachricht
- *SERVER_COMMAND_TYPE* – Denunziert den Befehl, den eine Nachricht beinhaltet

Die Enumeratoren werden als *readonly* Property der Network Message Interfaces interpretiert. Sie werden in der Klassendeklaration festgelegt, nicht über den Konstruktor. So ist jede Nachricht eindeutig markiert und ihr Type oder Command können während der Laufzeit nicht verändert werden.
Nicht alle Enumeratorwerte finden Anwendung; Diese dienen als Anhaltspunkte für mögliche Erweiterungen, die vom Entwickler vorgenommen werden können.

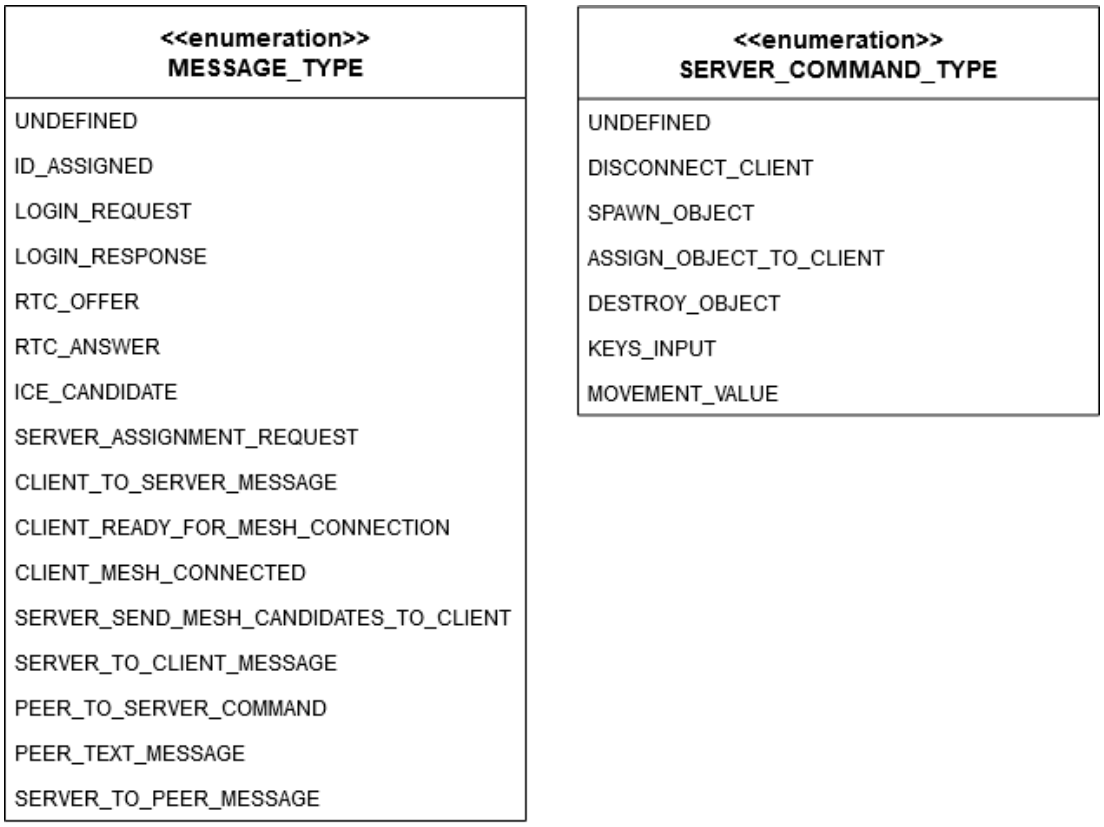


Abbildung 17: UML-Diagramm der globalen Enumerator Typen. Quelle: Eigene Darstellung

4.4.3 UiElementHandler

Diese statische Klasse dient als zentrale Sammelstelle für HTML-Elemente. So ist es dem Client und dem Server gleichermaßen möglich auf UI-Elemente zuzugreifen und ihre Events abzufangen oder auszulösen. In der ersten Iteration dient der UIElementHandler vor allem dazu, alle relevanten HTML-Elemente zu finden und so die Beispiele funktiontstüchtig zu machen. Außerdem fügt er für jeden neuen Client der dem Authoritative Server beitritt ein steuerbares Quadrat ein.

Der UIElementHandler ist nicht systemkritisch und kann gelöscht oder ersetzt werden. Funktionen der Netzwerkkomponenten müssen dann im Quellcode, nicht über HTML-Elemente, angesprochen werden.

<<static>> UiElementHandler	
+ electronWindow: Document	+ authoritativeServerelements: HTMLElement
+ signalingSubmit: HTMLElement	+ authoritativeServerStartSignalingButton: HTMLElement
+ signalingUrl: HTMLInputelement	+ authoritativeServerStopSignalingButton: HTMLElement
+ loginNameInput: HTMLInputElement	+ authoritativeServerBroadcastButton: HTMLElement
+ loginButton: HTMLElement	+ authoritativeSwitchToServerOrClientModeButton: HTMLElement
+ msgInput: HTMLInputElement	+ authoritativeServerMovingDiv: HTMLElement
+ chatbox: HTMLInputElement	+ authoritativeClientElements: HTMLElement
+ sendMsgButton: HTMLElement	+ authoritativeClientSignalingUrlInput: HTMLInputElement
+ connectToUserButton: HTMLElement	+ authoritativeClientConnectToServerButton: HTMLElement
+ usernameToConnectTo: HTMLInputElement	+ authoritativeClientLoginNameInput: HTMLInputElement
+ disconnectButton: HTMLElement	+ authoritativeClientLoginButton: HTMLElement
+ startSignalingButton: HTMLElement	+ authoritativeClientChatArea HTMLElement
+ peerToPeerHTMLElements: HTMLElement	+ authoritativeClientMessageInput: HTMLInputElement
+ authoritativeElements: HTMLElement	+ authoritativeClientSendMessageButton: HTMLElement
+ switchModeButton: HTMLElement	+ meshNetworkClientOrServerSwitch: HTMLElement
+ stopSignalingServer: HTMLElement	+ meshServerElements: HTMLElement
+ broadcastButton: HTMLElement	+ meshServerStartSignalingButton: HTMLElement
+ moveableBoxElement: HTMLElement	+ meshClientElements: HTMLElement
+ signalingelements: HTMLElement	+ meshClientSignalingURL: HTMLInputElement
+ webSocketServerChatBox: HTMLElement	+ meshClientSubmitButton: HTMLElement
+ webSocketServerMessageInput: HTMLInputElement	+ meshClientReadyButton: HTMLElement
+ getFundamentalDOMElements(): void	+ getMeshUiElements(): void
+ getPureWebSocketUiElements(): void	+ getAuthoritativeElements(): void
+ getAllUiElements(): void	+ getMeshUiElements(): void
+ addMovingDivForAuth(): HTMLElement	

Abbildung 18: UML-Diagramm der statischen UiElementHandler Klasse. Quelle: Eigene Darstellung

4.5 Electron Eintrittspunkte

Bei den Electron Eintrittspunkten handelt es sich um diejenigen Klassen, die innerhalb der HTML-Dateien eines Electron Programms eingebunden werden. In Electron Tutorials werden diese zumeist als „renderer.ts“ bezeichnet.
Zusätzlich gibt es noch die meist als „main.ts“ betitelte Klasse. In dieser ist Electron in der Lage Fenster zu öffnen und diese zu formatieren, beispielsweise in der Höhe und Breite. Außerdem kann dort eine HTML Datei festgelegt werden die nach Abschluss des Fensteraufbaus geöffnet wird
Ist die main.ts der Startpunkt, so ist die renderer.ts die Rennstrecke auf der alles passiert.
Zu finden sind diese Dateien im Ordner Scenes (siehe Anhang: Projektstruktur).

4.5.1 Fudge Network Entry Point

Diese Datei bezeichnet den Eintrittspunkt in die FudgeNetwork Beispielprogramme. Sie öffnet ein Fenster und ruft in diesem die „ChooseExample.html“ auf.

4.5.2 Example NetworkStructure

Die Dateien die diesem Namensschema folgen stellen die „renderer.ts“ für die verschiedenen Beispiele dar. Sie werden in den korrespondierenden HTML-Dateien eingebunden.
Möchte der Entwickler ein neues Programm entwickeln, so kann eine neue Datei in diesem Schema aufgebaut werden.

4.6 WebSocket Implementation

In diesem Abschnitt wird die Komponente für die reine WebSocket Kommunikation dargestellt und die Funktionsweise umrissen. Der Ablauf wird noch einmal genauer in den Aktivitätsdiagrammen im Anhang dargestellt.
Ist die Erstellung einer reinen WebSocket Struktur gewünscht, so müssen Client Manager und FudgeServer eingebunden werden. Diese können nach Belieben erweitert werden ohne die Verbindungserstellung zu gefährden.

4.6.1 Client Manager

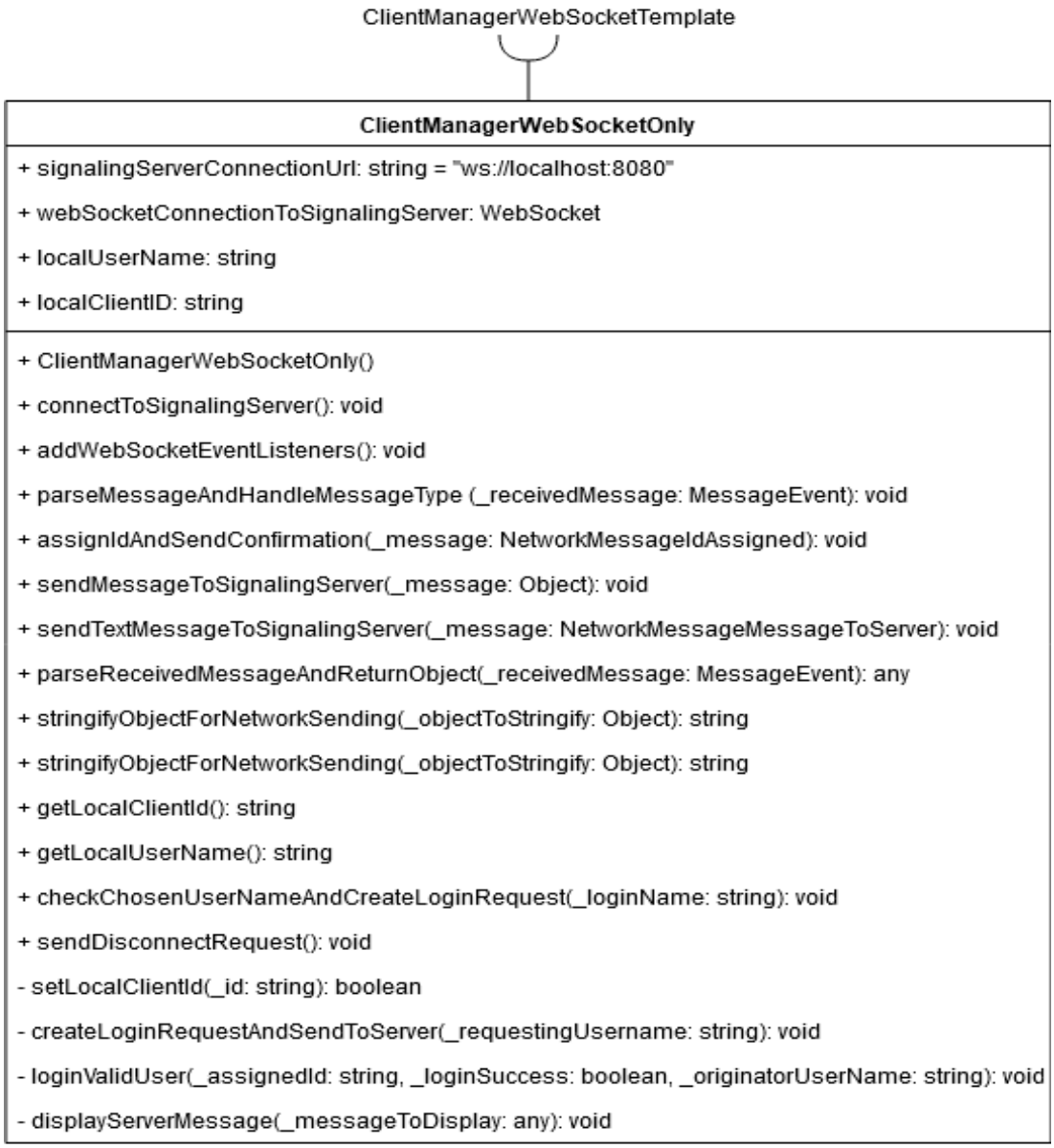


Abbildung 19: UML-Diagramm der WebSocket ClientManager Komponente. Quelle: Eigene Darstellung

Der Client Manager für die WebSocket Verbindung erstellt eine WebSocket Connection wenn er sich mit mit dem Signalingserver verbindet. Im gleichen Schritt fügt der Manager die Eventlistener für „open“, „close“ und „message“ hinzu. Der Server sendet eine Id-Assignment Nachricht (siehe Abbildung 21) an den Client Manager, worauf der Client Manager sich diese Id zuweist und eine Bestätigung an den Server schickt.

