Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelorarbeit

Björn Kulas

WebRTC basiertes Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

**Björn Kulas**

# WebRTC basiertes Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Hans Heinrich Heitmann
Zweitgutachter : Prof. Dr. Kai von Luck

Abgegeben am 09.04.2015

**Björn Kulas**

**Thema der Arbeit**

WebRTC basiertes Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung.

**Stichworte**

WebRTC, WebSocket, DataChannel, EKG, Web-Browser

**Kurzzusammenfassung**

Das Ziel dieser Bachelorarbeit ist die Entwicklung eines Echtzeitdaten-Managementsystem, welches Echtzeitdaten vom Sender zum Empfänger über eine Peer-to-Peer Verbindung überträgt und anschließend in einem Browser zur Betrachtung visualisiert. Die praktische Lösung wurde beispielhaft für die Übertragung von EKG Daten entworfen.

**Björn Kulas**

**Title of the paper**

WebRTC-based peer-to-peer real-time-data management system with browser supported visualization.

**Keywords**

WebRTC, WebSocket, DataChannel, ECG, Web-Browser

**Abstract**

The object of this bachelor thesis is the development of a real-time data-management system which real-time data from the transmitter to the receiver via peer-to-peer connection transmits and afterwards in a browser to observation visualizes. The practical solution is exemplified designed for the transmission of ECG data.

# Inhaltsverzeichnis


1 Einführung ........ 10

1.1 Motivation ........ 10

1.2 Zielsetzung ........ 11

1.3 Gliederung der Arbeit ........ 11

2 Eingesetzte Echtzeit Standards ........ 12

2.1 WebSocket ........ 12

2.1.1 Einleitung ........ 12

2.1.2 Hintergründe zur Entwicklung ........ 13

2.1.3 Beispiel: Was macht Echtzeit in Web-Anwendungen aus ........ 14

2.1.4 Vorteile und Eigenschaften des WebSocket-Protokolls ........ 16

2.1.5 Verbindungsaufbau einer WebSocket-Verbindung ........ 17

2.1.6 WebSocket Messages ........ 21

2.1.7 Payload Masking ........ 24

2.1.8 Abbau einer WebSocket-Verbindung ........ 28

2.2 WebRTC ........ 29

2.2.1 Einleitung ........ 29

2.2.2 Entwicklung ........ 30

2.2.3 Architektur ........ 30

2.2.4 WebRTC Protokolle ........ 33

2.2.5 Die RTCPeerConnection ........ 37

2.2.6 Signaling Channel ... 39
2.2.7 Herstellen der RTCPeerConnection ... 40
2.2.8 Der RTCDataChannel ... 42
2.2.9 Kommunikation über den RTCDataChannel ... 44
**3 Entwicklungs- & Anforderungsanalyse ... 45**
3.1 Entwicklungsszenario ... 45
3.2 Anwendungsfälle ... 47
3.3 Anforderungsanalyse ... 52
3.3.1 Funktionale Anforderungen ... 52
3.3.2 Nicht funktionale Anforderungen ... 61
**4 Design ... 69**
4.1 WebSocket-Server ... 71
4.1.1 Message-System ... 71
4.1.2 Peer Endpoint ... 73
4.1.3 Chat Endpoint ... 79
4.2 Web-Anwendung ... 82
4.2.1 Senderliste ... 82
4.2.2 Chat ... 83
4.2.3 EKG Kurve ... 83
4.3 EKG Daten Sender ... 84
**5 Realisierung ... 86**
5.1 WebSocket-Server in Java ... 86
5.1.1 Message-System ... 90
5.1.2 Encoder ... 91
5.1.3 Decoder ... 94
5.1.4 Konsolenausgabe ... 94
5.1.5 WebSocket Handshake ... 95
5.1.6 Chat Endpoint ... 96
5.1.7 Peer Endpoint ... 97

5.2 Web-Seite ........................................ 102
5.2.1 Senderliste ........................................ 103
5.2.2 EKG Kurve ........................................ 104
5.2.3 Chat ........................................ 111
5.2.4 Konsole ........................................ 113
5.2.5 Informationen ........................................ 113
5.3 EKG Daten Sender ........................................ 115
5.3.1 Vorwort ........................................ 115
5.3.2 Allgemeines Verhalten des EKG Daten Senders ........................................ 115
5.3.3 Implementierung für Web-Browser ........................................ 120
5.3.4 Implementierung für Node.JS ........................................ 121
5.4 Verwendung fremden Codes ........................................ 123
5.4.1 cbuffer.js ........................................ 123
5.4.2 adapter.js ........................................ 123
5.4.3 Node-webrtc ........................................ 123
5.4.4 WS ........................................ 123
5.4.5 Hashmap ........................................ 124
5.4.6 ECGSYN ........................................ 124

**6 Experimente ........................................ 125**

6.1 Datenvolumen einer Peer-to-Peer Verbindung ........................................ 125
6.1.1 Maximales Datenvolumen ........................................ 128
6.2 Simultaner Betrieb von Peer-to-Peer Verbindungen ........................................ 130
6.2.1 Datenvolumen mehreren Peer-to-Peer Verbindungen ........................................ 131
6.2.2 Auslastung der EKG Daten Sender Lösungen ........................................ 132
6.3 Datenfluss unter Auslastung des Uploads ........................................ 134

**7 Schlussfolgerung ........................................ 136**

7.1 Zusammenfassung ........................................ 136
7.2 Reflexion ........................................ 137
7.3 Fazit ........................................ 138
7.4 Ausblick ........................................ 139

# Abbildungsverzeichnis

Abbildung 2.1.3-1.: WebSocket Kommunikation: Aktienkurs ........ 14
Abbildung 2.1.5-2.: WebSocket URI’s, aus [RFC6455] ........ 17
Abbildung 2.1.5-3.: Einsatz einer verschlüsselten WebSocket-Verbindung, aus [KAAZING]. 18
Abbildung 2.1.5-4.: Clientseitiger Handshake beim Verbindungsaufbau, aus [RFC6455] ..... 19
Abbildung 2.1.5-5.: Serverseitiger Handshake beim Verbindungsaufbau, aus [RFC6455] .... 20
Abbildung 2.1.5-6.:Generierung des Werts für das Header-Feld Sec-WebSocket-Accept .... 21
Abbildung 2.1.6-7.: Ablauf einer WebSocket-Verbindung ........ 23
Abbildung 2.1.7-8.: Clientseitige WebSocket Text Message. ........ 24
Abbildung 2.1.7-9.: UTF-8 Darstellung eines String ........ 25
Abbildung 2.1.7-10.: Maskieren der Payload ........ 26
Abbildung 2.1.7-11.: Demaskieren der maskierten Payload ........ 27
Abbildung 2.1.7-12.: Darstellung von UTF-8 in String ........ 28
Abbildung 2.2.3-13.: Die Architektur von WebRTC, aus [WebRTCorg] ........ 31
Abbildung 2.2.3-14.: Differenz der Browser API's, aus [WebRTCorg] ........ 32
Abbildung 2.2.4-15.: WebRTC Protokolle, aus [BroNet] ........ 34
Abbildung 2.2.4-16.: Offer/Answer-Modell, aus [BroNet] ........ 37
Abbildung 2.2.6-17.: Mögliche Varianten eines Signaling Channels zwischen zwei Peers, aus [BroNet] ........ 40
Abbildung 2.2.7-18.: Offer/Answer Modell mit Einsatz des SDP, aus [BroNet] ........ 41
Abbildung 2.2.9-19.: Anwendungsfälle ........ 47
Abbildung 4-20.: Übersicht der Komponenten des Managementsystems ........ 69
Abbildung 4.1-21.: Komponente - WebSocket-Server ........ 71
Abbildung 4.1.1-22.: WebSocket-Server - Message-System ........ 72
Abbildung 4.1.2-23.: Peer Endpoint – Registrierung ........ 74
Abbildung 4.1.2-24.: Peer Endpoint – Verteilung der Senderliste ........ 76
Abbildung 4.1.2-25.: Peer Endpoint - Signaling ........ 78
Abbildung 4.1.3-26.: Chat Endpoint – Asynchrone Nachrichtenübermittlung ........ 80
Abbildung 4.1.3-27.: Chat Endpoint – Nachrichtenübermittlung an mehrere Instanzen ...... 81
Abbildung 4.2-28.: Komponente - Web-Anwendung ........ 82
Abbildung 4.3-29.: EKG Daten Sender - Komponente ........ 84
Abbildung 5.1-30.: Package Diagramm des WebSocket Server ........ 86
Abbildung 5.1-31.:WebSocket Server Klassen - POJO Endpoints ........ 88
Abbildung 5.1-32.: Java WebSocket Annotationen, aus [JavaEE7Tut] ........ 89
Abbildung 5.1.1-33.: Message-System - Realisierung ........ 91
Abbildung 5.1.2-34.: Encoder/Decoder - Realisierung ........ 93

Abbildung 5.1.4-35.: Konsolenausgabe eines EventConsoleWriter Objekts ........................ 95
Abbildung 5.2-36.: Screenshot der Web-Seite .................................................................. 102
Abbildung 5.2.1-37.: Senderliste - Einträge ...................................................................... 103
Abbildung 5.2.2-38.: Darstellung von EKG Kurve und Disconnect Button .......................... 104
Abbildung 5.2.2-39.: Session Description Protocol der Offer Message .............................. 107
Abbildung 5.2.3-40.: Darstellung des Chat ...................................................................... 112
Abbildung 5.2.3-41.: Darstellung von gesendeten und empfangenen Chatnachrichten .... 112
Abbildung 5.2.4-42.: Konsole mit Statusinformationen .................................................... 113
Abbildung 5.2.5-43.: Darstellung von Informationen zum Buffer und der aktuellen Verbindung zum EKG Daten Sender ........................................................................... 114
Abbildung 6.1-44.:Wireshark 10 Pakete mit 40 Samples .................................................... 126
Abbildung 6.1-45.: Upload Volumen über 10 Minuten ..................................................... 127
Abbildung 6.1.1-46.:Maximales Datenvolumen ............................................................... 128
Abbildung 6.2-47.: Test im HITeC e.V. 3S Labor ................................................................ 130
Abbildung 6.2.1-48.: Datenvolumen beim Senden an mehrere Instanzen der Web-Anwendung ....................................................................................................... 131
Abbildung 6.2.2-49.: Zeit beim Versenden der EKG Daten ............................................... 132
Abbildung A.1.1-50.: Darstellung einer clientseitigen Handshake-Einleitung, aus [RFC6455] ........................................................................................................................ 141
Abbildung A.1.1-51..: Darstellung einer serverseitigen Handshake-Antwort, aus [RFC6455] ........................................................................................................................ 144
Abbildung A.1.2-52..: Frame des Base-Framing-Protocol, aus [RFC6455] nach [RFC5234]. 148
Abbildung A.1.3-53..: Beispielsequenz bei der Verwendung von Polling, aus [WSAaEiW] . 150
Abbildung A.1.3-54..: Beispielsequenz bei der Verwendung von Long-Polling, aus [WSAaEiW] ....................................................................................................... 151
Abbildung A.1.4-55..: Vergleich der Kommunikation zwischen HTTP und Ajax, aus [ANAtWA] ........................................................................................................................ 154

# 1 Einführung

## 1.1 Motivation

Es gibt heutzutage ein großes Angebot an Web-Anwendungen, darunter sind soziale Netzwerke, Spiele, Content Communities, Office-Anwendungen und vieles mehr. Der größte Unterschied zur klassischen Anwendung ist die Auslagerung der Anwendung von der lokalen Festplatte ins Internet und die Integration der Benutzeroberfläche in einen Web-Browser.
Web-Anwendungen haben durch die Auslagerung der Anwendung ins Internet einen großen Vorteil gegenüber klassischen Anwendungen – Sie sind für viele Benutzer erreichbar. Dieser Vorteil hat dazu geführt, dass viele Web-Anwendungen als Mehrbenutzersysteme konzipiert wurden.
Neben der Abhängigkeit von einer Internetverbindung gibt es allerdings auch Nachteile die bei der Nutzung einer Web-Anwendung bestehen. Da bisher die meisten Web-Anwendungen ähnlich einem Client-Server-Modell konzipiert worden sind, muss die Kommunikation zweier Benutzer immer über den Server der Web-Anwendung ausgetauscht werden. Zusätzlich sieht das Internetprotokoll HTTP nicht vor, das ein Server einem Client ohne vorherige Anfrage eine Nachricht zusenden kann.
Der modellbedingte Kommunikationsweg und die Kommunikationseinschränkungen des Verwendeten HTTP, kosten Zeit, erzeugen viel Overhead und belastet so vor allem den Server der Web-Anwendung. Entwickler waren daher sehr kreativ, um Web-Anwendungen die gewünschte Dynamik zu verleihen und den Server der Web-Anwendung zu entlasten. Allerdings ist keine der entwickelten Lösungen kompromisslos – bisher.
So wurde Ende 2011 die Entwicklung des WebSocket-Protokolls abgeschlossen, was zwar weiterhin nur ein Client-Server-Model erlaubt aber die asynchrone Kommunikation zwischen Client und Server ermöglicht.
Noch in der Spezifikation befindlich aber trotzdem schon in aller Munde ist WebRTC ein modernes Framework, mit dem Multimediadaten in Echtzeit über eine Peer-to-Peer Verbindung kommuniziert werden können.
Die Entwicklung dieser beiden Protokolle hat unterschiedliche Ziele und könnte somit zukünftige Web-Anwendungen für Einsatzbereiche hervorbringen, für die sich die Entwicklung einer Web-Anwendung bisher als unrentabel oder unpraktisch dargestellt hat.
In dieser Bachelorarbeit werden die Fähigkeiten der beiden Protokolle, bei der Entwicklung eines Peer-to-Peer Echtzeitdaten-Managementsystems für die Beobachtung von Sensordaten, betrachtet.

## 1.2 Zielsetzung

Das Ziel dieser Bachelorarbeit ist die Entwicklung eines Peer-to-Peer Echtzeitdaten-Managementsystems mit Browser unterstützter Visualisierung in Form einer Web-Anwendung, unter Einsatz des WebSocket Protokolls und dem WebRTC Framework. Neben der Entwicklung ist auch die Betrachtung der beiden Protokolle beim Einsatz in einer Web-Anwendung ein Ziel dieser Bachelorarbeit.
Die Web-Anwendung nutzt neue Möglichkeiten, die das WebRTC Framework bietet. Die Verwendung von WebRTC ermöglicht ein Peer-to-Peer Kommunikationsmodell für Web-Anwendungen zu realisieren, ohne das die Installation eines Web-Browser Plug-In notwendig ist. Die Verwendung einer Peer-to-Peer Verbindung reduziert die Latenz bei der Datenübertragung, da die Daten nicht mehr über einen Server kommuniziert werden müssen, um von einem Peer zum anderen Peer zu gelangen. Der Verzicht auf einen Server bei der Datenübertragung reduziert neben der Latenz auch die Kosten beim Einsatz der Web-Anwendung.
Die Web-Anwendung bietet ein Daten-Managementsystem zur Betrachtung von visualisierten Echtzeitdaten an. Die Echtzeitdaten werden von Senderanwendungen, die auf einer Menge entfernter Peers ablaufen, verschlüsselt via Peer-to-Peer Verbindung über das Internet und Netzwerke kommuniziert.
Bei der Entwicklung werden eine möglichst hohe Plattformunabhängigkeit und ein kostengünstiger Einsatz angestrebt, um ein hohes Maß an Einsatzmöglichkeiten zu erzielen.
Für die praktische Lösung wurde ein Szenario gewählt, bei dem die Web-Anwendung für die Übertragung und Beobachtung von EKG Daten genutzt wird. Die zu übertragenden und betrachtenden EKG Daten entsprechen einem simulierten EKG Signal, welches aus einer lokalen Datei ausgelesen wird.

## 1.3 Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen der verwendeten Protokolle WebSocket und WebRTC betrachtet und dessen Arbeitsweisen beschrieben. Ein Entwicklungsszenario wird in Kapitel 3 festgelegt, um das in Kapitel 1.2 beschriebe System für ein bestimmtes Szenario anzupassen und zu entwickeln. Anschließend werden Anwendungsfälle für das gewählte Entwicklungsszenario ermittelt. Aus diesen und der Zielsetzung werden Anforderungen abgeleitet und in der Anforderungsanalyse beschrieben. In Kapitel 4 werden das Design des zu entwickelnden Systems und dessen Komponenten vorgestellt. Die Realisierung des entwickelten Systems wird in Kapitel 5 dargestellt und beschrieben. In Kapitel 6 erfolgt eine Beschreibung zu Experimenten, die mit dem entwickelten System durchgeführt wurden. Die Schlussfolgerung dieser Bachelorarbeit erfolgt in Kapitel 7.

# 2 Eingesetzte Echtzeit Standards

Das in Kapitel 2.1 beschriebene WebSocket Protokoll ist ein Netzwerkprotokoll zur asynchronen bidirektionalen Übermittlung von Daten zwischen Client und Server. Es eignet sich daher sehr für den Einsatz in Web-Anwendungen und wird in dieser Bachelorarbeit verwendet, um Lösungen für einen Signaling Server, die Verwaltung einer Senderliste und einen Multi-user Chat umzusetzen.
Das in Kapitel 2.2 beschriebene WebRTC ist ein Framework zur Echtzeit Übertragung von Multimediadaten über eine Peer-to-Peer Verbindung zwischen Web-Browsern. WebRTC bietet mit der Peer-to-Peer Verbindung zwischen Web-Browsern, eine neue Art der Übertragung von Daten in Echtzeit, wodurch neue Möglichkeiten für die Verwendung von Web-Browsern geschaffen wurden. WebRTC wird in dieser Bachelorarbeit für die Übertragung von EKG Daten, über eine verschlüsselte Peer-to-Peer Verbindung, an einen Web-Browser verwendet.

## 2.1 WebSocket

### 2.1.1 Einleitung

Das WebSocket-Protokoll wurde durch die Internet Engineering Task Force [IETF] im RFC6455 [RFC6455] spezifiziert und ist von der Internet Engineering Steering Group [IESG] freigegeben worden. Eine WebSocket-API [WSAPI] wurde vom World Wide Web Consortium [W3C], als Teil der HTML5 [HTML5WS] Spezifikation, entworfen.
Das WebSocket-Protokoll ist ein TCP [RFC793] basiertes Kommunikationsprotokoll zwischen Client und Server. Es unterstützt einen asynchronen und bidirektionalen Datenaustausch zwischen Client und Server. Für den Verbindungsauf- und Verbindungsabbau sieht das WebSocket-Protokoll einen Handshake zwischen Client und Server vor. Die Kommunikation zwischen Client und Server erfolgt in sogenannten Messages. Das genutzte Sicherheitsmodell ist herkunftsbasierend [RFC6454] und wird von Web-Browser unterstützt. Zusätzlich kann das Verschlüsselungsprotokoll TLS (Transport Layer Security) [RFC2818] für einen verschlüsselten Datenaustausch genutzt werden.
Mit der Entwicklung des WebSocket-Protokolls wollte man eine Möglichkeit bereitstellen, die es browserbasierte Anwendungen ermöglicht, bidirektional mit einem Server zu kommunizieren. Das WebSocket-Protokoll ermöglicht dies, ohne das dazu mehrere HTTP-Verbindung (Hypertext Transfer Protocol) [RFC2616] genutzt werden müssen.

In Zeiten des „Web 2.0“ ist das WebSocket-Protokoll sehr nützlich. Die asynchrone bidirektionale Kommunikation zwischen Client und Server entspricht den Anforderungen von modernen Web-Anwendungen. Web-Anwendungen können so ein sehr reaktionsfreundlich Verhalten bieten, sodass Nutzereingaben in „Echtzeit“ vom Server bearbeitet werden.
Mögliche Anwendungsmöglichkeiten sind Nachrichtenticker, Börsenticker, Messanger, Spiele und serverseitige Multi-User-Anwendungen. Weitere Anwendungsmöglichkeiten bietet unter anderem die Web-Seite websocket.org [wsDEMO] mit anschaulichen Beispielanwendungen, die mit dem Einsatz vom WebSocket-Protokoll umgesetzt wurden.

### 2.1.2 Hintergründe zur Entwicklung

Der zuvor genannte Begriff „Web 2.0“ [Web2_0] beschreibt eine massive Veränderung des World-Wide-Web, im folgendem Text nur noch „Web“ genannt. Die Veränderung bezieht sich auf die Verwendung, den Umgang und die Wahrnehmung durch die Benutzer. Das Web wird als eigenständige Plattform betrachtet. Es dient Nutzern nicht mehr nur als Informationenquelle wie es in der Vergangenheit war. Durch die Verwendung entsprechender Web-Anwendungen gibt es Nutzern die Möglichkeit eigene Inhalte im Web zur Verfügung zu stellen.
Web-Seiten dienen einer Web-Anwendung zur Visualisierung im Web-Browser des Benutzers. Web-Anwendungen erreicht ein Nutzer durch den Aufruf der entsprechenden Web-Seite. Eine solche Web-Seite ist die Benutzeroberfläche der Web-Anwendung. Es handelt sich also um Anwendungen, für dessen Nutzung lediglich einen Web-Browser benötigt wird. Die eigentliche Anwendung wird auf einem entfernten Server ausgeführt, zu dem der Web-Browser eine Verbindung herstellt. Web-Anwendungen wie Youtube, Twitter, Flickr, Google Docs oder Facebook sind heute fast jedem Menschen ein Begriff und werden von Millionen Benutzern täglich genutzt. Diese Web-Anwendungen ermöglichen es den Nutzer Inhalte im Internet für andere bereitzustellen, gemeinsam an ihnen zu arbeiten, zu spielen oder sich auszutauschen.
Für all diese Web-Anwendungen ist die Übertragung von Daten in „Echtzeit“ ein wichtiges Kriterium.
Ins Internet gestellte Daten sollen dort auch sofort für Andere zugänglich sein. Eingaben eines Nutzers sollen direkt von Web-Anwendungen zu einer Reaktion führen. Eine Web-Anwendung kann mit einer lokalen Anwendung nur konkurrieren, wenn sie dieser nicht im reaktionsfreudigen Verhalten nachsteht.

### 2.1.3 Beispiel: Was macht Echtzeit in Web-Anwendungen aus

Ein simples Beispiel ist ein Benutzer der einen Aktienkurs betrachtet. Beim betrachten wird erwartet, dass immer der aktuelle Aktienkurs zu sehen ist.

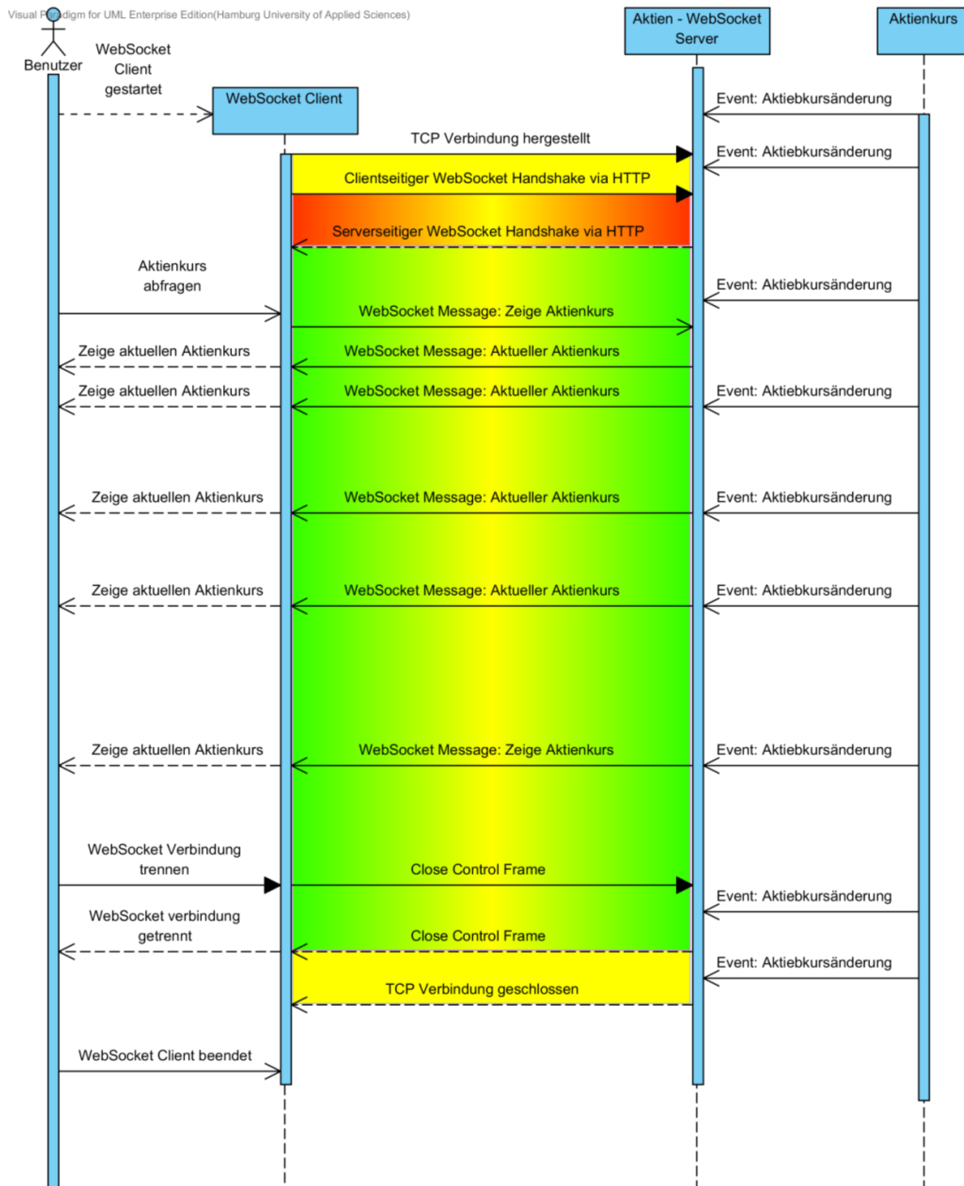



Abbildung 2.1.3-1.: WebSocket Kommunikation: Aktienkurs

In Abbildung 2.1.3-1 wird von einem Benutzer ein WebSocket-Client genutzt, der den Aktienkurs von einem WebSocket-Server erfragt.

Der Bereich mit dem gelben Hintergrund steht für eine TCP-Verbindung. Der rote Hintergrund entspricht der HTTP-Anfrage des clientseitigen Handshakes und der HTTP-Antwort des serverseitigen Handshakes. Der Bereich mit dem grünen Hintergrund steht für die WebSocket-Verbindung. Der gelbe Bereich der TCP-Verbindung schimmert auch durch den roten und grünen Bereich. Dies soll das Aufsetzen vom HTTP und WebSocket-Protokoll auf die TCP-Verbindung symbolisieren. Der Aufbau einer WebSocket-Verbindung per HTTP wird im späteren Kapitel 2.1.5 beschrieben.
Im Beispiel aus Abbildung 2.1.3-1 wird ein Verhalten beschrieben, bei dem das etablierte HTTP nicht mehr den aktuellen Anforderungen der Nutzer an das Web entspricht. Das HTTP ist ein Übertragungsprotokoll, das vor über 20 Jahren eingeführt wurde. Heute ist es das klassische Protokoll, wenn es darum geht, Web-Seiten in einen Web-Browser zu laden. Allerdings basiert es auf Anfragen, die vom Nutzer ausgelöst werden.
Wenn ein Nutzer eine Web-Seite im Web-Browser aufruft, wird eine entsprechende Anfrage vom Web-Browser an den Server geschickt, auf dem die Web-Seite gespeichert ist. Die Anfrage an den Server ist erforderlich, um die Daten der Web-Seite zu erhalten. Erhält der Web-Browser die angefragten Daten vom Server, nutzt er diese für die Darstellung der Web-Seite im Web-Browser. Der Nutzer sieht nun die aufgerufene Web-Seite in seinem Web-Browser.
Das Senden der angefragten Daten ist eine Reaktion des Servers auf die Anfrage des Web-Browser. Die Anfrage des Web-Browser ist eine Reaktion auf die Eingabe des Benutzers. Der Server benötigt also die Aktion des Benutzers, damit dieser die aktuellen Daten über seinen Web-Browser betrachten kann.
Ändert sich anschließend der Inhalt der Web-Seite, erfährt der Nutzer dies nicht. Es ist nötig dass der Nutzer wieder eine Eingabe am Web-Browser tätigt, damit dieser die aktuellen Daten vom Server erhält. Erst dann wird die im Web-Browser dargestellte Web-Seite aktualisiert. Es gibt also zwei nacheinander erfolgender unidirektionale Kommunikationswege: Anfragen vom Web-Browser an den Server und eine Antwort vom Server an den Web-Browser.
Dieses Verhalten eignet sich nicht für die bereits oben genannten modernen Anforderungen an Web-Anwendungen, die möglichst in „Echtzeit“ funktionieren sollen.
Es war bisher auch ohne WebSocket-Protokoll möglich, mit einer Web-Anwendung bidirektional zu kommunizieren. Web-Entwickler mussten dafür bisher allerdings in die Trickkiste greifen. Entstanden sind simple Techniken wie Long-Polling, beschrieben in Kapitel A.1.3, oder auch komplexere Lösungen wie die Ajax-Technologie, dessen Arbeitsweise im Kapitel A.1.4 beschrieben wird. Ideenreich mussten die Web-Entwickler sein, da HTTP als etabliertes Übertragungsprotokoll der Internetprotokollfamilie nicht für eine bidirektionale Kommunikation vorgesehen war. Das HTTP ist ein Anfrage/Antwort-Protokoll. Einer Antwort vom Server geht also immer eine vorherige Anfrage voraus. Diese verhalten ist synchron. Das WebSocket-Protokoll ermöglicht dem Server, Daten auch ohne vorherige Anfrage an einen Verbunden Client wie einen Web-Browser zu senden. Das Verhalten des WebSocket-Protokolls kann daher als asynchron bezeichnet werden.

### 2.1.4 Vorteile und Eigenschaften des WebSocket-Protokolls

Wie in der Einleitung dieses Kapitels beschrieben, ist das WebSocket-Protokoll TCP-basierend. Es setzt wie HTTP [RFC2616] und auch viele andere komplexe Protokolle auf TCP [RFC793] auf. Dies ist kein Zufall, ist TCP wohl das bisher erfolgreichste standardisierte Transportprotokoll. TCP wird unabhängig von gängigen Betriebssystemen eingesetzt, um zuverlässige Verbindungen zwischen zwei Hosts herzustellen. Es wird also immer eine TCP-Verbindung erstellt, wenn eine WebSocket-Verbindung aufgebaut wird.
Das WebSocket-Protokoll ist trotz der zuvor genannten Vorteile gegenüber HTTP nicht dessen Ersatz. Das WebSocket-Protokoll ist in aktueller Version sogar vom HTTP abhängig, was sich zukünftig aber ändern kann:

> *The WebSocket Protocol attempts to address the goals of existing bidirectional HTTP technologies in the context of the existing HTTP infrastructure; as such, it is designed to work over HTTP ports 80 and 443… However, the design does not limit WebSocket to HTTP, and future implementations could use a simpler handshake over a dedicated port without reinventing the entire protocol.*
>
> - The WebSocket Protocol [RFC6455]

Der Aufbau einer WebSocket-Verbindung wird mit einem HTTP Update Request eingeleitet. Dies bringt einen sehr großen Vorteil mit sich. So ist es möglich, dass WebSocket-Protokoll in einer Infrastruktur zu nutzen, die bisher nur für die Verwendung von HTTP ausgelegt war. Dies ermöglicht, bisher eingesetzte kompromissbehaftete Lösungen zur asynchronen bidirektionalen Kommunikation, zu ersetzen. Als kompromissbehaftete Lösungen können unter Anderen Polling, Long-Polling [RFC6202] XMLHTTPRequest [XMLHttpRequest] oder die Ajax-Technologie gezählt werden.
Diese Techniken sind alle entstanden, um Web-Seiten und Web-Anwendungen dynamischer erscheinen zu lassen. Allerdings konnten die Nachteile von HTTP meist nur reduziert aber nicht ausgeglichen werden. Genauere Informationen zur Arbeitsweis und den Problemen von Polling und Long-Polling liegen im Kapitel A.1.3 vor, die der Ajax-Technologie in Kapitel A.1.4.
Eine WebSocket-Verbindung hat deutliche Vorteile, die je nach verglichener Lösung mehr oder weniger deutlich ausfallen können. Ein Vorteil ist der geringere Overhead. WebSocket Messages haben im Gegensatz zu HTTP Headern einen geringen Overhead. Zudem braucht man bei der Verwendung des WebSocket-Protokolls, keine unnötigen Statusanfragen an den Server zu senden, um zu erfahren ob sich der Status geändert hat.
Ein weiterer Vorteil ist, die reduzierte serverseitige Verwaltungsarbeit. Durch den Einsatz des WebSocket-Protokolls, wird für gesamte Kommunikation nur eine TCP-Verbindung, pro verbundenen Client, benötigt.
Der größte Vorteil ist, dass das WebSocket-Protokoll eine echt asynchrone bidirektionale Verbindung erzeugt. Womit es dem Server und dem Client jederzeit möglich ist, eine

Message an den Remote Host zu senden, ohne das dieser vorher eine Anfrage senden muss, ein Event eintrifft oder ein periodischer Zeitraum verstrichen ist.
Wie schon erwähnt wurde das WebSocket-Protokoll so entworfen, dass es im Einklang mit HTTP ist. Es arbeitet dementsprechend auch über die Standardports von HTTP bzw. HTTPS. Daher ist der Einsatz des WebSocket-Protokolls in einer Netzinfrastruktur möglich, die bisher nur für HTTP vorgesehen war. Somit ist auch der Einsatz von nur einem Server für HTTP und das WebSocket-Protokoll einfach umzusetzen und in vielen Fällen auch sehr sinnvoll.

### 2.1.5 Verbindungsaufbau einer WebSocket-Verbindung

**WebSocket URI`s**

Um eine Verbindung mit einem Web-Browser zu einer Web-seite herzustellen, wird eine entsprechende Adresseingabe benötigt. Solche Adresse sind URI's (Uniform Resource Identifier) [RFC3986]. Die in Web-Browser am häufigen genutzten Typen sind wohl die des HTTP bzw. HTTPS. Wie bereits beschrieben, nutzt das WebSocket–Protokoll die Ports von HTTP und HTTPS. Daher gibt es auch zwei Arten von URI`s für das WebSocket-Protokoll, wie in Abbildung 2.1.5-2 dargestellt. Die folgende Beschreibung gilt für beide Arten von WebSocket-URI's, die dem Standardaufbau einer URI folgen.
Somit beschreibt die Angabe des Host, die Domain eines WebSocket-Servers. Die Angabe des Path beschreibt den Namen eines Endpoints. Ein Endpoint entspricht einem Dienst, den der WebSocket-Server anbietet. Ein WebSocket-Server kann eine Menge von Endpoints anbieten, die unterschiedlichen Diensten entsprechen und so für verschiedene Zwecke genutzt werden können.

```
ws-URI = "ws:" "//" host [ ":" port ] path [ "?" query ]
wss-URI = "wss:" "//" host [ ":" port ] path [ "?" query ]
```

**Abbildung 2.1.5-2.: WebSocket URI's, aus [RFC6455]**

Die beiden WebSocket-URI's unterscheiden sich nur durch die Angabe des Typs. Bei einer einfachen WebSocket-Verbindung wird der Typ mit „ws:“ bezeichnet und der Standard HTTP Port 80 genutzt. Für eine sichere WebSocket-Verbindung bzw. WebSocketSecure-Verbindung, bei der eine TLS Verschlüsselung für die Kommunikation eingesetzt wird, wird der Typ mit „wss:“ bezeichnet. In diesem Fall wird der Standard HTTP Port 443 genutzt. Empfehlenswert ist es, eine möglichst starke Verschlüsselung [WSC_UIG] auszuwählen.
Die Arbeitsweise einer TLS Tunnels funktioniert bei einer WebSocket-Verbindung genau wie bei einer HTTP-Verbindung. Beide Protokolle setzen auf TCP auf. Der TLS Tunnel verschlüsselt dabei die verwendete Verbindung, so dass über die genutzte TCP-Verbindung verschlüsselte Daten übertragen werden. Die Abbildung 2.1.5-3 verdeutlicht die genannte Arbeitsweise.

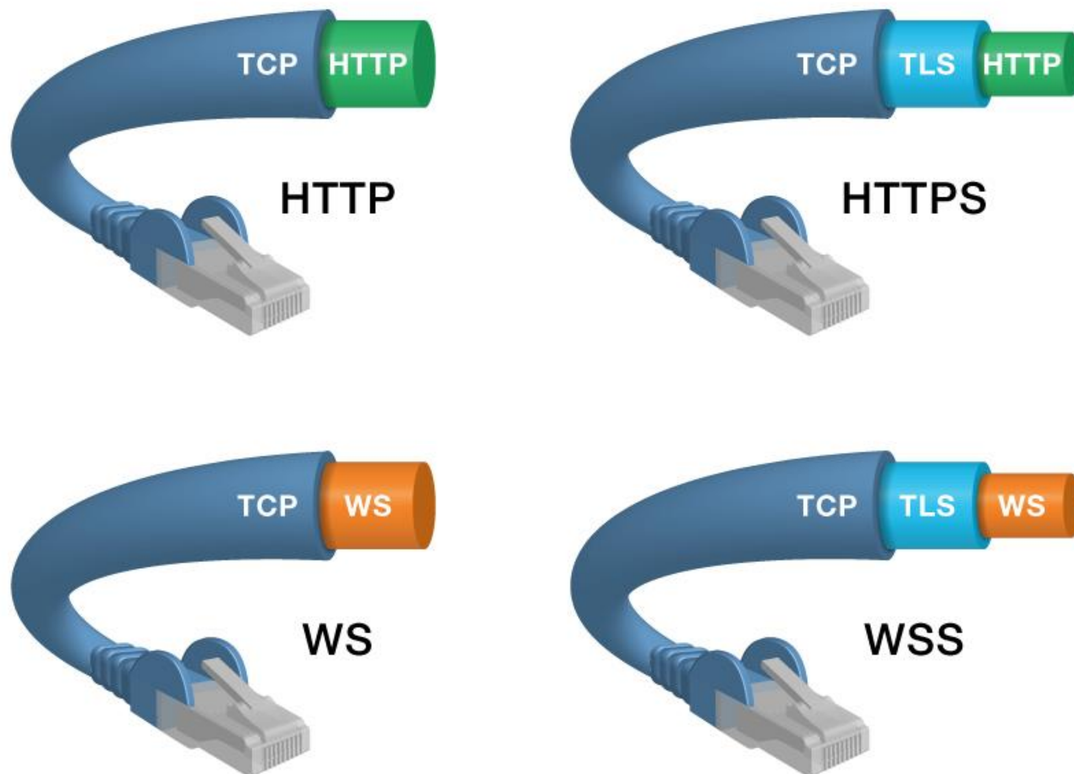



Abbildung 2.1.5-3.: Einsatz einer verschlüsselten WebSocket-Verbindung, aus [KAAZING]

Nach der Typ-Angabe folgt in einem URI die Angabe des Hosts. Die Angabe des Host kann optional mit einer Portangabe erweitert werden, wenn dieser vom Standard abweicht. Darauf folgt die Angabe des Pfads. Die Pfadangabe entspricht dem Pfad des gewünschten Endpunktes. Ein serverseitiger Endpunkt entspricht dabei einem Dienst, den ein Server für einen bestimmten Einsatzzweck anbietet. Ein Server kann eine Vielzahl von Endpunkten und damit Diensten anbieten. Optional kann nach der Pfadangabe eine Query-Angabe erfolgen.