Es ist nun eine Frage des Users, ob dieser wünscht einen Namen mit der Id zu verbinden. Falls ja, so kann der User einen Username für seine Verbindung eingeben. Diese wird auf Validität geprüft und anschließend wird ein *LoginRequest* (siehe Abbildung 15) generiert, zu JSON stringified und über die WebSocket Verbindung an den Server übertragen. Ist der Username gültig, so bekommt der Client eine *LoginResponse* (siehe Abbildung 15) als JSON zurück. Der Client Manager parsed diesen JSON String zurück in das Nachrichtenobjekt. War der Loginversuch erfolgreich, so ist der boolean loginSuccess in der Nachricht true und der Client Manager legt seinen UserNamen fest.

Sind diese Formalien beendet kann der Client Manager jederzeit eine Textnachricht an den Server senden. Dazu muss lediglich ein Text eingegeben und der Funktion „sendTextMessageToSignalingServer“ übergeben werden.

Nachrichten des Servers werden mit der Funktion „displayServerMessage“ dargestellt, sofern ein Textfeld im HTML implementiert und in den UiElementHandler geladen wurde.

In beiden Fällen wird ein eigener Nachrichtentyp zur Unterscheidung verwendet: *ClientToServer* und *ServerToClient* (siehe Abbildung 15).

4.6.2 Fudge Server

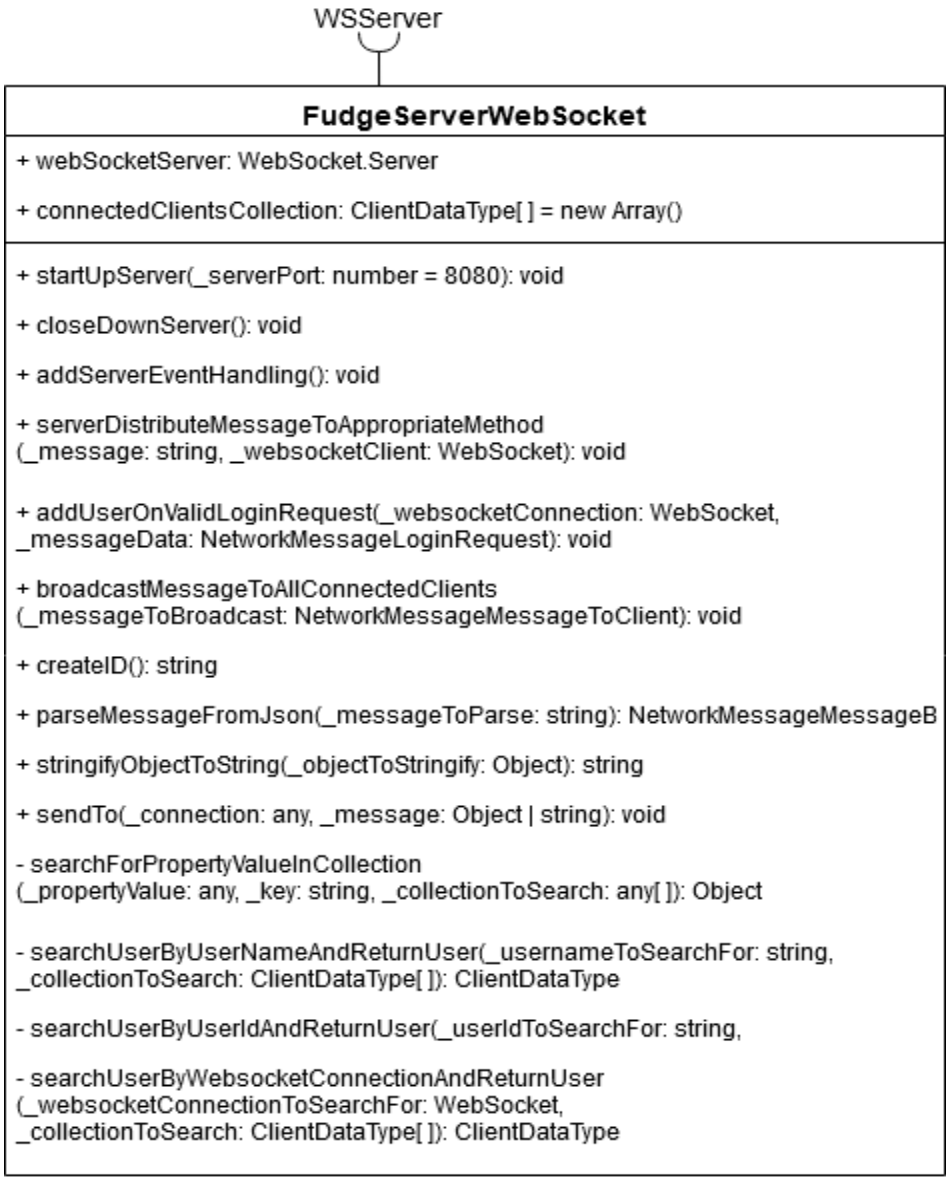


Abbildung 20: UML-Diagramm der WebSocket FudgeServer Komponente. Quelle: Eigene Darstellung

Der FudgeServer ist in der Lage einen WebSocketServer zu starten, zu stoppen, sowie das Eventhandling zu implementieren. Außerdem ist der FudgeServer in der Lage eine einzigartige ID zu generieren und neu verbundenen Clients zuzuweisen.

Der *FudgeServerWebSocket* prüft bei einem *LoginRequest*, ob der Username bereits an einen anderen Client vergeben wurde und gibt das Ergebnis der Prüfung als boolean an den Client zurück.

Nachrichten die der Server empfängt werden in einem Textfeld angezeigt sofern es im UiElementHandler existiert. Zudem werden empfangene Nachrichten als Teil der grundlegenden Implementierung an alle verbundenen Clients weitergeleitet. Dieses Verhalten kann in der „broadcastMessageToAllConnectedClients“ Funktion geändert und eingeschränkt werden.

4.6.3 Network Messages

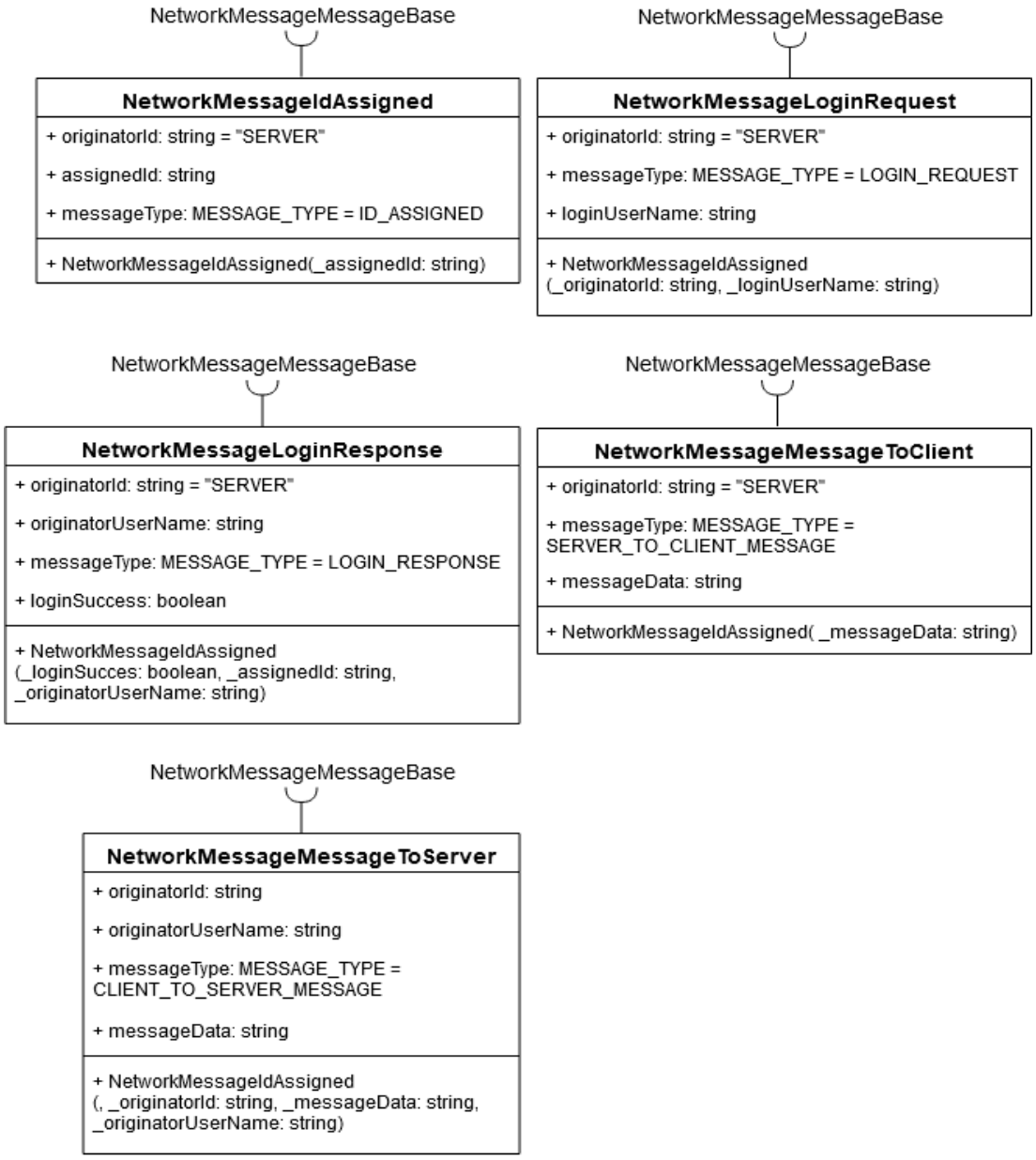


Abbildung 21: UML-Diagramm der für die WebSocket Komponente benötigten Network Message Komponenten. Quelle: Eigene Darstellung

4.6.4 Kommunikationsweise

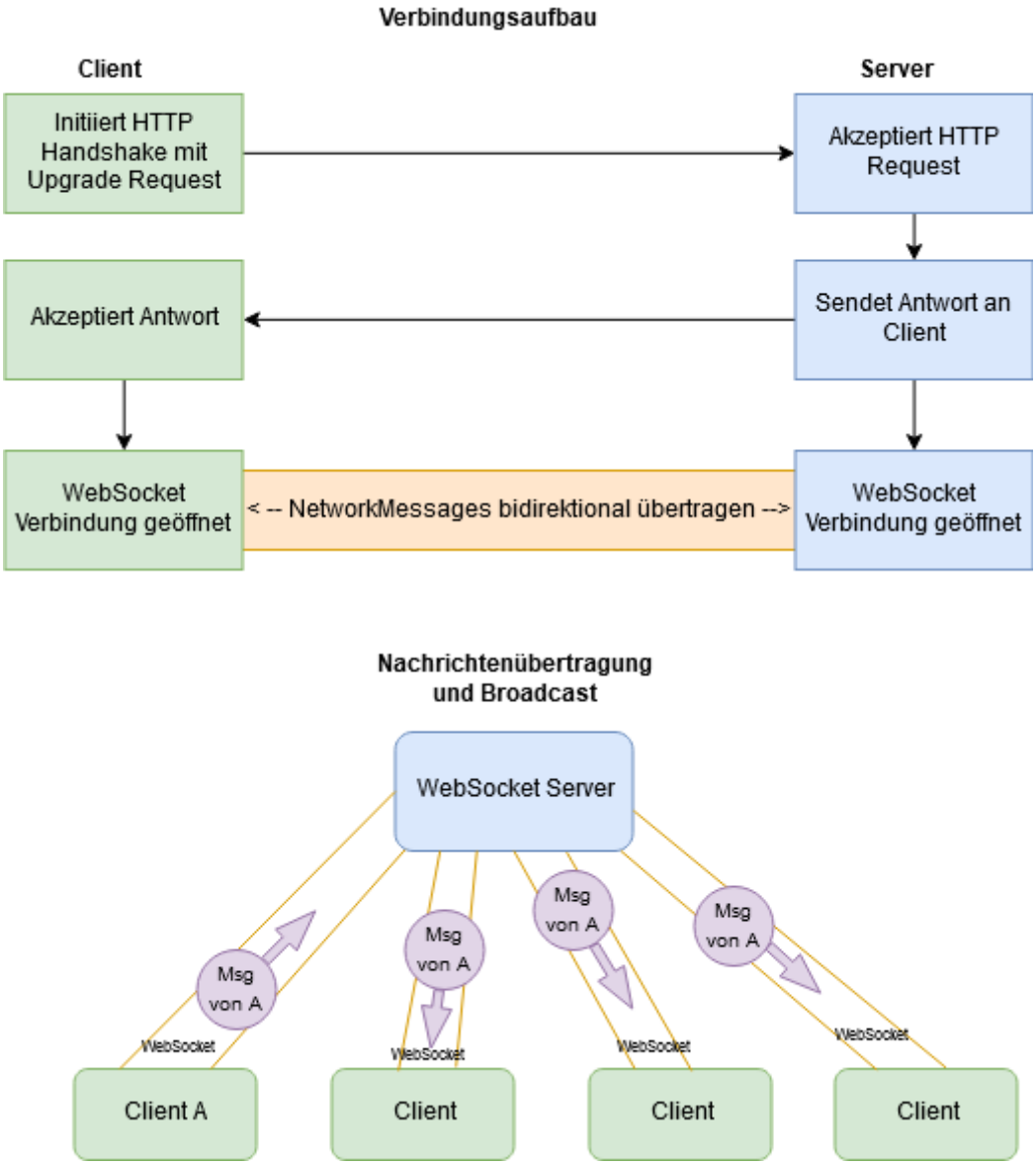


Abbildung 22: Ablauf Verbindungsaufbau einer WebSocket Verbindung in Fudge und der Broadcast Fähigkeit des WebSocket Server. Quelle: Eigene Darstellung

Wie in Abbildung 22 zu sehen, findet der Verbindungsaufbau via HTTP-Request-Response System statt. Anschließend wird die Verbindung auf WebSocket aufgerüstet.

Die Datenübertragung erfolgt dann von jedem Client über den WebSocket Kanal an den Server. Der Server, in dieser Implementation, schickt empfangene Nachrichten dann an jeden verbundenen Client weiter.

4.7 Single Peer Implementation

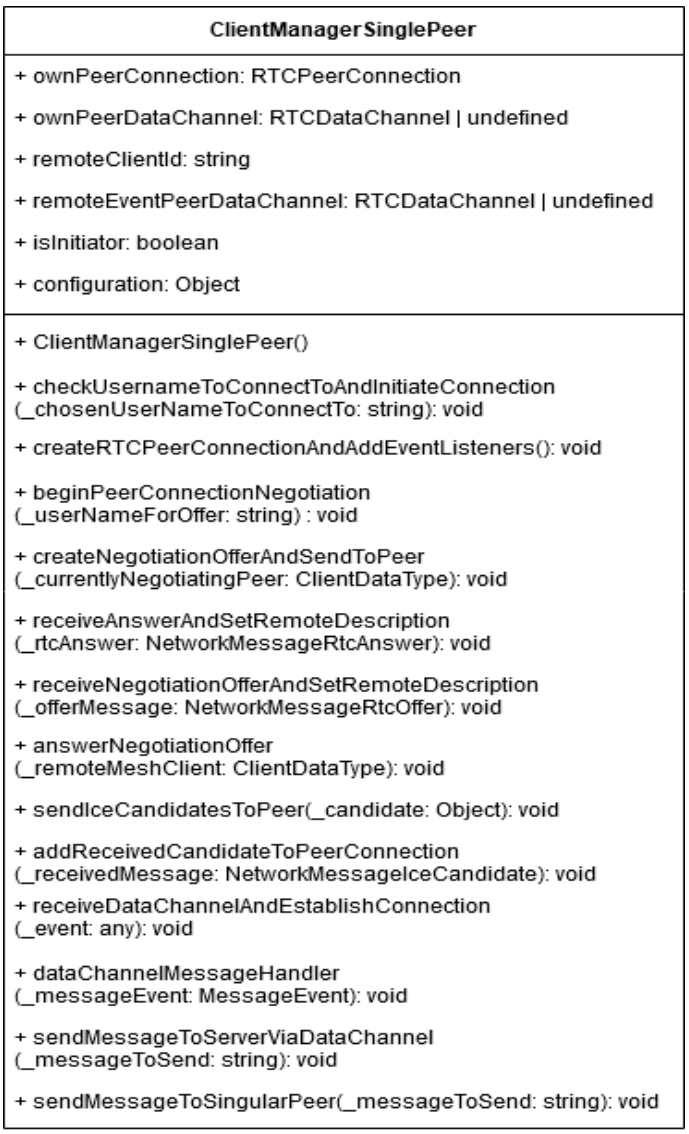


Abbildung 23: UML-Diagramm der Single Peer ClientManager Komponente. Quelle: Eigene Darstellung

Die Single Peer Implementation bedient sich der essentiellen Funktionen aus der WebSocket Implementation und erweitert diese um die notwendigen Nachrichtenhandler für eine vollständige WebRTC Verhandlung (siehe Abbildung 12). Der Übersichtlichkeit halber werden die Funktionen aus dem Interface WSServer nicht erneut dargestellt, stattdessen liegt der Fokus auf den für die WebRTC Verhandlung notwendigen Funktionen gelegt.

4.7.1 Client Manager

Im Gegensatz zum Client Manager für reine WebSocket Verbindungen muss der Client Manager für die Single Peer WebRTC Verbindung zwei Seiten der gleichen Aufgabe übernehmen:

1. Initiator des Verbindungsaufbaus
2. Empfänger des Verbindungsangebots

Um dies zu reflektieren existiert die „isInitiator“ Flag. Diese signalisiert dem Client Manager, ob Nachrichten über den eigenen oder einen empfangenen DataChannel gesendet werden. Dies ist notwendig da der Caller seinen DataChannel selbst generiert, während der Receiver den DataChannel aus dem „datachannel“ Event erhält und nicht selbst kontrolliert ob und wann dieser empfangen wird.

Auf der Callerseite muss der Identifier eines Receivers bekannt sein; im Beispiel ist das der Loginname. Daraufhin kann der Caller ein WebRTC Angebot generieren und daraus die lokale SDP-Description ableiten. Im Anschluss erstellt der Caller eine *RTCOffer* Nachricht und sendet diese an den angegebenen Receiver Username. Die Identifizierung des Clients übernimmt dabei der *FudgeServerSinglePeer*. In diesem Schritt wird der „isInitator“ Flag auf ‚true‘ gesetzt.
Dann erwartet der Caller eine Antwort. Mit Erhalt dieser wird die *remoteSDPDescription* gesetzt und der Austausch der *ICECandidates* beginnt. Ist der Austausch abgeschlossen und eine gemeinsame Kommunikationsmöglichkeit gefunden, schickt der Caller das *DataChannelEvent* über die *RTCPeerVerbindung* an den Receiver.

Auf der Receiverseite erwartet der Client eine RTCOffer Nachricht. Wird diese erhalten erzeugt der Receiver ein neues *RTCPeerConnection* Objekt und setzt die *SDPDescription* der *RTCOffer* Nachricht als *remoteSDPDescription*. Damit beginnt die Verbindungsverhandlung (siehe Abbildung 12).

Anschließend generiert der Receiver eine Antwort. Aus dieser Antwortet leitet er sich gleichzeitig die lokale SDP-Description ab. Abschließend erzeugt der Receiver eine neue RTCAnswer Nachricht und schickt diese an die ClientId aus dem RTCOffer. Damit beginnt der Austausch der ICE-Candidates.
Ist der Austausch abgeschlossen erhält der Receiver das „datachannel“ Event des Callers. Dieses Event beinhaltet einen DataChannel, den der Receiver als *remoteEventPeerDataChannel* speichert und fortan als Kommunikationskanal ansprechen kann.
Ab hier unterscheidet der Client Manager mithilfe des „isInitiator“ Flag welchen DataChannel er zur Kommunikation verwenden soll: *ownPeerDataChannel* oder *remoteEventPeerDataChannel*. Wird der Flag falsch gesetzt wird der Client Manager versuchen über einen nicht existenten oder nicht verbundenen DataChannel eine Nachricht zu schicken und so einen Fehler erzeugen.
Empfangene Peer Nachrichten werden, ähnliche wie bei der WebSocket Implementation, in einem entsprechenden Textfeld dargestellt.

4.7.2 Fudge Server

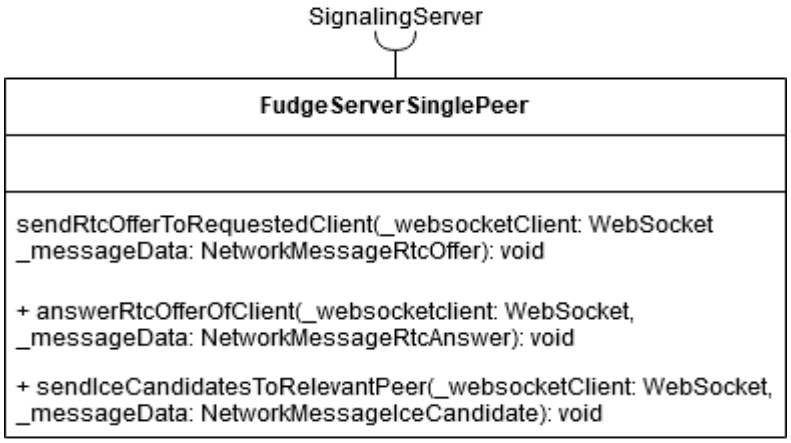


Abbildung 24: UML-Diagramm der Single Peer FudgeServer Komponente.
Quelle: Eigene Darstellung

In der Single Peer Struktur hat der Server lediglich die Aufgabe die Clients, wie in der WebSocket Implementation, eindeutig zu identifizieren, ihre erwünschen Loginnamen zu prüfen und zuzuweisen und die RTCOffer, RTCAnswer und

ICECandidate Nachrichten an den korrekten Client weiterzuleiten. Ist der DataChannel etabliert kann der Server heruntergefahren werden.

4.7.3 Network Messages

Zu den Nachrichten aus der WebSocket Implementation kommen nun noch die WebRTC Verhandlungsmessages hinzu, sowie eine Peer Message die speziell für das Beispiel nur einen String liefert, der in den Client Managern auf dem Bildschirm ausgegeben werden kann. Die Funktionsweise entspricht hier der eines Online Chatrooms.

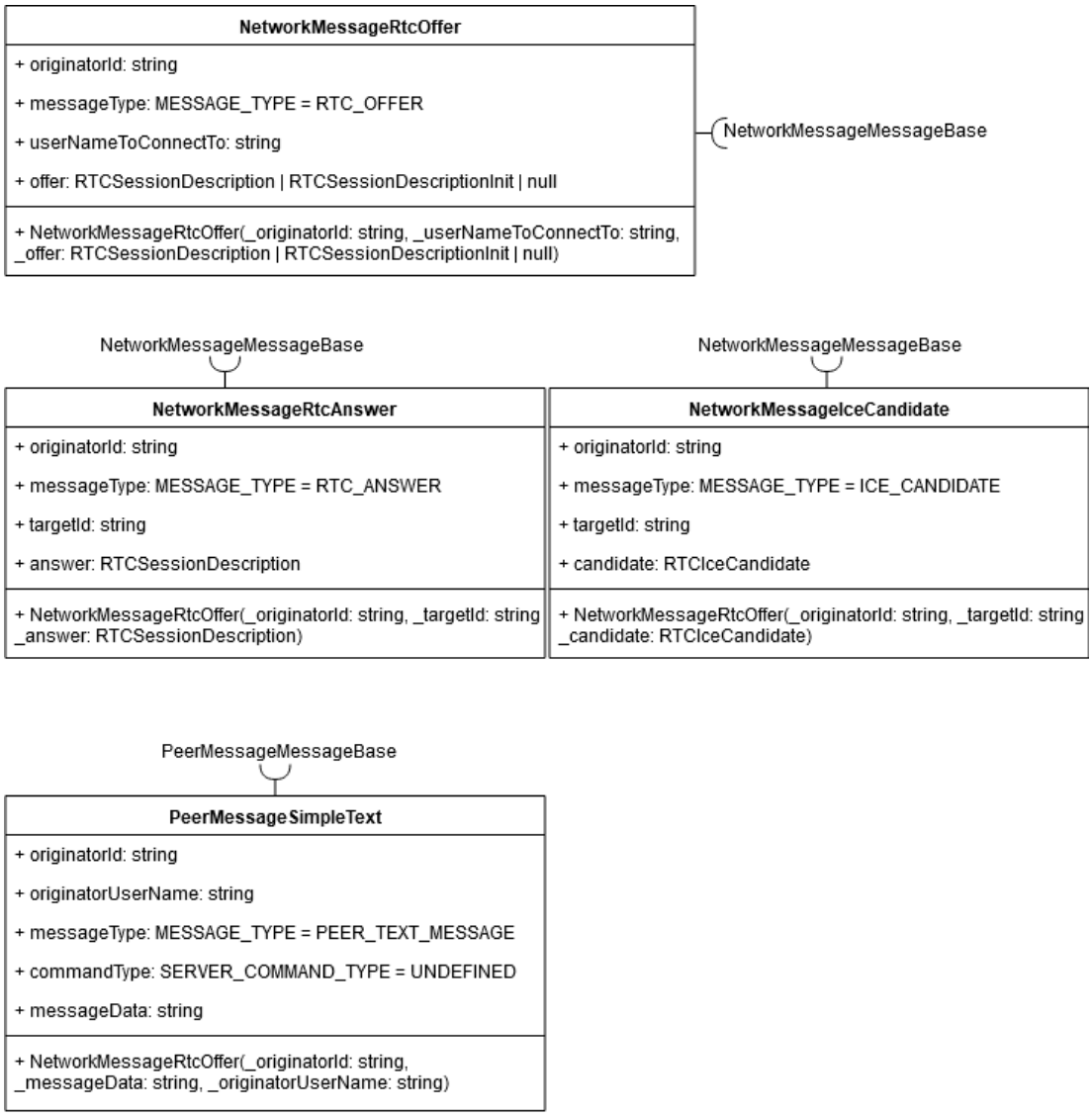


Abbildung 25: UML-Diagramm der für die Single Peer Verbindung benötigten NetworkMessage Komponenten. Quelle: Eigene Darstellung