**Clientseitiger Handshake beim verbindungsaufbau**

Der Handshake beim Verbindungsaufbau geht vom Client aus. Um den WebSocket-Handshake zu initiieren, sendet der Client ein HTTP Upgrade Request. In der Anfrage gibt es nach der ersten Zeile sogenannte Header-Fields bzw. Header-Felder [RFC2616], in denen alle für den Handshake nötigen und optionalen Angaben angegeben werden. Die Reihenfolge der Header-Felder ist unbedeutend und kann abhängig der Implementation variieren. Die Abbildung 2.1.5-4 entspricht einem Beispiel für einen clientseitigen Handshake, womit die WebSocket-Verbindung zum Server eingeleitet wird.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

**Abbildung 2.1.5-4.: Clientseitiger Handshake beim Verbindungsaufbau, aus [RFC6455]**

Im clientseitigen Handshake, welcher in der Abbildung 2.1.5-4 zu sehen ist, soll eine WebSocket-Verbindung zum Server Endpunkt „/chat" hergestellt werden. Der Client wünscht, die folgende Kommunikation über die WebSocket-Verbindung mit dem Sub-Protokoll „chat" oder „superchat" fortzuführen. Zudem wird im Feld „Sec-WebSocket-Key" ein zufälliger Base64 [RFC4648] codierter 16 Byte langer Wert an den Server geschickt.
Eine detaillierte Beschreibung aller Header-Felder des Clientseitigen Handshake liegt im Kapitel A.1.1. Der Handshake kann noch weitere Header-Felder [RFC6265] enthalten, beispielsweise zur Erweiterung des WebSocket-Protokolls, für die Verwendung von Cookies [RFC2616] oder zur Authentifizierung.
Nachdem der Client den initiierenden Teil des Handshakes zum Aufbau einer WebSocket-Verbindung an den Server gesendet hat, muss er auf dessen Antwort warten, bevor er weitere Daten senden darf.

**Serverseitige Überprüfung des empfangen clientseitigen Handshake**

Wenn ein WebSocket-Server den clientseitigen Handshake erhalten hat, muss der WebSocket-Server die Angaben aus den Header-Feldern prüfen. Die Prüfung erfolgt bevor der Server mit einer generierten akzeptierenden Handshake-Antwort antwortet und eine WebSocket-Verbindung hergestellt wird. Da im obigen Beispiel, aus Abbildung 2.1.5-4, alle nötigen Header-Felder mit zulässigen Werten angegeben wurden, kommt es nicht zum Abbruch des Handshake bzw. des Verbindungsaufbaus. Nur wenn das der Fall ist, antwortet der Server mit dem Statuscode 101 in der HTTP-Status-Line [RFC2616], wie es in Abbildung 2.1.5-5 zu sehen ist.
An dieser Stelle wird auf zwei wichtige Informationen eingegangen, die der WebSocket-Server überprüft. Eine detaillierte Beschreibung der serverseitigen Prüfung aller Header-Felder des Clientseitigen Handshake liegt im Kapitel A.1.1 vor.
Der Server überprüft, ob er eins der Sub-Protokolle des empfangen Handshake aus Abbildung 2.1.5-4 unterstützt.

Der Server kann auch das Header-Feld „Origin“ des empfangenen Handshakes aus Abbildung 2.1.5-4 auswerten. Dieses Header-Feld kann eine Herkunftsangabe enthalten, die die Herkunft des Handshake Initiators angibt.
Browser geben diese Information selbst an, eine Manipulation per JavaScript oder HTML ist über die API nicht möglich. Allerdings brauchen native Clients dieses Feld nach Spezifikation [RFC6455] nicht befüllen, da sie die Information fälschen könnten. Durch die Möglichkeit der Fälschung verliert die Information an Aussagekraft. Native Clients könnten sich so mit einer gefälschten aber korrekt aussehenden Angabe als Browser ausgeben und den Server so irritieren. Das liegt daran, dass der Server nicht unterscheiden kann, ob der Handshake von einem Browser oder von einem nativen Client initiiert wurde.
Das Header-Feld „Origin“ hat trotzdem noch Bedeutung und kann für folgende Möglichkeiten genutzt werden. Der Server kann Handshakes, anhand einer unerwünschten Herkunftsangabe, abbrechen. Dies ermöglicht die Nutzung einer White- oder Backlist. Eine weitere Möglichkeit ergibt sich für Verbindungen zu Browsern. Durch die korrekte Herkunftsangabe von Browsern, ist der Server vor unerwünschten cross-origin Zugriffen geschützt.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

**Abbildung 2.1.5-5.: Serverseitiger Handshake beim Verbindungsaufbau, aus [RFC6455]**

In der Abbildung 2.1.5-5 ist eine mögliche serverseitige Handshake Antwort an den Client zu sehen. In dieser bestätigt der Server die weitere Kommunikation über die WebSocket-Verbindung mit dem Sub-Protokoll „chat“. Der Server unterstützt also eins der vom Client gewünschten Sub-Protokolle für die Kommunikation.
Der Server erzeugt einen Wert für das Header-Feld „Sec-WebSocket-Accept“. Es ist in der Handshake Antwort, das Gegenstück zum Header-Feld „Sec-WebSocket-Key“, des clientseitigen Handshake. Der Server tut dies um dem Client zu beweisen, dass der Handshake vom korrekten WebSocket-Server empfangen und beantwortet wurde. Dies ist nötig, damit der Client eine WebSocket-Verbindung zum Server herstellt. Der Server muss dazu zwei Teilinformationen kombinieren und daraus den benötigten Wert bilden. Zu kombinieren sind der aus dem clientseitigen Handshake empfangene Wert aus dem Header-Feld „Sec-WebSocket-Key“ und ein konstanter Wert namens „Globally Unique Identifier“.
Im Folgenden ist in Abbildung 2.1.5-6 veranschaulicht, wie der Server für die Handshake Antwort an den Client aus Abbildung 2.1.5-5, den Wert für das Header-Feld „Sec-WebSocket-Accept“ erzeugt hat:

Abbildung 2.1.5-6.:Generierung des Werts für das Header-Feld Sec-WebSocket-Accept

Anschließend wird die erstellte serverseitige Handshake Antwort aus Abbildung 2.1.5-5 per HTTP an den wartenden Client gesendet.

**Clientseitige Überprüfung des empfangen serverseitigen Handshake**

Nachdem Empfang der serverseitigen Handshake-Antwort, muss der Client die Antwort prüfen. Im einfachsten oder optimalen Fall, wie im obigen Beispiel aus Abbildung 2.1.5-5, erhält der Client eine vollständige und mit gültigen Werten befüllte Handshake-Antwort vom Server zurück. Der Client kann der Handshake Antwort, den Wechsel auf das WebSocket-Protokoll und die Verwendung des Sub-Protokolls „chat“ entnehmen. Zusätzlich muss der Client den Inhalt des Header- Felds „Sec-WebSocket-Accept“ überprüfen.
Der Client kann den Wert vorhersagen. Der Client bestimmt den Wert, nach dem gleichen Schema wie der Server. Ist der Wert korrekt, ist für den Client bewiesen, dass der WebSocket-Server den Handshake beantwortet hat, an den der Client zuvor den Handshake initiiert hat.
Erst dann ist der Client bereit eine WebSocket-Verbindung herzustellen. Dieses Verhalten schützt den Server vor nicht-WebSocket-Verbindungen und vor Angreifern, die korrumpierten Pakete benutzen, welche XMLHTTPRequest [XMLHttpRequest] nutzen oder ausführbare Formulare beinhalten.

## 2.1.6 WebSocket Messages

Nachdem der verbindende Handshake zwischen Client und Server erfolgreich abgeschlossen wurde, wird eine WebSocket-Verbindung hergestellt. Bei der WebSocket-Verbindung handelt es sich um einen asynchronen bidirektionalen Kommunikationskanal, über den beide Hosts unabhängig vom Anderen Daten senden und empfangen können – Vollduplex. Die Abbildung 2.1.6-7 skizziert den kompletten Ablauf einer WebSocket-

Verbindung. Der im Kapitel 2.1.5 beschriebene Handshake zum Verbindungsaufbau per HTTP, ist mit den Sequenznummern 2 und 2.1 abgebildet.
Die Daten, die über eine WebSocket-Verbindung gesendet werden, werden „Messages“ [RFC6455] genannt. Eine Message kann aus einem oder mehreren Frames eines Typs bestehen. Ergänzende Informationen zum Inhalt und Aufbau von Frames werden im Kapitel A.1.2 aufgeführt. Die Fragmentierung der Frames wird auf dem Transportweg bestimmt.
Es gibt drei Typen von Frames: Text-, Binary- und Control-Frames. Der Typ eines Frames wird durch einen entsprechenden OP-Code, im Header eines Frames, definiert und identifiziert.
Daten vom Typ „Text“ werden in UTF-8 [RFC3629] interpretiert. Daten vom Typ „Binary“ sind durch Anwendungen, die die WebSocket-Verbindung nutzen, selbst zu interpretieren.
Control-Frames können Anwendungsdaten enthalten, sind aber nicht für die Nutzung von Anwendungen gedacht, sondern werden durch das WebSocket-Protokoll selbst genutzt. Control-Frames werden beispielsweise zur Verbindungstrennung, was im Kapitel 2.1.8 beschrieben wird, genutzt.
Das WebSocket-Protokoll nutzt „framing“ aus zwei Gründen. Der erste Grund ist, das Frame-basiertes arbeiten selbst. Der zweite Grund ist, die durchs „framing“ ermöglichte Unterscheidung zwischen Unicode- und Binärdaten.
Das WebSocket-Protokoll setzt mit dem Framing-Mechanismus auf TCP auf, um dessen Segment-Mechanismus zu nutzen, ohne aber von dessen Längenbegrenzung der Segmente beeinträchtig zu werden.
Werden beim Aufbau der WebSocket-Verbindung Protokollerweiterungen verhandelt, ist dessen Einfluss bzw. Manipulation auf Messages zu beachten. Nimmt eine Protokollerweitung Einfluss auf die Messages, muss dieser der Spezifikation der entsprechenden WebSocket-Protokollerweiterung entnommen werden. Den Einfluss von Protokollerweiterungen, gerade bei Verwendung mehrerer Protokollerweiterungen, ist im Kapitel A.1.1 beschrieben.

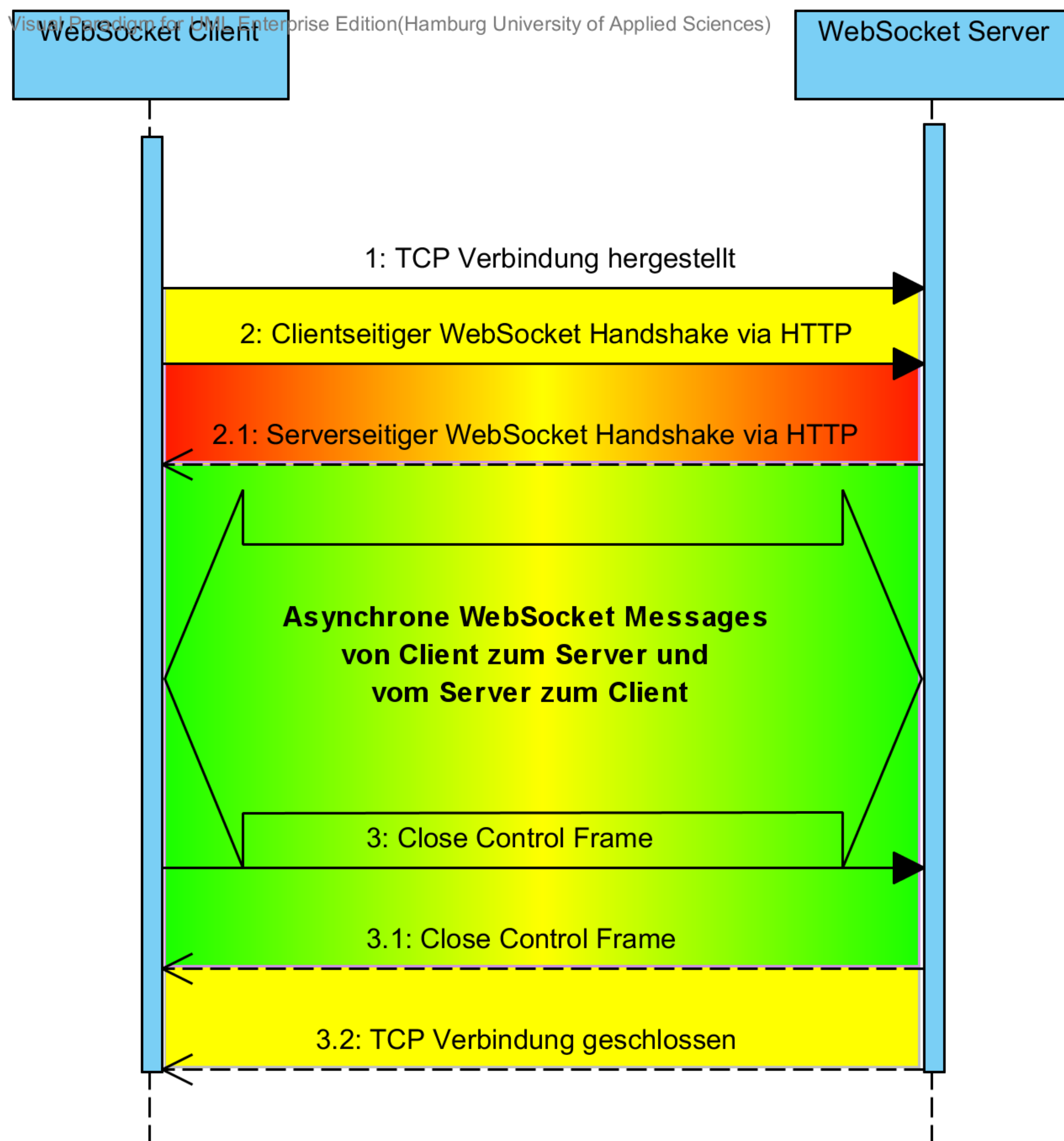



**Abbildung 2.1.6-7.: Ablauf einer WebSocket-Verbindung**

Daten werden in den oben genannten Messages transportiert. Messages bestehen aus Frames. Somit werden Daten indirekt in einer Sequenz von Frames übertragen. Zwischen Frames, die vom Client zum Server und vom Server zum Client gesendet werden, besteht ein Unterschied, der im folgenden Kapitel 2.1.7 aufgezeigt wird.

### 2.1.7 Payload Masking

Die Nutzdaten, die in einem Frame übertragen werden, werden „Payload“ genannt. Frames die vom Client an den Server gesendet werden, transportieren eine maskierte Payload. Maskiert bedeutet, dass die Daten nicht im Klartext bzw. lesbar transportiert werden. Die Maskierung dient dazu, dass die Daten nicht mehr ohne einen passenden Schlüssel gelesen werden können. Diese Maskierung ist unabhängig davon, ob die WebSocket-Verbindung durch einen verschlüsselten TLS-Kanal, also eine WebSocketSecure-Verbindung, genutzt wird oder nicht.
Denn die Maskierung ist keine Verschlüsselung. Der „masking key“ zur Demaskierung steckt im selben Frame, wie auch die Payload. Fängt man einen Frame ab, sieht man auch den „masking-key“. In der Abbildung 2.1.7-8 ist eine WebSocket-Message mit dem Programm „Wireshark“ dargestellt. Wendet man den „masking key“ in einem bestimmten Algorithmus auf die Daten an, kann man diese in Klartext einsehen. So ist es auch dem Programm möglich, die Payload demaskiert anzuzeigen.

```
⊟ WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x00
    .... 0001 = Opcode: Text (1)
    1... .... = Mask: True
    .011 0110 = Payload length: 54
    Masking-Key: 7a0dfd84
  ⊟ Payload
      Text: 012f94e05837dff31f6f8eed0e68dfa8586f8feb0d7e98f6...
  ⊟ Unmask Payload
      [Text unmask: {"id":"website","browserWSID":"","regType":1,"type":2}]
```

Abbildung 2.1.7-8.: Clientseitige WebSocket Text Message.

Das Maskieren der Payload hat einen wichtigen Grund. Während der Entwicklung des WebSocket-Protokolls wurde ein Experiment [TtYfFaP] durchgeführt, bei welchem demonstriert wurde, wie der Cache von Proxys-Servern mit schädlichem Code versehen werden kann, wenn clientseitige Messages nicht maskiert sind. Daraus resultierte der Vorschlag, der auch für das WebSocket-Protokoll übernommen wurde, die Daten vom Client an den Server zu maskieren. Durch das Maskieren der Payload ist es für einen Angreifer nicht möglich zu bestimmen, wie Daten im Frame aussehen. Damit wird unterbunden, dass ein Angreifer Messages erzeugen kann, die von einem Proxy-Server als HTTP-Anfrage missinterpretiert werden können.
Im folgenden Abschnitt wird das „Payload-Masking“ anhand eines Beispiels veranschaulicht.
Möchte man die Zeichenkette „Hamburg“ von einem Client über eine WebSocket-Verbindung zum Server übertragen, wird man dafür wahrscheinlich eine Message vom Typ „Text“ nutzen. Der Frame, aus dem die Message besteht, interpretiert diese Zeichenkette in UTF-8.

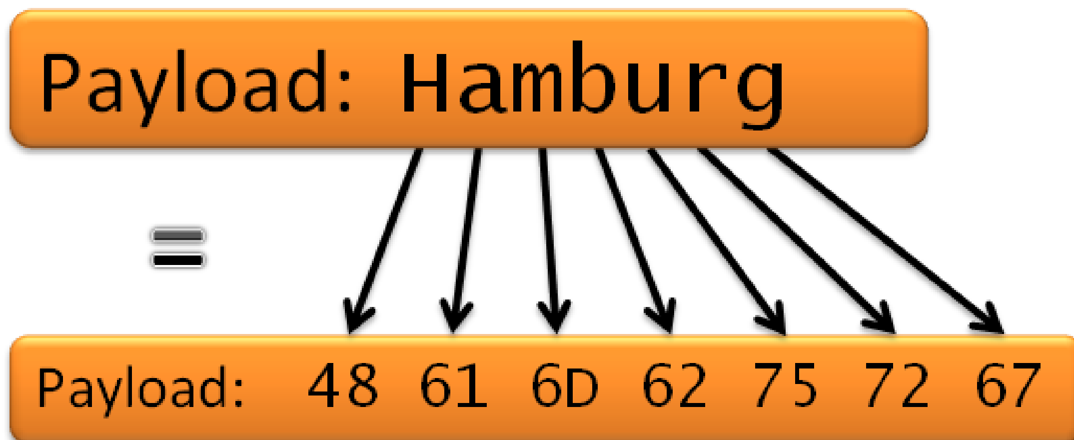



Abbildung 2.1.7-9.: UTF-8 Darstellung eines String

Jedes Zeichen der Zeichenkette entspricht einem Oktett. In Abbildung 2.1.7-9 wird die Zeichenkette „Hamburg“ in UTF-8 dargestellt.
Der Client wählt beim Versenden jedes Frames einen 32-bit langen „masking-key“ aus. Der „masking-key“ ist ein zufällig gewählter Wert aus einer endlichen Menge. Dieser wird, wie auch die Zeichen in der UTF-8-Codierung, als Oktett interpretiert. Die länge von 32-bit entspricht daher vier Oktetts. Folgender Algorithmus wird genutzt um eine Payload zu maskieren:

Masked Payload = Payload[i] XOR Masking-Key[i%4]

In Abbildung 2.1.7-10 ist die Maskierung der Payload dargestellt. Dazu wird die Payload aus Abbildung 2.1.7-9 verwendet. Jedes Oktett des Masking-Key wird dabei fortlaufend mit einem Oktett der Payload XOR-Verknüpft. Der Modulo-Operator aus dem Algorithmus gibt an, dass die Oktetts des Masking-Key dabei wiederholend, nach Anwendung auf vier Oktetts der Payload, auf die folgenden Oktetts der Payload angewandt werden.

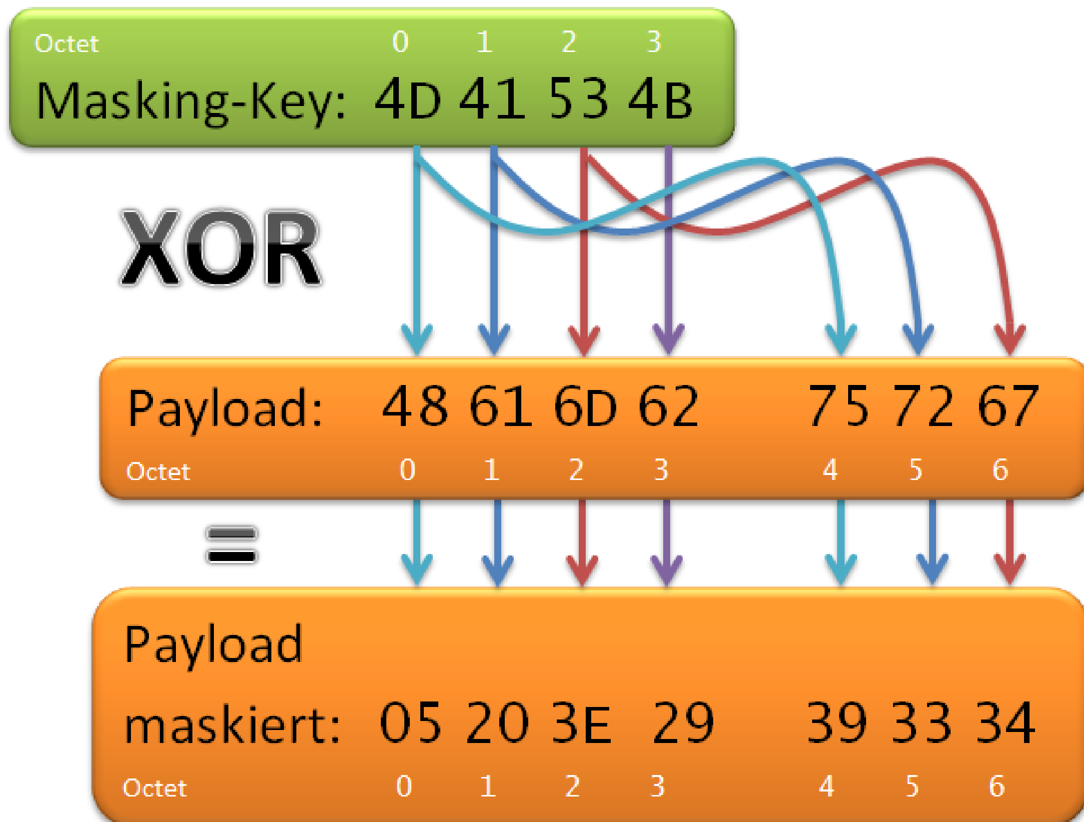



Abbildung 2.1.7-10.: Maskieren der Payload

Der Client sendet nun einen Frame mit der maskierten Payload und dem Masking-Key an den Server. Der Server wendet den gleichen Algorithmus auf die maskierte Payload an, um diese zu demaskieren:

Payload = Masked Payload[i] XOR Masking-Key[i%4]

Das Demaskieren der maskierten Payload wird in Abbildung 2.1.7-11 dargestellt. Der Server kann anschließend die übertragene Zeichenkette normal per UTF-8 interpretieren.

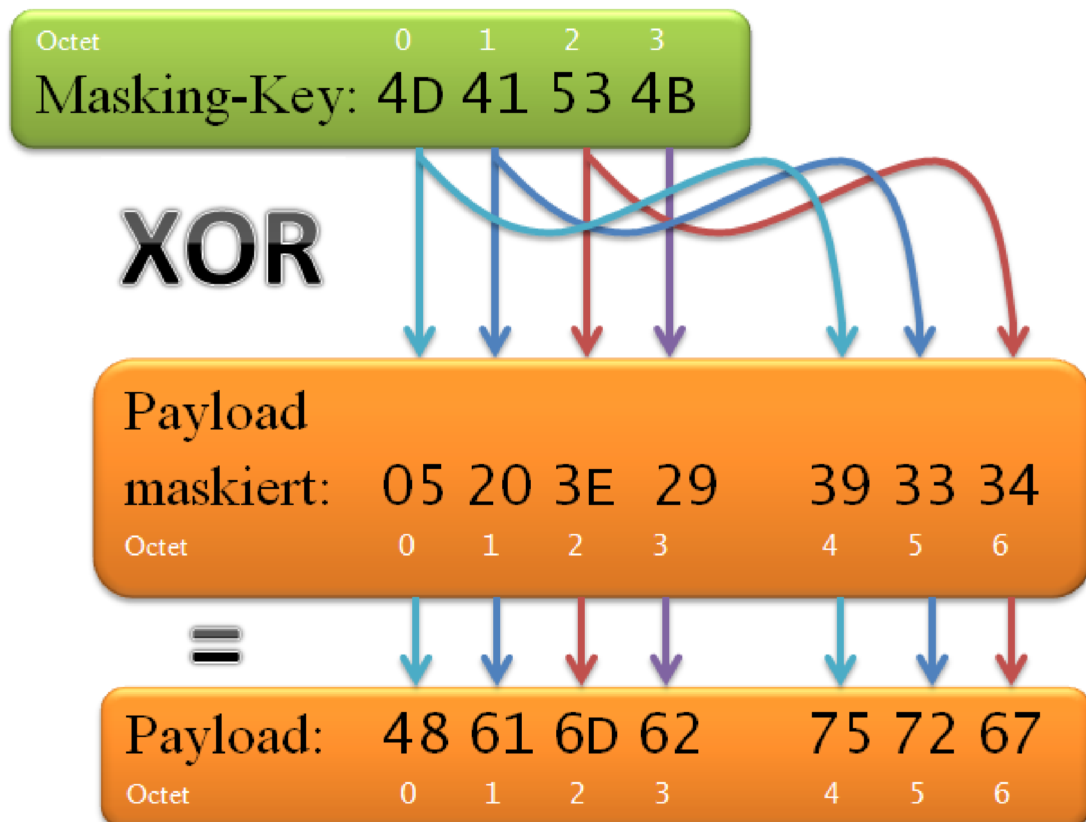



Abbildung 2.1.7-11.: Demaskieren der maskierten Payload

Nach der Konvertierung in Textform, wie in Abbildung 2.1.7-12 dargestellt, erhält man wieder die Zeichenkette, die vom Client versendet wurde.

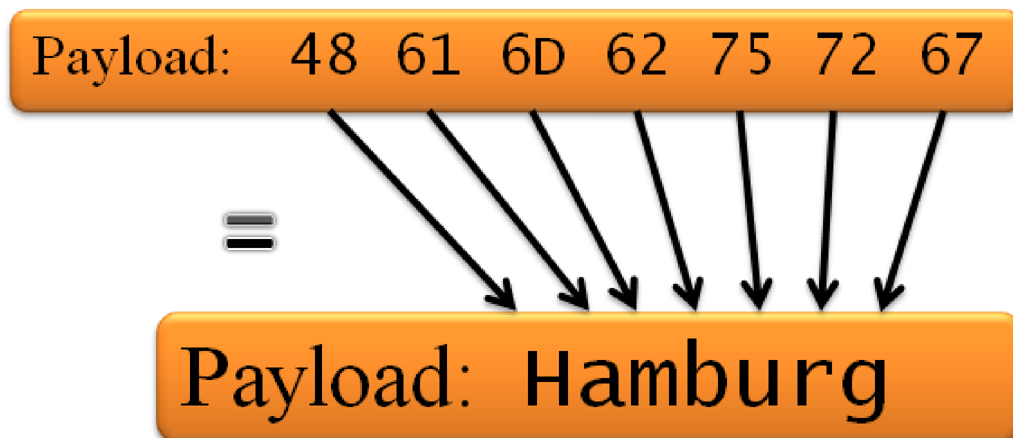



Abbildung 2.1.7-12.: Darstellung von UTF-8 in String

## 2.1.8 Abbau einer WebSocket-Verbindung

Im Gegensatz zum Handshake beim Verbindungsaufbau, der wie zuvor in Kapitel 2.1.5 beschrieben über das HTTP abgewickelt wird, wird der Handshake zur Verbindungstrennung über „Control Frames" des WebSocket-Protokolls vollzogen.
Der Ablauf ist deutlich simpler als beim Verbindungsaufbau. Wenn ein Host, der Client oder der Server die Verbindung trennen möchte, ist dieser Host der Initiator.
Der Initiator sendet ein Close-Control-Frame zum anderen Host und wartet bis von diesem Host auch ein Close-Control-Frame empfangen wurde. Empfängt ein Host einen Close-Control-Frame, antwortet dieser mit einem Close-Control-Frame, wenn er zuvor noch kein Control-Frame gesendet hat. Anschließende eingehende Daten werden verworfen von dem Host.
Nachdem ein Host einen Close-Control-Frame verschickt hat, sendet dieser keine Daten mehr.
Der Handshake ist dann abgeschlossen, wenn beide Host jeweils einen Close-Control-Frame erhalten und auch gesendet haben. Die genutzte TCP-Verbindung wird im Anschluss von einem Host geschlossen. Der Client sollte eine bestimmte Zeit warten bevor er die TCP-Verbindung trennt, damit die TCP-Verbindung möglichst vom Server geschlossen wird. Ein Sever sollte die Verbindung nach dem erfolgreichen Handshake zur Verbindungstrennung direkt schließen und so dem Client zuvorkommen.
Close-Control-Frames können Anwendungsdaten enthalten, darin ist es möglich den Grund der Verbindungstrennung anzugeben.

## 2.2 WebRTC

In den folgenden Kapiteln werden die Funktionen und Möglichkeiten von WebRTC äußerlich Betrachtet und Erklärt, so dass ein Basisverständnis für den Einsatz in dieser Arbeit entsteht. Es wird dabei nur auf Teile der WebRTC API eingegangen, die auch praktische Anwendung in der Arbeit gefunden haben.
Alle in diesen Kapiteln beschriebenen Eigenschaften können sich zukünftig Ändern. Denn wie im Kapitel 2.2.2 beschrieben wird, befindet sich WebRTC noch in der Entwicklung und ist noch nicht final Spezifiziert.

### 2.2.1 Einleitung

WebRTC steht für Web Real-Time Communication. Es handelt sich bei WebRTC um ein offenes Framework, mit passender Web-Browser API. WebRTC ermöglicht die Echtzeit Kommunikation zwischen modernen Browsern, ohne das Plug-ins oder weitere Software benötigt wird. Unterstützt wird auch das Betriebssystem Android, welches vor allem auf Smartphones eingesetzt wird. WebRTC kann so neben Web-Anwendungen für den Browser, auch für mobile Anwendungen eingesetzt werden.
Der Einsatz von WebRTC ermöglicht den Austausch von Video, Audio und beliebigen anderen Daten, in einer verschlüsselten Echtzeit Peer-to-Peer Verbindung. Eine solche Peer-to-Peer Verbindung ermöglicht eine asynchrone bidirektionale Übertragung von Daten, ohne dass die Daten über einen vermittelnden Server transportiert werden müssen. Um dies zu erreichen bringt das Framework WebRTC viel mit. Es nutzt effiziente und sichere Protokolle in Kombination, für die Datenübertragung. Die Wahl der verwendeten Protokolle hängt dabei vom Einsatz der Peer-to-Peer Verbindung ab. Es werden Protokolle für die NAT-Traverse, wie ICE, STUN und TURN genutzt. Erprobte Video- und Audio-Engines sind ebenfalls enthalten. Der Einsatz dieser Protokolle erlaubt auch die Integration von existierenden Kommunikationssystemen, die auf Basis von VOIP arbeiten. Alle diese Möglichkeiten und Funktionen sind in ihrer ganzen Komplexität hinter einer JavaScript API verborgen, was die Benutzung vereinfacht. Es braucht nur einige Zeilen Code, um eine Videokonferenz oder Datenübertragung per Browser zu ermöglichen.
Neben der offiziellen Implementierung der Browser- und Android-API, gibt es mittlerweile einige native oder auch Framework basierte Implementierungen der WebRTC API.
Die Verbreitung hält trotz der nicht abgeschlossenen Entwicklung, wie in Kapitel 2.2.2 genannt wird, weiter Einzug. Das Potential von WebRTC ist dank der genannten Möglichkeiten riesig und die Entwicklung wird von vielen Unternehmen verfolgt. Vor allem die Möglichkeit der Kommunikation mittels Web-Browser, wie Web-Telefonie oder Web-Videochat ist sehr interessant:

*„In other words, WebRTC is not only about bringing real-time communication to the browser, but also about bringing all the capabilities of the Web to the telecommunications world—a $4.7 trillion industry in 2012! Not surprisingly, this is a significant development and one that many existing telecom vendors, businesses, and startups are following closely. WebRTC is much more than just another browser API. "*

- Ilya Grigorik [BroNet]

### 2.2.2 Entwicklung

Die Entwicklung wurde Anfangs von Google begonnen. Im Verlauf der Entwicklung haben weitere Firmen teilgenommen, darunter sind Mozilla, Apple und Opera. Seit Mitte 2011 wird die Browser API von der WebRTC Working Group des World Wide Web Consortium [W3C] weiterentwickelt und befindet sich bis heute in der Entwicklung.
Die letzte veröffentlichte Version [WebRTCpub] ist von 2013, auf die sich aktuell noch viele Implementierungen beziehen. Die veröffentlichte Version gilt allerdings weder als stabil und für kommerzielle Zwecke geeignet, noch als abgeschlossen. Es wird noch eine signifikante Weiterentwicklung seitens der Web Real-Time Communications Working Group erwartet. Die heute aktuellste Version der Editor's Draft [WebRTCedi], ist von Ende 2014 und somit recht Aktuell.
Für die WebRTC Kommunikation wird eine ganze Reihe von Echtzeit-Protokollen und JavaScript API's genutzt. Die Protokollspezifikationen für Peer-to-Peer Verbindungen werden von der IETF [IETF] RTCWEB Group entwickelt. Die Entwicklung von API's, für den Zugriff auf Hardware, wird von der Media Capture Task Force [W3C] betrieben.

### 2.2.3 Architektur

Die Architektur von WebRTC besteht wesentlich aus zwei Teilen, wie in Abbildung 2.2.3-13 dargestellt ist. Dabei Handelt es sich um das WebRTC Framework und die Web-Browser API, welche folgend in diesem Kapitel beschrieben werden.

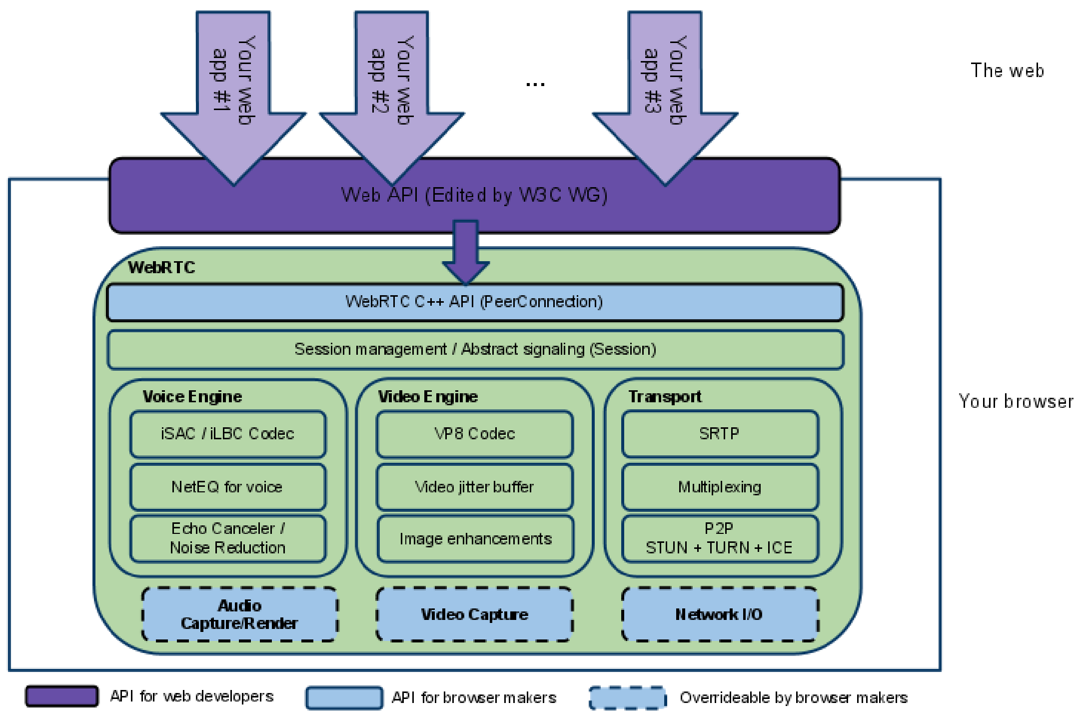


Abbildung 2.2.3-13.: Die Architektur von WebRTC, aus [WebRTCorg]

**Web-API für Web-Entwickler**

Die Web-API dient zur Entwicklung von Web-Anwendungen, bei denen WebRTC eingesetzt werden soll. Die API greift auf die Bestandteile des Frameworks zu und bietet entsprechende Methoden für den Web-Entwickler, in Form von einfach zu nutzenden JavaScript API's. Als Web-Entwickler ist zu beachten, das Chrome und Firefox aktuell nicht das standardisierte Interface der W3C API verwenden. Jeder Browser nutzt aktuell noch ein eigenes angepasstes Interface. Die Interfaces bieten die nötigen Funktionalitäten an. Allerdings unterscheiden sich die Namen der Funktionen, wie in Abbildung 2.2.3-14 zu sehen ist. Bei der Entwicklung ist daher auf eine einheitliche Verwendung eines Browser nötig oder die Integration aller Interfaces. Letzteres ist sinnvoller und durch vorhandene Adapter-Skripte auch einfach umzusetzen. In Googles WebRTC Repository wird ein Adapter-Skript unter folgendem Link [ADAP] angeboten.

```
W3C Standard           Chrome                   Firefox
--------------------------------------------------------------
getUserMedia           webkitGetUserMedia       mozGetUserMedia
RTCPeerConnection      webkitRTCPeerConnection  mozRTCPeerConnection
RTCSessionDescription  RTCSessionDescription    mozRTCSessionDescription
RTCIceCandidate        RTCIceCandidate          mozRTCIceCandidate
```

Abbildung 2.2.3-14.: Differenz der Browser API's, aus [WebRTCorg]

**Interface für Entwickler von Web-Browsern**

Das WebRTC Framework aus Abbildung 2.2.3-13 beschreibt alles was sich hinter den Funktionen der API verbirgt. Abgebildet ist als oberster Layer die „WebRTC C++ API“. Dieses Interface dient den Entwicklern von Web-Browsern, bei der Implementierung der Web-API in ihren Web-Browsern. Wie zuvor in diesem Kapitel erwähnt, nutzen die Web-Browser aktuell noch eigene angepasste Interfaces.
Auf der Abbildung des Framework ist unter der „WebRTC C++ API“ ein Bereich zu sehen, der mit „Session Management / Abstract signaling (Session)“ bezeichnet ist. Dieser symbolisiert eine abstrahierte Sitzungsschicht für die Konfiguration und Verwaltung von Verbindungen. Entwicklern wird es so ermöglicht, Einfluss auf die verwendeten Protokolle zu nehmen.