4.7.4 Kommunikationsweise

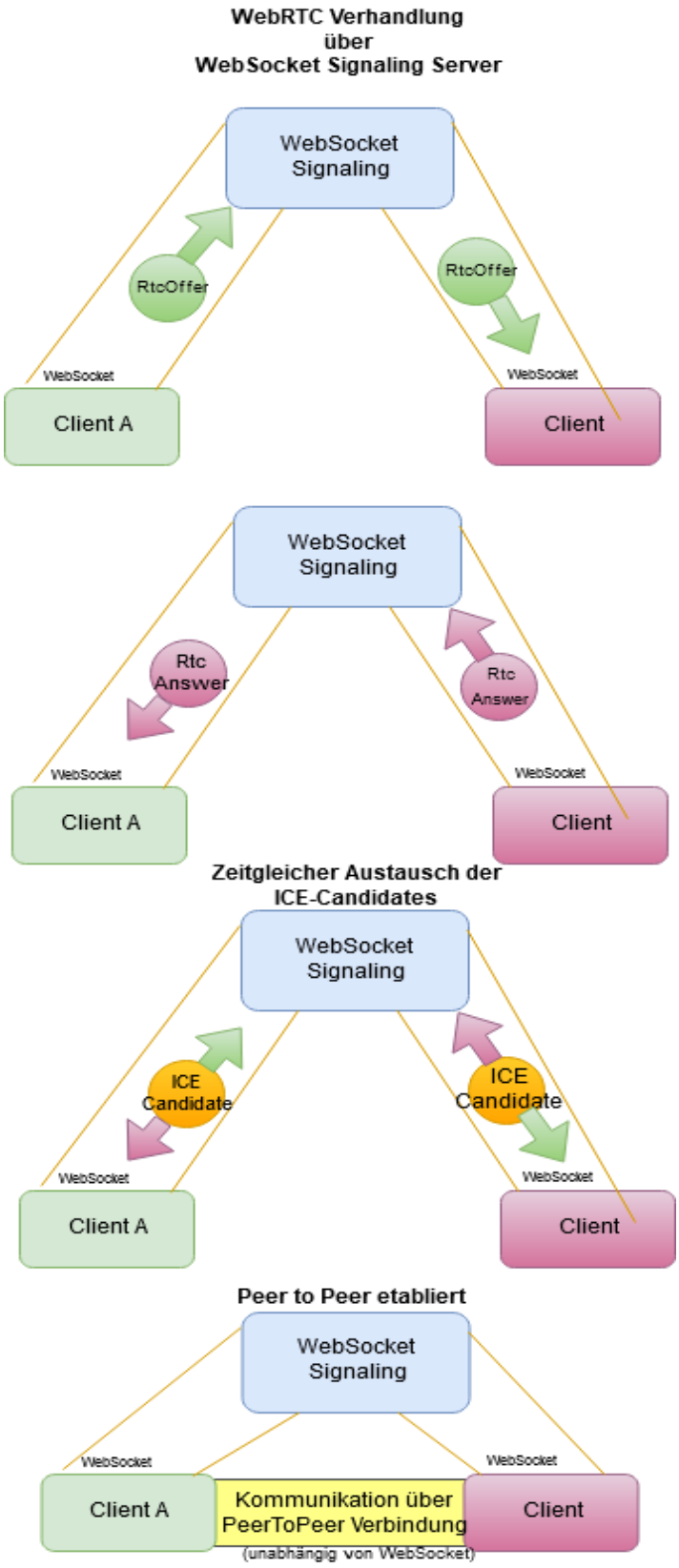


Abbildung 26: Etablierung einer Peer To Peer Connection über den FudgeServer WebSocket. Quelle: Eigene Darstellung

Wie in Abbildung 26 zu sehen, findet die WebRTC Verhandlung zur Etablierung einer Peer To Peer Connection über den WebSocket Signaling Server statt. Der Server identifiziert den gewollten Client aus der Id und leitet die Nachricht weiter.

Ist die Verhandlung abgeschlossen besteht eine direkte Kommunikationsverbindung zwischen den Clients. Ab nun ist der WebSocket Server nicht mehr notwendig, kann aber für die Wiederverbindung bei Verbindungsverlust verwendet werden.

4.8 Authoritative Server Implementation

4.8.1 Client Manager

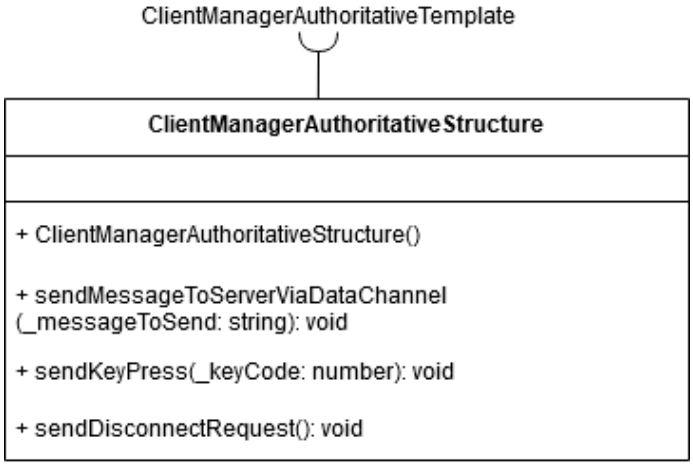


Abbildung 27: UML-Diagramm der für die Authoritative Structure notwendigen ClientManager Komponente. Quelle: Eigene Darstellung

Wie in den Client Manager Interfaces zu sehen (siehe Abbildung 11) basiert der Client Manager für die Authoritative Structure beinahe eins zu eins auf dem Client Manager für die Single Peer Connection.

Das liegt daran, dass der Client Manager selbst nur eine einzige WebRTC Verbindung, direkt zum Server, verwenden muss. Für die Implementation hat dieser Client Manager außerdem die Fähigkeit bekommen, Keypresses zu empfangen und zu versenden. Das Buttondown Event wird von dem Electron

renderer – ExampleAuthoritativeServer – abgefangen und der KeyCode der gedrückten Taste an den Client Manager weitergegeben. Dieser generiert eine Keypress Message und sendet diese an den Server.

Für die weitere Implementierung kann nun der Client Manager angepasst werden, um bestimmte Reaktionen auf Tastendrücke und Mausbewegungen zu zeigen und diese per spezialisierter Nachricht an den Server zu senden. Somit ist der Grundstein für ein vernetztes, digitales Client-Server Spiel gelegt.

4.8.2 Fudge Server

Abbildung 28: UML-Diagramm des Authoritative Signaling Server und des Authoritative Managers. Quelle: Eigene Darstellung

Der größte Unterschied zwischen dem Signaling Server aus der Single Peer Implementation und dem Signaling Server der Authoritative Implementation besteht in der zusätzlichen Referenz auf einen Authoritative Manager.
Dabei handelt es sich um den eigentlichen Authoritative Server. Diese Klasse verwaltet die Verbindungen zu den Clients und dient als zentraler „Peer“ für alle Spieler. So lässt sich die Client-Server Struktur mit Peer To Peer Verbindungen erreichen.
Um dies zu erreichen agiert der Manager wie ein herkömmlicher Client:
Er startet den WebRTC Verhandlungsprozess indem er ein Offer erstellt und es an den Receiver sendet. Dies tut der Manager für jeden Client der sich mit dem Signaling Server verbindet automatisch. Eine vollständige Verhandlung findet statt, wobei der Authoritative Signaling Server lediglich die Nachrichten an die Authoritative Manager Instanz weitergibt.
Der Authoritative Manager sammelt sämtliche Verbindungen in einem Array aus dem er bestimmte Clients per Id heraussuchen kann und so die Commands oder neue Objekte den jeweiligen Connections zuordnen kann.
Dies erlaubt es, wie in dieser Implementation demonstriert, Spielerobjekte zu erzeugen und die Steuerung nur dem besitzenden Client zu gestatten. Im Beispiel handelt es sich dabei um Blockelemente auf der HTML Seite, die für jeden Spieler mit einer eigenen Farbe erstellt werden und dann den Ids zugeordnet. Diese Zuordnung findet in der *divAndPlayerMap* statt.
Empfängt der Manager nun Tastendrücke die durch die Client Manager der verbundene Clients versendet werden, kann er diese den jeweiligen Blockelementen zuordnen und diese bewegen.
Somit ist der Grundstein gelegt für eine vollständige Spielverwaltung durch einen Authoritative Server und somit diese Art der Client-Server Netzwerkstruktur implementiert.

4.8.3 Network Messages

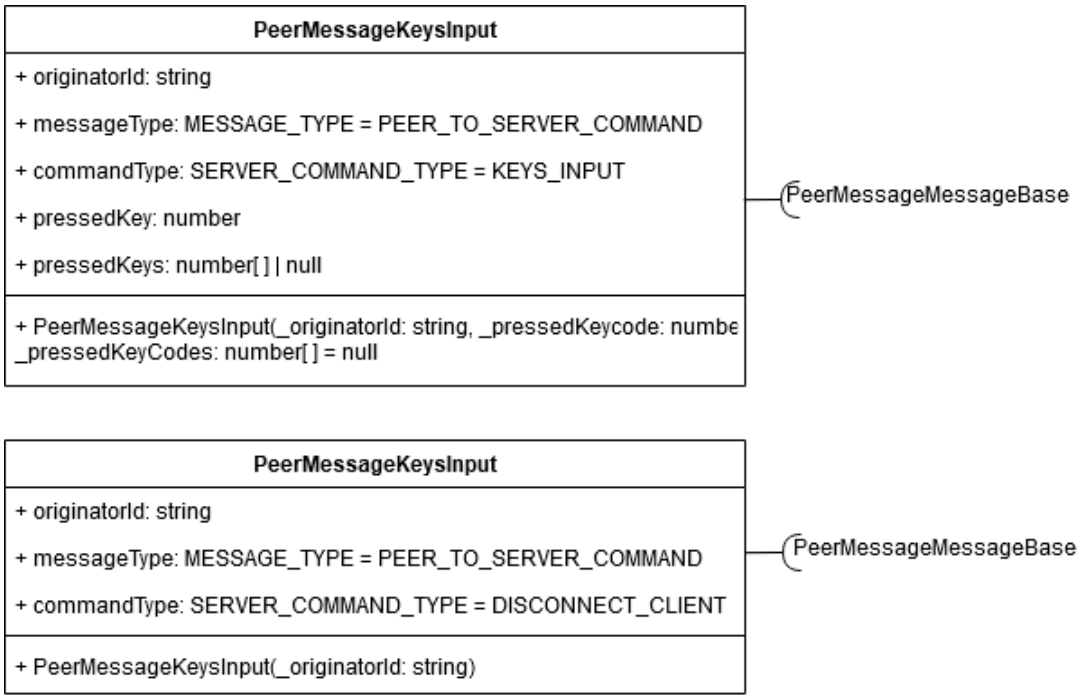


Abbildung 29: UML-Diagramm optionaler Nachrichtentypen für die Authoritative Client-Server Struktur. Quelle: Eigene Darstellung

4.8.4 Kommunikationsweise

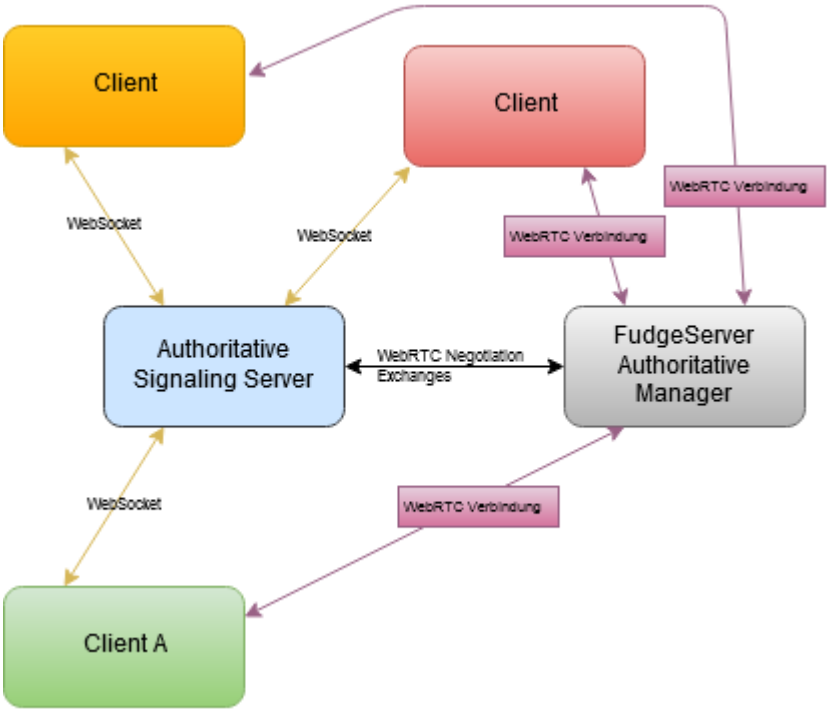


Abbildung 30: Darstellung der etablierten Verbindungen für Signaling Server und Authoritative Manager. Quelle: Eigene Darstellung

Wie in Abbildung 28 zu sehen, findet auch bei einem Authoritative WebRTC Server die Verhandlung via WebSocket statt. Der Unterschied besteht darin, dass die Verhandlungsdetails an eine Unterklasse, den Authoritative Manager weitergeleitet werden. Dieser Manager simuliert einen Peer im Netzwerk, mit dem sich jeder Client einzeln verbindet. Es entsteht eine Client-Server Struktur. Sind die Verbindungen etabliert findet die Kommunikation ausschließlich mit dem Authoritative Manager statt, der nun den zentralen Funktionen eines Authoritative Servers nachgeht: Nachrichten Validierung, Spielstand Verwaltung und Synchronisation.