**Audio- und Video-Engine**

Unter dem Layer mit der Bezeichnung „Session Management / Abstract signaling (Session)“ gibt es drei Bereiche. Diese drei Bereiche zeigen die Hauptfunktionalitäten des Frameworks und sind bezeichnet mit „Voice Engine“, „Video Engine“ und „Transport“. Das WebRTC Video- und Audio-Engines mitbringt, wurde in der Einleitung erwähnt.
Mit Hilfe dieser Engines und Codecs ist es möglich die komplexen Anforderungen der Echtzeitkommunikation mit einem Browser zu erfüllen, ohne auf Software anderer angewiesen zu sein. Problematische Prozesse, wie etwa die Synchronisierung der Medienströme zwischen den Peers bei schwankender Bitrate und Latenz, werden durch die gebotenen Engines behandelt und gelöst. Mehrere Codecs die für unterschiedliche Umstände optimiert sind, werden von den Engines geboten und automatisch eingesetzt. Die Engines sind daher in der Lage mediale Daten zu optimieren, zu codieren und decodiert an den Web-Browser zu übergeben. Unter den Bereichen der Video- und Audio-Engines in Abbildung 2.2.3-13 sind überschreibbare „Capture“-Bereiche dargestellt. Diese ermöglichen den Entwicklern von Web-Browsern, eigene Verfahren für das Einfangen von Audio- und Video-Material zu nutzen. Dies bietet den Entwicklern viele Möglichkeiten, die über eine Aufnahme der Web-Cam und dessen Mikrophones hinausgehen.

**Transport**

Der dritte Bereich des Frameworks ist auf Abbildung 2.2.3-13 rechts dargestellt und mit „Transport“ bezeichnet. Dieser ermöglicht den Aufbau und die Verwaltung einer RTCPeerConnection [RTCWEB JSEP] zwischen zwei Peers. Um eine Verbindung zwischen zwei Peers herzustellen, werden die Protokolle ICE [RFC5245], STUN [RFC5389] und TURN [RFC5766] genutzt.
ICE ist ein protokollgetriebenes Verfahren zur Herstellung einer Verbindung zwischen zwei Peers. Das STUN Protokoll wird genutzt, um eine Verbindung über die Grenzen von NAT’s hinweg herzustellen und aufrecht zu erhalten. Das TURN Protokoll wird verwendet, wenn es mit Hilfe des STUN Protokolls nicht möglich war eine Verbindung über die Grenzen von NAT’s hinweg herzustellen. In diesem Fall ist es eine Backup-Lösung, bei der die Daten zwischen den Peers über einen TURN-Server transportiert werden. Der Verbindungsaufbau einer Peer-to-Peer Verbindung, wird im Kapitel 2.2.7 beschrieben.
Über eine RTCPeerConnection kann jede Art von Daten transportiert werden, unabhängig davon ob Video-, Audio- oder Anwendungsdaten transportiert werden sollen. Das Framework ermöglicht auch den Datenaustausch von mehreren RTCPeerConnections in Echtzeit. Das ist beispielsweise bei einer Telefon- oder Videokonferenz mit mehreren Teilnehmern nötig. Bei solch einer Konferenz von mehreren Peers, sind mehrere RTCPeerConnection zu multiplexen, damit eine Kommunikation mit allen Teilnehmern parallel stattfinden kann.

### 2.2.4 WebRTC Protokolle

Im Einführungskapitel 2.2.1 wurde bereits erwähnt, dass WebRTC einige Protokolle mitbringt. Die auf der rechten Seite abgebildeten Protokolle in Abbildung 2.2.4-15, werden beim Einsatz von WebRTC verwendet. Die Grundlagen der Protokolle werden in diesem Kapitel zusammengefasst und deren Einsatz erklärt. Das Verständnis für das Zusammenspiel der Protokolle ist wichtig, um den Verbindungsaufbau zu verstehen. Die Beschreibung des Verbindungsaufbaus einer RTCPeerConnection mit einem symmetrischen RTCDataChannel wird in Kapitel 2.2.7 erklärt.

**UDP [RFC768]**

Das „User Datagram Protocol“ ist Teil der Internetprotokollfamilie. Es arbeitet auf der Transportschicht des OSI-Modells und setzt auf dem IPv4[RFC791] oder IPv6[RFC2460] auf. Es ist ein verbindungsloses und nicht-zuverlässiges Transportprotokoll ohne Verschlüsselung. Die Daten werden in Datagramen übertragen. UDP bietet die Dienstgüte „best effort“, also das Versenden der Daten unter bester Ausnutzung der Ressourcen. Damit wird eine möglichst geringe Latenz erwirkt und möglicher Paketverluste akzeptiert.

Somit bildet UDP die Grundlage für die zeitempfindliche Echtzeitkommunikation und wird häufig für Multimedia-Verbindungen eingesetzt.

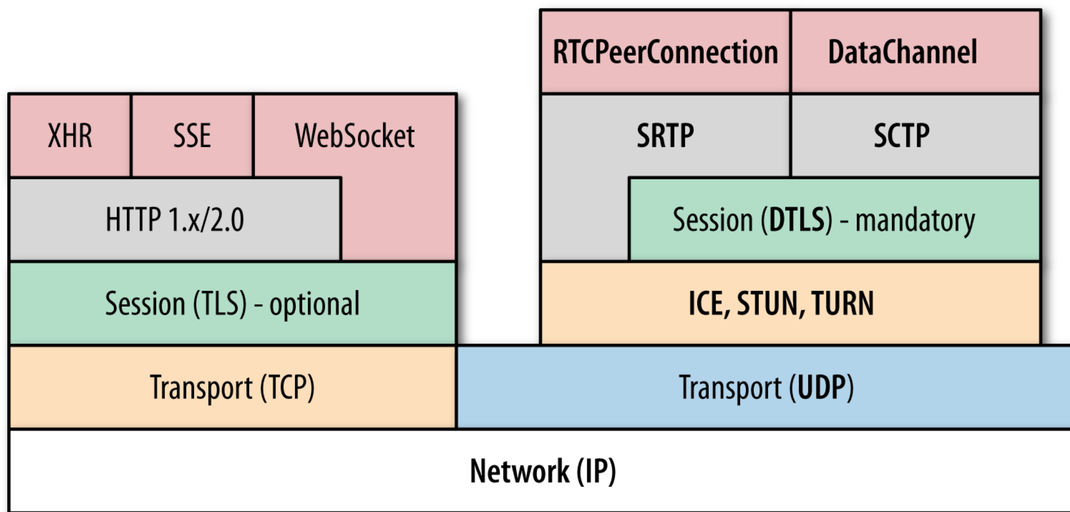



Abbildung 2.2.4-15.: WebRTC Protokolle, aus [BroNet]

**ICE [RFC5245]**

ICE bedeutet „Interactive Connectivity Establishment". Es handelt sich um ein Protokoll, welches eine UDP-Verbindung zwischen zwei Peers herstellt. Um dies zu ermöglichen wird ein Offer/Answer-Modell [RFC3264] genutzt, um das verbindungsbeschreibende SDP [RFC4566] zwischen beiden Peers auszutauschen. Für den Austausch der SDP wird ein dedizierter Signaling Channel genutzt. Ist der Austausch der SDP erfolgreich, ist es möglich eine Peer-to-Peer Verbindung herzustellen.
Das ICE Protokoll sieht die Ermittlung und Verwaltung von möglichen Routingpfaden vor. Verwaltet werden auch die beteiligten Peers, beispielsweise durch die Kontrolle der Konnektivität. Um eine Peer-to-Peer Verbindung herzustellen wird das STUN Protokoll [RFC5389] genutzt. Dies dient dem Sammeln von möglichen Routingpfaden und ermöglicht NAT Traversen. NAT-Traversen sind nötig, wenn die Peers in unterschiedlichen Netzwerken, die durch NAT's getrennt sind, arbeiten. Somit sind sie oft für die Herstellung einer Peer-to-Peer Verbindung nötig. Ist es mit der Hilfe des STUN Protokolls nicht gelungen, ein NAT zu durchqueren und so eine direkte Verbindung zwischen zwei Peers herzustellen, kann auf das TURN Protokoll [RFC5766] zurückgegriffen werden. Der Einsatz des TURN Protokolls bietet einen alternativen Verbindungsweg über einen Relais-Server. Dieser Weg dient als Backup-Lösung, um eine bestimmte Verlässlichkeit beim Verbindungsaufbau gewährleisten zu können. Denn die Herstellung einer ressourcenschonenden Peer-to-Peer Verbindung ist das Ziel beim Einsatz von WebRTC.

**STUN [RFC5389]**

STUN steht für „Session Traversal Utilities for NAT“. Es ist ein UDP-basiertes Protokoll und dient der NAT Traverse. Es ermittelt die öffentliche IP und offene Ports eines Peers. Zur Ermittlung der IP und nutzbaren Ports wird ein STUN-Server angesprochen und aufgefordert, die IP zurückzusenden, welche als Absender der Anfrage angegeben wurde. Diese Anfragen werden über verschiedene Ports durchgeführt.

**TURN [RFC5766]**

„Traversal Using Relays around NAT“ ist die Bedeutung von TURN. Es handelt sich um ein Protokoll, das eingesetzt wird, wenn mit dem Einsatz des STUN Protokolls keine NAT Traverse möglich war. Ist ein Peer hinter einem undurchdringlichen semmetrischen NAT oder einer Firewall, kann dieser Peer eine Verbindung zum TURN Relais-Server aufbauen und über diesen mit dem anderen Peer kommunizieren.

**SDP [RFC4566]**

Das „Session Description Protocol“ ist ein Anwendungsprotokoll. Es dient zur Beschreibung der Eigenschaften von Multimediaverbindungen, in einem definierten Format. Es beinhaltet Medien- und Verbindungsinformationen, die beim Verbindungsaufbau ausgehandelt werden können. Zur Aushandlung wird ein Offer/Answer-Model [RFC3264] benutzt.

**DTLS [RFC6347]**

DTLS ist ein Verschlüsselungsprotokoll, welches zur Verschlüsselung von Datagramen einer UDP-Verbindung genutzt wird. Es wird für Multimediaverbindungen von WebRTC eingesetzt, welche durch die Protokolle SRTP [RFC3711] und SCTP [RFC4960] realisiert werden. Beide Protokolle brauchen Unterstützung durch eine Verschlüsselung, um den Spezifikationen von WebRTC [WebRTCpub] gerecht zu werden.
DTLS wird zur Verschlüsselung der SCTP Verbindung genutzt. SCTP bietet selbst keine Verschlüsselung an, wird aber für den RTCDataChannel eingesetzt, dessen Spezifikation eine Verschlüsselung voraussetzt. Somit wird ein DTLS Tunnel eingesetzt, um die SCTP Verbindung zwischen den Peers zu verschlüsseln.
SRTP bietet zwar für den Medienstrom eine Verschlüsselung an, hat aber keinen eigenen Mechanismus zur verschlüsselten Übertragung der AES-Schlüssel. Daher werden die Nachrichten zum Übertragen der AES-Schlüssel mit DTLS verschlüsselt.
Die Sicherheit der Verschlüsselung entspricht TLS. Um die Funktion mit UDP als nicht-zuverlässigen Transportprotokoll zu ermöglichen, wurden nötigen Änderungen vorgenommen. Zu den eingeführten Änderungen gehören die Nummerierung der Pakete und die Möglichkeit den Handshake, der beim Start der Session genutzt wird, wiederherzustellen. Die Paketnummerierung und die Möglichkeit den Handshake

wiederherzustellen sind erforderlich, da zum Entschlüsseln der Nutzdaten alle Pakete benötigt werden. Um zu prüfen ob alle Pakete vorhanden sind, mussten diese Änderungen vorgenommen werden, da diese von UDP nicht geboten werden.

**SRTP [RFC3711]**

Beim „Secure Real-Time Transport Protocol“ handelt es sich um eine AES verschlüsselte Variante des RTP [RFC3550]. Es ist ein UDP basiertes Transportprotokoll für die Übertragung von Video- und Audiodaten in Echtzeit. SRTP bietet keinen eigenen Mechanismus zur verschlüsselten Übertragung der AES-Schlüssel an den Remote Peer an. Beim Transport von Mediadaten werden RTP Pakete in SRTP Pakete eingebettet und sind so geschützt.

**SCTP [RFC4960] erweitert durch Partial Reliability Extension [RFC3758]**

Das „Stream Control Transmission Protocol“ ist ein modernes IP-basiertes Transportprotokoll. Es arbeitet auf der Transportschicht, wie auch TCP und UDP. Es ist mit Unterstützung der Protokollerweiterung „Partial Reliability Extension“ ein zuverlässiges verbindungsorientiertes Protokoll und vereint die Vorteile von TCP und UDP. Es beherrscht Multi-Homing, nutzt parallele Datenströme für eine Verbindung und kann auch Datagramme versenden. Es ist sehr flexibel und lässt sich dem Einsatz entsprechend konfigurieren. Das API des RTCDataChannel bietet dem Entwickler Zugriff auf die Anpassungsmöglichkeiten zur Konfiguration beim Erzeugen des Objekts. Durch diese flexiblen Anpassungsmöglichkeiten und der Unterstützung durch die Protokollerweiterung wird es den Anforderungen an den RTCDataChannel der WebRTC Spezifikation [WebRTCpub] fast gerecht. Es bietet keine Verschlüsselung, welche jedoch gegeben sein muss. Daher wird der DTLS Tunnel zwischen den Peers genutzt. Mit der Unterstützung durch DTLS können die Anforderungen der WebRTC Spezifikation erfüllt werden.

Die obige Beschreibung der Protokolle verdeutlicht dessen nötiges Zusammenspiel, um der Spezifikation von WebRTC nachzukommen. Nachfolgend hierzu eine kurze Zusammenfassung.
Zur Herstellung und Verwaltung einer RTCPeerConnection zwischen zwei Peers, dienen die Protokolle ICE, STUN und TURN.
Zu betrachten ist zudem der vom ICE-Protokoll ausgeführte Austausch des SDP, durch das Offer/Answer-Modell [RTCWEB_JSEP]. Für den Austausch wird ein separater Signaling Channel benötigt, wie in Abbildung 2.2.4-16 dargestellt. Der Signaling Channel dient zur Kommunikation zwischen beiden Peers bevor eine RTCPeerConnection hergestellt wurde, die zur Übertragung von jeglichen Daten dienen kann. Der Einsatz des SDP und eine Beschreibung des Signaling Channels erfolgt im Kapitel 2.2.5.

Abbildung 2.2.4-16.: Offer/Answer-Modell, aus [BroNet]

Die Verwendung des DTLS Protokolls, ermöglicht die Verschlüsselung von UDP-Verbindungen und kann so zur Unterstützung für die beiden Arten von Multimediaverbindungen genutzt werden. Der Einsatz von DTLS unterscheidet sich bei der Anwendung für SCTP und SRTP, eine SCTP Verbindung wird komplett durch einen DTLS Tunnel verschlüsselt, bei einer verschlüsselten SRTP Verbindung werden nur die AES Schlüssel mit DTLS verschlüsselt übertragen.
Die Protokolle SRTP und SCTP werden zum Übertragen von Multimediadaten genutzt. Für die Übertragung von Video- und Audiodaten wird das SRTP eingesetzt, um beliebige Anwendungsdaten über einen DataChannel zu übertragen, das SCTP.

### 2.2.5 Die RTCPeerConnection

In diesem Kapitel werden die wichtigsten Komponenten der RTCPeerConnection beschrieben, um ein Verständnis für jede einzelne Komponente zu erlangen. Beschrieben werden auch dessen jeweilige Aufgaben und das Zusammenspiel miteinander.

**RTCPeerConnection**

Die RTCPeerConnection ist die grundlegende Verbindung bei Verwendung von WebRTC. Unabhängig davon, welche Art von Daten übertragen werden soll, ist es nötig eine RTCPeerConnection zwischen den Peers herzustellen. Bei der Erzeugung eines RTCPeerConnection Objekts muss mindestens eine STUN-Server Adresse als Parameter „IceConfig“ angegeben werden. Optional können darin auch mehrere STUN- [STUN_URI] wie auch TURN-Serveradressen [TURN_URI] angegeben werden. TURN-Server fungieren auch immer als STUN-Server. Der ICE Agent des RTCPeerConnection Objekts nutzt die bekannten STUN-Server und TURN-Server, um ICE Candidates zu sammeln [RFC5245]. Um einen Verbindungsweg mit einer möglichst geringen Latenz zu ermöglichen, ist ein STUN-Server, pro zu überwindender NAT-Schicht, optimal. Werden nicht öffentliche STUN- oder TURN-Server eingesetzt, ist es möglich entsprechende Authentifizierungsdaten bei der Erzeugung des RTCPeerConnection Objekts anzugeben. Die Angabe erfolgt auch im

Parameter IceConfig und ermöglicht dem ICE Agent der RTCPeerConnection die Nutzung der Server.
Je nach Einsatzzweck der RTCPeerConnection wird nach dem Erzeugen eines RTCPeerConnection Objekts, ein RTCDataChannel Objekt erstellt oder ein Medienstrom vom Browser bezogen und dem RTCPeerConnection Objekt übergeben. RTCDataChannel oder Medienstrom haben jeweils eigene Eigenschaften. Die Eigenschaften des RTCDataChannels werden in Kapitel 2.2.8 näher beschrieben. Durch die Übergabe an das RTCPeerConnection Objekt sind diese Eigenschaften bekannt und werden beim Erzeugen des SDP in diesem beschrieben.
Soll eine RTCPeerConnection getrennt werden, bietet das Interface dazu eine Close Methode an. Wird diese Methode aufgerufen, wird der dem Objekt zugehörige ICE Agent zerstört und die Verbindung zum Remote Peer unterbrochen, unabhängig davon ob noch RTCDataChannel oder ein MediaStream genutzt wird. Nach der Trennung werden alle benötigten Ressourcen wieder freigegeben.

**ICE Agent**

Jedem RTCPeerConnection Objekt gehört ein sogenannter ICE Agent an [RFC5245], der beim Erzeugen eines RTCPeerConnection Objekts entsteht. Der ICE Agent ermittelt mit Hilfe des STUN Protokolls, die eigene öffentliche IP und nutzbare Ports zur Herstellung der RTCPeerConnection. Die ermittelten Verbindungsinformationen, über die eine RTCPeerConnection hergestellt werden kann, heißen „ICE Candidates". Das Ermitteln bzw. Sammeln dieser ICE Candidates wird auch „ICE gathering" genannt. Der ICE Agent informiert das RTCPeerConnection Objekt über ermittelte ICE Candidates und koordiniert [RFC5245] diese. Die Ermittlung der ICE Candidates ist vom Austausch der Offer- und Answer-Nachrichten über den Signaling Channel unabhängig. Bei der Ermittlung der ICE Candidates ist der ICE Agent immer bemüht eine möglichst gute Verbindung zu erzielen. So kann die Ermittlung wiederholt werden, wenn die bereits ermittelte Verbindung nicht für die zu transportierenden Daten ausreicht. Der ICE Agent ist auch für Konnektivitätskontrollen zwischen den Peers zuständig. Der ICE Agent setzt die Konnektivitätskontrollen durch das Senden von Keepalive-Nachrichten, in Form von STUN Anfragen, um.

**ICE Candidates**

Neben dem Abschluss des Offer/Answer-Modell, ist es nötig das die ICE Agents mögliche ICE Candidates ermittelt haben. Über die ermittelten ICE Candidates kann eine RTCPeerConnection hergestellt werden. Das Ermitteln von ICE Candidates startet mit dem jeweiligen Erzeugen eines RTCPeerConnection Objekts, da damit jeweils ein ICE Agent erzeugt wird. Ermittelte ICE Candidates teilt der ICE Agent eines Peers seinem Remote Peer über den Signaling Channel mit.

Es ist möglich aber nicht empfehlenswert, die ermittelten ICE Candidates in den SDP mitzuteilen. Die Ermittlung der ICE Candidates ist ein eigenständiger Prozess des ICE Agent, welcher im Hintergrund abläuft. Die Ermittlung der ICE Candidates kann gerade bei Verwendung mehrerer STUN-Server andauern und so den Ablauf der Offer/Answer-Modell verzögern. Daher ist es sinnvoll die Mitteilung der ICE Candidates in Nachrichten durchzuführen, die unabhängig vom Offer/Answer-Modell übertragen werden.
Um die ICE Candidates außerhalb des SDP mitteilen zu können, wurde das ICE-Protokoll erweitert. Die Erweiterung nennt sich „ICE Trickling“ [RTCWEB_JSEP] und ermöglich die inkrementelle Erfassung der ICE Candidates. Durch die Notwendigkeit die ICE Candidates zusätzlich an den Remote Peer mitzuteilen, werden mehr Nachrichten über den Signaling Channel ausgetauscht. Da durch die Verwendung von ICE Trickling keine Verzögerung des Offer/Answer-Modells entsteht, ist die Verwendung zu empfehlen.
Empfängt ein Peer eine ICE Candidate Nachricht über den Signaling Channel vom jeweiligen Remote Peer, wird daraus ein lokales RTCIceCandidate Objekt erzeugt. Das RTCIceCandidate Objekt wird dem lokalem RTCPeerConnection Objekt übergeben. Der lokale ICE Agent wird so über die ermittelten ICE Candidates des Remote Peer informiert. Der ICE Agent koordiniert den Austausch von ICE Candidates. Ermittelt der ICE Agent einen ICE Candidate, wird die Callback Funktion „onicecandidate“ des RTCPeerConnection Objekts aufgerufen. In der Funktion wird eine ICE Candidate Nachricht für den Remote Peer erstellt und an diesen versendet. Sind genug passende ICE Candidates von beiden beteiligten ICE agents gefunden worden, wird der Sammeln- bzw. „ICE gathering“ Prozess abgeschlossen und die RTCPeerConnection initiiert.
Werden nicht genug ICE Candidates gefunden, gilt die Peer-to-Peer Verbindung als gescheitert. Die RTCPeerConnection kann dann nur mit Hilfe einer Relaislösung, über einen TURN-Server hergestellt werden. In diesem Fall erhöht sich die Latenz der RTCPeerConnection, da alle Daten über den TURN-Server kommuniziert werden. Dieser kann weit entfernt oder ausgelastet sein, beides würde die Latenz erhöhen.

### 2.2.6 Signaling Channel

Um eine Peer Connection zwischen zwei Peers herzustellen, wird der zuvor genannte Signaling Channel benötigt. Der Signaling Channel dient beiden Peers um miteinander kommunizieren können, bevor eine Peer-to-Peer Verbindung besteht. Die Kommunikation zwischen den Peers ist nötig, um die Offer/Answer-Nachrichten [RFC3264] auszutauschen und so eine Peer Connection herzustellen. Die Offer/Answer-Nachrichten beinhalten jeweils ein SDP, welches den Status des jeweils lokalen RTCPeerConnection Objekts angibt und so dem Remote Peer mitgeteilt wird.
Auf welche Art ein Signaling Channel realisiert wird ist nicht fest definiert. Die Verbindung die als Signaling Channel genutzt wird, kann sich daher in Implementierungen unterscheiden. Welche Art einer Verbindung für einen Signaling Channel genutzt wird, bleibt dem Entwickler überlassen.

Auf der rechten Seite der Abbildung 2.2.4-15 sind drei mögliche Protokolle aufgeführt. Es handelt sich um XHR bzw. XMLHttpRequest [XMLHttpRequest], SSE bzw. Server-Sent Event und WebSocket. In Abbildung 2.2.6-17 werden mögliche Variationen zur Realisierung eines Signaling Channel dargestellt.
In dieser Arbeit wird eine WebSocket Lösung genutzt, was nach Abbildung 2.2.6-17 der mittig dargestellten Custom signaling gateway Lösung entspricht. Informationen zu WebSocket sind im Kapitel 2.1 beschrieben.

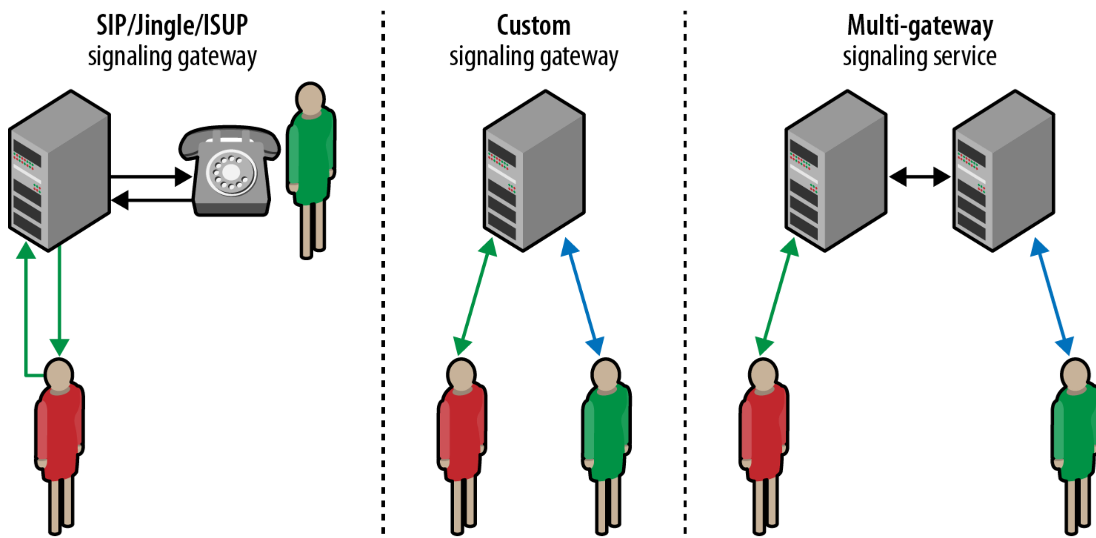



Abbildung 2.2.6-17.: Mögliche Varianten eines Signaling Channels zwischen zwei Peers, aus [BroNet]

Ist in diesem Fall die WebSocket Verbindung von beiden Peers zum Server hergestellt, dient der Weg über den WebSocket-Server als Signaling Channel. Damit der Signaling Channel zwischen zwei Peers korrekt arbeitet, benötigt der WebSocket-Server Informationen drüber, zwischen welchen Peers ein Signaling Channel besteht. Der Entwickler muss sich darum kümmern, dass dem WebSocket-Server bekannt ist, welche Peers miteinander kommunizieren.

### 2.2.7 Herstellen der RTCPeerConnection

Der in diesem Kapitel beschriebene Aufbau einer RTCPeerConnection beschreibt die Herstellung einer RTCPeerConnection für einen symmetrischen RTCDataChannel, da dieser auch praktisch in dieser Arbeit eingesetzt wird. Es gibt auch die Möglichkeit asymmetrische RTCDataChannels zu erstellen. Dabei erstellen beide Peers ein eigenes RTCDataChannel Objekt und der Verbindungsaufbauwird zwischen beiden Peers verhandelt.

**Offer Nachricht mit SDP**

Zum Initiieren des Offer/Answer-Modell sendet ein Peer eine Offer-Nachricht mit dem lokal generiertem SDP an den Remote Peer, wie in Abbildung 2.2.7-18 dargestellt.

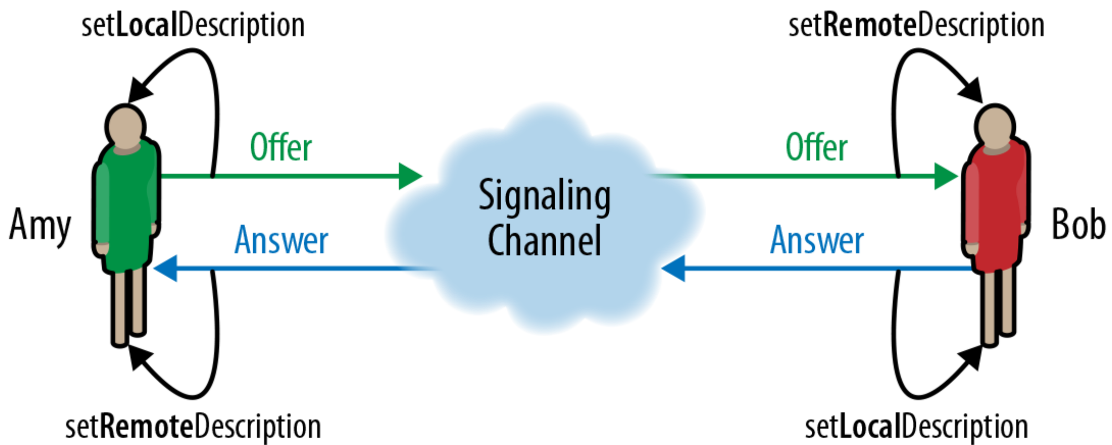



Abbildung 2.2.7-18.: Offer/Answer Modell mit Einsatz des SDP, aus [BroNet]

Um dem Remote Peer das SDP in Form einer Offer-Nachricht mitzuteilen ist es nötig, dass der Signaling Channel zum Remote Peer zuvor hergestellt wurde.
Zum Generieren des SDP bietet das Interface des RTCPeerConnection Objekts, die Methode „createOffer“ [RFC3264]. Das generierte SDP wird dem RTCPeerConnection Objekt mit dem Aufruf der Methode „setLocalDescription“ als Parameter übergeben und beschreibt so den lokalen Sendestrom des RTCPeerConnection Objekts.
Anschließend muss die Offer-Nachricht mit dem generierten SDP erstellt und dem Remote Peer über den Signaling Channel zugesendet werden.

**SDP des Remote Peer als Answer**

Die Umsetzung des Offer/Answer-Model ist mit dem Einsatz der WebRTC Web-API sehr symmetrisch. Wie in Abbildung 2.2.7-18 dargestellt, empfängt der Remote Peer die Offer-Nachricht mit dem SDP des anderen Peer über den Signaling Channel. Das empfangene SDP wird genutzt, um ein RTCSessionDescription Objekt zu erzeugen. Aus dem SDP lässt sich ermitteln, was der initiierende Peer für eine RTCPeerConnection herstellen möchte.
Das empfangene SDP wird dem RTCPeerConnection Objekt, durch den Aufruf der Methode „setRemoteDescription“, als Parameter übergeben. Auf diese Weise wird das lokale RTCPeerConnection Objekt über den Sendestrom des Remote RTCPeerConnection Objekts informiert bzw. beschreibt den lokalen Empfangsstrom.
Dem erzeugten RTCPeerConnection Objekt gehört ein ICE Agent an [RFC5245]. Der ICE Agent des Remote Peer arbeitet wie der des initiierenden Peers. Der ICE Agent beginnt

daher auch mit der Ermittlung der ICE Candidates, mit Einsatz von bekannten STUN- und TURN-Servern.
Nachdem das empfangene SDP dem eigenen RTCPeerConnection Objekt mitgeteilt wurde, muss ein eigenes SDP generiert werden. Der Remote Peer generiert ein SDP, durch den Aufruf der Methode „creatAnswer“ [RFC3264] des RTCPeerConnection Objekts. Das generierte SDP wird der RTCPeerConnection mit Aufruf der Methode „setLocalDescription“ mitgeteilt und beschreibt so den lokalen Sendestrom des RTCPeerConnection Objekts.
Der Remote Peer kennt nun das SDP des initiierenden Peers als Remote Description und das eigene SDP als Local Description. Das generierte SDP wird dann über den Signaling Channel, in einer Answer-Nachricht zurück an den initiierenden Peer geschickt.

**Abschluss des Offer/Answer Modells**

Der initiierende Peer empfängt über den Signaling Channel die Answer-Nachricht des Remote Peer, wie in Abbildung 2.2.7-18 dargestellt. Das empfangene SDP, wird verwendet um ein RTCSessionDescription Objekt zu erzeugen. Dieses wird dem RTCPeerConnection Objekt, mit dem Aufruf der Methode „setRemoteDescription“, als Parameter übergeben. Auf diese Weise wird das lokale RTCPeerConnection Objekt, über Sendestrom des Remote RTCPeerConnection Objekts bzw. den lokalen Empfangsstrom informiert.
Damit ist das Offer/Answer-Modell für den Austausch der SDP`s abgeschlossen. Peer und Remote Peer haben beide ein SDP erzeugt und dieses als Local Description gesetzt. Remote Peer und Peer haben beide auch das SDP des anderen empfangen und als Remote Description gesetzt. Beiden Peers ist somit der Status des lokalen und des remote RTCPeerConnection Objekts bekannt.
Erforderlich für den erfolgreichen Aufbau der RTCPeerConnection ist nun nur noch der Austausch der ermittelten ICE Candidates, in zusätzlichen Nachrichten über den Signaling Channel.

### 2.2.8 Der RTCDataChannel

Um einen RTCDataChannel nutzen zu können, wird eine hergestellte RTCPeerConnection benötigt, wie in Kapitel 2.2.7 beschrieben. Es können symmetrische und asymmetrische RTCDataChannels genutzt werden. Im Kapitel 2.2.7 wurde der Ablauf für einen symmetrischen RTCDataChannel beschrieben, welcher praktische Anwendung in dieser Arbeit findet. Folgende Beschreibung des RTCDataChannel bezieht sich nur auf einen symmetrischen RTCDataChannel.

**Konfiguration**

Der RTCDataChannel kann dem vorgesehenen Einsatz entsprechend angepasst werden, dafür gibt es sieben Optionen, die beim Erzeugen des RTCDataChannel Objekt konfiguriert

werden können und nachträglich für dieses Objekt nicht zu ändern sind. Beim erzeugen des RTCDataChannel Objekts, können dazu zwei Parameter übergeben werden. Der erste Parameter ist vom Typ String, welcher zur Bezeichnung des RTCDataChannel Labels dient. Der zweite optionale Parameter ist vom Typ RTCDataChannelInit, beinhaltet sechs Variablen und dient der Konfiguration des RTCDataChannel Objekts.
Die Variable id dient dazu, dem RTCDataChannel Objekt beim Erzeugen eine ID zuzuweisen, die zur Identifikation genutzt werden kann. Im Gegensatz zum Label der RTCDataChannels, ist die ID vom Typ unsigned short.
Die Variable ordered ist ein boolean, mit dem bestimmt wird, ob Daten garantiert in geordneter Reihenfolge versendet werden. Eine geordnete Reihenfolge entspricht der Reihenfolge, in der die Anwendung Daten mit dem Aufruf der send() Funktion versendet hat.
Es gibt zwei Variablen mit denen die Zuverlässigkeit bestimmt werden kann, es handelt sich um maxPacketLifeTime und maxReatransmits. Ein zuverlässiger RTCDataChannel versendet Daten solange neu, bis diese angekommen sind. Bei einem unzuverlässigen RTCDataChannel ist die Anzahl der möglichen Sendewiederholungen oder aber die Zeit, in der Sendewiederholungen stattfinden können, begrenzt. Die Variable maxRetransmits dient zur Begrenzung der Sendewiederholungen, die Variable maxPacketLifeTime zur Begrenzung der Zeit, in der Sendewiederholungen stattfinden dürfen. Wenn ein unzuverlässiger DataChannel erzeugt werden soll, kann nur eine der beiden Variablen gesetzt werden. Soll ein zuverlässiger DataChannel erzeugt werden, darf keiner der beiden Variablen gesetzt werden.
Zur Verwendung von Unterprotokollen bei der Kommunikation über einen RTCDataChannel, müssen diese in der Variablen protocol angegeben werden. Zu beachten ist, dass beiden Peers das gewählte Unterprotokoll bekannt sein muss und zu Verfügung steht.
Mit der Variable negotiated wird bestimmt, ob der RTCDataChannel zwischen den beiden Peers verhandelt wird oder nicht. Wird verhandelt, erzeugen beide Peers ein eigenes RTCDataChannel Objekt, mit eventuell unterschiedlicher Konfiguration. Dies kann dann zu asymmetrischen RTCDataChannels führen, die nur die ID teilen und darüber verknüpft sind. Wenn nicht über den RTCDataChannel verhandelt wird, wird nur ein einziges RTCDataChannel Objekt durch den Verbindungsinitiator erzeugt, womit die Konfiguration für beide Peers identisch ist. Dies führt zu einem symmetrischen RTCDataChannel, wie im Kapitel 2.2.5 beschrieben und in dieser Arbeit praktisch angewandt.

**Symmetrischer RTCDataChannel**

Beim Aufbau einer RTCPeerConnection für einen symmetrischen RTCDataChannel wird vom initiierenden Peer das einzige RTCDataChannel Objekt erstellt. Das RTCDataChannel Objekt wird dem RTCPeerConnection Objekt des Remote Peer, nach Herstellung der RTCPeerConnection, mitgeteilt. Das Interface eines RTCPeerConnection Objekt besitzt dafür das Eventhandler Attribut „ondatachannel“. Dieses Attribut des Remote Peers wird durch

ein Event gesetzt. Dass auslösende Event ist die Mitteilung des RTCDataChannel Objekt vom initiierenden Peer. Nachdem der Remote Peer den RTCDataChannel mitgeteilt bekommen hat, kann er diesen auch verwenden.

### 2.2.9 Kommunikation über den RTCDataChannel

**Empfang von Nachrichten**

Das Interface des RTCDataChannel nutzt Eventhandler [HTML5PUB] Attribute, um auftretende Ereignisse zu verarbeiten. Diese Lösung entspricht dem Verfahren, was auch die WebSocket API nutzt. Es gibt vier auftretende Events, die durch die Eventhandler verarbeitet werden. Events gibt es für das Herstellen des RTCDataChannel, für ankommende Nachrichte, für auftretende Fehler und beim Schließen des RTCDataChannel.
Das eventgetriebene Empfangen von Nachrichten unterstreicht die asynchrone Arbeitsweise des RTCDataChannel. Events können jederzeit nach dem Aufbau und vor dem Abbau der Verbindung auftreten.
Sendet nun ein Remote Peer eine Nachricht, empfängt der Peer ein Event, welches die Nachricht des Remote Peer enthält. In der Callback Funktion des Events wird dann die Nachricht aus dem Event gelesen und verarbeitet. Der Typ der Nachricht, lässt sich dem Auftretenden Event entnehmen und so identifizieren. Dem Entwickler ist es möglich die empfangene Nachricht passend zu verarbeiten.

**Senden von Nachrichten**

Zum Senden von Nachrichten bietet das Interface des RTCDataChannel vier Methoden, die sich durch den Typ der zu sendenden Daten unterscheiden. Unterstützt werden die Typen string, blob, arrayBuffer und arrayBufferView.

**Trennung des RTCDataChannel**

Ist die Kommunikation zwischen beiden Peers abgeschlossen oder möchte ein Peer die Verbindung trennen, kann er dies mit dem Aufruf einer Close Methode einleiten. Unabhängig davon ob der auslösende Peer selbst Erzeuger des RTCDataChannel Objekts war, kann die Funktion aufgerufen werden. Der Remote Peer wird über ein auftretendes Close Event informiert.

# 3 Entwicklungs- & Anforderungsanalyse

Wie im Kapitel 1.2 erwähnt ist das Ziel dieser Bachelorarbeit, die Betrachtung vom WebSocket-Protokoll und des WebRTC Framework bei dessen Anwendung. Um das WebSocket-Protokoll und das WebRTC Framework anzuwenden soll zur Demonstration ein Daten-Managementsystem entwickelt werden, das genutzt werden kann um Echtzeitdaten zu betrachten, die von entfernten Sendern verschlüsselt über eine Peer-to-Peer Verbindung kommuniziert werden. Das zu entwickelnde System, wurde bewusst abseits einer reinen Web-Anwendung gewählt, bei der die Kommunikation auf Web-Browser beschränkt ist.
Im Kapitel 3.1 wird das Entwicklungsszenario präsentiert, in diesem wird beschrieben wie und wofür das Daten-Managementsystem eingesetzt wird. Das präsentierte Entwicklungsszenario beschreibt damit die praktisch zu entwickelte Web-Anwendung. Die Beschreibung des Entwicklungsszenarios ermöglicht die Ableitung von Anforderungen an das zu entwickelnde Daten-Managementsystem. Durch die Definition des Entwicklungsszenarios können in Kapitel 3.2 Anwendungsfälle für die Komponenten der zu entwickelnden Web-Anwendung beschrieben werden, woraus weitere Anforderungen abgeleitet werden können. In Kapitel 3.3 erfolgt dann eine Anforderungsanalyse, in der aus den Anwendungsfällen aus Kapitel 3.2, die Anforderungen für die Realisierung abgeleitet werden.

## 3.1 Entwicklungsszenario

Das gewählte Szenario spielt im medizinischen Umfeld und beschreibt ein mögliches Einsatzfeld für das Daten-Managementsystem, welches in dieser Bachelorarbeit entwickelt wird.
In diesem Szenario handelt es sich um eine medizinische Einrichtung, wie beispielsweise eine Arztpraxis, eine medizinische Zentrale einer Pflegeeinrichtung oder einer Krankenhausstation.
Das Szenario sieht medizinisches Personal vor, das die EKG Daten jedes Patienten in einem Web-Browser am Arbeitsplatz abrufen kann, um diese in der visualisierten Form einer EKG Kurve zu beobachten. Die Beobachtung ist unabhängig davon, ob sich ein Patient einen Raum weiter oder zuhause befindet. Ein EKG ist immer als entfernt und internetfähig zu

betrachten. Notwendig ist nur, dass der Arbeitsplatz zum Beobachten und der Ort an dem sich ein Patient mit seinem EKG aufhält, eine Verbindung ins Internet bietet.
Es ist zudem möglich die Beobachtung auf weitere Arbeitsplätze auszuweiten. Dies ermöglicht die gemeinsame Beobachtung der EKG Kurve eines Patienten oder die Beobachtung von jeweils einer EKG Kurve eines Patienten pro Arbeitsplatz. Die Benutzer können über einen Chat miteinander kommunizieren, um die Beobachtung schnell und direkt zu analysieren und sich auszutauschen.
Die EKG Kurve entspricht der Darstellung eines einkanaligen EKG. Die EKG Kurve soll in einer Auflösung von 400 Samples pro Sekunde geplottet werden, was der nativen Auflösung des EKG Daten Senders entsprechen soll.
Die Patienten sind mit mobilen EKG Geräten ausgestattet, welche eine Verbindung ins Internet nutzten. Diese Geräte sind fiktiv und selbst nicht Thema dieser Bachelorarbeit. Simuliert werden die mobilen EKG Geräte durch eine Anwendung, die EKG Daten sendet und im Folgenden auch EKG Daten Sender genannt wird.

## 3.2 Anwendungsfälle

In diesem Kapitel werden Anwendungsfälle beschrieben, die die aktiven Bedienmöglichkeiten und das passive Verhalten der zu entwickelnden Lösung beschreiben. Alle in Abbildung 2.2.9-19 dargestellten Anwendungsfälle dienen der Beschreibung von gewünschten Funktionen, um im Kapitel 3.3 entsprechende Anforderungen abzuleiten. Das Verhalten in Fehlerfällen wird nicht beschrieben und ist auch nicht Bestandteil dieser Bachelorarbeit.

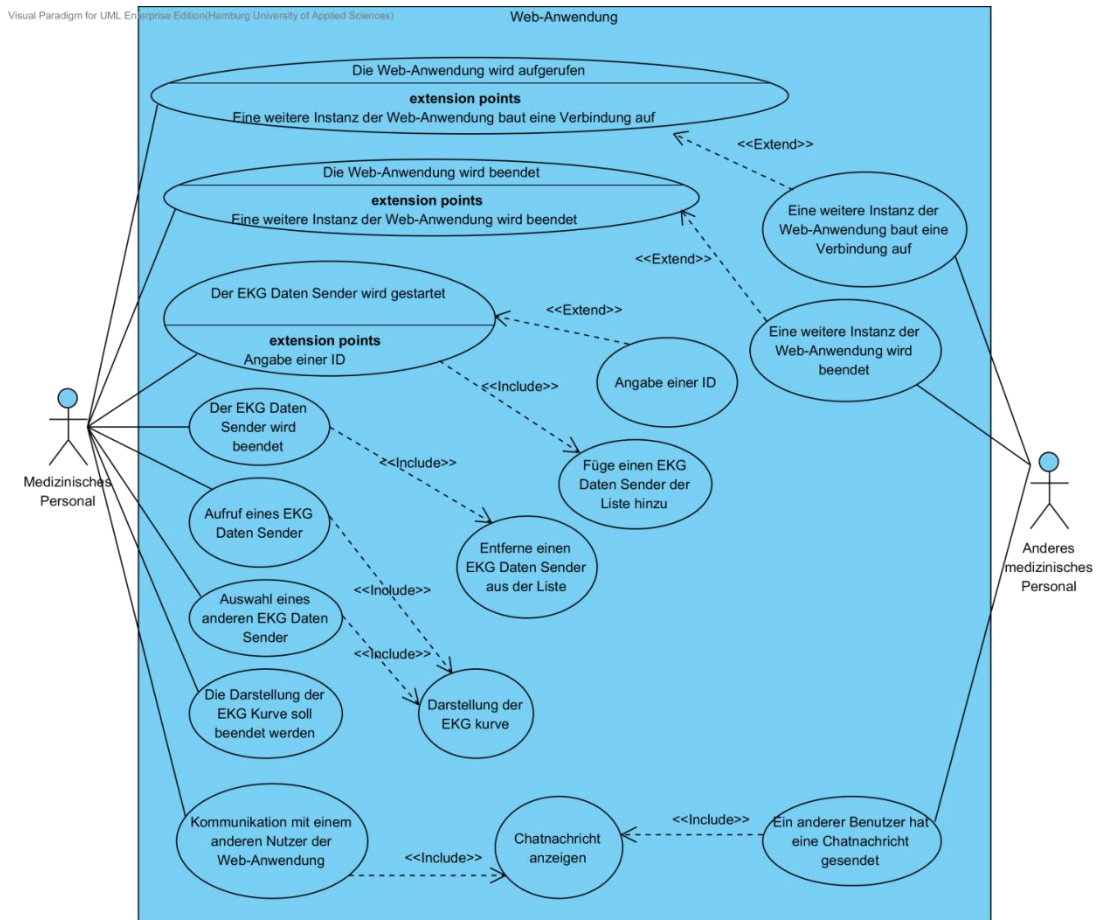



Abbildung 2.2.9-19.: Anwendungsfälle

**Die Web-Anwendung wird aufgerufen**

| | |
|---|---|
| Nummer des Anwendungsfalls | 1 |
| Kurzbeschreibung des Anwendungsfalls | Für den Aufruf der Web-Anwendung ist die Eingabe einer entsprechenden URL in einem Web-Browser nötig.<br>Anschließend baut sich die Web-Anwendung im Web-Browser auf und kann von aufrufenden Benutzern bedient werden. |

**Die Web-Anwendung wird beendet**

| | |
|---|---|
| Nummer des Anwendungsfalls | 2 |
| Kurzbeschreibung des Anwendungsfalls | Möchte ein Benutzer die Web-Anwendung beenden, soll dies jederzeit durch das Schließen des verwendeten Web-Browser oder des genutzten Tabs möglich sein. |

**Aufruf eines EKG Daten Sender**

| | |
|---|---|
| Nummer des Anwendungsfalls | 3 |
| Kurzbeschreibung des Anwendungsfalls | Hat der Benutzer die Web-Anwendung im Browser aufgerufen und möchte die EKG Kurve eines Patienten betrachten, wählt er dazu den entsprechenden Eintrag in einer Liste aus, die alle verfügbaren EKG Daten Sender der Patienten beinhaltet. |

**Auswahl eines anderen EKG Daten Sender**

| | |
|---|---|
| Nummer des Anwendungsfalls | 4 |
| Kurzbeschreibung des Anwendungsfalls | Betrachtet ein Benutzer die EKG Kurve eines Patienten und wählt einen anderen EKG Daten Sender aus der Liste in der Web-Anwendung aus, dann soll die EKG Kurve des als letzten ausgewählten EKG Daten Senders angezeigt werden. Der Wechsel zwischen den EKG Daten Sender soll jederzeit möglich sein. |

**Kommunikation mit einem anderen Nutzer der Web-Anwendung**

| | |
|---|---|
| Nummer des Anwendungsfalls | 5 |
| Kurzbeschreibung des Anwendungsfalls | Möchte ein Benutzer anderen Benutzern der Web-Anwendung eine Nachricht im Chat zukommen lassen, ist zuvor einmalig die Angabe eines Namens nötig, der anderen Benutzern als Absender angezeigt wird. Anschließend kann der Benutzer eine Chatnachricht verfassen und absenden, so dass alle anderen Benutzer der Web-Anwendung die Nachricht im Chat erhalten und den gewählten Namen des Autors dazu angezeigt bekommen. |

**Die Darstellung der EKG Kurve soll beendet werden**

| | |
|---|---|
| Nummer des Anwendungsfalls | 6 |
| Kurzbeschreibung des Anwendungsfalls | Wenn ein Benutzer die Darstellung einer EKG Kurve betrachtet und anschließend keine weitere Darstellung der EKG kurve mehr wünscht, soll es die Möglichkeit geben die Darstellung zu beenden, ohne das dazu der Web-Browser oder der verwendete Tab geschlossen werden muss. |

**Eine weitere Instanz der Web-Anwendung baut eine Verbindung auf**

| | |
|---|---|
| Nummer des Anwendungsfalls | 7 |
| Kurzbeschreibung des Anwendungsfalls | Wird bei einer parallelen Nutzung der Web-Anwendung, der identische EKG Daten Sender von mehreren Benutzern ausgewählt, soll der EKG Daten Sender jeder verbundenen Instanz der Web-Anwendung die gleichen aktuellen EKG Daten übermitteln. |

**Eine weitere Instanz der Web-Anwendung wird beendet**

| | |
|---|---|
| Nummer des Anwendungsfalls | 8 |
| Kurzbeschreibung des Anwendungsfalls | Wird eine Instanz der Web-Anwendung beendet, an die der EKG Daten Sender gerade EKG Daten übermitteln, sollen das Versenden der EKG Daten zu der entsprechenden Instanz eingestellt werden. Übermittelt ein EKG Daten Sender Daten an weitere Instanzen der Web-Anwendung, soll die Übermittlung der Daten an diese Instanzen weitergeführt werden. |

**Ein anderer Benutzer hat eine Chatnachricht gesendet**

| | |
|---|---|
| Nummer des Anwendungsfalls | 9 |
| Kurzbeschreibung des Anwendungsfalls | Wurde von einem anderen Benutzer eine Chatnachricht versendet, wird diese sofort mit einem Autorenname angezeigt. |

**Der EKG Daten Sender wird gestartet**

| | |
|---|---|
| Nummer des Anwendungsfalls | 10 |
| Kurzbeschreibung des Anwendungsfalls | Wenn der EKG Daten Sender vom Benutzer gestartet wird soll der Benutzer dem Sender einen Namen oder eine ID zuteilen, damit der Sender in der Web-Anwendung identifiziert werden kann.<br>Da in dieser Lösung kein echtes EKG zum Einsatz kommt, soll auch die Quelldatei mit EKG Daten ausgewählt werden können.<br>Hat der Benutzer beide benötigten Informationen angegeben, soll der EKG Daten Sender als wählbarer Listeneintrag in der Web-Anwendung dargestellt werden. |

**Der EKG Daten Sender wird beendet**

| | |
|---|---|
| Nummer des Anwendungsfalls | 11 |
| Kurzbeschreibung des Anwendungsfalls | Wird der EKG Daten Sender durch einen Benutzer beendet, dann soll der vorhandene Listeneintrag in der Web-Anwendung entfernt werden. |

**Die Quelldatei ist komplett ausgelesen**

| | |
|---|---|
| Nummer des Anwendungsfalls | 12 |
| Kurzbeschreibung des Anwendungsfalls | Ist die Quelldatei, die die EKG Daten beinhaltet, komplett ausgelesen und wurden alle EKG Daten versendet, dann soll der Inhalt der Quelldatei wiederholt versendet werden. |

## 3.3 Anforderungsanalyse

In diesem Kapitel werden aus dem in Kapitel 3.1 genannten Entwicklungsszenario und den in Kapitel 3.2 beschriebenen Anwendungsfällen, Anforderungen für die Entwicklung des Daten-Managementsystem abgeleitet. Die Anforderungen werden dabei in funktionale und nicht funktionale Anforderungen unterteilt.
Bei der Beschreibung der Realisierung in Kapitel 5 werden alle Anforderungen berücksichtigt, um eine Web-Anwendung zu realisieren, die der Zielsetzung in Kapitel 1.2, dem Entwicklungsszenario in Kapitel 3.1 und den Anwendungsfällen aus Kapitel 3.2 entspricht.

### 3.3.1 Funktionale Anforderungen

**Erreichen der Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 1 |
| Quell-Anwendungsfall | Anwendungsfall 1 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt deutlich die Entwicklung einer Web-Anwendung die mit der Eingabe einer URL in einem Web-Browser aufgerufen wird. |
| Auswirkung auf Komponente | Gesamtsystem |
| Info | Daraus ergibt sich auch eine nicht funktionale Anforderung, dabei handelt es sich um Anforderung 22. |

**Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 2 |
| Quell-Anwendungsfall | Anwendungsfall 1 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt eine Web-Anwendung die in einem Web-Browser bedient wird. |
| Auswirkung auf Komponente | Gesamtsystem |
| Info | Daraus ergibt sich auch eine nicht funktionale Anforderung, dabei handelt es sich um Anforderung 22. |

**Schließen der Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 3 |
| Quell-Anwendungsfall | Anwendungsfall 2 |
| Beschreibung der Anforderung | Aus dem Anwendungsfall lässt sich entnehmen, dass das plötzliche Schließen des Web-Browser oder von verwendeten Tabs, unabhängig vom aktuellen Zustand, fehlerfrei möglich ist. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Dies erfordert auch, dass eine eventuell genutzte Verbindung zu einem EKG Daten Sender beendet werden muss. |

**EKG Daten Sender List**

| | |
|---|---|
| Nummer der Anforderung | 4 |
| Quell-Anwendungsfall | Anwendungsfall 3 |
| Beschreibung der Anforderung | Im Anwendungsfall wird beschrieben, dass die Web-Anwendung gestartete EKG Daten Sender der Patienten in einer Liste dargestellt werden, aus der die Benutzer den gewünschten EKG Daten Sender auswählen können. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | EKG Daten Sender müssen in der Liste identifiziert werden können. |

**Visualisierung der EKG Daten**

| | |
|---|---|
| Nummer der Anforderung | 5 |
| Quell-Anwendungsfall | Anwendungsfall 3 |
| Beschreibung der Anforderung | Der Anwendungsfall gibt vor, dass die EKG Daten, in Form einer EKG Kurve dargestellt werden sollen. Um die empfangenen Daten zu veranschaulichen, soll eine Visualisierung der EKG Daten erfolgen. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | EKG Daten sind in roher Form wenig aussagekräftig. Allein die Menge an Werten die pro Sekunde gemessen werden, lässt kaum schnelle Rückschlüsse über das Verhalten des Herzens eines Patienten zu. Eine Visualisierung der EKG Daten in Form einer EKG Kurve ist dagegen aussagekräftiger. Medizinisch geschulte Personen erhalten durch den Blick auf eine EKG Kurve, schnell einen Eindruck zum Verhalten des Herzens ihres Patienten. |

**Wechsel des EKG Daten Sender**

| | |
|---|---|
| Nummer der Anforderung | 6 |
| Quell-Anwendungsfall | Anwendungsfall 4 |
| Beschreibung der Anforderung | Die Beschreibung des Anwendungsfalls gibt vor, dass der Wechsel auf einen anderen EKG Daten Sender, jeder Zeit möglich sein muss, um dessen EKG Daten als EKG Kurve zu beobachten auch während der Betrachtung einer EKG Kurve. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Die Web-Anwendung muss sich entsprechend um den Verbindungabbau der alten Verbindung und den Aufbau der neuen Verbindung kümmern. |

**Sichtbarkeit der EKG Daten Sender Liste**

| | |
|---|---|
| Nummer der Anforderung | 7 |
| Quell-Anwendungsfall | Anwendungsfall 4 |
| Beschreibung der Anforderung | Der Anwendungsfall gibt an, dass die Liste aus Anforderung 3 auch bei der Darstellung der EKG Kurve sichtbar sein muss. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Die Web-Anwendung muss die Liste der EKG Daten Sender und die Darstellung der EKG Kurve immer darstellen. Eine Unterteilung in Frames ist sinnvoll. |

**Chat**

| | |
|---|---|
| Nummer der Anforderung | 8 |
| Quell-Anwendungsfall | Anwendungsfall 5 |
| Beschreibung der Anforderung | Im Anwendungsfall wird vorgegeben, dass die Web-Anwendung einen Chat bieten soll. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Der dient den Benutzern der Web-Anwendung als Kommunikationskanal. |

**Chatbenutzer müssen einen Namen angeben**

| | |
|---|---|
| Nummer der Anforderung | 9 |
| Quell-Anwendungsfall | Anwendungsfall 5 |
| Beschreibung der Anforderung | Der Anwendungsfall sieht vor, dass ein Benutzer den Chat erst nach der Eingabe eines Namens nutzen kann. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Diese Anforderung ist nötig, um Anforderung 15 erfüllen zu können. |

**Chat Nachrichten gehen an alle Benutzer der Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 10 |
| Quell-Anwendungsfall | Anwendungsfall 5 |
| Beschreibung der Anforderung | Dem Anwendungsfall wird entnommen, dass eine Chatnachricht immer an alle anderen Nutzer der Web-Anwendung versendet wird. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Das bedeutet auch, dass der Chat nicht genutzt werden kann, um nur gezielt mit einen anderen Benutzer der Web-Anwendung zu kommunizieren. |

**Beendung der EKG Kurven Darstellung**

| | |
|---|---|
| Nummer der Anforderung | 11 |
| Quell-Anwendungsfall | Anwendungsfall 6 |
| Beschreibung der Anforderung | Der Anwendungsfall sieht vor, dass die Web-Anwendung einen Mechanismus bieten soll, mit dem die laufende Darstellung einer EKG Kurve beendet werden kann. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Wenn die Darstellung der EKG Kurve beendet wird müssen keine EKG Daten mehr abgebildet und somit auch keine EKG Daten mehr übertragen werden. |

**Gemeinsames beobachten eines EKG**

| | |
|---|---|
| Nummer der Anforderung | 12 |
| Quell-Anwendungsfall | Anwendungsfall 7 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass mehrerer Benutzer mit eigenen Instanzen der Web-Anwendung einem EKG Daten Sender gemeinsam beobachten können. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Der EKG Daten Sender muss mehrere parallele Verbindungen unterstützen, um an diese die aktuellen EKG Daten zu senden. |

**Weiterer Datenversand beim Verbindungsabbau**

| | |
|---|---|
| Nummer der Anforderung | 13 |
| Quell-Anwendungsfall | Anwendungsfall 8 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass wenn ein EKG Daten Sender mehrere Verbindungen zu Instanzen der Web-Anwendung hält und eine davon geschlossen wird, weiterhin auf den nicht geschlossenen Verbindungen die aktuellen EKG gesendet werden sollen. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Der Sendevorgang darf nicht unterbrochen werden. |

**Identifikation der Verbindungen**

| | |
|---|---|
| Nummer der Anforderung | 14 |
| Quell-Anwendungsfall | Anwendungsfall 8 |
| Beschreibung der Anforderung | Dem Anwendungsfall ist zu entnehmen, dass der EKG Daten Sender in der Lage sein muss die Verbindungen zu identifizieren. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Die Identifikation ist notwendig, um Anforderung 13 erfüllen zu können. Verbindungen müssen identifiziert werden können, um über die Übermittlung von EKG Daten zu entscheiden. |

**Chat Nachrichten haben Autoren**

| | |
|---|---|
| Nummer der Anforderung | 15 |
| Quell-Anwendungsfall | Anwendungsfall 9 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass eine Chatnachricht immer mit dem Namen des Autors versehen ist. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Dies ermöglicht anderen Benutzern die Zuordnung der empfangenen Chatnachricht. |

**Zustellung von Chat Nachrichten**

| | |
|---|---|
| Nummer der Anforderung | 16 |
| Quell-Anwendungsfall | Anwendungsfall 9 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass eine Chatnachricht immer sofort zugestellt werden muss. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Die direkte Zustellung von Chat Nachrichten dient der Kommunikation untern den Benutzern. |

**Angabe einer ID beim Anwendungsstart**

| | |
|---|---|
| Nummer der Anforderung | 17 |
| Quell-Anwendungsfall | Anwendungsfall 10 |
| Beschreibung der Anforderung | Aus dem Anwendungsfall lässt sich entnehmen, dass beim Starten eines EKG Daten Senders eine ID oder ein Name eingegeben werden muss. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Die Angabe einer ID oder eines Namens beim Start der EKG Daten Sender Anwendung ist sinnvoll, damit dieser passend zum Patienten angegeben werden kann. |

**EKG Daten Sender werden für Listen identifiziert**

| | |
|---|---|
| Nummer der Anforderung | 18 |
| Quell-Anwendungsfall | Anwendungsfall 10 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass die Web-Anwendung die verbundenen EKG Daten Sender für die Benutzer in eine Liste führen soll. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Um die in Anforderung 3 genannte Liste zu ermöglichen, ist zur Verwaltung der Liste die Identifikation der EKG Daten Sender durch die Web-Anwendung notwendig. |

**Angabe einer Quelldatei für EKG Daten beim Anwendungsstart**

| | |
|---|---|
| Nummer der Anforderung | 19 |
| Quell-Anwendungsfall | Anwendungsfall 10 |
| Beschreibung der Anforderung | Der Anwendungsfall beschreibt, dass wenn die EKG Daten Sender Anwendung gestartet wird auch eine Quelldatei mit den EKG Daten angegeben werden muss. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Die Angabe der Quelldatei ist sinnvoll, da so verschiedene Quelldateien mit der gleichen EKG Daten Sender Anwendung genutzt werden können. |

**Entfernung des EKG Daten Sender aus der Liste**

| | |
|---|---|
| Nummer der Anforderung | 20 |
| Quell-Anwendungsfall | Anwendungsfall 11 |
| Beschreibung der Anforderung | Im Anwendungsfall wird vorgegeben, dass beim Beenden der EKG Daten Sender Anwendung der in der Web-Anwendung entsprechende Eintrag in der Liste entfernt werden muss. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Dies ist nötig, damit kein Benutzer einen Listeneintrag eines EKG Daten Sender auswählen kann, der schon beendet ist. |

**Wiederversandt der EKG Daten aus der Quelldatei**

| | |
|---|---|
| Nummer der Anforderung | 21 |
| Quell-Anwendungsfall | Anwendungsfall 12 |
| Beschreibung der Anforderung | Der Inhalt, der in Anforderung 18 genannten Quelldatei soll wiederholt versendet werden, wenn der komplette Inhalt der Quelldatei versendet wurde. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Das ermöglicht einen Dauerbetrieb der EKG Daten Sender Anwendung, ohne das neue EKG Daten benötigt werden. |

### 3.3.2 Nicht funktionale Anforderungen

Die nicht funktionalen Anforderungen sind dem Entwicklungsszenario aus Kapitel 3.1 sowie der Zielsetzung in Kapitel 1.2 entnommen und werden in diesem Kapitel begründet.

**Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 22 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Es soll eine Anwendung entwickelt werden, die in einem Web-Browser bedient wird und damit als Web-Anwendung bezeichnet werden kann.<br>Der Web-Browser soll zur Bedienung eines Daten-Managementsystems dienen, welches für EKG Daten genutzt wird. Die Web-Anwendung stellt alle verfügbaren EKG Daten Sender dar, so dass der Benutzer diese auswählen kann. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | 1. Die Entscheidung zur Entwicklung einer Web-Anwendung hat mehrere Gründe. Ein Grund ist die hohe Unabhängigkeit der verwendeten Plattform. Web-Browser werden für unterschiedliche Betriebssysteme entwickelt und können daher auf nahezu allen Hardware-Plattformen genutzt werden. Hierzu zählen unter anderem Smartphones, Netbooks und klassische PC's.<br>2. Ein weiterer Grund eine Web-Anwendung zu entwickeln ist, dass Web-Anwendungen in der sicheren Umgebung eines Web-Browsers ausgeführt werden, die den Umgang mit modernen Netzwerk-Protokollen meist rasch beherrschen.<br>3. Um dem Ziel eines kostengünstigen Einsatzes nachzukommen, ist die Realisierung einer Web-Anwendung sinnvoll. Das Angebot an Web-Browsern ist groß und die Verwendung in der Regel kostenfrei. Gute Beispiele für kostenfreie und betriebssystemübergreifende Web-Browser sind Firefox und Chrome, welche weltweit am meisten Verwendung finden und daher als Referenz für die Funktionalität dienen sollen. |

**Darstellung der Web-Anwendung**

| | |
|---|---|
| Nummer der Anforderung | 23 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Die Web-Anwendung muss in mindestens drei Komponenten dargestellt werden. Es wird eine Komponenten benötigt, um alle verfügbaren EKG Daten Sender anzuzeigen, die EKG Daten senden können. Eine weitere Komponente wird zur Darstellung der EKG Daten benötigt, welche in Form einer EKG Kurve umgesetzt werden sollen. Die dritte benötigte Komponente dient dem Chat der Kommunikation mit anderen Benutzern der Web-Anwendung. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Die Darstellung der Web-Anwendung ist praxis- aber nicht designbezogen. Die Darstellung aller Komponenten soll der Übersicht dienen. |

**Visualisierung der EKG Daten**

| | |
|---|---|
| Nummer der Anforderung | 24 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Die Entscheidung eine Web-Anwendung zu entwickeln ermöglicht den Einsatz von HTML. Mit der Verwendung von HTML bietet ein Web-Browser eine große Menge an Möglichkeiten, um ganz unterschiedliche Arten von Daten visualisieren zu können. |
| Auswirkung auf Komponente | Web-Anwendung |
| Info | Auf den Einsatz von kostenintensiver Software Dritter kann daher verzichtet werden, was die Plattformunabhängigkeit unterstreicht und die Kosten für einen Einsatz nicht erhöht. |

**Einsatz in fremden Netzwerken**

| | |
|---|---|
| Nummer der Anforderung | 25 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Die EKG Daten Sender sollen in fremden Netzwerken eingesetzt werden können, auf welche kein Einfluss genommen werden kann. Dies ist keine triviale Anforderung, da fast jedes Netzwerk durch ein NAT vom Adressraum des Internet entkoppelt ist und eigenen Regeln unterliegt. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Wie in Kapitel 2.2.4 beschrieben, bietet WebRTC mit der Verwendung des STUN Protokolls die Möglichkeit der NAT-Traverse. Damit wird es möglich eine Verbindung zu einem EKG Daten Sender, der sich in einem fremden Netzwerk befindet, herzustellen und die Anforderung zu erfüllen. Die Möglichkeit eine Verbindung in fremde nicht zu beeinflussenden Netzwerke herzustellen, ermöglicht einen flächendeckenden Einsatz in vielen privaten und öffentlichen Netzwerken. |

**Verschlüsselung**

| | |
|---|---|
| Nummer der Anforderung | 26 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Die Echtzeit Übertragung von sensiblen Daten soll auch ermöglicht werden, womit eine Verschlüsselung der Daten notwendig wird.<br>Da es sich bei EKG Daten um persönliche personengebundene Daten handelt besteht die Notwendigkeit, die Daten verschlüsselt zur Web-Anwendung zu übertragen. Somit bedient das gewählte Entwicklungsszenario auch das Ziel der Verschlüsselung. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Die Übertragung von verschlüsselten Daten ist in vielen Einsatzbereichen unumgänglich, vor allem wenn die Sensordaten in fremden Netzwerken und über das Internet übertragen werden sollen. |

**Peer-to-Peer Verbindung**

| | |
|---|---|
| Nummer der Anforderung | 27 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Der Empfang von Daten vieler Sender an einem zentralen Punkt soll ohne den Einsatz teurer Server ermöglicht werden, weshalb ein Peer-to-Peer Verbindung zu den Sendern angestrebt wird. Eine Peer-to-Peer Verbindung vermeidet den Einsatz eines Servers, der den Traffic aller eigesetzten Sender empfängt und dazu viele parallele Verbindungen halten und verwalten muss. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Der Einsatz einer Peer-to-Peer Verbindung für die Übertragung der Daten ist unabhängig vom gewählten Szenario und dient allein der kostengünstigen Realisierung. Durch den Einsatz einer Peer-to-Peer Verbindung, kann auf die Verwendung eines kostenintensiven Servers, der die Daten vom EKG Daten Sender zur Instanz der Web-Anwendung weitersendet, verzichtet werden. Die Möglichkeit auf die Weiterversendung eines Servers verzichten zu können, spart die Hälfte des benötigten Übertragungsvolumens ein. Die Verzögerung einer direkten Peer-to-Peer Verbindung ist zudem bis um die Hälfte kürzer und damit deutlich kleiner, was der Betrachtung in Echtzeit entgegenkommt. |

**Echtzeit**

| | |
|---|---|
| Nummer der Anforderung | 28 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Um die Beobachtung von zeitkritischen Daten zu ermöglichen, soll die Übertragung der Daten den Anforderungen der weichen Echtzeit genügen. Damit können Daten nahezu verzögerungsfrei verarbeitet und beobachtet werden. Die Übermittlung von EKG Daten erfolgt in Echtzeit, da die Darstellung von verzögerten EKG Daten für eine direkte Beobachtung nicht sinnvoll ist. Damit im Notfall schnell reagiert werden kann, müssen die Daten in Echtzeit dargestellt werden und dem aktuellen Moment entsprechen. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Zu berücksichtigen ist allerdings, dass die wachsende Verzögerung, bei wachsender Entfernung zwischen Sender und Empfänger, nicht verhindert werden kann. Die Möglichkeit EKG Sender Daten in Echtzeit zu Visualisieren und zu betrachten erweitert das Einsatzgebiet dieser Web-Anwendung zusätzlich, da die Verarbeitung von Daten in Echtzeit in vielen Fällen angestrebt wird.<br>Zusätzlich ist die folgende Anforderung 29 zu beachten, die den Ausgleich von Jitter beschreibt. |

**Ausgleich von Jitter**

| | |
|---|---|
| Nummer der Anforderung | 29 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Durch den Einsatz in fremden Netzwerken, die über das Internet erreicht werden sollen, muss die variierende Paketlaufzeit einer Internetübertragung berücksichtig werden. Die schwankende Laufzeit von Daten und das damit unregelmäßige Eintreffen von Daten soll ausgeglichen werden und der Ausgleich möglichst minimiert werden, damit die Beobachtung in Echtzeit weiterhin möglich ist.<br>Die zu entwickelnde Lösung soll eine Übertragung über das Internet ermöglichen. Da das Laufzeitverhalten einer solchen Lösung nie exakt vorbestimmt werden kann, wird ein flexibler Mechanismus zum Ausgleich des Jitters benötigt. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Als Jitter wird die variierende Laufzeit von Daten, bei der Übertragung vom Sender zum Empfänger, bezeichnet. Die zu entwickelnde Anwendung soll dazu genutzt werden können, Echtzeitdaten entfernter EKG Daten Sender zu betrachten, womit immer eine Laufzeit besteht. Der Ausgleich von Jitter ist bei Echtzeitdaten kritisch und muss gut überlegt sein.<br>Echtzeitdaten sollen möglichst schnell übertragen werden und einem bestimmten Zeitverhalten entsprechen. Der Ausgleich von Jitter, durch die Verwendung eines Buffers, beeinflusst das Zeitverhalten und muss daher an die gegebenen Umstände angepasst werden. |

**Plattformunabhängigkeit**

| | |
|---|---|
| Nummer der Anforderung | 30 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Die Entscheidung eine Web-Anwendung zu entwickeln, unterstreicht die Plattformunabhängigkeit, da die Referenz Web-Browser Chrome und Firefox auf vielen Plattformen angeboten werden. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Plattformunabhängigkeit ist bei der Entwicklung eines nicht fest spezifizierten Systems fast immer erwünscht, um das System möglichst flexibel einsetzen zu können. Daher soll eine möglichst Plattformunabhängig Lösung in dieser Bachelorarbeit entwickelt werden. |

**Kostengünstiger Einsatz**

| | |
|---|---|
| Nummer der Anforderung | 31 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Neben der Plattform Unabhängigkeit gelten viele der oben genannten Ziele einer Erweiterung des Einsatzgebiets. Daher soll der Einsatz dieser Lösung auch mit möglichst geringen Kosten ermöglicht werden. Auf den Einsatz von kostenpflichtiger Software Dritter und kostenintensiver Hardware soll daher verzichtet werden. |
| Auswirkung auf Komponente | Web-Anwendung, EKG Daten Sender |
| Info | Durch einen kostengünstigen Einsatz soll das Einsatzgebiet auf Gesellschaften und Projekte ausgeweitet werden, für die kostenintensive Anschaffungen und Unterhaltskosten nicht möglich oder gerechtfertigt sind. |

**EKG Daten Sender**

| | |
|---|---|
| Nummer der Anforderung | 32 |
| Quelle | Entwicklungsszenario, Zielsetzung |
| Beschreibung | Zur Entwicklung gehört neben der Web-Anwendung selbst auch ein System, welches zum Senden der EKG Daten an die Web-Anwendung dient. |
| Auswirkung auf Komponente | EKG Daten Sender |
| Info | Es werden keine gemessenen EKG Daten gesendet. Die EKG Daten die versendet werden stammen aus einer Datei, in der zuvor simulierte EKG Daten gespeichert wurden. |

# 4 Design

In diesem Kapitel wird das Design des WebRTC basierten Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung beschrieben. Dies soll das Zusammenspiel der einzelnen Komponenten und deren Informationswege aufzeigen.
Dabei wird zunächst das gesamte Managementsystem mit seinen Komponenten beschrieben und in Abbildung 4-20 dargestellt.