In der derzeitigen Implementation erschafft der Manager neue HTML Objekte, weißt diese jeweils einem Spieler zu, empfängt Tastendrücke der Clients, leitet diese an die entsprechenden Objekte weiter und bewegt sie. Die Darstellung findet dabei auf dem Server selbst statt.

4.9 Mesh Network Implementation

4.9.1 Client Manager

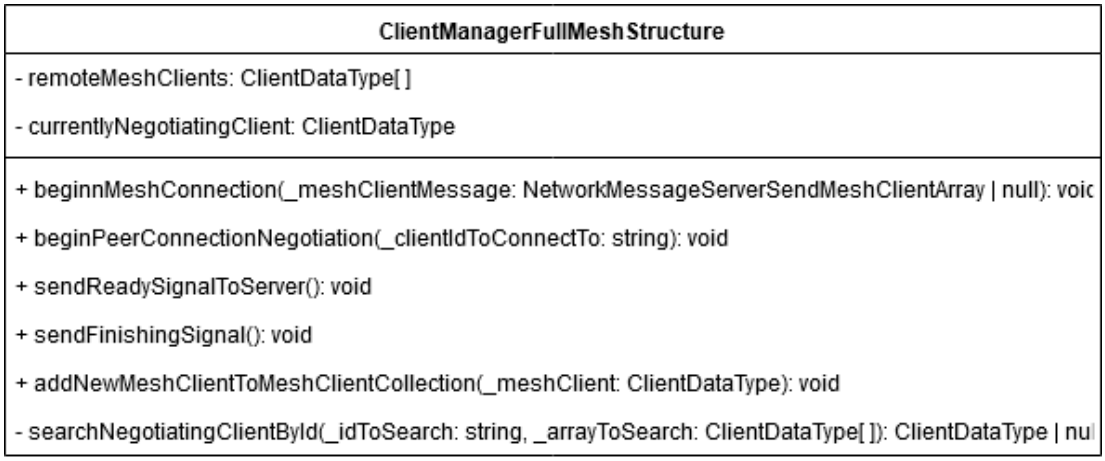


Abbildung 31: UML-Diagramm der für ein Mesh Network notwendigen ClientManager Komponente. Quelle: Eigene Darstellung

Bei der Mesh Network Structure speichert der Client Manager aufgrund des fehlenden Servers selbst eine Liste der Clients im Netzwerk. Diese Liste erstellt er bei der Verbindungsverhandlung aus den erhaltenen Ids der Clients. So kann jeder Mesh Client eindeutig identifiziert werden.

Die Beginn der Mesh Verhandlungen wird durch eine Servernachricht ausgelöst: *ServerSendMeshClientArray*. Dieses Array wird der Funktion *beginnMeshConnection* übergeben. Der erste Client wird *beginPeerConnectionNegotiation* übergeben und die Verhandlung beginnt. Dabei wird eine RTCPeerVerbindung erstellt, ein DataChannel mit der Id des Clients erstellt und sämtliche relevanten EventListener angefügt. Abschließend beginnt die Verhandlung mit dem Versand einer RtcOffer Nachricht.

Ist die Verbindung etabliert springt der Quellcode zurück in die *beginnMeshConnection* Funktion. Dort wird der erste Eintrag der Array Liste, der Client mit dem eine Verbindung etabliert wurde, entfernt und die Funktion ruft sich erneut selbst auf. Dieser rekursive Aufruf wird durchgeführt bis die Liste leer ist.

Ist die Liste leer wird eine *ClientIsMeshConnected* Nachricht an den Server gesendet, der daraufhin dem nächsten Client das Startsignal sendet.

Der Client Manager übernimmt ebenfalls wie die anderen zuvor auch die Funktionen der WebSocket Etablierung, WebRTC Verhandlung und des Empfangs von Nachrichten über WebSocket und RTCDataChannel. Diese werden hier daher nicht erneut dargestellt.

4.9.2 Fudge Server

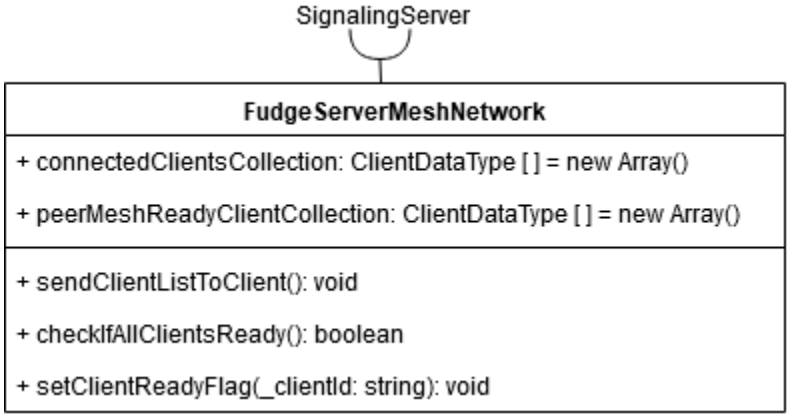


Abbildung 32: UML-Diagramm der für ein Mesh Network notwendigen SignalingServer Komponente. Quelle: Eigene Darstellung

Abbildung 32 zeigt die zusätzlichen Eigenschaften und Funktionen des Mesh Network Servers. Er fungiert grundsätzlich wie die anderen Signaling Server. Er leitet die Verhandlungs Network Messages an die korrekten Clients weiter. Der Mesh Network Server verarbeitet dazu noch drei weitere Network Messages:

- *ClientReady* – Der sendende Client ist bereit für das Mesh Network
- *ServerSendMeshClientArray* – Das peerMeshReadyClientCollection Array wird an den ersten Client in der Liste gesendet
- *ClientIsMeshConnected* – Signal, dass der Client seine Mesh Network Erstellung abgeschlossen hat

Der Server erstellt ein Array, in dem alle mit dem Server verbundenen Clients gespeichert sind. Außerdem enthält der Server ein Array, in das alle Clients übertragen werden, sobald die ready flag für alle Clients gesetzt wurde.

Anschließend sendet der Server die Liste der Clients an den ersten Client in der Liste, nachdem der erste Client aus der Liste entfernt wurde.
Ist der Client mit der Erstellung des um ihn zentrierten Mesh Networks fertig, so erhält der Server eine Client Is Mesh Connected Nachricht. Dies teilt dem Server mit, den vorherigen Schritt zu wiederholen, den nächsten Client aus der Liste zu entfernen und die neue Liste an diesen Client zu senden, der wiederum, nachdem alle Verbindungen etabliert wurde, eine Client Is Mesh Connected Nachricht absetzt. Dieser Vorgang wiederholt sich, bis die Clientliste auf der Serverseite leer ist.
Ist die Liste leer sind alle Clients miteinander und untereinander verbunden.

4.9.3 Network Messages

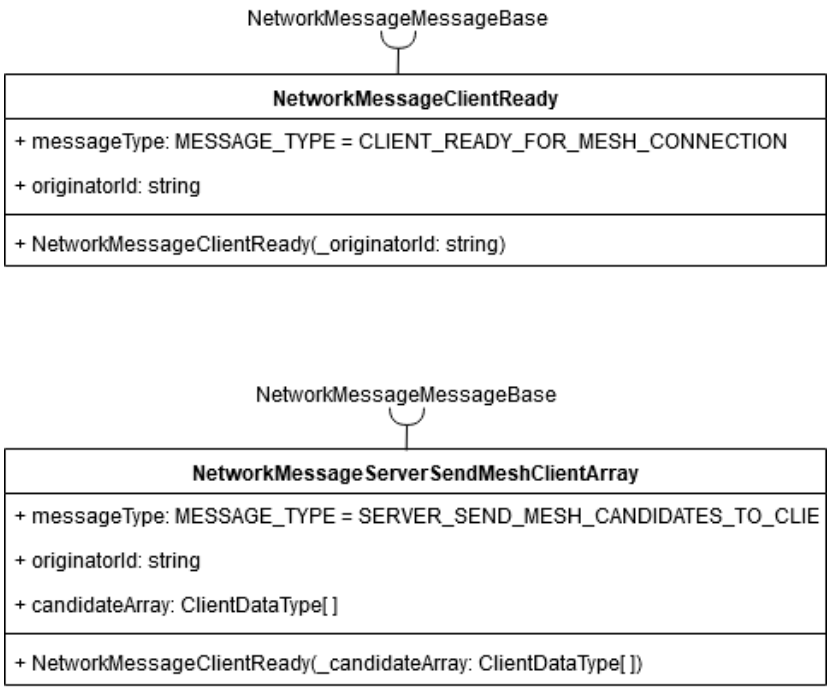


Abbildung 33: UML-Diagramm der für ein Mesh Network notwendigen NetworkMessage Komponenten. Quelle: Eigene Darstellung

4.9.4 Kommunikationsweise

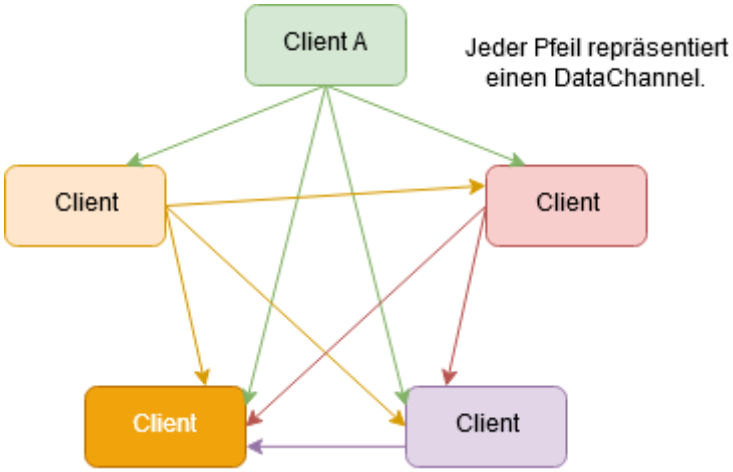


Abbildung 34: Vollständige Mesh Verbindung in Fudge. Quelle: Eigene Darstellung

Wie in Abbildung 34 zu sehen, nimmt die Anzahl der initierten WebRTC Verbindungen ab, je weiter hinten der Client sich in der Liste befindet. Es beginnt mit vier für den ersten Client, der vorletzte Client muss nur noch eine einzige Verbindung erstellen und der letzte Client gar keine.

Das bedeutet aber nicht, dass der erste Client die meisten Verbindungen verwalten muss. Jeder Client muss für jede eintreffende Verbindung eine eigene RTCPeerConnection verwalten. So ist die Anzahl der Peer Connections pro Client immer mindestens: **RTCPeerConnection * Anzahl der Clients – 1**

Daher ist eine Mesh Verbindung in der maximalen Anzahl der Clients beschränkt, da mit zunehmender Clientenzahl die Performance Ansprüche durch die Verwaltung der RTCPeerConnections zunehmen. In einer Client-Server Struktur kann das durch Serverhardware und Internetinfrastruktur ausgeglichen werden. Der Server ist hier speziell darauf ausgelegt viele Verbindungen zu verwalten und die Netzwerkbandbreite ist dementsprechend angepasst.

5. Diskussion und Ausblick

Im Rahmen dieser Bachelorarbeit wurden funktionierende Netzwerkkomponenten für die Game-Engine Fudge konzipiert und implementiert. Das Ergebnis zeigt, dass entsprechend der Spezifikationen von WebRTC und Chromium, UDP Kommunikation direkt zwischen Peers möglich ist, und sogar eine Serverstruktur aufgebaut und in Electron implementiert werden kann. Die Verwendung von TypeScript hat die Fehlersuche stark vereinfacht und es erlaubt, einfache Strukturen zu erstellen, die Skalierbarkeit, Wartbarkeit und Modularität gewährleisten Dies zeigt auch eine gewisse Flexibilität in der Herangehensweise. Die hier erstellten Komponenten könnten anderweitig umgesetzt werden, auch wenn die Kommunikation mit dem UDP-Protokoll in einer Browserumgebung ausschließlich über WebRTC erfolgen kann.

In dieser Arbeit wurde dabei die grundlegende Erstellung von Kommunikationswegen über UDP zwischen zwei Clients, seien beide Peers oder sei es eine Client-Server Struktur, fokussiert. Die Kombination von Webtechnologien mit Servertechnologien im Rahmen einer Browserumgebung hat aufwendige Fehlersuche notwendig gemacht. Die Kommunikation über UDP innerhalb einer modernen Browserumgebung ist aus Sicherheitsgründen eigentlich unmöglich gemacht. Dadurch sind entgegen der Erwartungen Algorithmen zur Serverseitigen-Prediction, dem Austausch von Gamestates und weitläufigen Lobbysystemen im Rahmen der Arbeit nicht möglich gewesen.