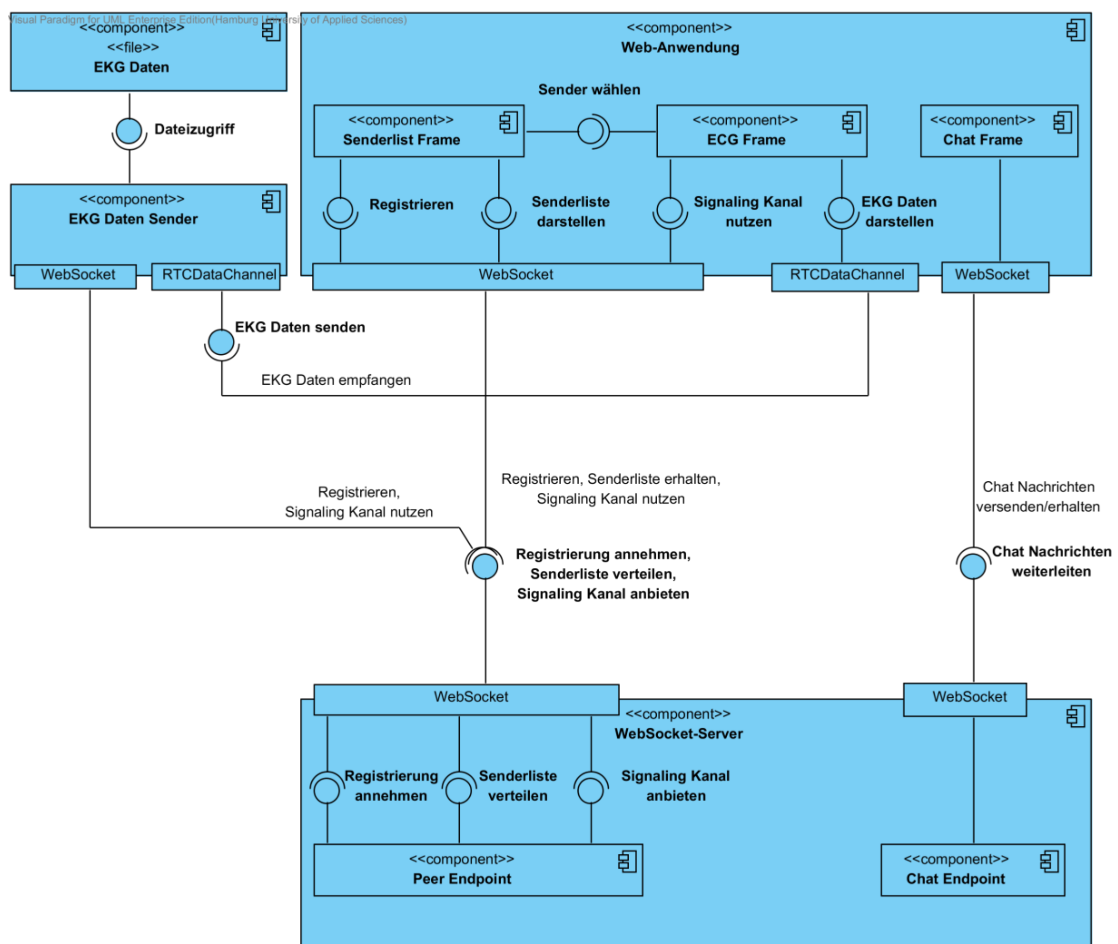



Abbildung 4-20.: Übersicht der Komponenten des Managementsystems

Wie der Abbildung 4-20 zu entnehmen ist, gibt es drei Komponenten die zur Entwicklung des WebRTC basiertes Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung nötig sind.
Der links oben angeordnete EKG Daten Sender liest die EKG Daten aus einer Datei ein und kann diese über einen RTCDataChannel versenden. Nach dem Start des EKG Daten Sender baut dieser eine WebSocket-Verbindung zum Peer Endpoint des WebSocket-Server auf, um sich bei diesem zu registrieren. In Kapitel 4.3 wird der EKG Daten Sender genauer beschrieben.
Die rechts oben angeordnete Web-Anwendung soll wie zuvor in Kapitel 3 beschrieben, eine Liste mit verbundenen EKG Daten Sendern anzeigen, eine EKG Kurve darstellen und einen Chat für die Benutzer anbieten. Die Web-Anwendung besteht somit aus drei Hauptkomponenten und wird in Kapitel 4.2 beschrieben. Die Web-Anwendung dient dem Benutzer zudem dazu, eine Peer-to-Peer Verbindung, in Form eines RTCDataChannel, zum EKG Daten Sender herzustellen. Dies ermöglicht den direkten Austausch der EKG Daten und somit den Verzicht vom Einsatz teurer Server, wie es in Anforderung 27 beschrieben wird.
Nach dem Start einer Web-Anwendungsinstanz registriert sich diese, wie zuvor der EKG Daten Sender, beim Peer Endpoint des WebSocket-Server und bekommt daraufhin eine Senderliste von diesem zugeschickt, in der der bereits registrierte EKG Daten Sender enthalten ist. Nun kann der Benutzter der Web-Anwendung, über die dargestellte Senderliste den registrierten EKG Daten Sender auswählen, um eine direkte Peer-to-Peer Verbindung zu diesem aufzubauen. Daraufhin sendet der EKG Daten Sender die EKG Daten über die Peer-to-Peer Verbindung, welche dann im Web-Browser, der die Web-Anwendungsinstanz ausführt, in Form einer EKG Kurve dargestellt und vom Benutzer beobachtet werden kann.
Der in Abbildung 4-20 unten angeordnete WebSocket-Server bietet einen Peer Endpoint und einen Chat Endpoint an, welche in den Kapiteln 4.1.2 und 4.1.3 genauer beschrieben werden. Endpoints sind, wie in Kapitel 2.1.5 erwähnt, Dienste des WebSocket-Servers. Die Dienste der WebSocket-Endpoints werden von der rechts oben angeordneten Web-Anwendung und dem links oben angeordneten EKG Daten Sender genutzt. Der Peer Endpoint des WebSocket-Server nimmt die Registrierung von EKG Daten Sender und Web-Anwendungsinstanz entgegen, verteilt die Senderliste mit registrierten EKG Daten Sendern an die registrierten Instanzen der Web-Anwendung und bietet einen Signaling Kanal zwischen EKG Daten Sender und Web-Anwendungsinstanzen. Der Chat Endpoint dient hingegen nur den Instanzen der Web-Anwendung und leitet eine Chatnachricht, die von einer Instanz versendet wurde, an alle anderen Instanzen der Web-Anwendung weiter.

## 4.1 WebSocket-Server

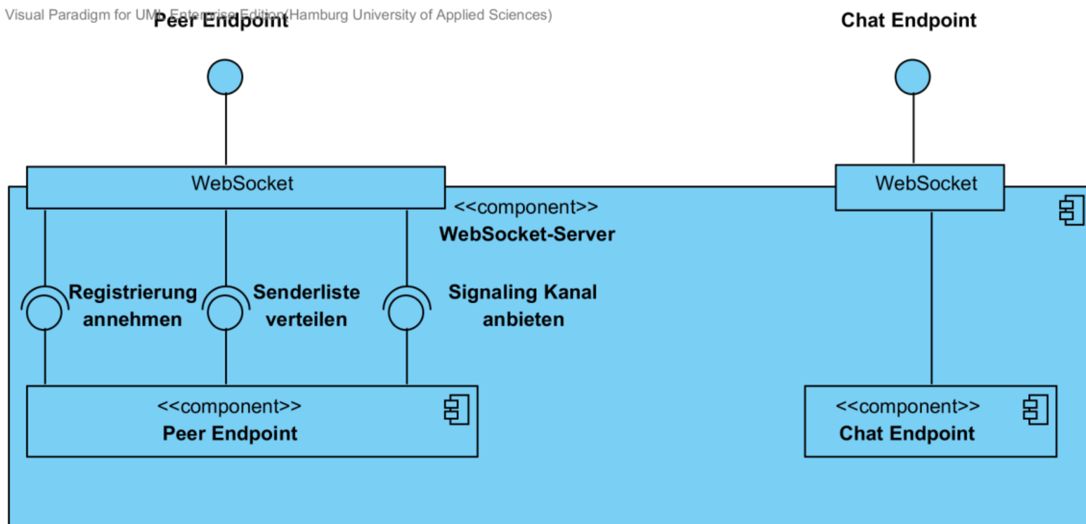



Abbildung 4.1-21.: Komponente - WebSocket-Server

Der WebSocket-Server bietet zwei Endpoints an und damit zwei Dienste für unterschiedliche Einsatzzwecke. Es handelt sich um einen Chat Endpoint und um einen Peer Endpoint, welche beide in den folgenden Kapiteln beschrieben werden und in Abbildung 4.1-21 dargestellt sind.
Der Chat Endpoint dient der Übermittlung von Chat Nachrichten unter allen Instanzen der Web-Anwendung.
Der Peer Endpoint hat ein breiteres Aufgabengebiet und dient als Signaling Kanal für den Aufbau von RTCDataChannels, nimmt Registrierungen von EKG Daten Sendern wie auch von Instanzen der Web-Anwendung an und verteilt die Senderliste mit registrierten EKG Daten Sendern an registrierte Instanzen der Web-Anwendung.

### 4.1.1 Message-System

Für die auf dem WebSocket-Server eingesetzten Endpoints wird ein Endpoint übergreifendes Message-System entwickelt, was aus eigens entworfenen Klassen besteht, um Messages einfach zu identifizieren und passend zu verarbeiten.
Jede Message gehört dabei einer Kategorie an, die den Einsatz der Message beschreibt (s. Abbildung 4.1.1-22). Die Kategorisierung von Messages ermöglicht es, diese ihrem Einsatzzwecks zuzuordnen.
Die in Abbildung 4.1.1-22 beschrieben Messages, decken den kompletten Bedarf an Messages ab, die über die entwickelten Endpoints versendet werden.

Für Nutzdaten sieht das WebSocket-Protokoll zwei Typen von Messages vor. Dabei handelt es sich, wie in Kapitel 2.1.6 beschrieben, um Messages vom Typ „Text“ oder „Binary“. Auf dem Transportweg sollen Messages vom Typ „Text“ eingesetzt werden, die zuvor JSON codiert wurden. Die JSON Datencodierung der WebSocket-Messages wird serverseitig von Decoder- und Encoder-Klassen umgesetzt. JSON bietet eine geordnete Struktur die gut lesbar ist, einen geringen Overhead mitbringt und einfach unter Java und JavaScript eingesetzt werden kann.

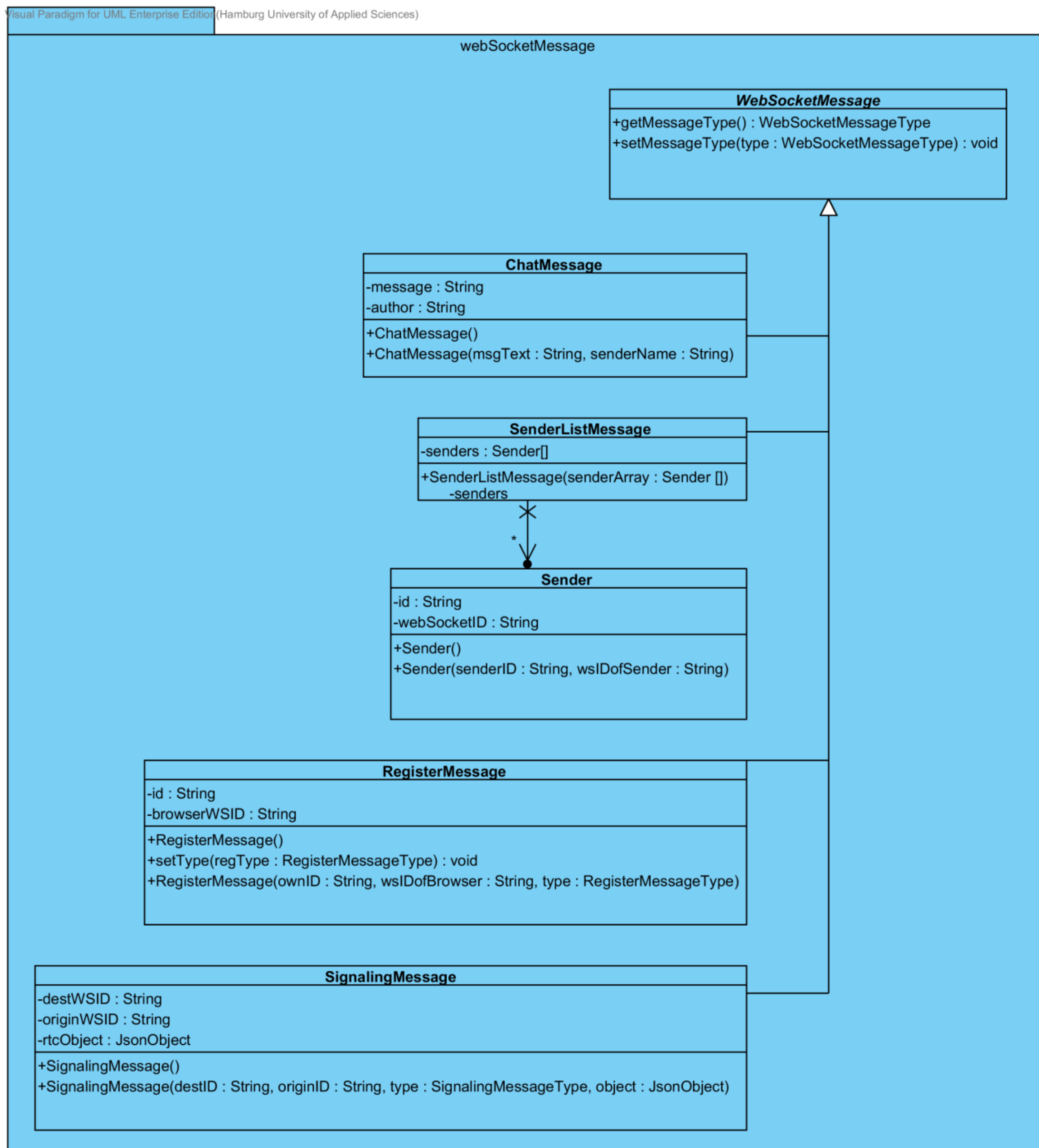



Abbildung 4.1.1-22.: WebSocket-Server - Message-System

### 4.1.2 Peer Endpoint

Die Aufgaben des Peer Endpoints können in drei Kategorien aufgeteilt werden: Registrierung, Senderliste und Signaling Kanal.
Der Peer Endpoint dient zur Registrierung von Instanzen der Web-Anwendung und Instanzen des EKG Daten Senders. Die Registrierung von Instanzen der Web-Anwendung EKG Daten Sendern dient dem Peer Endpoint zur Identifizierung der aufgebauten WebSocket-Verbindungen.
Um Instanzen der Web-Anwendung über verfügbare registrierte EKG Daten Sender in Form einer Senderliste zu informieren, kommuniziert der Peer Endpoint diesen den Instanzen der Web-Anwendung direkt.
Der Peer Endpoint dient, in Form eines Signaling Kanal, zur Kommunikation zwischen den Instanzen der Web-Anwendung und den Instanzen des EKG Daten Senders über welchen Signaling Messages ausgetauscht werden, um eine Peer-to-Peer Verbindung herzustellen.

**Registrierung**

Jede Instanz der Web-Anwendung und jede Instanz eines EKG Daten Senders, baut nach dem Start eine Verbindung zum Peer Endpoint auf. Dem Peer Endpoint ist somit die Anzahl hergestellter WebSocket-Verbindungen bekannt. Dem Peer Endpoint ist aber nicht bekannt, wie viele der offenen WebSocket-Verbindungen von Web-Browsern und wie viele von EKG Daten Sendern hergestellt worden sind.
Erkannt werden könnte dies beim WebSocket-Handshake, da Web-Browser, wie in Kapitel 2.1.5 beschrieben, immer das Header-Feld Origin nutzen. Da aber auch native Clients das Header-Feld Origin benutzen können und der Inhalt des Header-Felds manipuliert werden kann, wäre die Identifizierung anhand des Header-Felds unsicher.
Daher ist eine simplere und zuverlässigere Methode zur Identifizierung der WebSocket-Verbindungen, in Form einer Registrierung vorgesehen, welche in Abbildung 4.1.2-23 dargestellt ist.
Instanzen der Web-Anwendung und EKG Daten Sender senden nach dem Aufbau der WebSocket-Verbindung zum Peer Endpoint eine Register Message, die es dem Peer Endpoint ermöglicht den Initiator der neue Verbindung zu identifizieren.
Der Peer Endpoint speichert dabei die WebSocket Session IDs der WebSocket Verbindungen, über die eine Register Message eingegangen ist. Die Register Message enthält die Information, ob es sich um die Registrierung eines EKG Daten Sender oder eines Web-Browsers handelt, der eine Instanz der Web-Anwendung ausführt. Anhand der Information wird die WebSocket Session ID in einer Map namens „SenderMap“ oder in einer Liste namens „BrowserList“ gespeichert.
Wird ein EKG Daten Sender oder eine Instanz der Web-Anwendung beendet, wird zum Abbau der WebSocket Verbindung, wie in Kapitel 2.1.8 beschrieben, ein Handshake ausgeführt. Nach Empfang eines Close Control Frame, der Teil des Handshake zum Abbau

einer WebSocket-Verbindung ist, wird die entsprechende WebSocket Session ID aus der BrowserList oder der SenderMap entfernt.

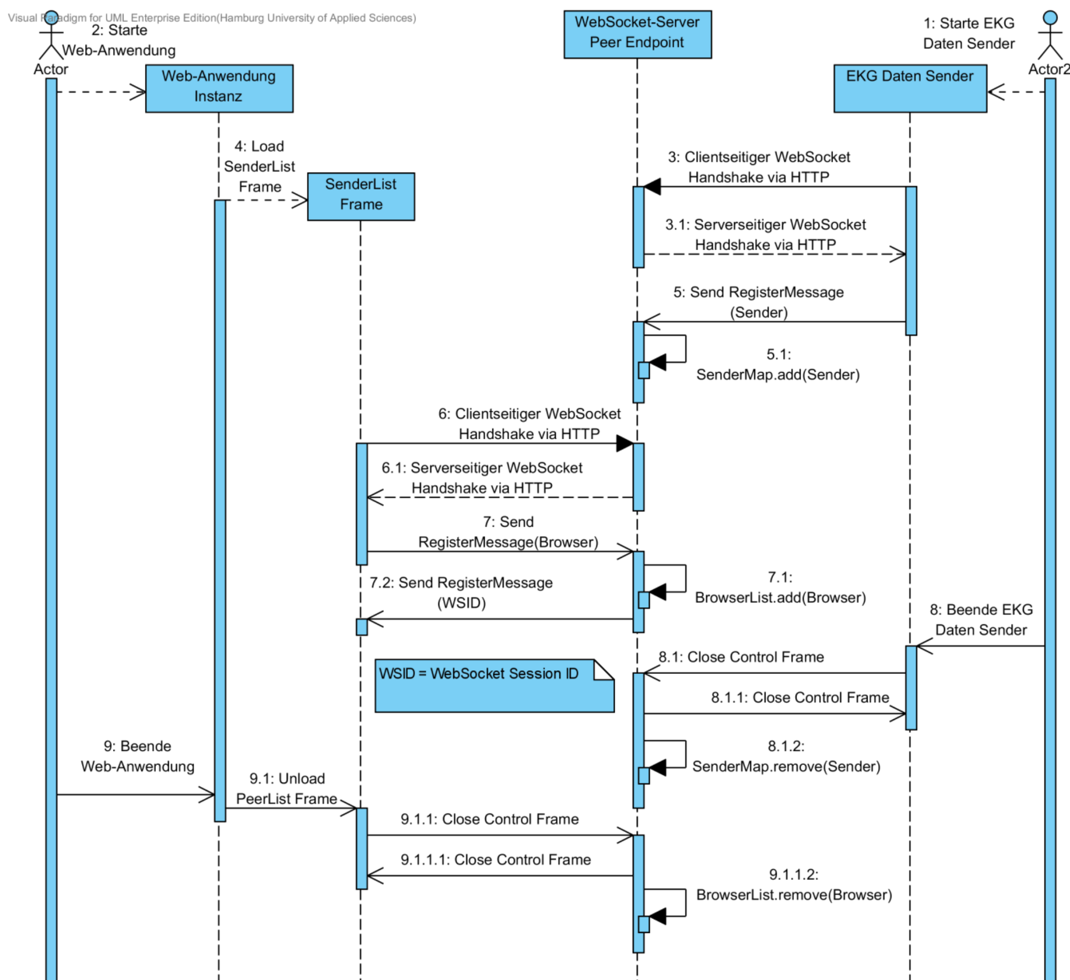



**Abbildung 4.1.2-23.: Peer Endpoint – Registrierung**

### Zusatz bei der Registrierung von Web-Anwendungen

In Abbildung 4.1.2-23 ist eine Reaktion auf die Register Message einer Web-Anwendungsinstanz in Form einer Message (Sequenznummer 7.2) zu sehen. In der Message wird der Web-Anwendungsinstanz die WebSocket Session ID mitgeteilt, unter der dem Peer Endpoint die WebSocket Verbindung zur Web-Anwendungsinstanz bekannt ist.
Jede Instanz der Web-Anwendung nutzt die eigene WebSocket Session ID des Peer Endpoint, um damit die Label von RTCDataChannels zu setzen. Bei der Initiierung eines RTCDataChannel durch eine Web-Anwendungsinstanz wird das Label mit der WebSocket

Session ID versehen. Auf diese Weise sind die RTCDataChannel Label einer Web-Anwendungsinstanz immer identisch.
Die Instanz eines EKG Daten Sender, zu der ein RTCDataChannel aufgebaut wird, nutzt das Label um die RTCDataChannel zu identifizieren, was nötig ist, wenn mehrere Instanzen der Web-Anwendung einen RTCDataChannel zu einer Instanz eines EKG Daten Senders aufbauen. In Kapitel 4.3 wird erläutert, wie der EKG Daten Sender die RTCDataChannels anhand des Label identifiziert.

**Senderliste**

Die Kommunikation zwischen WebSocket-Server und den Instanzen der Web-Anwendung wird genutzt, um den Instanzen der Web-Anwendung die verfügbaren EKG Daten Sender mitzuteilen und auf aktuellem Stand zu halten.
Jeder EKG Daten Sender und jede Instanz der Web-Anwendung registriert sich, wie In Abbildung 4.1.2-24 dargestellt, beim Start am Peer Endpoint des WebSocket-Servers. Dem Peer Endpoint ist daher immer bekannt, wie viele Instanzen der Web-Anwendung geöffnet und wie viele EKG Daten Sender eingesetzt werden.
Ändert sich die Anzahl an eingesetzten EKG Daten Sender, wird das allen registrierten Instanzen der Web-Anwendung mitgeteilt. Die Mitteilung erfolgt in Form einer aktuellen Liste der registrierten EKG Daten Sender, der sogenannten Senderliste. Die Senderliste besteht aus Einträgen von registrierten EKG Daten Sender, mit der jeweiligen ID des EKG Daten Senders und der jeweiligen WebSocket Session ID, unter der ein EKG Daten Sender dem Peer Endpoint bekannt ist.
Erhöht sich die Anzahl geöffneter Instanzen der Web-Anwendung, wird die Senderlist mit verfügbaren EKG Daten Sendern direkt an alle Instanzen der Web-Anwendung übertragen. Auf diese Weise erhält eine neue Instanz der Web-Anwendung immer eine aktuelle Senderliste, die eine Liste aller registrierter EKG Daten Sender beinhaltet.
Die Verwendung von WebSocket-Verbindungen ermöglicht die direkte Reaktion des Servers auf Änderungen von verfügbaren EKG Daten Sendern. Änderungen ergeben sich durch Registrierungen und Abmeldungen von EKG Daten Sendern, worauf der Peer Endpoint mit dem Versand einer aktuellen Senderliste an alle geöffneten Web-Anwendungsinstanzen reagiert. Die Senderliste stellt so immer den aktuellen Stand an verfügbaren EKG Daten Sendern dar, ohne das die Benutzer der Web-Anwendung tätig werden müssen.

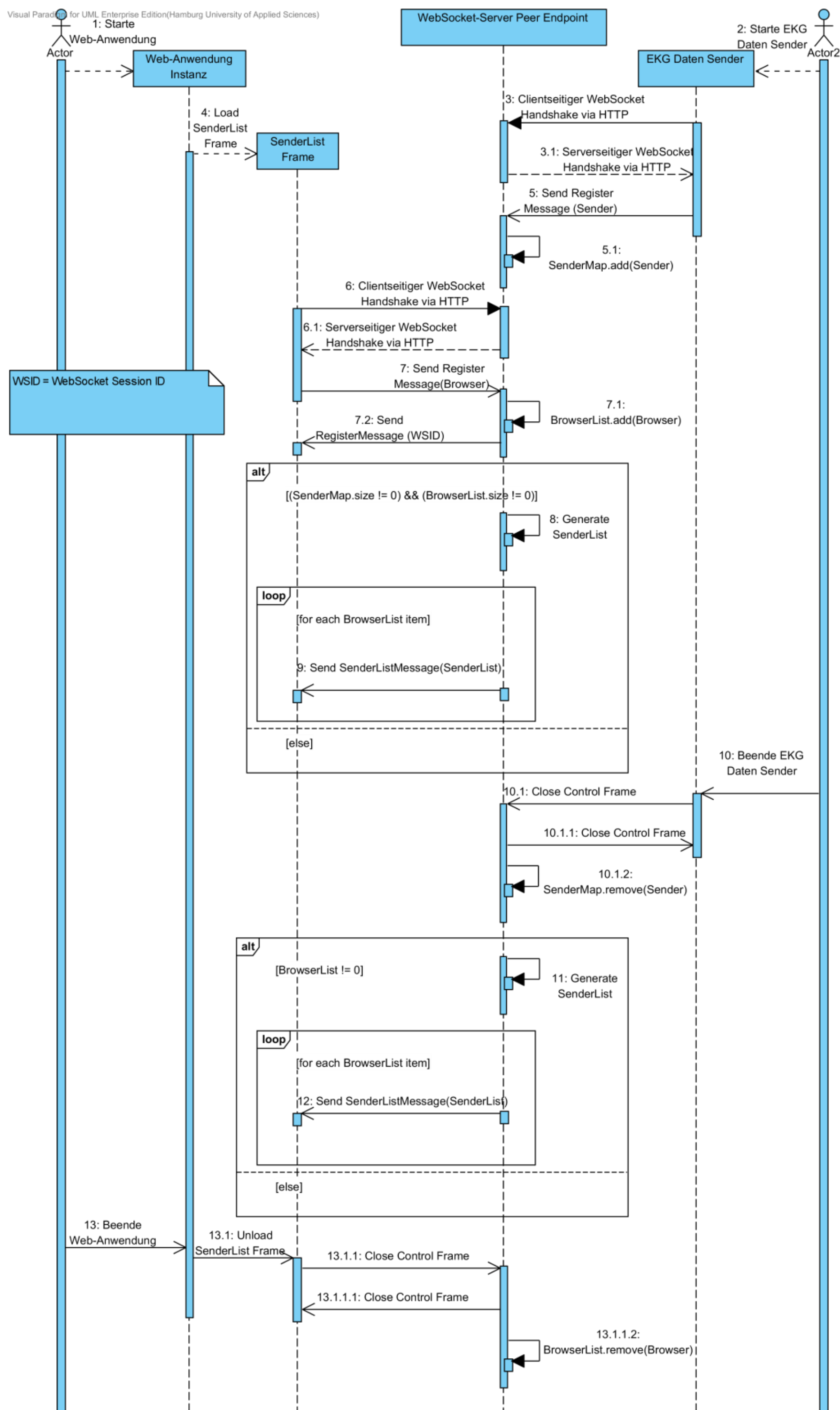



Abbildung 4.1.2-24.: Peer Endpoint – Verteilung der Senderliste

**Signaling Kanal**

Wie zuvor erwähnt, dient der Peer Endpoint auch der Kommunikation zwischen Instanzen der Web-Anwendung und Instanzen der EKG Daten Sender Anwendung. Dieser Kommunikationsweg ist notwendig, wenn ein Benutzer der Web-Anwendung eine Verbindung zu einem EKG Daten Sender herstellen möchte, da es sich dabei um den Aufbau einer RTCPeerConnection handelt. Wie in Kapitel 2.2 beschrieben, wird beim Aufbau einer RTCPeerConnection ein Signaling Kanal benötigt, der den beteiligten Peers zum Austausch von Offer-, Answer- und ICE Candidate Nachrichten dient.
Wie im obigen Absatz zur Registrierung beschrieben, registrieren sich die EKG Daten Sender und die Instanzen der Web-Anwendung beim Peer Endpoint, daher wird der Signaling Kanal über den Peer Endpoint realisiert, da dieser über WebSocket-Verbindungen zu beiden Peers verfügt und so die Signaling Messages einfach vermittelt kann.
In Abbildung 4.1.2-25 ist der vereinfachte Kommunikationsablauf zwischen einer Instanz der Web-Anwendung und einem EKG Daten Sender dargestellt. Die Abbildung stellt die Registrierung und eine folgende Übermittelung der Senderliste an die Instanz der Web-Anwendung dar. Anschließend wählt ein Benutzer den EKG Daten Sender aus der Senderliste aus, um eine Verbindung zu diesem herzustellen.
Alle Signaling Messages beinhalten die WebSocket Session ID des Empfängers, an den die Message weitergeleitet werden soll.
Die Instanz der Web-Anwendung erhält die WebSocket Session ID des EKG Daten Sender, mit der Senderliste, welche der Peer Endpoint an die Instanz der Web-Anwendung schickt, nachdem sich diese registriert hat.
Wählt der Benutzer den EKG Sender aus der Senderliste aus, kann dem Peer Endpoint anhand der WebSocket Session ID mitgeteilt werden, an welchen EKG Daten Sender die Offer-, Answer- und ICE Candidate Nachrichten weitegeleitet werden sollen.
Empfängt der Peer Endpoint eine Signaling Message, ist die WebSocket Session ID des Absenders bekannt. Der Peer Endpoint fügt die WebSocket Session ID des Absenders in die Signaling Message ein und schickt Signaling Messages anschließend an den Empfänger weiter.
Dadurch das die erste Signaling Message immer von einer Instanz der Web-Anwendung an den EKG Daten Sender gesendet wird und der Peer Endpoint die WebSocket Session ID des Absenders, also der Web-Anwendungsinstanz, in die Signaling Message einfügt, erfährt der empfangende EKG Daten Sender die WebSocket Session ID der Web-Anwendungsinstanz. Der EKG Daten Sender kann so beim Versenden eigener Signaling Messages, die WebSocket Session ID der Web-Anwendungsinstanz angeben, die der Peer Endpoint zur Weiterleitung benötigt.
Die Peers, die zum Aufbau einer RTCPeerConnection, Signaling Messages über den Peer Endpoint versenden, werden von diesem immer anhand der WebSocket Session ID identifiziert. Durch Einfügen der WebSocket Session ID des Absenders in die Signaling Message, muss nur die initiierende Instanz der Web-Anwendung, die WebSocket Session ID des EKG Daten Senders kennen, welche in der Senderliste enthalten ist.

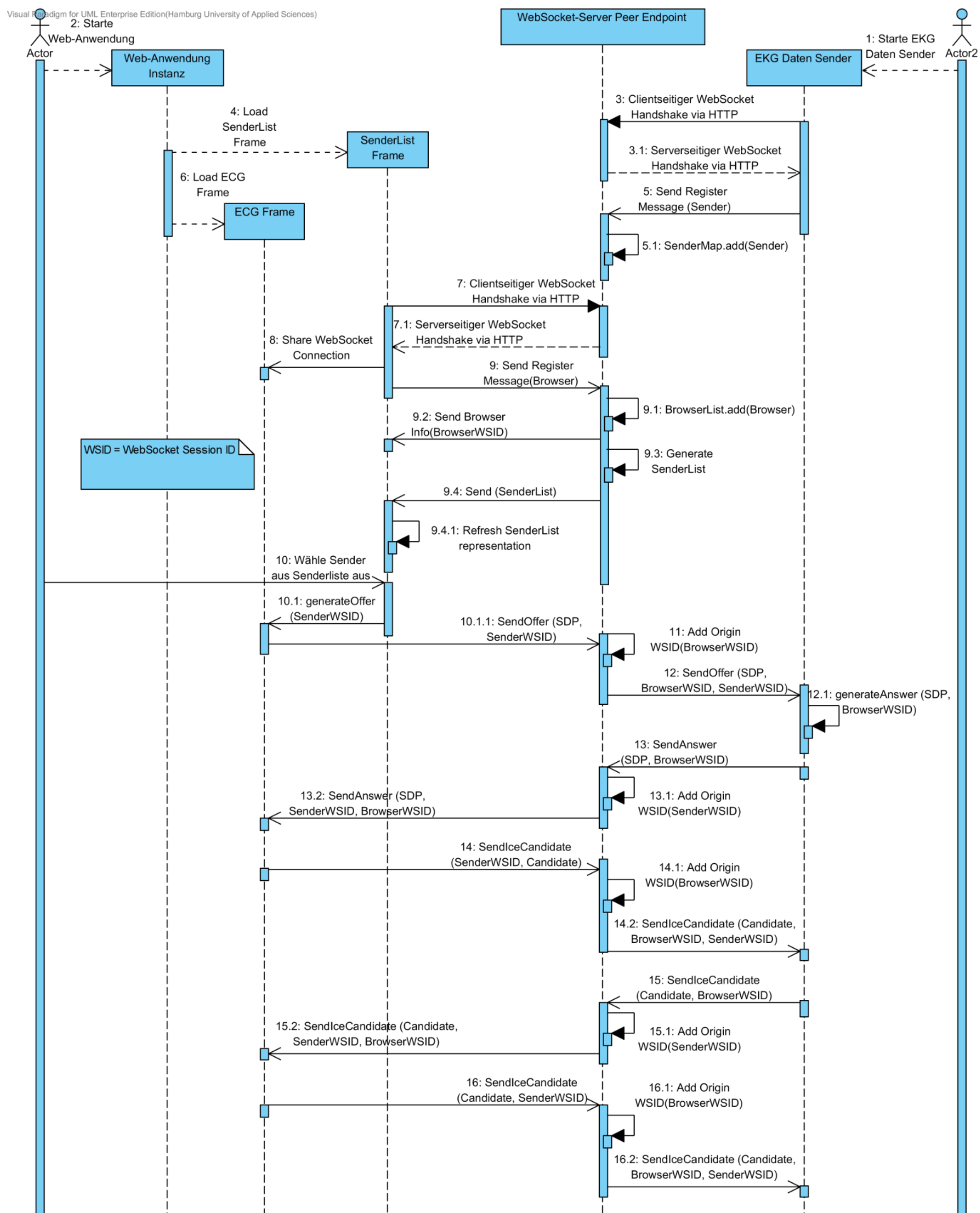



**Abbildung 4.1.2-25.: Peer Endpoint - Signaling**

Die Kommunikation über den Signaling Kanal kann auch einen anderen unregelmäßigeren Verlauf nehmen, als in Abbildung 4.1.2-25 dargestellt. Wie in Kapitel 2.2 beschrieben, ermittelt der ICE Agent einer RTCPeerConnection ICE Candidates. Die Ermittlungsdauer ist

unbestimmt und von Faktoren, wie beispielsweise Verzögerung zum STUN- oder TURN-Server oder die Anzahl zu durchquerender NATs, abhängig.
Wann und in welcher Reihenfolge der Austausch von ICE Candidate Nachrichten erfolgt ist daher nicht vorhersagbar, das Gleiche gilt für deren Anzahl. Da WebSocket-Verbindungen asynchronen bidirektionalen Datenaustausch unterstützten, können Signaling Messages jederzeit an den Peer Endpoint geschickt und von diesem direkt weitergesendet werden. Ein unregelmäßiger Kommunikationsverlauf ist daher unproblematisch.

### 4.1.3 Chat Endpoint

Wie in Kapitel 2.1.1 beschrieben wurde, ist das WebSocket-Protokoll für die direkte Kommunikation von Nachrichten zwischen Client und Server ausgelegt. Die Eigenschaften einer WebSocket-Verbindung, die eine bidirektionale asynchrone Übermittlung von Messages jederzeit ermöglichen, sind die Gründe dass für den Chat der Web-Anwendung eine Lösung mit Einsatz des WebSocket-Protokolls entwickelt wird.
Der Chat Endpoint wird genutzt um Chat Nachrichten zwischen den Instanzen der Web-Anwendung auszutauschen. Die Eigenschaften des WebSocket-Protokolls passen genau zu den Anforderungen einer Chat Anwendung, die jederzeit von einem Benutzter genutzt werden kann, um eine Chat Nachricht an andere Benutzer zu senden ohne dass zuvor eine Nachricht empfangen worden sein muss.

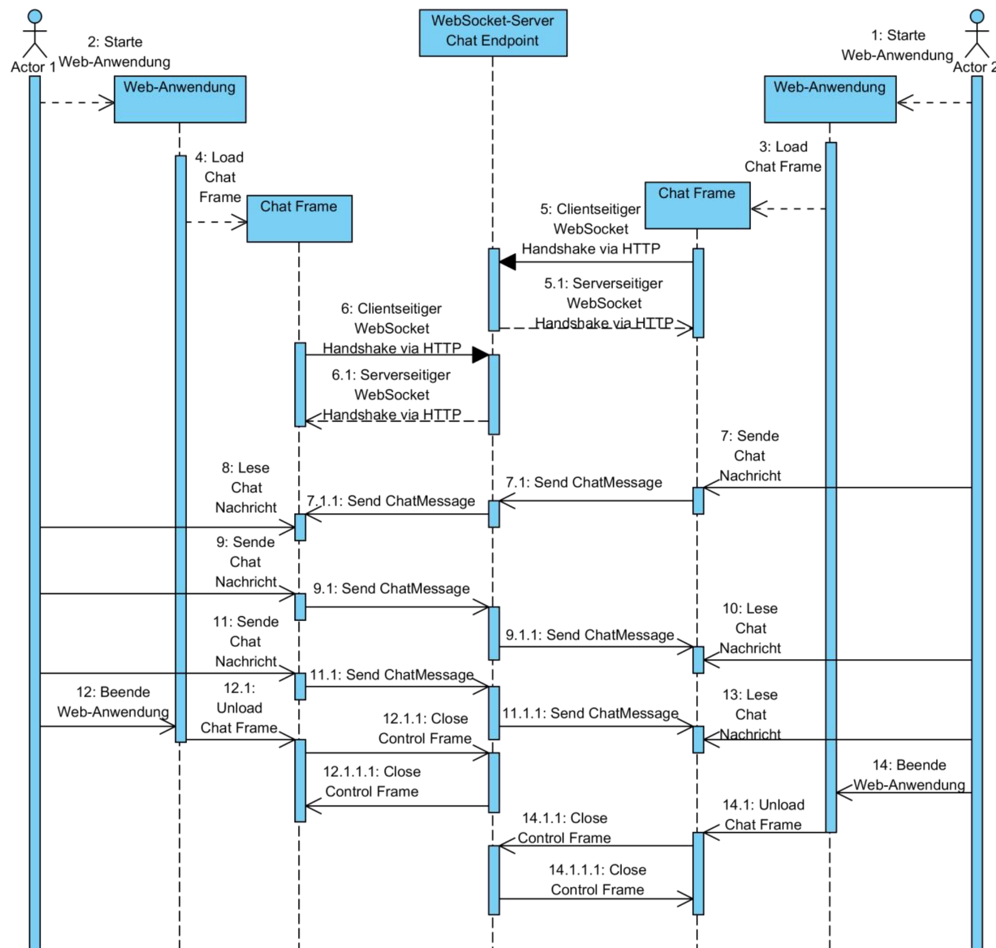



Abbildung 4.1.3-26.: Chat Endpoint – Asynchrone Nachrichtenübermittlung

Da es sich beim Einsatz des WebSocket-Protokolls um ein Client-Server-Modell handelt, werden alle Messages an den Chat Endpoint des WebSocket-Server gesendet und von diesem an alle anderen Instanzen der Web-Anwendung weitergeleitet.
In Abbildung 4.1.3-26 ist die Weiterleitung von Chat Nachrichten über den Endpoint und das asynchrone Senden und Empfangen von Chat Nachrichten dargestellt.

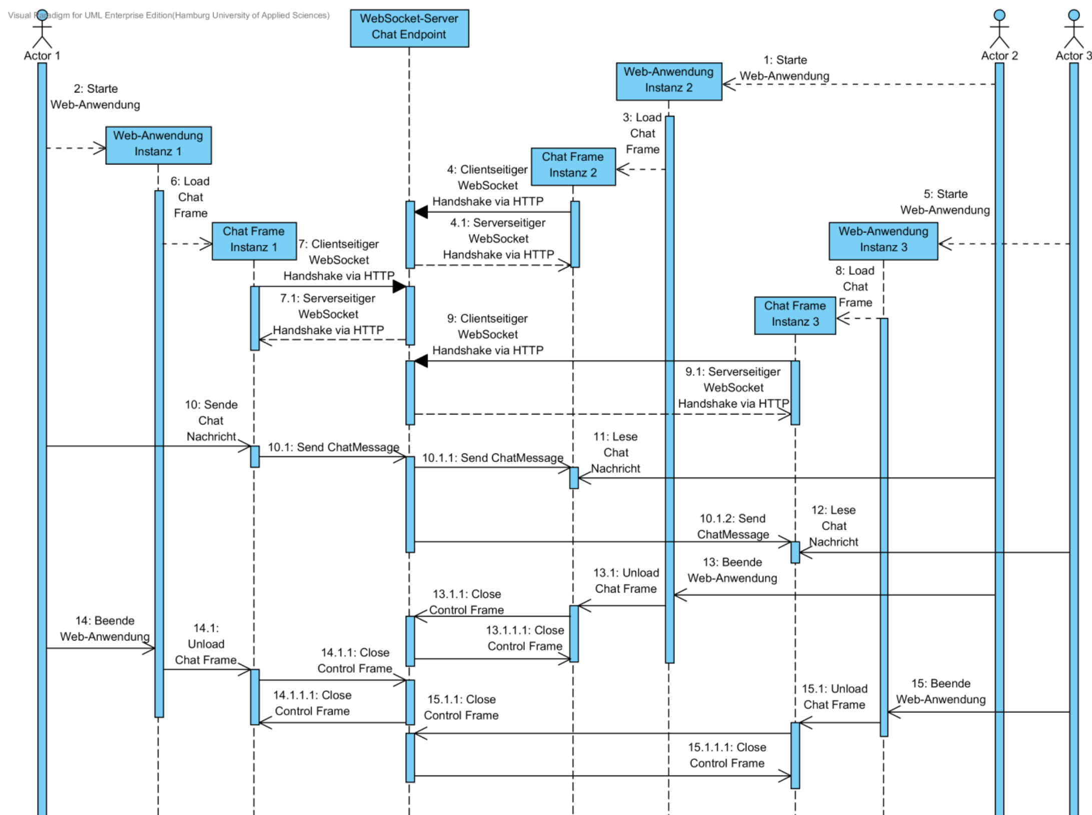



Abbildung 4.1.3-27.: Chat Endpoint – Nachrichtenübermittlung an mehrere Instanzen

Jede Instanz der Web-Anwendung hat einen Chat Frame in dem der Chat realisiert ist. Der Chat Frame stellt nach dem Start der Web-Anwendungsinstanz, eine WebSocket-Verbindung zum Chat Endpoint des WebSocket-Servers her.
Wird eine Chat Nachricht in Form einer ChatMessage an den Chat Endpoint gesendet, leitet dieser die ChatMessage an alle WebSocket-Verbindungen weiter, mit Ausnahme der WebSocket-Verbindung des Chat Frame von dem die Chat Nachricht verfasst und abgesendet wurde.
In Abbildung 4.1.3-27 ist die Arbeitsweise des Chat Endpoint mit mehreren Benutzern der Web-Anwendung dargestellt. In der Abbildung ist zu sehen, dass eine Chat Nachricht vom Chat Frame der Web-Anwendung an den Chat Endpoint des WebSocket Server gesendet und von diesem an alle anderen verbundenen Instanzen der Web-Anwendung weitergeleitet wird.

## 4.2 Web-Anwendung

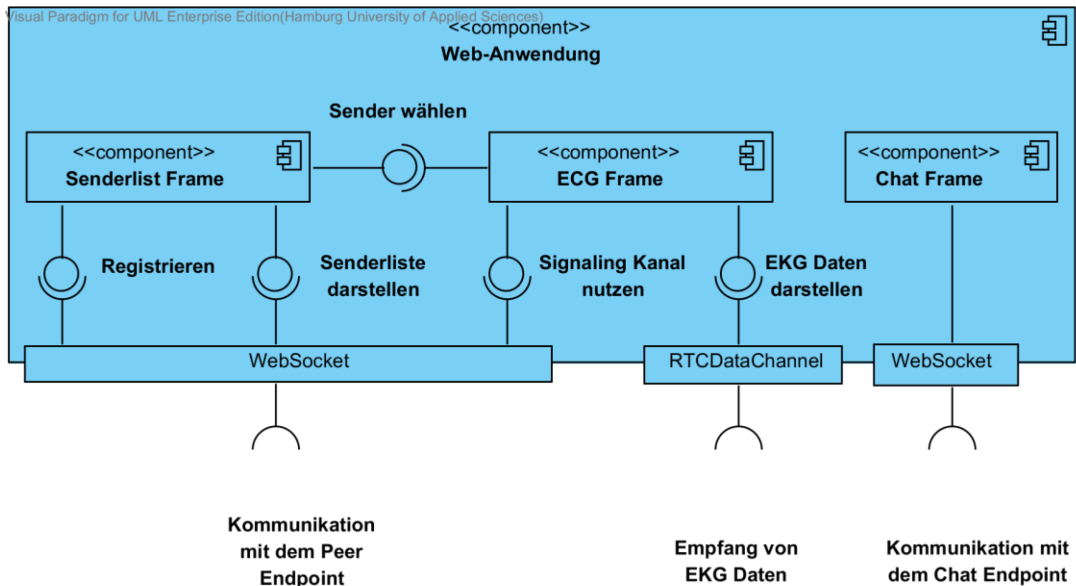



Abbildung 4.2-28.: Komponente - Web-Anwendung

Die Web-Anwendung ist in Form einer Web-Seite über einen Web-Browser erreichbar und wird von diesem dargestellt. Ein Benutzer kann über die im Web-Browser dargestellte Web-Seite die Web-Anwendung bedienen und benutzen. Wie in Abbildung 4-20 dargestellt soll die Web-Seite mindestens drei Funktionen erfüllen. Es soll eine Senderliste dargestellt werden, worin aktuell eingesetzte EKG Daten Sender geführt werden, es soll einen Chat geben, über den der Nachrichtenaustausch mit andere Benutzern der Web-Anwendung möglich ist und es sollen nach Auswahl eines EKG Daten Senders aus der Senderliste EKG Daten in Form einer Kurve dargestellt werden. Diese drei Bereiche werden in den folgenden Unterkapiteln beschrieben und sind in Abbildung 4.2-28 als innere Komponenten dargestellt.

### 4.2.1 Senderliste

Die Darstellung der Senderliste wird durch einen Frame der Webseite umgesetzt, der immer für den Benutzer sichtbar ist. Die Übermittlung der registrierten EKG Daten Sender erfolgt in Form einer Senderliste vom Peer Endpoint, wie in Kapitel 4.1.2 beschrieben und durch Abbildung 4.1.2-24 dargestellt.

Die einzelnen EKG Daten Sender sind in der Listendarstellung durch eine ID für den Benutzer identifizierbar. Bei der ID handelt es sich um die EKG Daten Sender ID, die beim Starten eines EKG Daten Senders angegeben wird.

### 4.2.2 Chat

Wie im Kapitel 4.1.2 beschrieben, nutzt der Chat eine WebSocket-Verbindung zum Chat Endpoint, um Chat Nachrichten zu versenden und zu empfangen. Nach Aufruf der Web - Anwendung stellt der Chat Frame eine WebSocket-Verbindung zum Chat Endpoint her. Anschließend kann der Chat vom Benutzer der Web-Anwendung genutzt werden.
Der Chat wird in einem kleinen Teil der Web-Seite dargestellt und besteht nur aus einer Eingabezeile und einem Fenster für Chat Nachrichten. Da jede Chat Nachricht an alle Instanzen der Web-Anwendung verschickt wird gibt es keine Benutzerliste wie man sie aus Instant-Messangern kennt. Die Eingabezeile wird als Erstes dazu genutzt, um seinen Namen anzugeben, so dass beim Versenden einer Chat Nachricht dieser Name als Absender der Nachricht angezeigt werden kann. Nach der Eingabe eines Chat Namen wird die Eingabezeile genutzt um Chat Nachrichten zu verfassen.

### 4.2.3 EKG Kurve

Der Frame der Web-Anwendung, in dem die EKG Kurve dargestellt wird, wird den größten Bereich ausmachen. Die EKG Kurve wird eine Auflösung von 400 Samples/s betragen und daher, bei der Kurvendarstellung einiger Sekunden die Breite eines üblichen PC Monitors mit Full-HD Auflösung füllen. Der Bereich wird mit einem Gitternetz hinterlegt, so dass die EKG kurve leichter abgelesen und interpretiert werden kann.
In Kapitel 4.1.2 wurde beschrieben, dass die EKG Daten über eine RTCPeerConnection zwischen Instanz der Web-Seite und einem EKG Daten Sender übertragen werden. Die empfangenen EKG Daten werden in Form einer Kurve dargestellt, wobei dir Kurve fortlaufend von links nach rechts gezeichnet wird. Ist der Bereich für die EKG Kurve vollständig mit der fortlaufenden EKG Kurve ausgenutzt, wird wieder mit der Darstellung der EKG Kurve auf der linken Seite des Bereichs begonnen. Die zuvor gezeichneten EKG Daten werden beim wiederholten Nutzen des Bereichs entfernt. Auf diese Weise ist der Bereich der EKG Kurve immer mit den EKG Daten der letzten Sekunden gefüllt und der Darstellung eines herkömmlichen einkanaligen EKGs ähnlich.

## 4.3 EKG Daten Sender

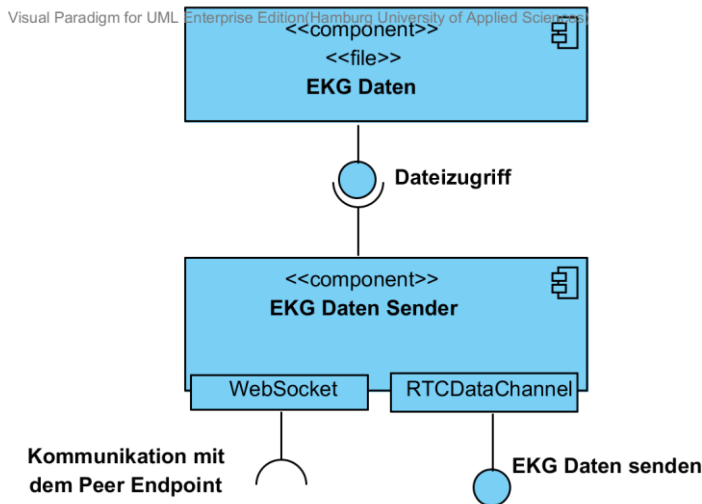



Abbildung 4.3-29.: EKG Daten Sender - Komponente

Beim EKG Daten Sender handelt es sich um eine Anwendung, die ein mobiles internetfähiges EKG simuliert. Die EKG Daten werden, wie in Abbildung 4.3-29 dargestellt, aus einer Datei ausgelesen, in der zuvor generierte EKG Daten gespeichert wurden.
Wird der EKG Daten Sender gestartet, registriert sich dieser, wie in Abbildung 4.1.2-23 gezeigt, beim Peer Endpoint. Die Senderliste von laufenden Instanzen der Web-Anwendung wird anschließend um einen Listeneintrag erweitert, der anhand der ID des EKG Daten Senders identifiziert werden kann. Wählt ein Benutzer der Web-Anwendung den Listeneintrag aus, wird eine RTCPeerConnection zum EKG Daten Sender hergestellt.
Für den Austausch der Signaling Messages, die beim Aufbau der RTCPeerConnection nötig sind, wird vom Peer Endpoint ein Signaling Kanal gestellt, wie in Abbildung 4.1.2-25 abgebildet, der vom EKG Daten Sender genutzt wird.
Beim Erzeugen der RTCPeerConnection und dem darauf folgenden Empfang des RTCDataChannel, werden diese in Maps gespeichert, wobei die WebSocket Session ID der Web-Anwendungsinstanz als Key verwendet wird. So können die Pärchen aus RTCPeerConnection und RTCDataChannel identifiziert werden.
Nach Herstellung der RTCPeerConnection und dem Empfang des RTCDataChannels werden die ausgelesenen EKG Daten über den DataChannel versendet.
Der EKG Daten Sender ist fähig, mehrere RTCPeerConnections zu Instanzen der Web-Anwendung parallel zu halten und über deren RTCDataChannels die aktuellen EKG Daten zu senden. Beim Versenden der EKG Daten wird die Map mit den RTCDataChannels durchlaufen, um über deren Send Funktion die EKG Daten zu versenden.
Im Kapitel 4.1 wurde beschrieben, dass die Label der RTCDataChannel nach WebSocket Session ID benannt sind. Nachfolgend wir erklärt, warum dies nötig ist.

Beim Beenden des RTCDataChannel wird eine entsprechende Callback Funktion des RTCDataChannel durch ein Event aufgerufen. Beim Aufruf der Callback Funktion, gibt es nur einen Bezug zum RTCDataChannel, nicht aber zu dessen RTCPeerConnection.
Da die WebSocket Session ID im Label das RTCDataChannel steht, ermöglicht dies die zuvor in Maps gespeicherten RTCDataChannel und RTCPeerConnection aus den Maps zu entfernen, da als Key die entsprechende WebSocket Session ID genutzt wurde.
Werden alle RTCDataChannel zu den Instanzen der Web-Anwendung getrennt, stoppt der EKG Daten Sender das Versenden von EKG Daten. Durch die weiter bestehende WebSocket-Verbindung zum Peer Endpoint, bleibt der EKG Daten Sender weiterhin erreichbar.

# 5 Realisierung

In diesem Kapitel wird die Realisierung des WebRTC basiertes Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung beschrieben. Die Beschreibung der Realisierung bezieht sich auf die Anforderungen aus Kapitel 3.3 und die Umsetzung des Designs aus Kapitel 4.

## 5.1 WebSocket-Server in Java

Der WebSocket-Server ist in Java implementiert und wird auf einem GlassFish 4 Server gehostet. Java und der GlassFish Server können kostenfrei plattformübergreifend eingesetzt werden und erfüllen damit die Anforderung 30 und Anforderung 31.

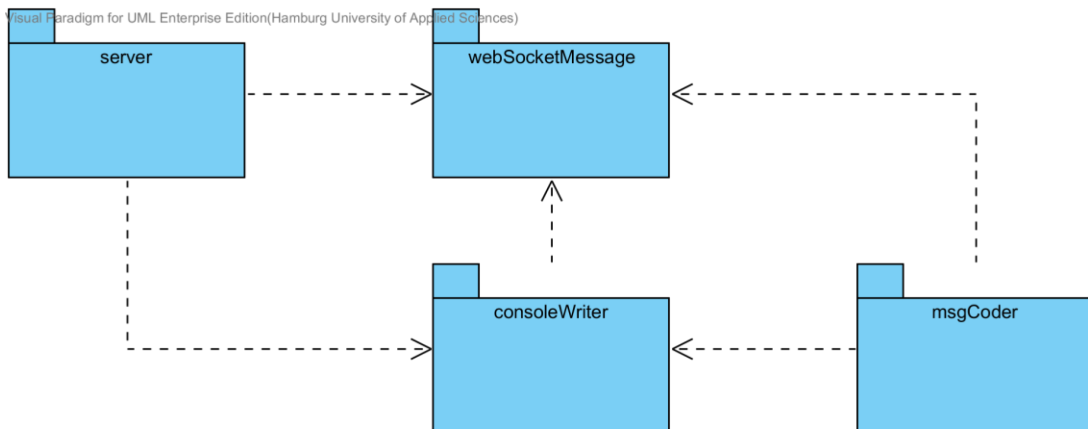



Abbildung 5.1-30.: Package Diagramm des WebSocket Server

In den folgenden Kapiteln wird die Implementierung der in Abbildung 5.1-30 dargestellten Packages, welche den Hauptkomponenten der Implementierung entsprechen, vorgestellt und die Entscheidungen in der Entwicklung begründet.
Genutzt wird die Java Referenz Implementierung, die als JSR 356 [JSR356] im Rahmen von Java EE 7 veröffentlicht wurde und ihren Ursprung im Open Source Projekt Project Tyrus [Tyrus] hat.
Bei der Entwicklung dieser Implementation der WebSocket Spezifikation [RFC6455] für Java wurden zwei Modelle definiert, die Entwickler nutzen können um WebSockets in Java

Anwendungen zu verwenden. Die Java EE 7 API für WebSockets bietet ein Interface- und ein annotationsgetriebenes Modell für die Entwicklung von WebSocket Endpoints an, welche auch Interface-driven/Programmatic Endpoint und Annotated Endpoints bezeichnet werden. Die Endpoints des WebSocket-Servers dieser Bachelorarbeit sind mit einem annotationsgetriebenen bzw. Annotated Endpoint Modell realisiert worden, weil dies der WebSocket JavaScript API [WSAPI] im Aufbau ähnlich ist.

Annotated Endpoints sind POJO's [POJO], die weder ein Interface implementieren noch eine Erweiterung einer anderen Klasse darstellen und somit eine niedrige Kopplung und eine hohe Kohäsion aufweisen. In Abbildung 5.1-31 sind die Endpoint Klassen des Servers dargestellt, die keine Assoziationen zu anderen Klassen oder Interfaces aufweisen.

Der Aufbau eines Annotated Endpoint ist überschaubar. Die Annotation selbst beschreibt die Art des Endpoint und damit, ob es sich um einen Server- oder einen Client Endpoint handelt.

Die Annotation beinhaltet auch Deklarationen von Eigenschaften, die für den Endpoint gelten. In den Eigenschaften muss der URI Pfad angegeben werden, der angibt, wo der Endpoint veröffentlicht wird und unter welcher URI er erreichbar ist. Optional können Encoder- und Decoder Klassen angegeben werden, die genutzt werden um Messages beim Empfang zu dekodieren und beim Versand zu kodieren. Die optionale Angabe von unterstützen Subprotokollen ist möglich sowie die optionale Angabe einer Konfigurator Klasse für Endpoints.

In einer solchen Endpoint Konfigurator Klasse, können unter anderem Eigenkreationen für Algorithmen, der Zugriff auf den WebSocket Handshake und unterstütze Erweiterungen für den Endpoint implementiert werden.

Neben den Endpoint Klassen und der Endpoint Konfigurator Klasse gibt es auch eine Konfigurator Klasse für den Server. In dieser werden die Endpoints eines WebSocket-Servers angegeben, die vom Server veröffentlicht und angeboten werden. In Abbildung 5.1-31 ist die Endpoint Konfigurator Klasse mit ServerEndpointConfigurator und die Server Konfigurator Klasse mit ServerAppConfig benannt.

Nach der Annotation folgt ein üblicher Klassenrumpf, der bei einem Annotated Endpoint und nur einen Konstruktor ohne Argumente beinhalten darf. Anschließend werden die eigentlichen WebSocket Methoden aufgeführt, die jeweils auch eine Annotation besitzen, welche auch Lifecycle Annotations genannt werden [JavaEE7Tut]. Davon gibt es vier Stück, die jeweils einem Event zugeordnet sind. Tritt ein Event auf, wird die entsprechend annotierte Methode aufgerufen. In Abbildung 5.1-32 sind die vier Annotationen aufgezeigt.

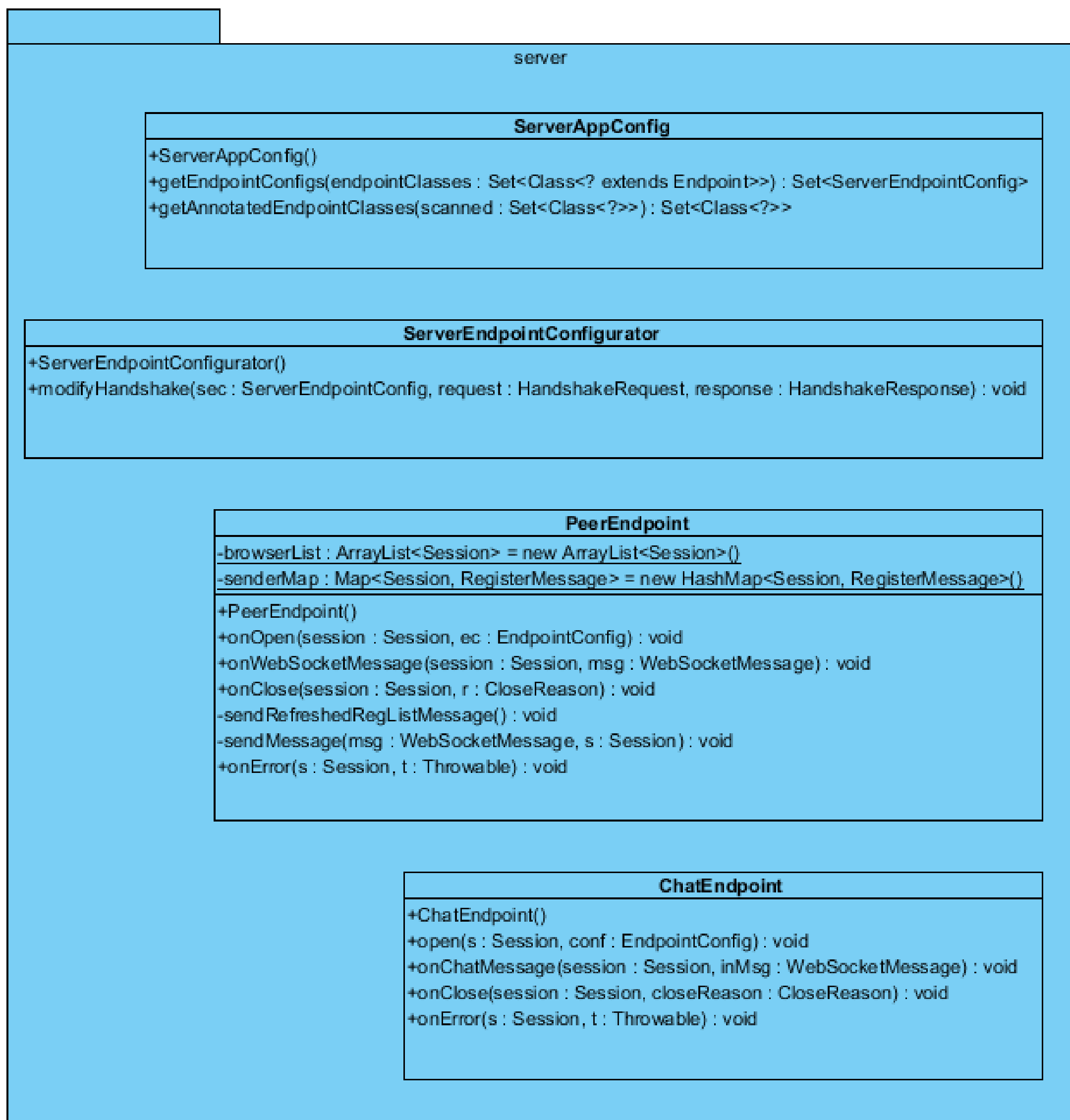



**Abbildung 5.1-31.:WebSocket Server Klassen - POJO Endpoints**

Um Messages zu versenden werden zwei Arten von Aufrufen angeboten. Ein synchroner und damit blockierender Aufruf sowie ein asynchroner und damit nicht blockierender Aufruf.
In dieser Lösung wird der asynchrone nicht blockierende Aufruf genutzt, bei dem nicht der aufrufende Thread die Message versendet, sondern ein anderer Thread genutzt wird um die Message zu versenden. Dies hat den Vorteil dass der Aufruf zum Senden beendet ist bevor die Message tatsächlich versendet wurde. Der Aufruf zum Senden einer sehr langen Message dauert daher genau so lange wie der Aufruf zum Senden einer sehr kurzen Message.

| Annotation | Event | Example |
| --- | --- | --- |
| OnOpen | Connection opened | `@OnOpen`<br>`public void open(Session session, EndpointConfig conf) { }` |
| OnMessage | Message received | `@OnMessage`<br>`public void message(Session session, String msg) { }` |
| OnError | Connection error | `@OnError`<br>`public void error(Session session, Throwable error) { }` |
| OnClose | Connection closed | `@OnClose`<br>`public void close(Session session, CloseReason reason) { }` |

Abbildung 5.1-32.: Java WebSocket Annotationen, aus [JavaEE7Tut]

Beim Verbindungsaufbau einer asynchronen WebSocket-Verbindung können mehrere Attribute bestimmt werden. Dazu gehören die maximale Idle-Zeit und die Zeit die vergehen darf, bis der Versand einer asynchronen Message mit einem Timeout ausläuft.
In dieser Realisierung werden beide Zeiten beim Verbindungsaufbau gesetzt wobei eine Verbindung nach einer Woche Idle-Zeit beendet wird und das Versenden einer asynchronen Message nach 5 Minuten als fehlgeschlagen gilt.
Neben der Unterscheidung zwischen synchronem oder asynchronem Versand stehen für die verschiedenen WebSocket Message Typen entsprechende Methoden bereit. Trotz dessen das in dieser Lösung JSON codierte Nachrichten verschickt werden, wird eine Sende-Methode vom Typ „Object“ genutzt, nicht „Text“. Dies ist darin begründet, dass ein eigenes Message-System genutzt wird und die Objekte der Messages JSON codiert versendet werden. Das Message-System ist eine Menge an Message-Klassen, die jeweils für einen bestimmten Zweck eingesetzt werden. Die verschiedenen Message-Klassen werden in Kapitel 5.1.1 vorgestellt.

### 5.1.1 Message-System

Das entwickelte Message-System beinhaltet die in Abbildung 5.1.1-33 dargestellt Message-Klassen. Die Realisierung orientiert sich stark am angedachten Entwurf, der in Kapitel 4.1.1 beschrieben und in der Abbildung 4.1.1-22 dargestellt ist.
Wie der Abbildung 5.1.1-33 zu entnehmen ist, sind Messages-Klassen für jede Art der Kommunikation vorhanden. Alle Message-Klassen sind von der abstrakten Klasse WebSocketMessage abgeleitet und erben daher das Attribute type, was zur Unterscheidung der folgenden Message Klassen dient.
Für Chatnachrichten gibt die Klasse ChatMessage. Es ist die einfachste Message, da diese nur die Chat Nachricht und den Autor der Chat Nachricht enthält. Eine weitere Unterteilung des Einsatzzwecks ist nicht nötig, da alles Chat Nachrichten und damit auch die Chat Messages gleich sind.
Für das Verteilen der registrierten EKG Daten Sender in Form einer Senderliste, gibt es die „SenderListMessage“. Diese wird seitens des Peer Endpoint genutzt, um an registrierte Instanzen der Web-Anwendung die Senderliste zu verteilen. Die SenderListMessage enthält eine Array an Sender Objekten, welche wiederum Informationen zu den registrierten EKG Daten Sendern enthalten. Ein Objekt der Klasse Sender enthält Attribute für eine ID und eine WebSocket ID welcher der WebSocket Session ID entspricht. Beide Informationen benötigt die Web-Anwendung, um eine Senderliste der EKG Daten Sender darzustellen. Die ID dient zur Identifikation des EKG Daten Sender in der Senderliste durch den Benutzer der Web-Anwendung. Die WebSocket Session ID dient der Web-Anwendung dazu, den Empfänger einer Signaling Message anzugeben, so dass der Peer Endpoint diese an den gewählten EKG Daten Sender weiterleiten kann.
Zum Registrieren am Peer Endpoint nutzen die Instanzen der Web-Anwendung und der EKG Daten Sender die Klasse RegisterMessage. Dessen Einsatz kann anhand eines Enum in drei Arten unterschieden werden. Durch den Einsatz des Enum wird unterschieden, ob sich ein Web-Browser, der eine Instanz der Web-Anwendung ausführt oder ein EKG Daten Sender registriert. Zusätzlich wird diese Message auch genutzt, um den Web-Browser nach der Registrierung über die WebSocket Session ID zu informieren, unter der der Peer Endpoint die WebSocket-Verbindung zum Web-Browser angelegt hat. Bei der Registrierung von EKG Daten Sendern nutzen diese das Attribut ID, um die eigene ID an den Peer Endpoint mitzuteilen. Das Attribut browserWSID dient dem Peer Endpoint dazu, der Web-Anwendung die eigene WebSocket Session ID mitzuteilen.
Zur Realisierung eines Signaling Kanals müssen alle Arten von Signaling-Messages zwischen zwei Peers ausgetauscht werden. Daher bietet die Klasse SignalingMessage eine Unterscheidung der Messages in allen drei Arten an. Signaling Messages können unter Einsatz eines Enum in Offer-, Answer- und ICE Candidate Messages unterschieden werden. Die Message bietet zwei Attribute, die genutzt werden, um WebSocket Session IDs des Absenders und des Empfängers zu transportieren. Zusätzlich gibt es noch das Attribut rtcObject, in dem die SDP der Offer- und Answer Messages transportiert wird.

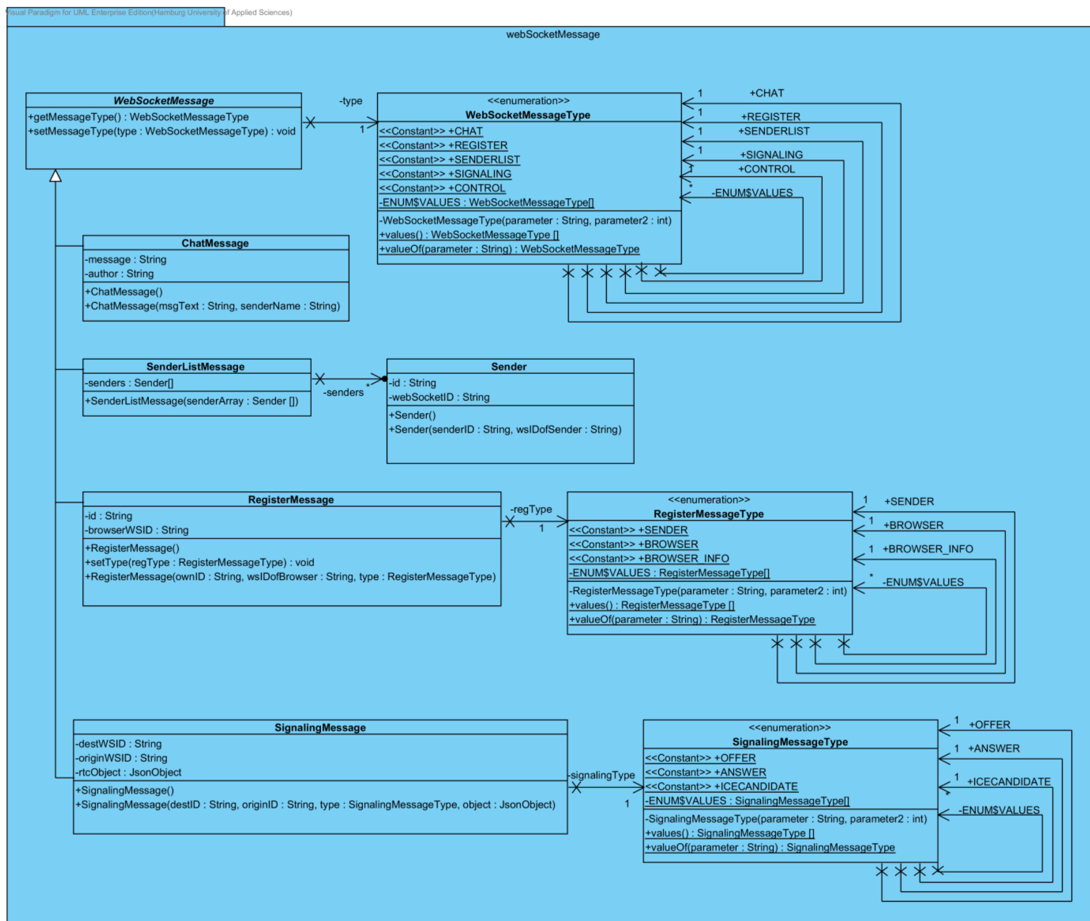


Abbildung 5.1.1-33.: Message-System - Realisierung

## 5.1.2 Encoder

Encoder sind Klassen, die dazu dienen, die in Kapitel 5.1.1 genannten Klassen von Messages für die Übertragung in eine JSON codierte Textform zu bringen. Für jede Message Klasse existiert ein eigenen Encoder.

Wie in Kapitel 5.1 beschrieben, werden Encoder in der Annotation der WebSocket Endpoint Klassen angegeben. Jeder Endpoint benötigt die Angabe der Encoder für die Message Klassen die über den Endpoint versendet werden. So beinhaltet die Annotation der Klasse, die den Chat Endpoint realisiert, wie im Folgenden Code-Abschnitt zu sehen ist, nur einen Encoder für Chat Messages.

```
@ServerEndpoint(value = "/chat",
        encoders = EncoderChatMessage.class,
        decoders = DecoderWebSocketMessage.class,
        configurator = ServerEndpointConfigurator.class
)
public class ChatEndpoint {
```

Der Peer Endpoint versendet mehrere verschiedene Klassen von Messages. Daher sind mehrere Encoder in der Notation angegeben, wie im Folgenden Code-Abschnitt dargestellt ist.

```
@ServerEndpoint(value = "/peer",
        encoders = { EncoderSenderListMessage.class,
        EncoderRegisterMessage.class, EncoderSignalingMessage.class},
        decoders = DecoderWebSocketMessage.class,
        configurator = ServerEndpointConfigurator.class
)
public class PeerEndpoint {
```

Zu sehen sind alle Encoder Klassen mit Ausnahme die der Chat Messages, da diese nicht vom Peer Endpoint, sondern vom Chat Endpoint empfangen und versendet werden.

Durch die Angabe der Encoder in der Annotation verwendet der Endpoint die Encoder automatisch sobald eine Message der entsprechenden Klasse versendet werden soll. Ein manueller Aufruf und die Erzeugung eines entsprechenden Objekts ist nicht nötig.
In Abbildung 5.1.2-34 ist das Package msgCoder mit allen Encodern dargestellt. Der Aufbau einer Encoder Klasse ist immer identisch. Es sind drei Methoden des implementierten Encoder.Text<t> Interfaces mit einer Implementierung zu Überschreiben. Die Methoden destroy und init sind in dieser Realisierung zu vernachlässigen, da diese keine Funktion implementieren. Es wird hierzu lediglich ein Hinweis auf der Konsole ausgegeben. Beim Aufruf der Methode encode wird ein entsprechendes Message Objekt übergeben. Die Methode liefert einen JSON codierten String des übergebenen Message Objekts zurück.

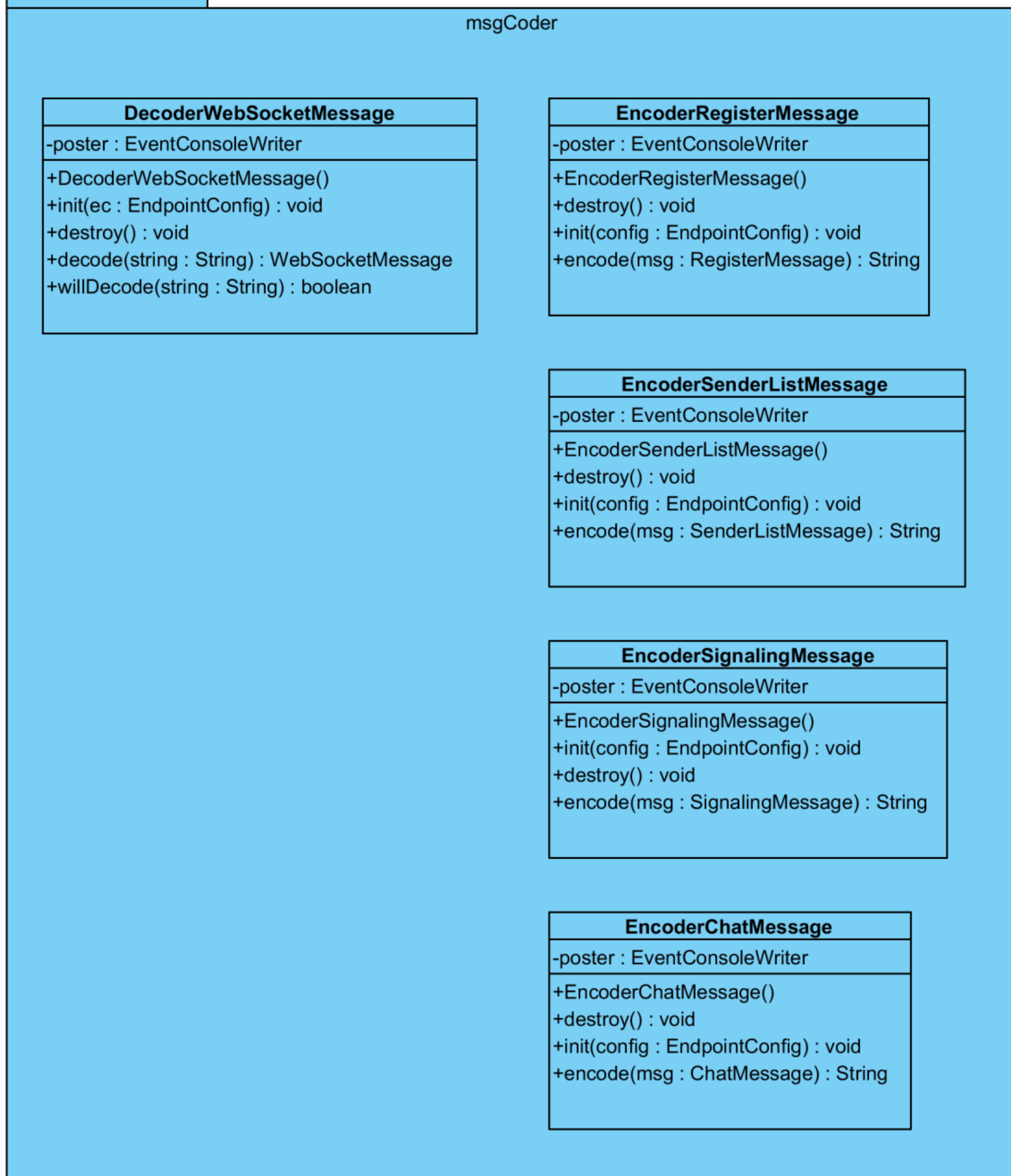



**Abbildung 5.1.2-34.: Encoder/Decoder - Realisierung**

### 5.1.3 Decoder

Im Gegensatz zu den Encodern, von denen es je eine Message Klasse gibt, ist nur ein Decoder für die Message Top Klasse WebSocketMessage implementiert, von der alle Message Klassen abgeleitet sind. Die Decoder Klasse ist auch in Abbildung 5.1.2-34 abgebildet und gehört dem Package msgCoder an.
Der Grund, dass nur eine Decoder Klasse implementiert wurde, ist ein Hinweis im Java EE 7 Tutorial [JavaEE7Tut]. Dieser besagt, dass nur jeweils eine Decoder Klasse in der Annotation angegeben werden darf, die das Interface Decoder.Text<msg> oder Decoder.Binary<msg> implementiert.
Allerdings taucht dieser Hinweis in der Java EE 7 API [JavaEE7API] nicht mehr auf.
Die Decoder Klasse wird wie die Encoder Klassen in der Notation angegeben, was den Endpoint befähigt, entsprechende Messages zu empfangen. Die Angabe des Decoders ist in folgenden Code-Abschnitt zu sehen.

```
@ServerEndpoint(value = "/chat",
        encoders = EncoderChatMessage.class,
        decoders = DecoderWebSocketMessage.class,
        configurator = ServerEndpointConfigurator.class
)
public class ChatEndpoint {
```

Der Decoder verarbeitet den empfangenen JSON String der ein Objekt einer Message darstellt und erzeugt daraus das entsprechende Message Klassen Objekt, welche dann an den Endpoint gegeben wird.