Problematisch ist auch die direkte Einbindung mit Fudge. Da Fudge vom TypeScript Compiler im Schema „system“ kompiliert wird, die Netzwerkkomponenten allerdings im Schema „commonjs“ kompiliert werden müssen um funktionstüchtig zu sein, muss ein Teil des Projekts angepasst werden. Da Fudge außerdem auf traditionellen ‚namespaces‘ basiert und die Netzwerkkomponenten auf das TypeScript eigene Barrelsystem angewiesen sind ist eine Konvertierung notwendig, ehe die Komponenten einschränkungsfrei eingebunden werden können.

Gleichzeitig bieten sich die Komponenten, da sie auch außerhalb von Fudge voll funktionstüchtig sind, für weiterführende Arbeiten an, beispielsweise um

serverseitige Prediction zu konzipieren und umzusetzen, Lobbysysteme zu entwickeln und serverseitige Gamestate Validierung zu implementieren. Sie können außerdem für andere Electron Projekte verwendet werden, die nichts mit digitalen Spielen zu tun haben.

Es können auch spezialisierte Netzwerkprotokolle entwickelt werden, die genau auf die didaktischen Anforderungen von Fudge zugeschnitten sind und so die Abhängigkeit von UDP umgehen. Außerdem können optimierte Kommunikationswege erarbeitet werden, die die Latenz bei der Kommunikation zwischen Clients und Server weiter reduziert.

Die für diese Arbeit verwendeten Technologien sind weltweit anerkannt und viel genutzt, beispielsweise JavaScript und Chromium, beide unterstützt durch den Technologiekonzern Google, TypeScript entwickelt von Microsoft, und Electron welches auf Technologie von Microsoft basiert. So besteht für Fudge keine Gefahr in näherer Zukunft obsolete Webtechnologien oder Komponenten deren Kompatibilität nicht mehr gegeben ist zu beherbergen.

Literaturverzeichnis

Blink. (o.D.). Abgerufen am 03. April, 2019, von https://www.chromium.org/blink

Brown, K. (2018, 04. September). JavaScript: How Did It Get So Popular? Abgerufen am 07. August 2019, von https://news.codecademy.com/javascript-history-popularity/

By Example. [Online Dokumentation] (o.D.). Abgerufen am 27. März, 2019, von https://www.typescriptlang.org/docs/handbook/declaration-files/by-example.html

Chopra, Varun (2015): WebSocket essentials, building apps with HTML5 WebSockets. build your own real-time web applications using HTML5 WebSockets. Birmingham, UK: Pack Publishing. E-Book

Electron Documentation. (o.D.). Abgerufen am 15.April 2019, von https://electronjs.org/docs/tutorial/support

Factory.hr. (2018). HTTP/2: the difference between HTTP/1.1, benefits and how to use it. Abgerufen am 07. Juli, 2019, von https://medium.com/@factoryhr/http-2-the-difference-between-http-1-1-benefits-and-how-to-use-it-38094fa0e95b

Gorski, P. L. / Lo Iacono, L. / Nguyen, H. V. (2017): WebSockets. Moderne HTML5-Echtzeitanwendungen entwickeln. München: Carl Hanser Verlag GmbH & Co. KG

Introduction to web APIs. [Online Dokumentation] (o.D.). Abgerufen am 20. April 2019, von https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction

Introduction to WebRTC protocols. [Online Dokumentation] (o. D.). Abgerufen am 20. April, 2019, von https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

JSTutorials Team. (2019). Simple Websocket Example with Nodejs. Abgerufen am 08. Juli, 2019 von https://www.js-tutorials.com/nodejs-tutorial/simple-websocket-example-with-nodejs/

July 2019 Web Server Survey. [Online Survey] (2019, 26. Juli). Abgerufen am 20. August, 2019, von https://news.netcraft.com/archives/2019/07/26/july-2019-web-server-survey.html

Kazimier, M. / de Visser, J. (2017): Video games. Playtime is over; with revenue surpassing one billion euro in 2018, video games are serious business and here to stay. Abgerufen am 27. Juni, 2019, von https://www.pwc.nl/en/publicaties/dutch-entertainment-and-media-outlook-2017-2021/videogames.html

Luding, S. / Garza, J. (2017): Learning HTTP/2. A practical Guide for Beginners. Farnham, UK: O'Reilly UK Ltd.

Mandl, P. (2018): TCP und UDP Internals. Protokolle und Programmierung. Wiesbaden: Springer Vieweg.

Mesh Topology. [Webseite] (o.D.). Abgerufen am 28. Juli, 2019, von http://webpage.pace.edu/ms16182p/networking/mesh.html

Messner, S. (2017, 18. August). The rise and fall of For Honor. Abgerufen am 15. August, 2019, von https://www.pcgamer.com/the-rise-and-fall-of-for-honor/

Panday, K.K. (2013). SSL Handshake and HTTPS Bindings on IIS. Abgerufen am 20. August, 2019, von https://blogs.msdn.microsoft.com/kaushal/2013/08/02/ssl-handshake-and-https-bindings-on-iis/

Patel, N. (2018, 10. April). Unity vs Unreal Engine? No more Confusion for Game development. Abgerufen am 12. Juni 2019, von https://www.linkedin.com/pulse/unity-vs-unreal-engine-more-confusion-game-nisha-patel

Patel, P. (2018, 18. April). What exactly is Node.js?. Abgerufen am 28. März, 2019, von https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/

Porter, J. (2017, 04. Februar). For Honor developer responds to networking and framerate criticisms. Abgerufen am 15. August, 2019, von https://www.techradar.com/news/for-honor-developer-responds-to-networking-and-framerate-criticisms

Rieseberg, Felix (2018): Introducing Electron. Desktop apps with JavaScript. Sebastapol, CA: O‘Reilly Media. E-Book

Ristic, Dan (2015): Learning WebRTC. Develop interactive real-time communication applications with WebRTC. Birmingham, UK: Packt Publishing. E-Book

Sergiienko, A. (2014): WebRTC blueprints. Develop your very own media applications and services using WebRTC. Birmingham, UK: Packt Publishing. E-Book

Signaling and video calling. [Online Dokumentation] (o. D.). Abgerufen am 21. April, 2019, von https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling

Signaling and video calling. [Online Dokumentation] (o.D.). Abgerufen am 19. April, 2019, von https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling

TCPSocket. [Online Archiv] (2019, 23. März). Abgerufen am 25. August, 2019, von https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/API/TCPSocket

Usage statistics of JavaScript as client-side programming language on websites. [Online Survey] (o.D.). Abgerufen am 10. August, 2019, von https://w3techs.com/technologies/details/cp-javascript/all/all

Usage statistics of Node.js. [Online Survey] (o.D.). Abgerufen am 10. August, 2019, von https://w3techs.com/technologies/details/ws-nodejs/all/all

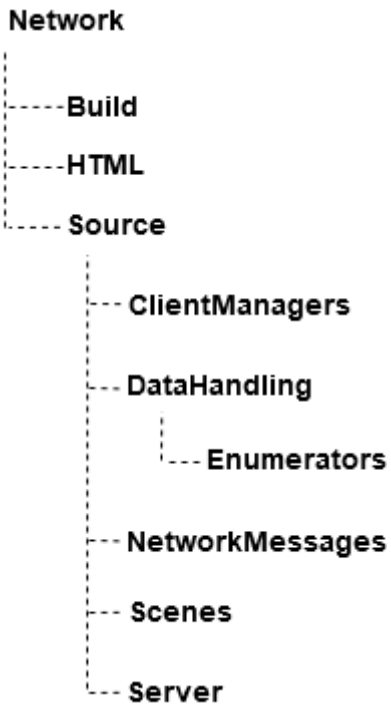
Vogl, A. (2018, 06. März). Network Connection Types in Online Games and How Do They Affect You? Abgerufen am 17. Mai, 2019, von https://www.gamingweekender.com/network-connection-types-online-games-affect/

Warcholinski, M. (o.D). What is Electron JS? Abgerufen am 29. März, 2019, von https://brainhub.eu/blog/what-is-electron-js/

Wijman, T. (2018). Mobile Revenues Account for More Than 50% of the Global Games Market. Abgerufen am 03. Juli, 2019, von
as It Reaches $137.9 Billion in 2018.
https://newzoo.com/insights/articles/global-games-marketreaches-137-9-billion-in-2018-mobile-games-take-half/

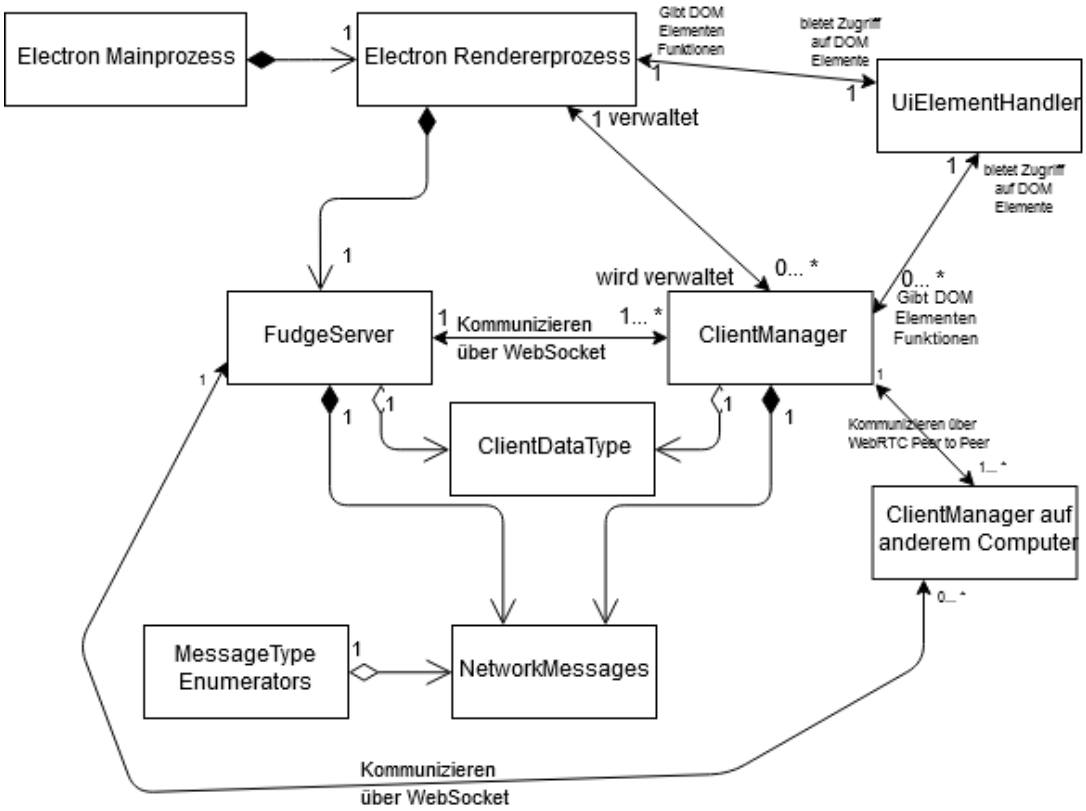
Anhang

Anhang 1: Projektstruktur der Netzwerkkomponenten



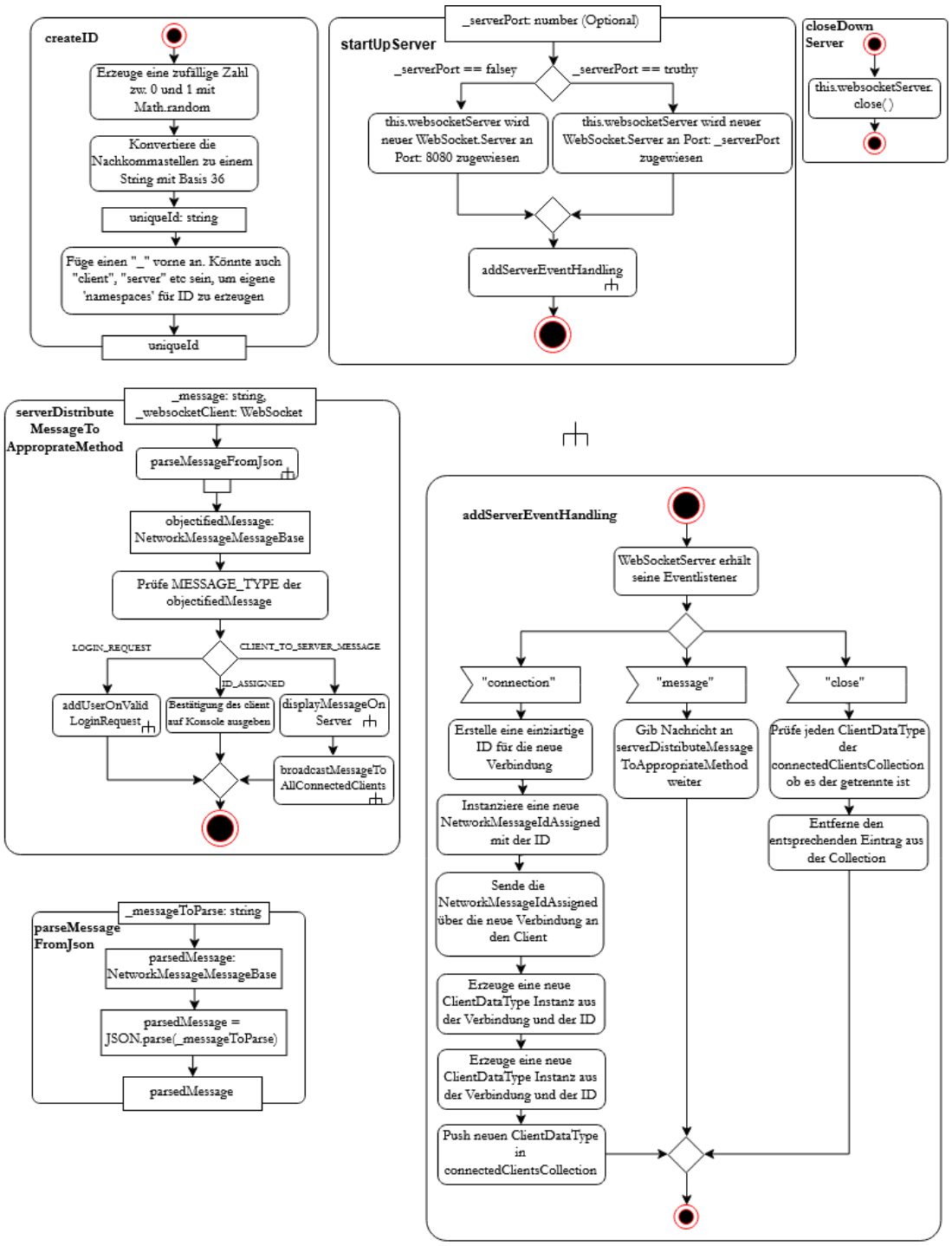
- **Network**: Root Ordner
- **Build**: Aus Source TypeScript Dateien kompilierte JavaScript Dateien
- **HTML** Dateien für Darstellung in Electron
- **Source**: Quellcode als TypeScript Dateien
- **ClientManagers**: Logik für Netzwerk Clients, nicht Server
- **DataHandling**: Unterstützende Logik und Datentypen für die Verarbeitung
- **NetworkMessages**: Durch Interfaces festgelegte Nachrichten für den Versand über das Netzwerk
- **Scenes**: Beispielszenen und Einstiegpunkt für Electron. Äquivalent zu main und renderer in Electron Tutorials
- : Logik für Server, hier auch alle verschiedenen Strukturarten