### 5.1.4 Konsolenausgabe

Die Konsolenausgabe des WebSocket-Servers wurde aus einer Kombination aus einer eigenen Implementierung und der Verwendung einer Logger Instanz realisiert. Die eigene Implementierung dient dazu, Strings für die Konsolenausgabe möglichst zentral zu Kapseln und aus dem Quellcode der anderen Klassen auszulagern. Der zentrale Ort für Konsolennachrichten vereinfacht die Pflege, gestaltet die Übersicht der Ausgaben besser, bietet einen zentralen Platz um die Hinweise von Exceptions zu erweitern und ist ein einfacher Weg, um Nachrichten mit möglichst identischen Aufbaus, auf der Konsole auszugeben.
Es wurde dazu die Klasse EventConsoleWriter implementiert, die fast ausschließlich aus vorgefertigten und parametrisierten Textbausteinen besteht, die per Methoden Aufruf von einem StringBuilder Objekt zusammengesetzt und mit Verwendung des Loggers auf der Konsole ausgegeben werden.

Jede andere eigens implementierte Klasse des WebSocket-Servers nutzt die Klasse EventConsoleWriter für die Ausgabe auf der Konsole konsequent, weshalb die Klasse nach einem Singleton Double Looking Pattern entworfen wurde, was verfälschte Ausgaben auf der Konsole verhindert.
Die Logger Instanz wird verwendet, da diese im Gegensatz zum Ausgabestream System.out, die Ausgabe und das Leeren der intern verwendeten Buffer sicher organisiert.
In Abbildung 5.1.4-35 ist ein Beispielauszug der Konsolenausgabe, vom WebSocket Handshake beim Verbindungsaufbau, zu sehen.

```
[#|2015-03-16T19:08:20.278+0100|INFO|glassfish 4.1||_ThreadID=27;_ThreadName=Thread-8;_TimeMillis=1426529300278;_LevelValue=800;|
  ---   ---   ---   ---   ---   ---   ---   ---   ---   ---   ---   ---
ON OPEN Server Message:
        Called in class                 = server.PeerEndpoint
        Handshake Request:
                accept                = [text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8]

                accept-encoding       = [gzip, deflate]

                accept-language       = [de, en-US;q=0.7, en;q=0.3]

                cache-control         = [no-cache]

                connection            = [keep-alive, Upgrade]

                host                  = [vybz.asuscomm.com:8080]

                origin                = [http://vybz.asuscomm.com]

                pragma                = [no-cache]

                sec-websocket-key     = [CIv5EpDz1IfFVxVyL6Uo4A==]

                sec-websocket-version = [13]

                upgrade               = [websocket]

                user-agent            = [Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101 Firefox/36.0]

        Handshake Response:
                Connection           = [Upgrade]

                Sec-WebSocket-Accept = [4VBtj9rWwZLpp41LUJ7vKiwaZAU=]

                Upgrade              = [websocket]

        New Session-conncetion established:
                Session ID                            = 6d5a068c-bc59-4d0d-87af-52b63bcfb124
                Is connection of Session secure(wss)  = NO
                Current number of Open Sessions       = 1
                Connection Max Idle Timeout           = 604800000 ms
                Connection Max Message Buffer Size    = 2048 bytes
                Connection Max Binary Buffer Size     = 0 bytes
                Connection Async Send Timeout         = 300000 ms
---   ---   ---   ---   ---   ---   ---   ---   ---   ---   ---   ---|#]
```

Abbildung 5.1.4-35.: Konsolenausgabe eines EventConsoleWriter Objekts

### 5.1.5 WebSocket Handshake

In Kapitel 5.1 wurde bereits die Verwendung des ServerEndpointConfigurator erwähnt, welcher Bestandteil des Package Server ist, das in Abbildung 5.1-31 zu sehen ist. Das Objekt der Klasse ServerEndpointConfigurator wird genutzt, wie in Java EE 7 Tutorial [JavaEE7Tut] beschrieben, um den Inhalt des WebSocket Handshake beim Verbindungsaufbau auszulesen.
Ein Beispiel für den Inhalt des WebSocket Handshake ist in Abbildung 5.1.4-35 dargestellt, in der ein erfolgreicher Verbindungsaufbau zum Peer Endpoint zu sehen ist und Einblick auf alle Header-Felder des browserseitigen HTTP Request und serverseitigen HTTP Response gegeben wird.

### 5.1.6 Chat Endpoint

Der Chat Endpoint ist eine der simpelsten Varianten einer Annotated Endpoint Implementierung, womit er sich gut als Anschauungsmaterial eignet. Im folgenden Code-Abschnitt sind alle Methoden, ausgenommen der zum Empfangen von Pong Messages, mit einer WebSocket Annotation abgebildet.

```
@OnOpen
public void open(Session s, EndpointConfig conf) {
        s.setMaxBinaryMessageBufferSize(0);
        s.setMaxTextMessageBufferSize(32768);
        s.getContainer().setAsyncSendTimeout(fiveMin);
        s.setMaxIdleTimeout(oneWeek);
        sendPing(s);
        poster.postOnOpen(s, conf);
}

@OnMessage
public void onChatMessage(Session session, WebSocketMessage inMsg) {
        poster.postOnMessage((ChatMessage) inMsg, session);
        ChatMessage outMsg = new ChatMessage(((ChatMessage)
        inMsg).getMessage() , ((ChatMessage) inMsg).getAuthor());
        for (Session sess : session.getOpenSessions()) {
                if (sess.getId() != session.getId) {
                        poster.postOnSend(outMsg, sess);
                        try {
                                sess.getAsyncRemote().sendObject(outMsg);
                        } catch (IllegalArgumentException e) {
                                poster.postException("sendObject()",
                                ExceptionType.ILLEGALARGUMENT, e);
                        } catch (IllegalStateException e) {
                                poster.postException("getAsyncRemote()",
                                ExceptionType.ILLEGALSTATE, e);
                        }
                }
        }
}

@OnClose
public void onClose(Session session, CloseReason closeReason) {
        poster.postOnClose(session, closeReason);
}

@OnError
public void onError(Session s, Throwable t) {
        poster.postOnError(s, t);
}
```

Die Methode „open“ ist mit der Annotation @OnOpen versehen. Daher wird die Methode jedes Mal aufgerufen, wenn eine WebSocket-Verbindung zu Endpoint Chat erfolgreich hergestellt wird. Es werden die Buffergrößen für WebSocket Messages vom Typ „Text“ und „Binary“ gesetzt, wobei letzterer auf 0 gesetzt wird, da keine Messages vom Typ „Binary“ genutzt werden. Im Anschluss werden die Idle Zeit der Session und die maximale Übertragungsdauer einer Message festgelegt. Der Aufruf der sendPing() Methode dient dem Senden eines Ping Control Frames, welcher vom Web-Browser mit einem Pong Control Frame beantwortet wird. In einer nicht dargestellten Methode wird auf empfangene Pong Messages, mit zeitverzögerten Senden einer neuen Ping Message reagiert, womit ein Idle Timeout der WebSocket-Verbindung verhindern soll, wenn der Chat nicht genutzt wird. Das gleiche Verfahren wird auch vom Peer Endpoint genutzt. Abschließend wird eine parametrisierte Methode für die Konsolenausgabe von neuen Verbindungen aufgerufen, bei der das Session Objekt der neuen Verbindung und ein EndpointConfig Objekt übergeben werden. Dies ermöglicht eine Ausgabe auf der Konsole, wie sie in Abbildung 5.1.4-35 dargestellt ist.

Die Annotation @OnMessage ziert die Methode onChatMessage und wird dementsprechend immer dann aufgerufen, wenn eine WebSocketMessage beim Chat Endpoint eintrifft. Die Methode erhält vom Decoder ein WebSocketMessage Objekt, da der Decoder nur den Typ WebSocketMessage dekodiert und der Typ entsprechend angegeben werden muss. Es handelt sich bei dem Objekt allerdings um den abgeleiteten Typ ChatMessage, zu der als erstes eine Konsolenausgabe stattfindet.

Anschließend wird die empfangene ChatMessage direkt an alle anderen Sessions des Chat Endpoint weitergeleitet, womit Anforderung 16 erfüllt wird. Die Sessions entsprechen dabei den WebSocket-Verbindungen zu Instanzen der Web-Anwendung. Als Erstes wird dazu ein neues ChatMessage Objekt erzeugt, wobei die Attribute aus der empfangenen ChatMessage kopiert werden. Für den Versand wird ein Set, das alle Sessions des Chat Endpoint beinhaltet, durchlaufen und geprüft, dass die Session nicht dem Absender entspricht. Trifft dies für eine Session zu, wird das zuvor erzeugte ChatMessage Objekt an die entsprechende Session versendet und die nächste Session des Sets geprüft. Der Versand wird mit einem asynchronen Aufruf umgesetzt, wobei die Message nur an die unterliegende Verbindung weitergegeben wird, ohne auf dessen abgeschlossenen Versand zu warten. Zusätzlich ist zu beachten, dass zum Versenden eine Methode genutzt wird, die den Typ Object verschickt, nicht „Text“. Beim Versenden wird ein Encoder für den Typ ChatMessage genutzt. Erst der Encoder kodiert das ChatMessage Objekt in einen JSON String der dann in einer WebSocket-Message vom Typ „Text“ verschickt wird.

### 5.1.7 Peer Endpoint

Der Peer Endpoint hat verschiedene Aufgaben, wie bereits im Kapitel 4.1.2 erklärt wurde. Er ist dafür zuständig die Registrierungen von EKG Daten Sendern und Web-Anwendungen entgegenzunehmen, so dass alle registrierten Instanzen der Web-Anwendung eine aktuelle

Senderliste führen und dient zwischen zwei Peers als Signaling Kanal. Der Peer Endpoint kann daher mit mehreren Typen von Messages umgehen und reagiert entsprechend unterschiedlich. Der Umgang auf die verschiedenen Message Typen werden im Folgenden beschrieben, womit auch die Bewältigung der oben genannten Aufgaben veranschaulicht wird.

**Registrierung & Senderliste**

Der folgende Code-Abschnitt zeigt die Reaktion und den Umgang des Peer Endpoint auf den Eingang von RegisterMessages.

```
@OnMessage
public void onWebSocketMessage(Session session, WebSocketMessage msg){
	/* A sender or a browser will register */
	if (msg.getMessageType() == WebSocketMessageType.REGISTER) {
		poster.postOnMessage((RegisterMessage) msg, session);
		/* if the register message of type SENDER then put the
		* session in senderMap
		*/
		if (((RegisterMessage) msg).getRegType() ==
		RegisterMessageType.SENDER) {

			senderMap.put(session, (RegisterMessage) msg);
		/* if the register message of type BROWSER then add the
		 * session in browserList
		 */
		} else if (((RegisterMessage) msg).getRegType() ==
		RegisterMessageType.BROWSER) {
			browserList.add(session);

			RegisterMessage browserInfoMsg = new
			RegisterMessage("", session.getId(),
			RegisterMessageType.BROWSER_INFO);

			sendMessage(browserInfoMsg, session);
		} else {
			poster.postOnError(session, new Exception("Recv unknown
			Register Message"));
		}
		/*
		 * check to send a refreshed sender list
		 */
		if ((browserList.size() != 0) && (senderMap.size() != 0)) {
			sendRefreshedSenderListMessage();
		}
}
```

Nach Empfang einer WebSocketMessage wird deren Attribut „MessageType“ überprüft, um zu ermitteln, um was für eine WebSocketMessage es sich handelt. Handelt es sich um eine RegisterMessage wird eine Konsolenausgabe ausgelöst, die den Inhalt der Message enthält. Anschließend wird das Attribut „RegTyp“ der RegisterMessage überprüft, da dieses Attribut aussagt, ob ein EKG Daten Sender oder ein Web-Browser die RegisterMessage geschickt hat, worauf unterschiedlich reagiert werden muss. Handelt es sich um die RegisterMessage eines EKG Daten Senders, wird dessen RegisterMessage in einer Map gespeichert. Als Key wird die WebSocket Session ID der WebSocket-Verbindung genutzt, da diese nicht doppelt vorkommen kann.
Wurde eine RegisterMessage eines Web-Browsers empfangen, in dem eine Instanz der Web-Anwendung läuft, dann wird die WebSocket Session ID der WebSocket -Verbindung in einer Liste gespeichert, in der nur die Session IDs von registrierten Web-Browsern gespeichert werden. Im Anschluss wird dem Browser mit einer RegisterMessage, die seine WebSocket Session ID enthält, geantwortet.
Nachdem die Verarbeitung der RegisterMessage abgeschlossen ist, wird geprüft ob sich bereits mindestens ein EKG Daten Sender und ein Web-Browser registriert haben. Denn dann muss nach Empfang einer RegisterMessage eine aktuelle Senderliste an die registrierten Web-Browser gesendet werden, da sich entweder ein neuer EKG Daten Sender registriert hat oder ein Web-Browser, dem noch keine Senderliste zugesendet wurde.
Auch wenn eine Verbindung zu einem Web-Browser oder EKG Daten Sender getrennt wird, wird geprüft, ob eine aktuelle Senderliste an die Web-Browser gesendet werden muss, wie im folgenden Code-Abschnitt zu sehen ist.

```
@OnClose
public void onClose(Session session, CloseReason r) {
	poster.postOnClose(session, r);

	if (senderMap.containsKey(session)) {
		senderMap.remove(session);
		/*
		 * check if a browser there that need a refreshed senderList
		 */
		if (browserList.size() != 0) {
			sendRefreshedSenderListMessage();
		}

	} else if (browserList.contains(session)) {
		browserList.remove(session);
	} else {
		poster.postOnError(session, new Exception("Recv OnClose from
		an unknown Session"));
	}
}
```

Nach dem Aufruf der Methode mit der @OnClose Annotation, die beim Verbindungsabbau aufgerufen wird, wird geprüft, ob in der Map, in der die RegisterMessages der EKG Daten Sender gespeichert werden, die WebSocket Session ID enthalten ist, die den Aufruf der Methode ausgelöst hat.
Ist die WebSocket Session ID in der Map enthalten, wird der entsprechende Eintrag aus der Map gelöscht. Anschließend wird geprüft, ob noch weitere Web-Browser registriert sind, welchen dann eine aktuelle Senderliste zugesandt wird.
Ist die WebSocket Session ID nicht in der Map in der die RegisterMessages der EKG Daten Sender gespeichert werden, wird geprüft ob die WebSocket Session ID in der Liste ist, in dem die WebSocket Session IDs der Web-Browser gespeichert werden. Ist die WebSocket Session ID enthalten wird diese aus der Liste gelöscht.
Ist die WebSocket Session ID, die die OnClose Message gesendet hat, weder in der Liste der Web-Browser Session IDs noch in der Map der EKG Daten Sender, wird ein Exception geworfen und auf der Konsole ausgegeben.
Der Peer Endpoint führt somit immer eine Liste, in der die aktuell verbundenen Web-Browser geführt werden, sowie eine Map, in der die aktuell verbundenen EKG Daten Sender enthalten sind. Dies ist nötig um jederzeit eine aktuelle Senderliste erstellen zu können, die alle verbundenen EKG Daten Sender enthält, welche an alle registrierten Web-Browser gesendet wird, somit Anforderung 20 erfüllt ist. Im folgenden Abschnitt wird gezeigt, wie die Senderliste erstellt wird.

```
private void sendRefreshedSenderListMessage() {
	poster.postServerInfo("sendRefreshedRegListMessage");
	List<Sender> regSenderList = new ArrayList<Sender>();

	Set<Session> sessions = senderMap.keySet();
	for (Session s : sessions) {
		Sender registeredSender = new Sender(senderMap.get(s).getId()
		, s.getId());

		regSenderList.add(registeredSender);
	}

	SenderListMessage outMsg = new
	SenderListMessage(regSenderList.toArray(new Sender[0]));

	/*
	 * send a regSenderlist message to each registered browsers, all
	 * registered sender in the regSenderlist
	 */
	for (Session sess : browserList) {
		sendMessage(outMsg, sess);
	}
}
```

Um eine Senderliste zu erstellen, die an alle registrierten Instanzen der Web-Anwendung gesendet wird, wird als erstes eine ArrayList erstellt, die Objekte vom Typ Sender speichern kann. Anschließend wir die Map, in der die RegisterMessages der verbundenen EKG Daten Sender enthalten sind, durchlaufen. Dabei wird für jeden Eintrag der Map ein neues Objekt vom Typ Sender erzeugt, um darin die entsprechende Session ID und die ID des jeweiligen EKG Daten Sender zu speichern. Alle erzeugten Sender Objekte werden der zu Beginn erstellten ArrayList hinzugefügt, die nun eine Liste mit allen verbunden EKG Daten Sendern darstellt. Nun kann eine SenderListMessage erzeugt werden, die den Inhalt der ArrayList in Form eines Array aufnimmt. Abschließend wird die List durchlaufen, in der die Session IDs verbundener Web-Browser enthalten sind, um diesen die SenderListMessage zuzusenden.

**Signaling Kanal**

Um zwischen zwei WebSocket Clients bzw. WebRTC Peers einen Signaling Kanal zu realisieren, bedarf es nur der korrekten Weiterleitung von SignalingMessages. Beide WebSocket Clients sind mit dem Peer Endpoint verbunden, können aber noch nicht direkt miteinander kommunizieren. Daher ist es nach Empfang einer SignalingMessage nötig, diese korrekt weiterzuleiten und den Empfänger über den Absender zu informieren.
Wie die Weiterleitung der SignalingMessages umgesetzt wird und die Benachrichtigung des Empfängers über den Absender erfolgt, ist in folgendem Abschnitt zu sehen.

```
else if (msg.getMessageType() == WebSocketMessageType.SIGNALING) {
	poster.postOnMessage(msg, session);
	/*
	* set the session ID of the origin, so that destination peer
	* knows the origin
	*/
	((SignalingMessage) msg).setOriginWSID(session.getId());
	/* try to push the signaling message to destination */
	for (Session sess : session.getOpenSessions()) {
		if(sess.getId().equals(((SignalingMessage)msg).getDestWSID())
		){
			sendMessage(msg, sess);
		}
	}
}
```

Nach dem Empfang einer SignalingMessage wird eine Methode für eine entsprechende Konsolenausgabe aufgerufen. Danach wird die Session ID des Absenders in die SignalingMessage eingetragen, so dass der Empfänger der SignalingMessage erfährt, unter welcher WebSocket Session ID der Absender dem Peer Endpoint bekannt ist. Abschließend werden alle WebSocket Sessions des Peer Endpoint durchlaufen und geprüft, ob die Empfänger WebSocket Session ID unter ihnen ist. Ist dass der Fall wird die SignalingMessage an diese Session weitergesendet.

## 5.2 Web-Seite

Die entwickelte Web-Seite wird als bedienbare Benutzeroberfläche der Web-Anwendung genutzt, welche sich aus der Web-Seite und dem WebSocket-Server zusammensetzt. Die Web-Anwendung bietet dem Benutzer zum beobachten von EKG Daten ein Management-System, in Form einer EKG Kurve. Die EKG Daten werden von einem entfernten EKG Daten Sender über eine verschlüsselte Peer-to-Peer Verbindungen übertragen, nachdem der Benutzer einen EKG Daten Sender aus der Senderliste ausgewählt hat. Die realisierte Web-Anwendung erfüllt damit Anforderung 22. Gehostet wird die Web-Seite, wie der WebSocket-Server, auf dem eingesetzten GlassFish 4 Server. Die Web-Seite ist so über einen Web-Browser erreichbar und bedienbar, womit Anforderung 1 und Anforderung 2 erfüllt werden.
Die Web-Seite ist in HTML geschrieben und bedient sich an einem minimalen Anteil CSS für die visuelle Gestaltung. Für die Nutzung der Web-API‘s von WebRTC und WebSocket, wie auch für den Einsatz eines Buffers und die JSON Codierung von Messages, wird JavaScript verwendet.
Die Unterteilung der Web-Seite in verschieden Bereiche erfolgt mithilfe von Frames und Framesets, die nicht mehr von HTML5 unterstützt werden. Es handelt sich um fünf Bereiche, die eine Senderliste, einen Chat, eine Konsole, ein Infofenster und einen Bereich zur Darstellung der EKG Daten in Kurven Form bieten, womit Anforderung 23 erfüllt wurde. Auf Abbildung 5.2-36 sind die farblich unterschiedlich gestalteten Frames zu sehen, welche in den folgenden Kapiteln genauer beschrieben werden.

Abbildung 5.2-36.: Screenshot der Web-Seite

### 5.2.1 Senderliste

Die Senderliste nimmt einen Bereich über die gesamte linke Seite der Web-Seite ein, um immer für den Benutzer sichtbar und erreichbar zu sein, womit Anforderung 7 erfüllt wird. In Abbildung 5.2-36 ist die Senderliste auf der linken Seite der Web-Seite zu sehen. In der Senderliste werden registrierter EKG Daten Sender, mit Hilfe einer Liste aus Buttons dargestellt, wie es Anforderung 18 vorgibt. Die Buttons sind mit der ID des jeweiligen EKG Daten Sender beschriftet, womit die Identifizierung der EKG Daten Sender realisiert und Anforderung 4 erfüllt wird.
Der Frame, in dem die Senderliste dargestellt wird, baut eine WebSocket-Verbindung zum Peer Endpoint auf und registriert sich anschließend bei diesem mit einer entsprechenden WebSocket Message. Ein Label zeigt den Verbindungsstatus zum Peer Endpoint an. Dessen Beschriftung wechselt entsprechend des Verbindungsstatus und zeigt ein rötlichen Disconnected oder grünlichen Connected Schriftzug an. Sendet der Peer Endpoint eine Liste von registrierten EKG Daten Sendern, wie in Kapitel 5.1.7 beschrieben ist, wird jeder Listeneintrag in Form eines Buttons in der Senderliste, wie in Abbildung 5.2.1-37 zu sehen ist, dargestellt.

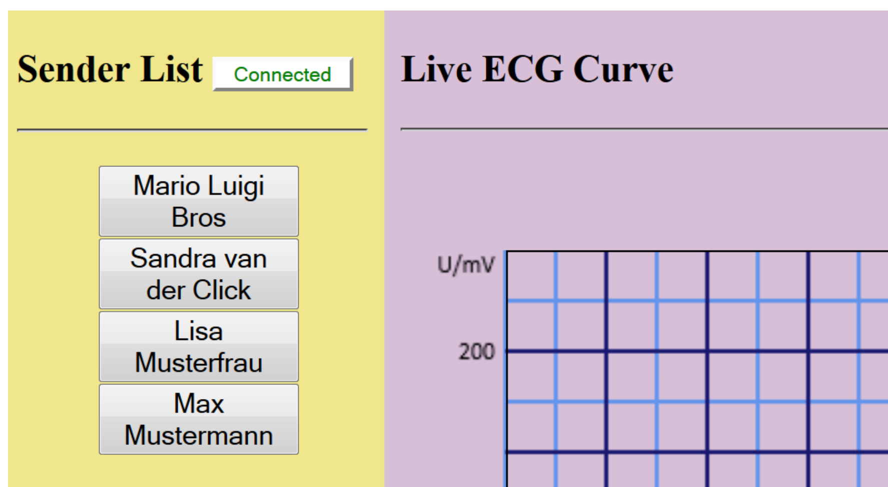


**Abbildung 5.2.1-37.: Senderliste - Einträge**

Benutzer können durch einen Klick auf einen Button die EKG Kurve des EKG Daten Sender abrufen. Durch den Klick auf einen Button wird die Funktion disconnectDataChannel() aufgerufen, um eine eventuell bestehende Verbindung zu einem EKG Daten Sender abzubauen. Anschließend wird die Funktion generateOffer() aufgerufen, womit der Verbindungsaufbau initiiert wird. Dies ermöglicht den Wechsel zwischen den EKG Daten Sendern mit sauberem Verbindungsabbau, wie in Anforderung 6 gefordert. Beim Aufruf der Funktion generateOffer() werden neben der WebSocket Session ID und der ID des gewählten EKG Daten Senders auch die WebSocket Session ID des Senderlisten Frames an die Funktion übergeben. Die Funktion generateOffer() initiiert den Aufbau einer

RTCPeerConnection zum gewählten EKG Daten Sender und ist Bestandteil des Frames, in dem die EKG Kurve dargestellt wird.

### 5.2.2 EKG Kurve

Der Frame der Web-Seite, in dem die EKG Kurve dargestellt wird, ist in Abbildung 5.2-36 violett Dargestellt. Der Frame baut keine eigene WebSocket-Verbindung auf um Signaling Messages zu senden und zu empfangen, sondern nutzt die WebSocket-Verbindung, die der Frame der Senderliste hergestellt hat.
Besteht eine Verbindung zu einem EKG Daten Sender, werden die empfangenen EKG Daten in Form einer EKG Kurve dargestellt, die als rote Linie auf ein im Hintergrund liegendes Gitternetz gezeichnet wird, womit Anforderung 5 erfüllt ist. Darunter ist ein Button mit der Bezeichnung Disconnect dargestellt, der es dem Benutzer ermöglicht die Darstellung der EKG Kurve zu beenden, wie es Anforderung 11 vorsieht, in dem die Peer-to-Peer Verbindung zum sendenden EKG Daten Sender abgebaut wird. Abbildung 5.2.2-38 zeigt die Darstellung von EKG Daten in Form einer EKG Kurve und den Disconnect Button.

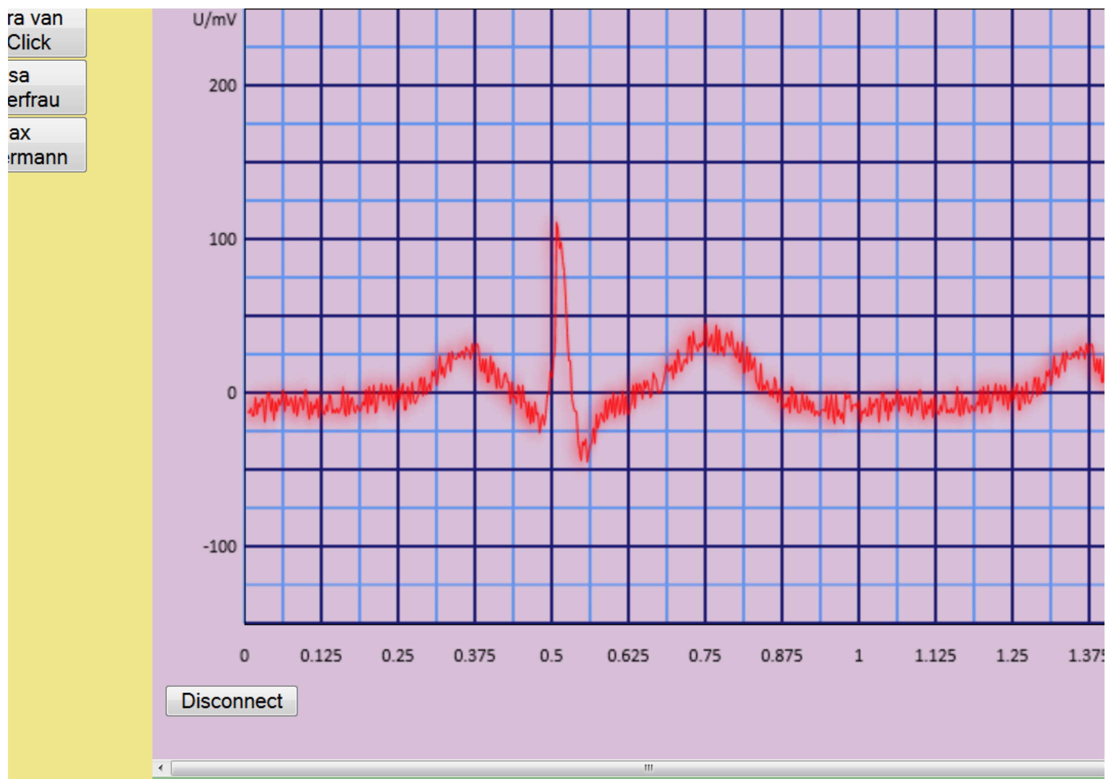



Abbildung 5.2.2-38.: Darstellung von EKG Kurve und Disconnect Button

In den folgenden Abschnitten werden die Hauptfunktionen des Frames beschrieben, die aus dem Aufbau der RTCPeerConnection, der Regulierung eines Buffer für EKG Daten und dem Zeichnen der EKG Daten in Form einer Kurve entsprechen.

**Initiieren der Peer-to-Peer Verbindung durchs Senden einer Offer Message**

Wie in Kapitel 5.2.1 genannt, wird durch einen Klick auf einen Button der Senderliste die Funktion generateOffer() aufgerufen, die den Verbindungsaufbau einer RTCPeerConnection zum EKG Daten Sender initiiert.
Der folgende Code-Abschnitt zeigt den rein funktionalen Code der generateOffer() Funktion.

```
function generateOffer(ownWSID, remotePeerWSID, NameOfRemotePeer) {
	remotePeerName = NameOfRemotePeer;
	parent.statsFrame.setRemotePeerName(NameOfRemotePeer);

	peerConnection = new RTCPeerConnection(iceConfig);

	if (!dataChannel) {
		dataChannel = peerConnection.createDataChannel(
		ownWSID,dataChannelOptions);
		peerConnection.createOffer(function(description) {
			peerConnection.setLocalDescription(description,
			function() {
				sendOffer(remotePeerWSID,description);
			}, onError);
		}, onError);

		dataChannelHandle(dataChannel);
		peerConnection.ondatachannel = dataChannelHandle;
	}

	peerConnection.onicecandidate = function(event) {
		if (!event.candidate) {
			return;
		} else {
			var candidate = event.candidate;
			sendIceCandidate(remotePeerWSID, candidate);
		}
	}
}
```

Beim Aufruf der Funktion generateOffer() werden drei Parameter übergeben.
Der erste Parameter mit dem Namen ownWSID, beinhaltet die WebSocket Session ID unter der dem Peer Endpoint die WebSocket-Verbindung dieser Web-Anwendungsinstanz

bekannt ist. Die WebSocket Session ID der Web-Anwendungsinstanz wird beim Erzeugen des RTCDataChannel genutzt, um die Bezeichnung des RTCDataChannel Labels zu setzen.
Der zweite Parameter trägt den Namen remotePeerWSID. Es handelt sich um die WebSocket Session ID des EKG Daten Senders, welcher zuvor in der Senderliste angeklickt wurde. Die WebSocket Session ID des EKG Daten Senders wird beim Versenden der Offer Message genutzt wird, um dem Peer Endpoint das Ziel der Offer Message mitzuteilen, an welches der Peer Endpoint die Offer Message weiterleitet.
NameOfRemotePeer ist der dritte Parameter, welcher die ID des EKG Daten Senders beinhaltet und der Beschriftung des angeklickten Senderlisten Buttons entspricht. Genutzt wird die ID für Konsolenausgaben und zur Ausgabe des aktuell verbundenen EKG Daten Senders im „statsFrame“. Da die Konsolenausgaben auch außerhalb dieser Funktion ausgeführt werden, wird der Parameter in der globalen Variable „remotePeerName“ gespeichert.
Nach dem Speichern des Funktionsparameter NameOfRemotePeer in einer globalen Variablen und einem Funktionsaufruf, der die ID des EKG Daten Senders in einem anderen Frame anzeigt, wird ein RTCPeerConnection Objekt erzeugt. Der Parameter iceConfig enthält Angaben von STUN- und TURN Servern, die vom ICE Agent des erzeugten RTCPeerConnection Objekts für die NAT Traverse genutzt werden. Die Durchquerung von NATs ist nötig, um einen RTCPeerConnection zu einem entfernten EKG Daten Sender, der in einem anderen Netzwerk betrieben wird, herzustellen und damit Anforderung 25 zu erfüllen.
Nachdem ein RTCPeerConnection Objekt erzeugt wurde, wird ein RTCDataChannel erstellt. Dabei werden die zwei Parameter ownWSID und dataChannelOptions angegeben. Der Parameter ownWSID beschreibt das Label des RTCDataChannel. Die Bezeichnungen der RTCDataChannel Labels, die von einer Instanz der Web-Anwendung gesetzt werden, sind immer identisch, bis die Instanz eine neue WebSocket-Verbindung zum Peer Endpoint aufbaut. Der Parameter dataChannelOptions beinhaltet die Konfiguration des RTCDataChannels, wie in Kapitel 2.2.8 beschrieben. Das RTCDataChannel Objekt wird so konfiguriert, dass es partiell unzuverlässig und in geordneter Reihenfolge sendet. Partielle Unzuverlässigkeit bedeutet in diesem Fall, das nur innerhalb von 200ms ein Wiederversenden stattfindet. Die partielle Unzuverlässigkeit ist nötig, um den Datenempfang für einen bestimmten Zeitraum gewährleisten zu können und so der weichen Echtzeit nachzukommen, um Anforderung 28 zu erfüllen. Anschließend wird mit dem Aufruf der Funktion createOffer() eine lokale SDP erzeugt und dem RTCPeerConnection Objekt mit Aufruf der Funktion setLocalDescription() übergeben. Ist die Übergabe erfolgreich, wird die Funktion sendOffer() aufgerufen, die eine Offer Message über den Signaling Kanal an den EKG Daten Sender versendet.
In Abbildung 5.2.2-39 ist das SDP zu sehen, wie es in einer Offer Message an einen EKG Daten Sender geschickt wird. Der Ausdruck „c=IN IP4 0.0.0.0“ gibt an, dass ICE Trickling verwendet wird, was in Kapitel 2.2.5 beschrieben wird. Der Ausdruck „m=application 9 DTLS/SCTP 5000“, gibt die Verwendung eines DTLS verschlüsselten RTCDataChannel, wie es von Anforderung 26 gefordert ist, auf Basis einer SCTP Verbindung an.

```
v=0
o=Mozilla-SIPUA-36.0 21565 0 IN IP4 0.0.0.0
s=SIP Call
t=0 0
a=ice-ufrag:88732baf
a=ice-pwd:c824688991fa72638ba3d7b91eac2d6e
a=fingerprint:sha-256
14:BD:C4:7C:0E:7B:1C:48:6C:A4:84:F5:0A:B2:CE:73:C2:33:74:2B:0B:33:A7:60:F7:90:7B:AB:A6:8
0:DF:7B
m=application 9 DTLS/SCTP 5000
c=IN IP4 0.0.0.0
a=sctpmap:5000 webrtc-datachannel 256
a=setup:actpass
```

**Abbildung 5.2.2-39.: Session Description Protocol der Offer Message**

Das RTCDataChannel Objekt wird anschließend an ein Handle übergeben, welches Funktionen zur Reaktion auf eintreffenden Events des RTCDataChannel Objekts bietet, wie etwa beim Eintreffen von Messages.
Anschließend wir das Handle dem RTCPeerConnection Objekt mit dem Aufruf der Funktion ondatachannel() übergeben.
Abschließend wird eine Callback Funktion eingesetzt, um auf eintreffende RTCPeerConnection Events zu reagieren, die der ICE Agent beim ICE gathering Prozess auslöst. Tritt das Event ein und beinhaltet es einen ICE Candidate, wird als Reaktion eine ICE Candidate Message über den Signaling Kanal versendet, um dem EKG Daten Sender den gefundenen ICE Candidate mitzuteilen. Tritt das Event auf und enthält keinen ICE Candidate bedeutet dies, das der ICE Agent den ICE Gathering Prozesses abgeschlossen hat. Durch den erfolgreichen Abschluss des ICE gathering Prozesses, ist die RTCPeerConnection bereit zur Datenübertragung, was durch das Auftreten eines OnOpen Event des RTCDataChannel komplettiert wird.

**Empfang der Answer Message**

Nach dem Versenden einer Offer Message ist eine Answer Message vom EKG Daten Sender als Reaktion zu erwarten, welches ein SDP enthält. Trifft die Answer Message ein, wird ein neues RTCSessionDescription Objekt erzeugt. Dabei wird das empfangene SDP aus der Answer Message übergeben. Anschließend wird das neu erzeugte RTCSessionDescription Objekt, mit dem Aufruf der Funktion setRemoteDescription(), an das RTCPeerConnection Objekt übergeben, wie der folgende Code-Abschnitt zeigt.

```
function setDescription(description) {
	var rtcSessionDescription = new RTCSessionDescription(description);
	peerConnection.setRemoteDescription(rtcSessionDescription);
}
```

**Empfang einer ICE Candidate Message**

Wird eine Message über den Signaling Kanal, die einen gefundenen ICE Candidate des ICE Agent vom EKG Daten Senders enthält, empfangen, wird mit diesem ein neues RTCIceCandidate Objekt erzeugt und dem RTCPeerConnection Objekt mit Aufruf der Funktion addIceCandidate() übergeben, wie in folgendem Code-Abschnitt zu sehen ist.

```
function addIceCandidate(candidate) {
	peerConnection.addIceCandidate(new RTCIceCandidate(
		{
			sdpMLineIndex : candidate.sdpMLineIndex,
			candidate : candidate.candidate
		})
	);
}
```

**Regulierung des Buffer Füllstandes**

Nach dem mit Einsatz von WebSocket Messages, SDPs und ICE Candidates über den Signaling Kanal ausgetauscht und eine RTCPeerConnection erfolgreich hergestellt werden konnte, beginnt der EKG Daten Sender mit dem Versenden von EKG Daten über den RTCDataChannel.
Um Jitter bei der Übertragung der EKG Daten auszugleichen und das Zeichnen der EKG Kurve vom Empfang der EKG Daten zu entkoppeln wird ein Buffer in Form eine Queue eingesetzt, womit Anforderung 29 erfüllt wird. Über den RTCDataChannel empfangene EKG Samples werden direkt in den Buffer geschrieben. Zum Zeichnen der EKG Kurve wird eine variable Anzahl an EKG Samples aus dem Buffer entnommen, die durch die Regulierung des Buffer beeinflusst wird.
Der Buffer soll möglichst klein gehalten werden, damit EKG Daten nach nur kurzer Verzögerung dargestellt werden aber groß genug sein, um die als Jitter bekannten Schwankungen beim Datenempfang über das Internet auszugleichen. Um dies zu erreichen findet eine Regulierung der Buffergröße statt und es wird eine variable Auslesegeschwindigkeit des Buffer genutzt.
Die Regulierung wird in der Funktion oneSecInterval() realisiert, bei der es sich um eine Intervall Funktion handelt, die periodisch ein Mal pro Sekunde aufgerufen wird. Damit wird sie deutlich seltener aufgerufen als die shiftAndPaint() Funktion, die zum Auslesen des Buffer und anschließendem Zeichnen der EKG Kurve alle 25ms aufgerufen wird. Die Funktion zur Regulierung wird nur einmal in der Sekunde aufgerufen, da deutlich mehr Operationen und rechenintensivere Funktionen aufgerufen werden, als beim Auslesen der EKG Daten aus dem Buffer und dem anschließenden Zeichnen auf ein Canvas Element.