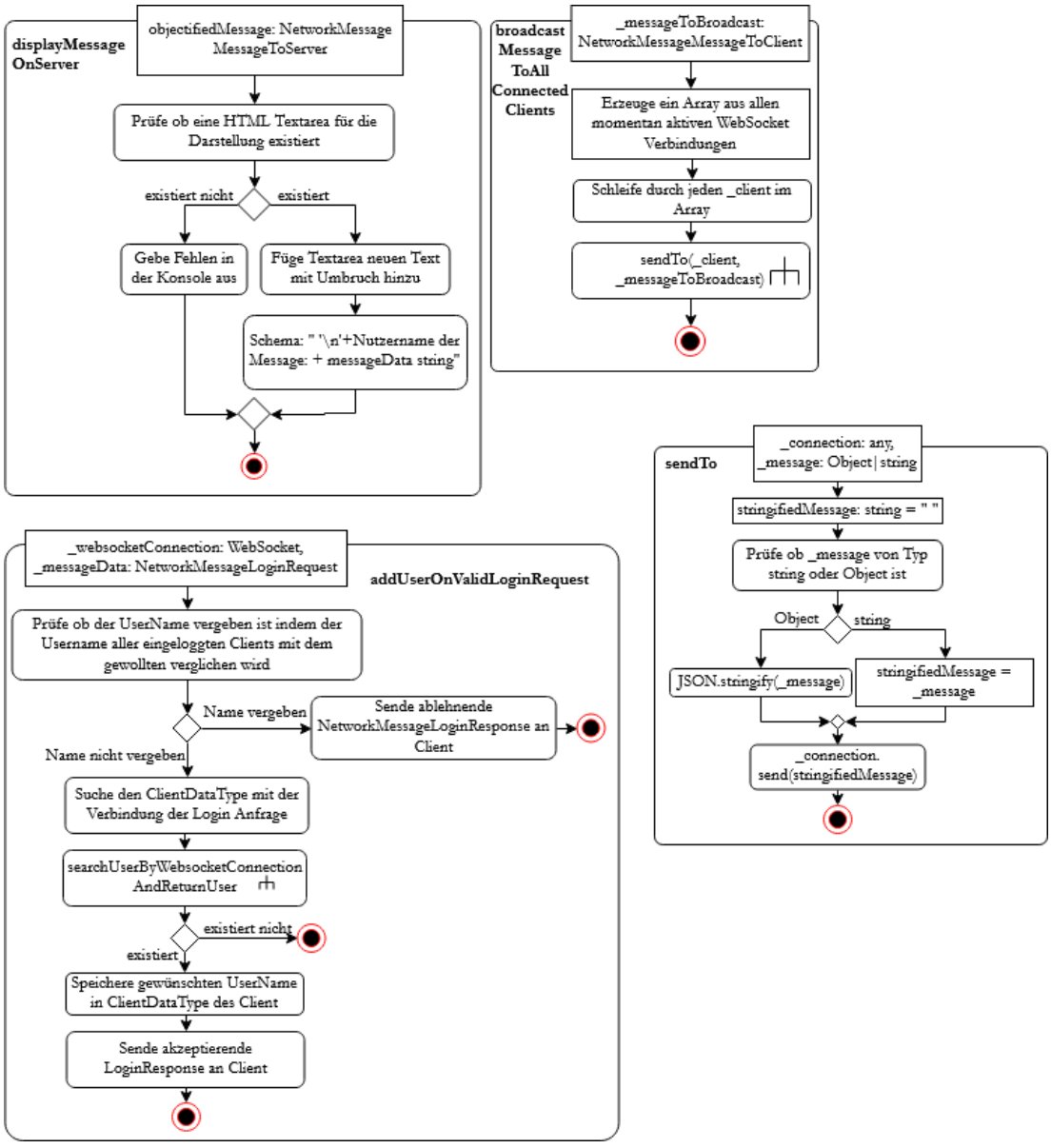
Anhang 2: Gesamtstruktur einer Electron Anwendung mit FudgeNetwork



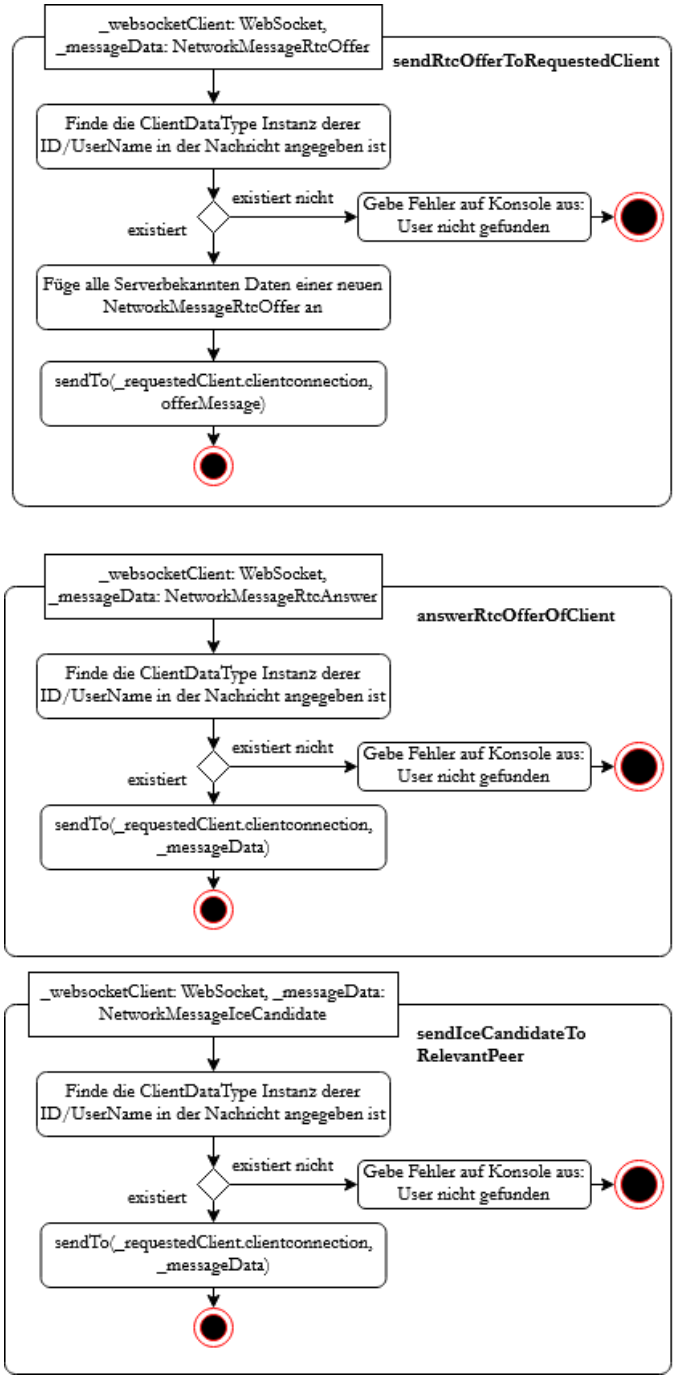
Anhang 3: Aktivitätsdiagramme

Anhang 3.1: FudgeServer WebSocket

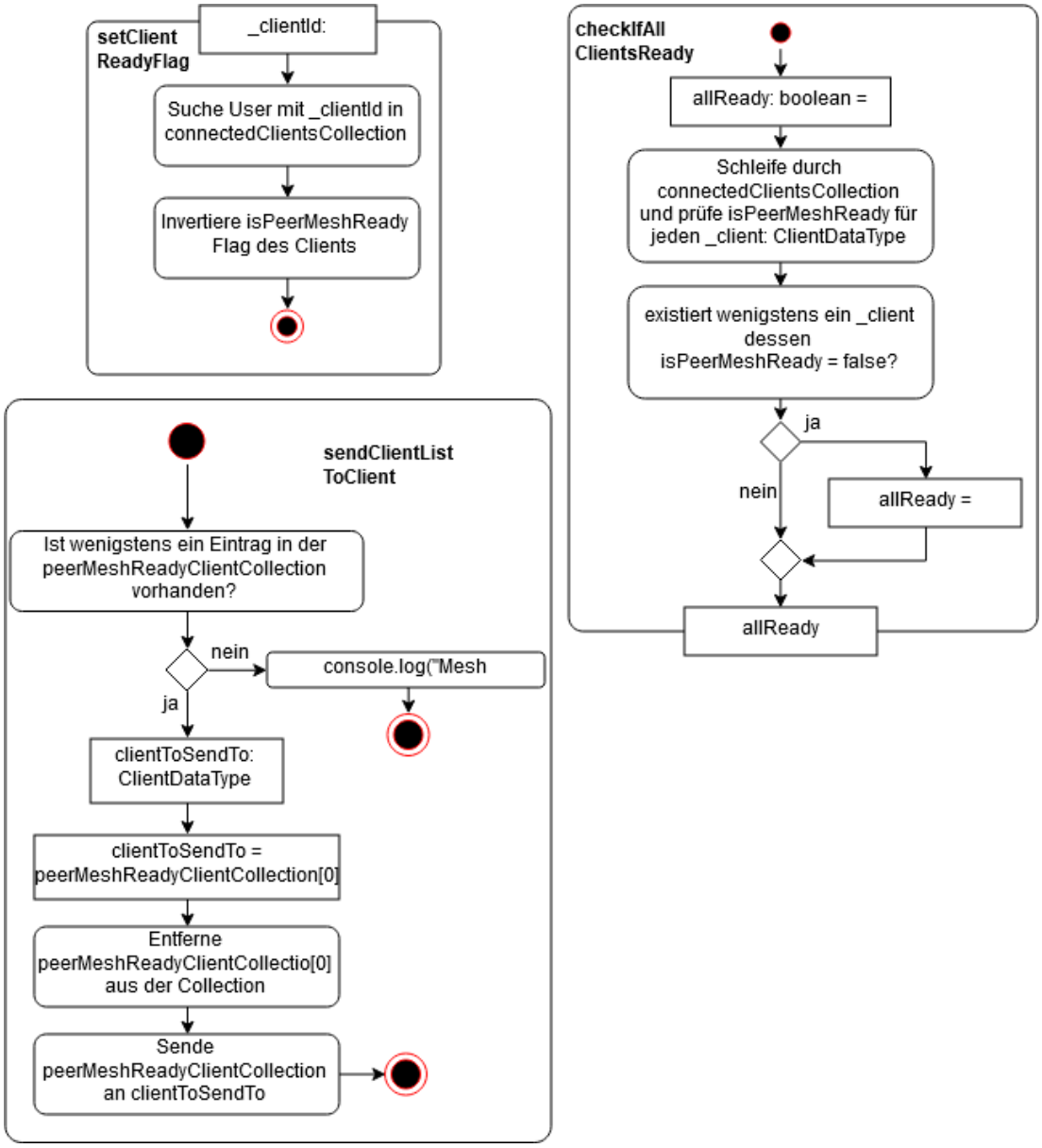



displayMessage OnServer
objectifiedMessage: NetworkMessage MessageToServer
Prüfe ob eine HTML Textarea für die Darstellung existiert
existiert nicht
existiert
Gebe Fehlen in der Konsole aus
Füge Textarea neuen Text mit Umbruch hinzu
Schema: " '\n'+Nutzername der Message: + messageData string"
broadcast Message ToAll Connected Clients
_messageToBroadcast: NetworkMessageMessageToClient
Erzeuge ein Array aus allen momentan aktiven WebSocket Verbindungen
Schleife durch jeden _client im Array
sendTo(_client, _messageToBroadcast)
sendTo
_connection: any, _message: Object | string
stringifiedMessage: string = " "
Prüfe ob _message von Typ string oder Object ist
Object
string
JSON.stringify(_message)
stringifiedMessage = _message
_connection. send(stringifiedMessage)
_websocketConnection: WebSocket, _messageData: NetworkMessageLoginRequest
addUserOnValidLoginRequest
Prüfe ob der UserName vergeben ist indem der Username aller eingeloggten Clients mit dem gewollten verglichen wird
Name vergeben
Sende ablehnende NetworkMessageLoginResponse an Client
Name nicht vergeben
Suche den ClientDataType mit der Verbindung der Login Anfrage
searchUserByWebsocketConnection AndReturnUser
existiert nicht
existiert
Speichere gewünschten UserName in ClientDataType des Client
Sende akzeptierende LoginResponse an Client

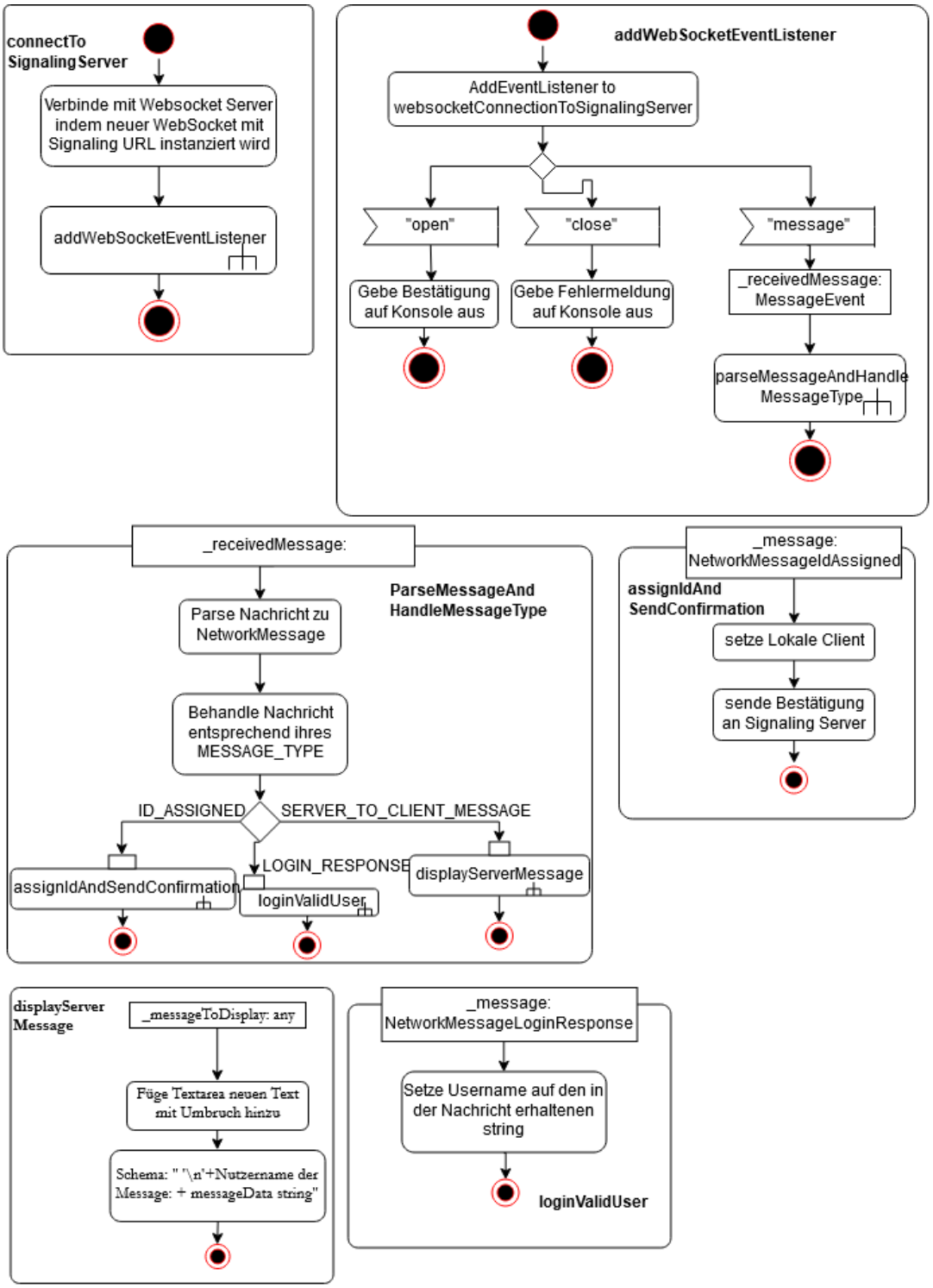
Anhang 3.2: FudgeServer SinglePeer



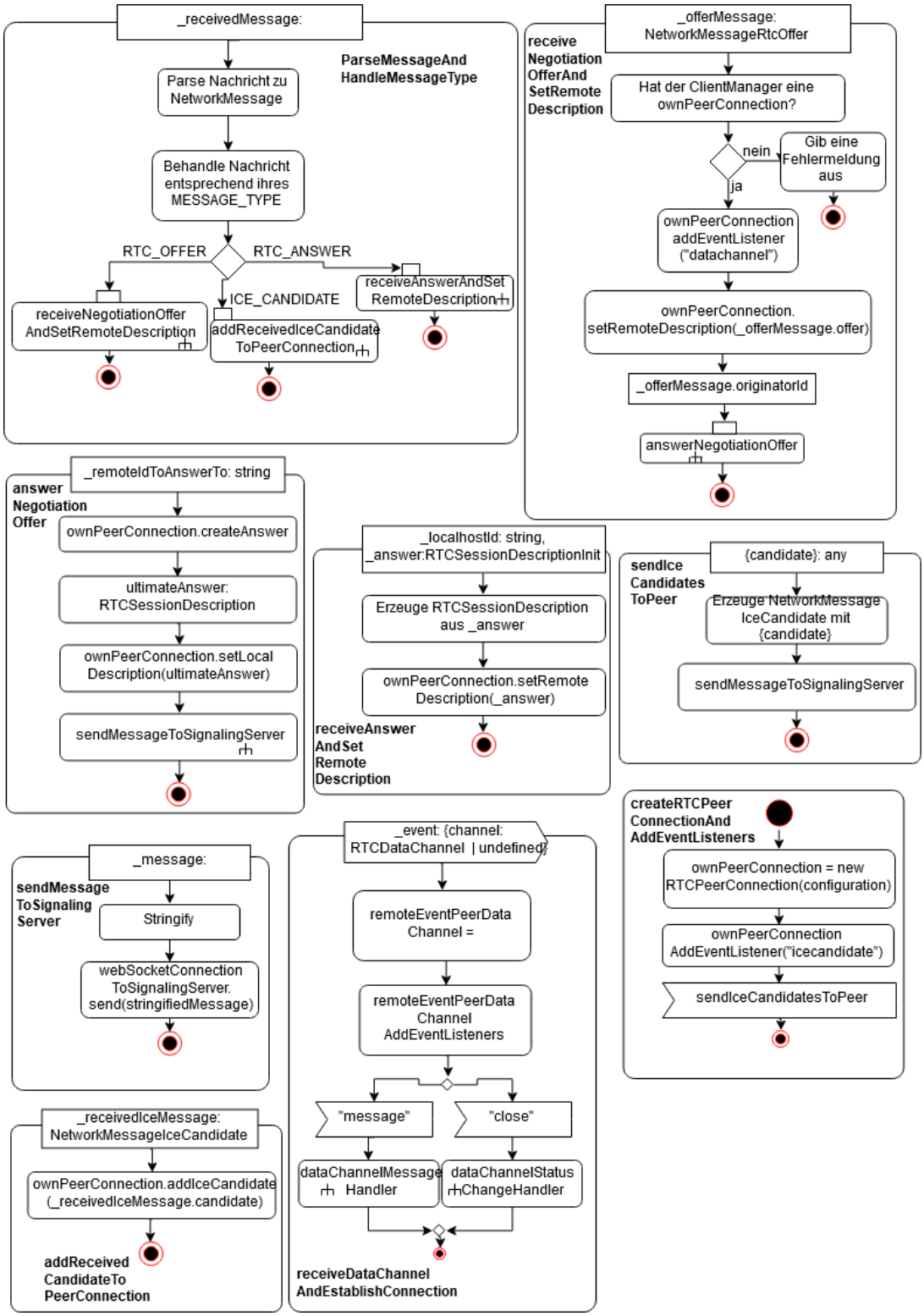
Anhang 3.3: FudgeServer Mesh

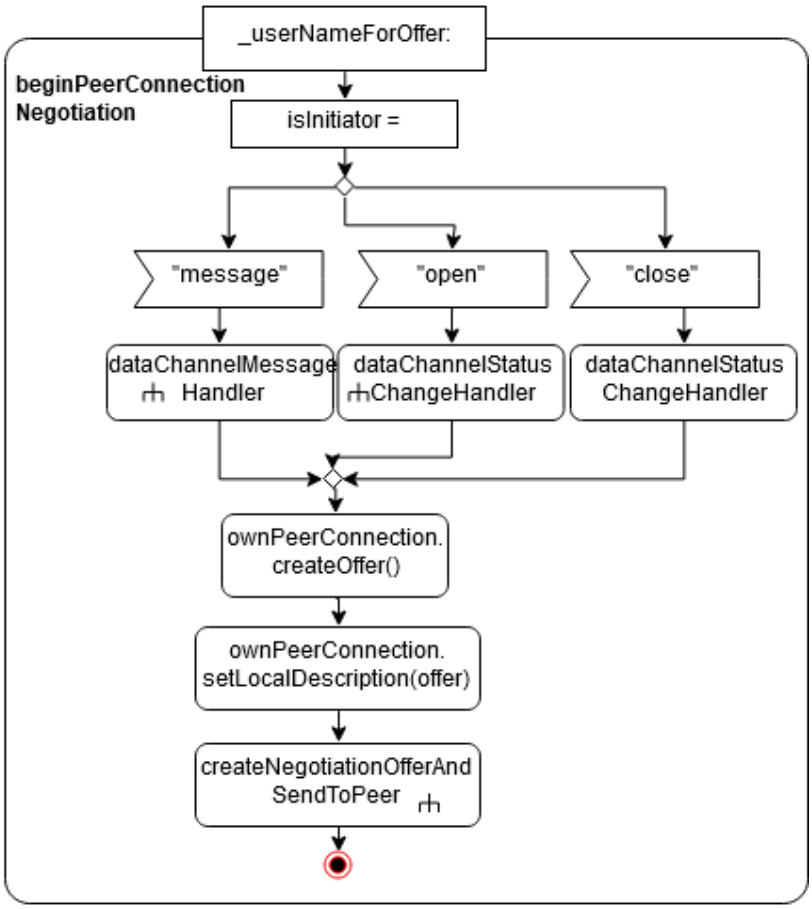


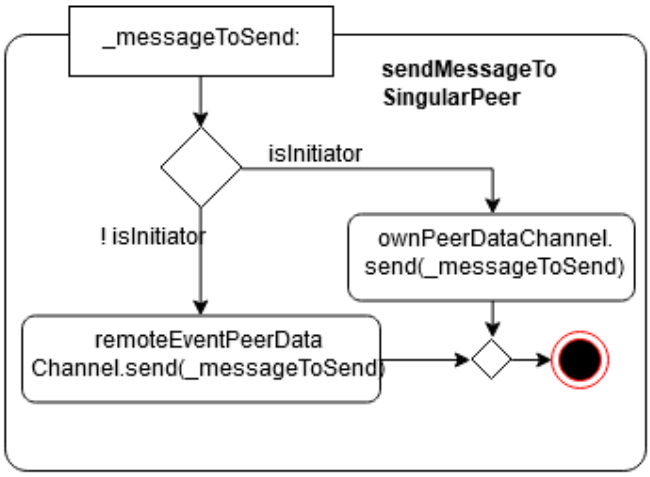
Anhang 3.4: ClientManager Pure WebSocket



Anhang 3.5: ClientManager Single Peer




_userNameForOffer:
beginPeerConnection
Negotiation
isInitiator =
"message"
"open"
"close"
dataChannelMessage
Handler
dataChannelStatus
ChangeHandler
dataChannelStatus
ChangeHandler
ownPeerConnection.
createOffer()
ownPeerConnection.
setLocalDescription(offer)
createNegotiationOfferAnd
SendToPeer


_messageToSend:
sendMessageTo
SingularPeer
isInitiator
! isInitiator
ownPeerDataChannel.
send(_messageToSend)
remoteEventPeerData
Channel.send(_messageToSend)

Eidesstaatliche Erkärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Thesis selbständig und ohne unzulässige fremde Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel, sind angegeben.

Ort und Datum Unterschrift

Furtwangen, den.