Die Berechnung des Wertes offset, der zum Regulieren der Auslesegeschwindigkeit beim Zeichnen der EKG Kurve genutzt wird und die Entscheidung zur Reduzierung der Buffergröße, werden in folgendem Code-Abschnitt gezeigt.

```
function oneSecInterval() {
	var flussDiff = Math.round(updateRecvSampleRate());
	var standDiff = Math.round(updateBufferFillState());
	var posOffsetBorder = (numMessagePerInterval * 0.2);
	var negOffsetBorder = (numMessagePerInterval * -0.2);

	offset = Math.round(((flussDiff * 0.1) + (standDiff * 0.9)) / 40);

/*zu schnelle/langsames lesen verhindern, gerade nach buffer
verkleinerung*/
	if (offset > posOffsetBorder) {
		offset = posOffsetBorder;
	} else if (offset < negOffsetBorder) {
		offset = negOffsetBorder;
	}

	if (bufferReaderInterval == null) {
		//buffering
	} else {

		if ((offset < posOffsetBorder)&&(offset > negOffsetBorder)){
			holdCounter++;
		} else {
			holdCounter = 0;
		}
/*holdcounter um eine entsprechende zeit auf ein filllevel zu warten*/
		if ((holdCounter > 3) && (sampleRate < bufferSize)) {
			holdCounter = 0;
			reduceBufferSize();
		}
	}
}
```

Die Funktion oneSecInterval() wird einmal in der Sekunde aufgerufen. Als Erstes werden vier Variablen gesetzt: flussDiff, standDiff, posOffsetBorder und negOffsetBorder.
Die Variable flussDiff enthält einen Wert, der die Abweichung empfangener EKG Daten vom Sollwert 400 Samples/s in der letzten Sekunde beziffert. In der Variable standDiff ist ein Wert enthalten, der die aktuelle Abweichung des Buffer Füllstandes vom Sollwert 50% angibt.
Die beiden Variablen posOffsetBorder und negOffsetBorder sind freigewählte Grenzwerte, die zur Begrenzung der Geschwindigkeitsregulierung dienen, in der der Buffer ausgelesen und die EKG Kurve gezeichnet wird.

Die Variable offset ist global und hat direkten Einfluss auf die Geschwindigkeit, in der die Funktion shiftAndPaint() die EKG Daten aus dem Buffer entnimmt und auf das Canvas Element zeichnet.
Der Wert der Variable offset wird mit Hilfe der beiden Variablen flussDiff und standDiff berechnet. Die Variable offset ist der Wert des Quotienten, wobei der Dividend eine gerundete Summe zweier gewichteter Multiplikationen ist und der Divisor dem Wert 40 entspricht. Die Summe des Dividenden wird durch den Faktor dividiert, der durch die Funktion shiftAndPaint() häufiger aufgerufen wird, als die Funktion oneSecInterval(). Der Faktor entspricht dem Wert 40 und die Division ist eine entsprechende Abschwächung.
Die beiden Multiplikationen im Dividend sind Gewichtungen für die Werte, die die zuvor ermittelten Abweichungen von den Sollwerten des Datenempfangs und des Buffer Füllstand beziffern. Die Gewichtung durch die Multiplikation beschreibt den Einfluss der jeweiligen Abweichung auf das regulierende Ergebnis in der Variable offset. Da die Variable standDiff mit 0.9 multipliziert wird, hat sie einen entsprechend höhere Gewichtung und damit Einfluss auf den Wert des Quotienten, der in der Variable offset gespeichert wird. Es wird daher deutlich stärker auf eine Abweichung des Buffer Füllstandes reagiert, als auf Abweichungen beim Datenempfang, da der Buffer kurzzeitige Abweichungen des Datenempfangs ausgleicht. Ob mehr oder weniger Samples pro Sekunde aus dem Buffer ausgelesen und auf das Canvas Element gezeichnet werden, hängt somit in erster Linie vom Füllstand des Buffers ab. Ist der Füllstand zu hoch, werden mehr Samples in der Sekunde gezeichnet, ist der Füllstand zu niedrig, werden weniger Samples in der Sekunde gezeichnet.

**Beispiel zum Verhalten beim Empfang von zu vielen Samples/s**

Besteht eine positive Abweichung des Datenempfangs für längere Zeit, steigt der Füllstand des Buffer an, worauf mit einer höheren Geschwindigkeit beim Auslesen des Buffers und Zeichnen der EKG Kurve reagiert wird. Würde der Füllstand des Buffers trotzdem weiter steigen, weil die Erhöhung der Auslesegeschwindigkeit nicht ausreicht, wird der Buffer vergrößert. Die Vergrößerung des Buffer wird in einer Callback Funktion umgesetzt, die der Buffer beim Überlaufen aufruft.

**Beispiel zum Verhalten beim Empfang von zu wenigen Samples/s**

Besteht hingegen über längere Zeit eine negative Abweichung des Datenempfangs, sinkt der Füllstand des Buffer ab, worauf mit einer geringeren Geschwindigkeit beim Auslesen des Buffers und Zeichnen der EKG reagiert wird. Wäre die Abweichung so stark, dass der Füllstand des Buffers trotzdem sinken würde, würde der Buffer leer laufen. Ist der Buffer leer, wird der Aufruf der Intervall Funktion shiftAndPaint() beendet, womit keine EKG Daten mehr aus dem Buffer ausgelesen und die EKG Kurve nicht weiter gezeichnet wird. Der Füllstand des Buffers würde wieder ansteigen. Die Intervall Funktion shiftAndPaint() wird erst ab einem Füllstand von 50% wieder gestartet.

Die zuvor genannten Variablen posOffsetBuffer und negOffsetBuffer grenzen dabei den Wert der Variable offset im IF-Statement ein, um nicht zu viele oder wenige Samples in der Sekunde zu zeichnen. Eine zu starke Änderung der Geschwindigkeit beim Zeichnen der EKG Kurve, ist zu dem unangenehm zu betrachten.
Abschließend wird noch geprüft ob eine Reduzierung der Buffergröße sinnvoll ist. Dazu darf der Wert der Variable offset für drei Sekunden keine der beiden Grenzen, posOffsetBuffer und negOffsetBuffer, überschreiten.
Die Reduzierung der Buffergröße wird auf minimal 400 Samples begrenzt, was einem Buffer von einer Sekunde EKG Daten entspricht. Da die Berechnung zur Regulierung nur einmal in der Sekunde aufgerufen wird und damit träge arbeitet, ist ein kleinerer Wert kritisch.
Eine Erweiterung des Buffer ist auch möglich. Diese wird aber aufgrund der Trägheit der Regulierung über eine Callback Funktion des Buffers ausgelöst, die aufgerufen wird, wenn es zum Buffer Overflow kommt. Daten die beim Overflow des Buffers verloren gehen würden, werden wie der Inhalt des vollen Buffer in den vergrößerten Buffer gespeichert.

**Darstellungsbereich der EKG Kurve**

Das Gitternetz und die EKG Kurve werden auf zwei unterschiedliche übereinanderliegende Canvas Elemente gezeichnet. Dies ermöglicht eine unabhängige Gestaltung der zu zeichnenden Darstellung und dem Hintergrund, womit Anforderung 24 erfüllt wird. Zusätzlich hilft die Trennung von der Darstellung der EKG Kurve und dem Hintergrund, um den Aufwand beim Zeichnen der EKG Kurve möglichst gering zu halten.

**Schließen der Web-Anwendung**

Um Anforderung 3 zu erfüllen ist eine Handler Funktion implementiert, die eine hergestellte RTCPeerConnection beim Schließen des genutzten Tabs oder des eingesetzten Web-Browser sauber abbaut. Die Handler Funktion wird durch das Auftreten des unload Events des Web-Browsers oder des onbeforeunload Events des Windows Objekts aufgerufen.

### 5.2.3 Chat

Der Chat wird in einem grünlich gestalten Frame dargestellt, wie in Abbildung 5.2.3-40 zu sehen ist. Der Chat dient den Benutzern zur Kommunikation, womit Anforderung 8 erfüllt wird. Beim Versenden einer Nachricht werden ausschließlich WebSocket Messages genutzt, die über den WebSocket-Server an alle anderen Instanzen der Web-Anwendung weitergesendet werden, wie es Anforderung 10 vorsieht.

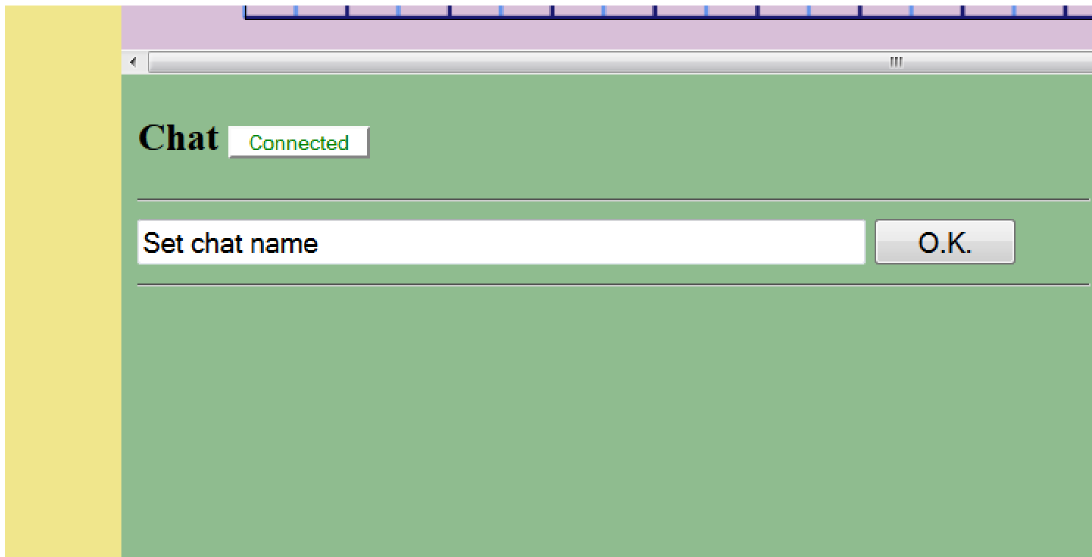



Abbildung 5.2.3-40.: Darstellung des Chat

Der Benutzer muss einen Namen in die Eingabezeile eingeben und diesen bestätigen. Anschließend kann der Chat genutzt werden, was Anforderung 9 entspricht.
Chatnachrichten werden in die Eingabezeile eingegeben und nach Bestätigung des Buttons, der nach der Eingabe eines Namens mit send beschriftet ist, versendet.
Die Abbildung 5.2.3-41 zeigt zwei Chats aus unterschiedlichen Instanzen der Web-Anwendung und die Darstellung der eigenen versendeten und der empfangenen Chatnachrichten. Auf der linken Seite der Abbildung ist die Instanz von Benutzer 1 und auf der rechten Seite die Instanz von Benutzer 2 abgebildet.

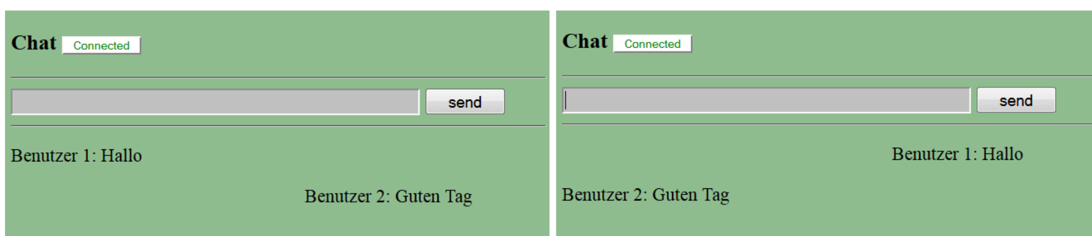



Abbildung 5.2.3-41.: Darstellung von gesendeten und empfangenen Chatnachrichten

In beiden Instanzen werden Chatnachrichten in gleicher Form dargestellt. Unabhängig ob der Benutzer die Nachricht selbst geschrieben hat, wird erst der angegebene Name und dann die Nachricht dargestellt, was Anforderung 15 entspricht.

### 5.2.4 Konsole

In einem Frame der Web-Seite wurde eine Konsole zur Ausgabe von System-Nachrichten realisiert, die in Abbildung 5.2.4-42 dargestellt ist. Die Ausgabe wird genutzt um Fehlermeldungen aber auch Statusmeldungen des Systems an den Benutzer ausgeben zu können, so dass dieser nicht auf die Konsole des verwendeten Browser zurückgreifen muss.
Zum Debugging kann die Konsole nur bedingt genutzt werden, da immer nur die 1000 aktuellsten Nachrichten im Textfeld gehalten werden.
Es wurden Funktionen zum Scrollen an den Anfang und an das Ende des Konsolenverlaufs implementiert, welche über Buttons genutzt werden können. Über weitere Buttons können, der komplette Konsolenverlauf gelöscht und das standardmäßig aktivierte Auto-Scrolling deaktiviert werden.

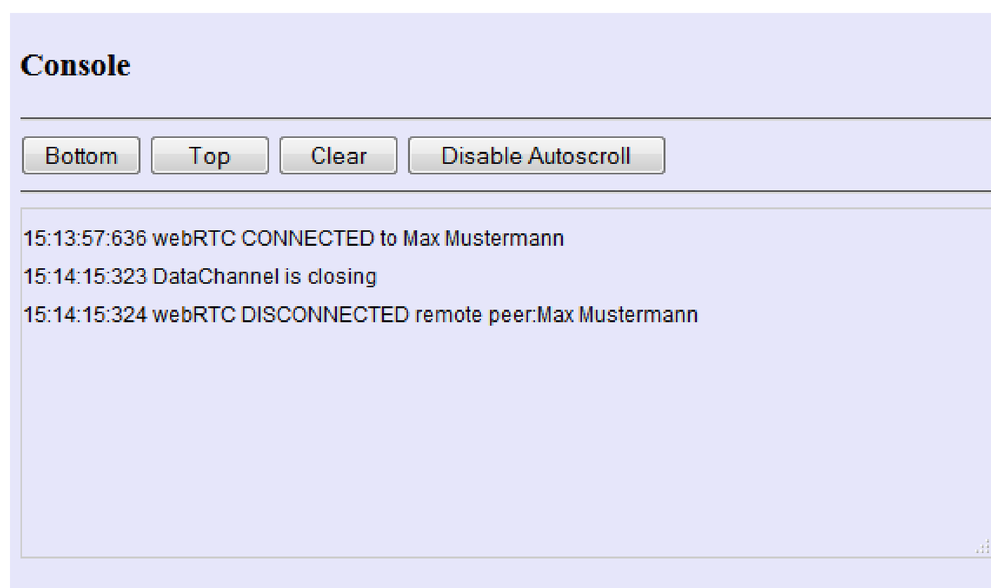



Abbildung 5.2.4-42.: Konsole mit Statusinformationen

### 5.2.5 Informationen

Neben der Konsole wurde ein weiterer Frame Implementiert, der keine Anforderungen erfüllt aber den Benutzer hilfreich unterstützen kann, wenn es darum geht das Verhalten der Web-Anwendung zu deuten.
Der Frame dient der Anzeige von Informationen zum Buffer und der Verarbeitung von EKG Daten.
Wie in Abbildung 5.2.5-43 dargestellt ist, werden Informationen zur Buffer Größe, der Anzahl an enthaltenen Samples, dem Füllstand und der damit verbundenen Verzögerung angezeigt.

Zu jeder hergestellten Verbindung wird die ID des EKG Daten Senders, der aktuell empfangene Wert, die Gesamtanzahl empfangener EKG Samples und die aktuelle Empfangsrate von EKG Samples angezeigt. Zusätzlich wird noch die Rate angegeben, mit der die EKG Daten in Kurvenform gezeichnet werden.

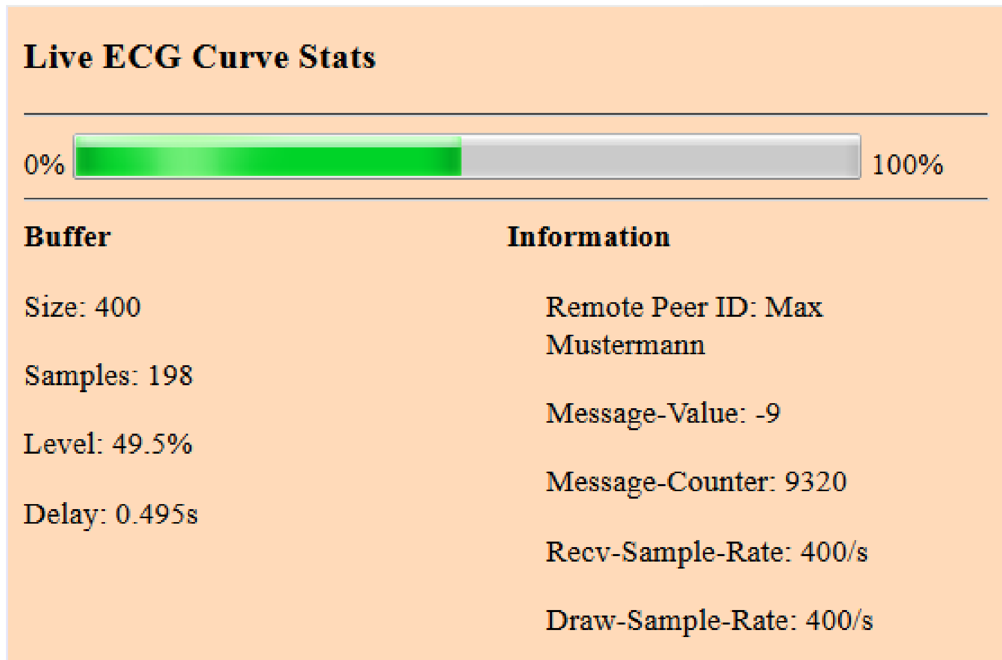



Abbildung 5.2.5-43.: Darstellung von Informationen zum Buffer und der aktuellen Verbindung zum EKG Daten Sender

## 5.3 EKG Daten Sender

### 5.3.1 Vorwort

Der EKG Daten Sender ist eine Anwendung, die ein mobiles internetfähiges EKG simuliert, welches die EKG Daten über eine verschlüsselte RTCPeerConnection Verbindung des WebRTC Framework sendet.
Entwickelt und implementiert wurde eine JavaScript Lösung für die Web-Browser Chrome und Firefox, da diese die aktuell gebotene WebRTC Web-API implementieren und daher als Referenz dienen.
Eine angepasste JavaScript Lösung ist für das Framework Node.JS implementiert worden, welches auf einem 64-bit Ubuntu 14.04.2 LTS System zum Einsatz kommt.
Es wurden zwei Lösungen entwickelt, da eine von der Web-Browser API unabhängige Implementierung für die Realisierung des zu Beginn genannten EKGs zweckmäßiger ist. Für ein mobiles EKG, was EKG Daten über eine RTCPeerConnection sendet, wäre eine von der Web-API unabhängig native Implementierung optimal, um möglichst performant zu arbeiten. Es würde beispielsweise den Einsatz auf einem kostengünstigen mobilen SOC ermöglichen, ohne auf die Verwendung eines Web-Browsers angewiesen zu sein.
Es gibt einige Implementierungen des WebRTC Framework, für das Node.JS Framework auf x86 Systemen, die ohne Verwendung eines Web-Browsers eingesetzt werden können. Das Einsatzgebiet von Node.JS ist groß, da es plattformunabhängig eingesetzt werden kann, was Anforderung 30 erfüllt. Node.JS wird zudem von einer weltweiten Community mit verschiedensten kostenfreien Implementierungen unterstützt, was Anforderung 31 entspricht. Die Node.JS Implementierungen für WebRTC sind einer nativen Implementierung am nächsten und sind daher als Beispiel für den Einsatz von WebRTC, ohne Verwendung eines Web-Browser und der gebotenen Browser-API, anzusehen. Eine solche Implementierung kommt zum Einsatz, um einen Eindruck von zukünftigen nativen Implementierungen zu bekommen.
Die beiden Lösungen wurden möglichst identisch entwickelt, dessen Leistungen wird in Kapitel 6.2 miteinander verglichen.

### 5.3.2 Allgemeines Verhalten des EKG Daten Senders

Trotz der zwei Lösungen für den Einsatz auf Unterschiedlichen Systemen, sind sich beide Implementierungen sehr ähnlich und haben eine fast identische Arbeitsweise. In den folgenden Abschnitten wird die Arbeitsweise eines EKG Daten Senders beschrieben. In den Kapiteln 0 und 5.3.4 wird auf die Unterschiede in der Implementierung hingewiesen.

**Start des EKG Daten Senders**

Um Anforderung 17 nachzukommen, muss ein Benutzer beim Starten eines EKG Daten Senders eine ID für den EKG Daten Sender vergeben, die in der Senderliste der Web-Anwendung angezeigt wird. Zusätzlich können die Datei die EKG Daten beinhaltet und der URI des Peer Endpoint angegeben werden. Nach dem Start wird die gewählte Quelldatei eingelesen, in der die EKG Daten stehen, um diese in einem Array zu speichern und für den Versand vorzubereiten. Für eine reale Implementierung wäre es notwendig wiederholend aktuelle EKG Daten des Sensors auszulesen.
Nachdem die eingelesenen EKG Daten in einem Array gespeichert wurden, wird eine WebSocket-Verbindung zum Peer Endpoint hergestellt. Sobald die Verbindung erfolgreich hergestellt wurde, wird die Callback Funktion für das OnOpen Event aufgerufen, aus der eine Register Message an den Endpoint gesendet wird, um sich zu registrieren. Folgender Code-Abschnitt zeigt die Callback Funktion des WebSocket Objekts, in der ein JSON String erstellt wird, der die RegisterMessage repräsentiert und anschließend versendet wird.

```
function onOpen(evt) {
	webSocketSend(JSON.stringify({
		"id": senderID,
		"browserWSID": "",
		"regType": REGISTERTYPE.SENDER,
		"type": WEBSOCKETTYPE.REGISTER
	}));
}
```

Der EKG Daten Sender führt keinen weiteren Code aus und wartet, bis eine neue WebSocket Messages eintrifft, worauf dann die Callback Funktion onMessage() aufgerufen wird.
Wählt ein Benutzer der Web-Anwendung den EKG Daten Sender aus, um dessen EKG Daten zu beobachten, wird durch die Web-Anwendung der Versand einer Offer Message über den Signaling Kanal an den gewählten EKG Daten Sender ausgelöst.
Wird vom EKG Daten Sender eine WebSocket Message empfangen, wird anhand des genutzten Message-Systems die Art der Message bestimmt und mit dem Aufruf einer entsprechenden Funktion reagiert.

**Erhalt der Offer Message**

Trifft die Offer Message einer Web-Anwendung ein, wird die Funktion generateAnswer() aufgerufen, welche in folgendem Code-Abschnitt präsentiert wird.

```
function generateAnswer(peerWSID, description) {
	var pc = new RTCPeerConnection(iceConfig);
	peerConnections.set(peerWSID, pc);

	var rtcSessionDescription = new RTCSessionDescription(description);
	pc.setRemoteDescription(rtcSessionDescription);

	pc.createAnswer(function(aDescription) {
		pc.setLocalDescription(aDescription, function() {
			sendAnswer(peerWSID, aDescription);
		}, logError);
	}, logError);

	pc.onicecandidate = function(event) {
		if (!event.candidate) {
			if (event.target.iceGatheringState == "complete") {
			return;
			}
		} else {
			var candidate = event.candidate;
			sendIceCandidate(peerWSID, candidate);
		}
	}

	pc.ondatachannel = function(evt) {
		var dataChannel = evt.channel;
		dataChannels.set(peerWSID, dataChannel);
		dataChannelHandle(dataChannels.get(peerWSID));
	};
};
```

Beim Aufruf der Funktion generateAnswer() werden zwei Parameter übergeben.
Der erste Parameter peerWSID beinhaltet die WebSocket Session ID des Absenders der empfangenen Offer Message. Die WebSocket Session ID wurde vom Peer Endpoint beim Weiterleiten der Offer Message in die Message eingefügt. Beim Erstellen einer Antwort kann die WebSocket Session ID für die Angabe des Empfängers genutzt werden.
Der zweite Parameter description beinhaltet die SDP der initiierenden Web-Anwendung.
Als Erstes wird ein RTCPeerConnection Objekt erzeugt, wobei der Parameter iceConfig die zu verwendende Adresse von STUN- und TURN Server beinhaltet.
EKG Daten Sender müssen den Umgang mit mehreren RTCPeerConnections beherrschen, damit die Benutzer mehrerer Web-Anwendungsinstanzen parallel die EKG Daten eines EKG

Daten Senders beobachten können. Aus diesem Grund wird das RTCPeerConnection Objekt in einer Map gespeichert und die WebSocket Session ID als Key verwendet.
Nachdem das RTCPeerConnection Objekt in der Map gespeichert wurde, wird ein RTCSessionDescription Objekt erzeugt. Beim Erzeugen des Objekts wird die mit der Offer Message empfangene SDP als Parameter übergeben. Das erzeugte Objekt wird dem RTCPeerConnection Objekt anschließend mit dem Aufruf der Funktion setRemoteDescription() übergeben.

**Versenden der Answer Message**

Im Anschluss wird die Funktion createAnswer() des RTCPeerConnection Objekts aufgerufen, welche eine SDP erstellt, die dem RTCPeerConnection Objekt mit dem Aufruf der Funktion setLocalDescription übergeben wird. Anschließend wird die SDP zusammen mit der empfangenen WebSocket Session ID der Web-Anwendung beim Aufruf der Funktion sendAnswer() übergeben. Durch den Aufruf der Funktion sendAnswer() wird die Answer Message über den Signaling Kanal an die initiierende Instanz der Web-Anwendung versendet, die das neu erzeugte SDP enthält.
Die Callback Funktion des RTCPeerConnection Objekts mit der Bezeichnung onicecandidate wird vom den ICE Agent des RTCPeerConnection Objekts, durch das Auslösen eines entsprechenden Events, aufgerufen. Der ICE Agent löst ein Event aus, wenn ein ICE Candidate gefunden wurde oder die der ICE gathering Prozess abgeschlossen ist. Der Unterschied liegt im Inhalt des Events. Daher wird der Inhalt des Events, nach Aufruf der Callback in einem IF-Statement überprüft.

**Empfang einer IceCandidate Message**

Enthält dass Event einen ICE Candidate, wird dieser und der Parameter peerWSID an die Funktion sendIceCandidate() übergeben. Der Aufruf dient dazu, eine IceCandidate Message an die initiierende Instanz der Web-Anwendung zurückzusenden und den gefunden ICE Candidate mitzuteilen.
Wird im IF-Statement ein Event ohne Candidate ermittelt und der iceGatheringState hat den Status complete, ist die Suche nach passenden Routingpfaden in Form von ICE Candidates abgeschlossen und die Callback Funktion ondatachannel des RTCPeerConnection Objekts wird durch ein entsprechende Event aufgerufen.

**Empfang des RTCDataChannel**

Wir die Callback Funktion ondatachannel durch ein Event aufgerufen beinhaltet das Event ein RTCDataChannel Objekt. An dieser Stelle wird deutlich, dass ein symmetrischer DataChannel verwendet wird, der durch den initiierenden Peer definiert wird, wie in Kapitel 2.2.7 beschrieben.

Das im Event enthaltene RTCDataChannel Objekt wird in einer Map gespeichert, wobei als Key wieder die WebSocket Session ID der initiierenden Instanz der Web-Anwendung verwendet wird. Nun sind RTCPeerConnection Objekt und RTCDataChannel Objekt in zwei unterschiedlichen Maps gespeichert, wobei die gleiche WebSocket Session ID als Key genutzt wurde. Die Identifikation ist so anhand der WebSocket Session ID immer möglich, was vor allem beim Abbau der Verbindung wichtig ist, was nachfolgend in diesem Kapitel erläutert wird.
Das RTCDataChannel Objekt wird dann an einen Handle übergeben, womit es möglich ist, auf die Events des RTCDataChannel Objekts zu reagieren.

**Senden der EKG Daten**

Nach der Übergabe des RTCDataChannel Objekts an das Handle wird das OnOpen Event des RTCDataChannel die entsprechende Callback Funktion des Handle aufrufen. Darin wird ein Intervall gestartet, welches die Funktion sendECGData() alle 100ms periodisch aufruft.
In der Funktion werden die zu sendenden EKG Daten aus einem Array, das die EKG Samples aus der Quelldatei beinhaltet, kopiert und in eine Variable geschrieben. Da die Funktion alle 100ms aufgerufen wird, werden bei jedem Aufruf 40 EKG Samples aus dem Array kopiert. Auf diese Weise sendet der EKG Daten Sender 400 EKG Samples pro Sekunde, was dem Sollwert entspricht, der sich an der Auflösung eines echten EKG orientiert.
Sind alle Daten aus dem Array, das die EKG Samples aus der Quelldatei beinhaltet, versendet worden, wird wieder am Anfang des Array mit dem Kopieren fortgesetzt, was einen Dauerbetrieb des EKG Daten Senders ermöglicht und Anforderung 21 erfüllt.
In einer Schleife wird über die Map iteriert, in der alle vorhandenen RTCDataChannel Objekte gespeichert sind. Bei jeder Iteration wird die Funktion send() der RTCDataChannel Objekte aufgerufen und die Variable mit den 40 EKG Samples übergeben, so dass über alle RTCDataChannel die gleichen EKG Daten versendet werden. Dies ermöglicht den Benutzern von Instanzen der Web-Anwendung die EKG Daten eines EKG Daten Senders gemeinsam zu beobachten, womit Anforderung 12 erfüllt ist.

**Abbau der Peer-to-Peer Verbindung**

Der Abbau der RTCPeerConnection erfolgt nur seitens der Web-Anwendung wenn, Benutzer den Disconnect Button nutzen, den zu beobachtenden EKG Daten Sender wechseln, den verwendeten Tab oder Web-Browser schließen. Wird die Darstellung der EKG Kurve durch eine der genannten Möglichkeiten beendet, schließt die Instanz der Web-Anwendung den RTCDataChannel. Das Schließen des RTCDataChannels löst ein OnClose Event aus, wodurch auch die entsprechende Funktion auf Seiten des EKG Daten Senders aufgerufen wird. Zusätzlich wird beim Senden überprüft, ob der RTCDataChannel offen ist. Wird erkannt dass ein RTCDataChannel nicht mehr offen ist oder tritt das OnClose Event auf, wird das entsprechende RTCDataChannel Objekt und das zugehörige RTCPeerConnection Objekt aus der jeweiligen Map gelöscht.

Der EKG Daten Sender kann das RTCDataChannel an dessen Label identifizieren, wie es Anforderung 14 vorsieht. Das Label entspricht einer WebSocket Session ID, welche als Key für die Maps verwendet wurde als das RTCPeerConnection Objekt und das RTCDataChannel Objekt gespeichert wurden. Es wird geprüft, ob die Labelbezeichnung einem Key der Maps entspricht. Trifft dies zu, werden die entsprechenden Einträge in der Maps gelöscht. Nach dem Löschen der Objekte werden keine EKG Daten mehr über das gelöschte RTCDataChannel Objekt versendet und die Ressourcen wieder freigegeben. Gibt es weiter Einträge in den Maps, wird der Sendevorgang für diese fortgeführt, wie es Anforderung 13 vorsieht. Gibt es keine weiteren Einträge in den Maps, wird das Sendeintervall, dass die Funktion sendECGData() aufruft, beendet und der Sendevorgang eingestellt. Erst mit dem Aufbau einer neuen Peer-to-Peer Verbindung, würde der Sendeintervall wieder gestartet werden.

### 5.3.3 Implementierung für Web-Browser

**Dateityp**

Die Implementierung des EKG Daten Sender für die Referenz Web-Browser Firefox und Chrome erfolgt in einer HTML Datei, die einen definierten Bereich für JavaScript enthält.

**Einlesen der Quelldatei**

Beim Starten wird ein Input Element angezeigt, in die der Benutzer die zu verwendende ID des EKG Daten Senders eintragen muss. Über ein zweites Input Element kann die URI zum Peer Endpoint angeben werden, sonst wird eine vordefinierte URI Angabe verwendet.
Anschließend wird eine Formulareingabe angezeigt, über die eine lokale Datei ausgewählt werden muss, die EKG Daten enthält, womit Anforderung 19 erfüllt wird. Das Einlesen der Quelldatei erfolgt über die mit HTML 5 eingeführte File API Spezifikation [FILEAPI]. Nach Auswahl der Datei wird ein entsprechenden Event ausgelöst, welche von einem Eventlistener an eine Handle Funktion übergeben wird. Dass an die Funktion übergebene Event beinhaltet ein Filelist Objekt, was Zugriff auf die gewählte Datei gewährt. Mit Hilfe eines FileReader Objekts wird über das Filelist Objekt iteriert. Dem FileReader Objekt wird eine Funktion übergeben, die durch das Event onloaded aufgerufen wird, was geschieht, sobald alle Daten des Filelist Objekts eingelesen sind. In der durch das onloaded Event aufgerufenen Funktion wird der vom FileReader Objekt erstellte String in EKG Samples zerteilt, welche dann in ein Array gespeichert werden.
Die durch ein Intervall periodisch aufgerufene Funktion sendECGData, welche im Kapitel 5.3.2 erklärt wurde, nutzt die Daten des Arrays um die EKG Samples an die Instanzen der Web-Anwendung zu versenden.
Durch das komplette Einlesen der Quelldatei und Erstellung eines Array mit den ausgelesenen EKG Samples, kann das Versenden schnell erfolgen. Allerdings ist dies keine

Lösung die genutzt werden kann, wenn in einer Datei immer neue EKG Daten von einem Sensor geschrieben werden.

**Verwendung der API's**

Die Implementierung für Web-Browser kann auf die angebotenen Web APIs für WebSockets und das WebRTC Framework zugreifen, die in den Spezifikationen gut dokumentiert sind. Daher kann auf fremde Implementierungen der API's verzichtet werden.

### 5.3.4 Implementierung für Node.JS

**Dateityp**

Die Implementierung für Node.JS erfolgt ausschließlich in JavaScript und wird in einer Konsole ausgeführt.

**Einlesen der Quelldatei**

Beim Start der EKG Daten Sender Anwendung muss der Benutzer ein Argument für die ID des EKG Daten Senders angeben. Als Argument kann auch der Dateiname der Datei angegeben werden, die die zu versendenden EKG Daten enthält, womit Anforderung 19 erfüllt wird. Optional kann per Argument auch ein URI zum Peer Endpoint angegeben werden, sonst wird eine vordefinierte URI Angabe genutzt.
Um auf das Dateisystem zugreifen zu können, wurde das Node.JS Modul Filesystem [FS]verwendet, was einen Wrapper um die Standard POSIX Funktionen realisiert und so den Zugriff auf lokale Dateien ermöglicht.
Nach dem Start des EKG Daten Senders wird die Funktion readFile() aufgerufen, in welcher die Quelldatei geöffnet und ausgelesen wird. Beim Auslesen werden die EKG Samples in ein Array gespeichert.
Die durch ein Intervall periodisch aufgerufene Funktion sendECGData, welche im Kapitel 5.3.2 erklärt wurde, greift auf das Array zu, wenn es die EKG Samples an die Instanzen der Web-Anwendung versendet.
Auch für diese Lösung gilt, dass das Einlesen der kompletten Quelldaten nur in diesem Modell möglich ist. Beim Einsatz eines realen EKGs, welches seine Sensordaten in eine Datei schreibt, müsste das Auslesen der EKG Daten periodisch erfolgen.

**Verwendung der API‘s**

Durch den Einsatz von Node.JS [NODEJS] in der Version 0.10.37, ist der Zugriff auf eine Vielzahl von Implementierungen der unterschiedlichsten API’s gegeben. Dies sind zum Teil von Node.JS angebotene Module aber auch oft Community-Projekte, die noch in der Entwicklung sind. Dies trifft auch auf die gewählte Implementierung der WebRTC API [WRTC] und der WebSocket API [WS] zu. Allerdings konnte durch den Einsatz dieser Implementierungen ein EKG Daten Sender für JavaScript realisiert werden.

## 5.4 Verwendung fremden Codes

Bei der Realisierung des entwickelten Systems, wurde an einigen Stellen fremder Code verwendet. In diesem Kapitel wird daher eine Aufzählung des nicht selbst geschrieben eingesetzten Codes erfolgen.

### 5.4.1 cbuffer.js

Die JavaScript Implementierung eines Ringbuffers namens CBuffer [CBUF] wurde in der Version 1.1.0, bei der Realisierung der Web-Seite für die Web-Anwendung verwendet. Der Ringbuffer wird eingesetzt, um die EKG Daten vor der Darstellung als EKG Kurve zwischen zu speichern und Jitter entgegenzuwirken.
Der CBuffer steht unter der MIT License (MIT) und das Copyright liegt bei Trevor Norris.

### 5.4.2 adapter.js

Das JavaScript adapter.js vom 23.04.2013, wird in Googles Repository [ADAP] angeboten und adaptiert die noch unterschiedlichen Interfaces, die von Firefox und Chrome genutzt werden, auf den W3C Standard. Dies ermöglicht die Implementierung des WebRTC Frameworks, ohne weiter auf die unterschiedlichen Interfaces Rücksicht nehmen zu müssen, wie in Kapitel erwähnt 2.2.3. Das Adapterskript steht unter einer BSD-style License, das Copyright liegt bei The WebRTC project authors, wobei im Root Ordner folgende Autoren genannt werden: Google Inc., Mozilla Foundation, Intel Corporation, Vonage Holdings Corp., MIPS Technologies, Ben Strong, Martin Storsjo, Jie Mao, Anil Kumar, Opera Software und Silviu Caragea.

### 5.4.3 Node-webrtc

Die Implementierung des WebRTC Frameworks für Node.JS wurde in Version 0.0.49, für die Realisierung des EKG Daten Senders für Node.JS [NODEJS] verwendet. Node-webrtc [WRTC] ist auf GitHub veröffentlicht, das Copyright liegt bei Alan Kligman.

### 5.4.4 WS

WS ist die Bezeichnung einer Implementierung der WebSocket Spezifikation für Node.JS, die in Version 0.7.1 für die Realisierung des EKG Daten Senders für Node.JS eingesetzt

wurde. WS ist auf GitHub veröffentlicht [WS], steht unter der MIT License (MIT) und das Copyright liegt bei Einar Otto Stangvik.

### 5.4.5 Hashmap

Hashmap ist der Name einer JavaScript Klasse für Node.JS und Web-Browser, die eine Implementierung einer Hashmap ist. Verwendet wird sie in der Version 2.0.0, bei der Realisierung der EGK Daten Senders für Node.JS, um die RTCPeerConnection und RTCDataChannel Objekte der hergestellten Verbindungen zu speichern. Hashmap ist auf GitHub veröffentlich [HASH], steht unter der MIT License (MIT), das Copyright liegt beim Autor Ariel Flesler.

### 5.4.6 ECGSYN

Für die Generierung von EKG Daten, die von EKG Daten Sendern versendet werden, wurde das Java Applet ECGSYN genutzt. Es wird auf der Web-Seite von PhysioNet [ECGSYN] veröffentlicht. Das Applet wurde von Mauricio Villarroel entwickelt, welcher auch das Copyright besitzt und steht unter einer GNU General Public License 2 (GNU GLP2). Es basiert auf einer Entwicklung von Patrick McSharry und Gari Clifford.

# 6 Experimente

In diesem Kapitel wird die entwickelte Realisierung der Web-Anwendung und die Lösungen des EKG Daten Sender untersucht und getestet, um einen Einsatz in einer realen Umgebung einschätzen und die Lösungen des EKG Daten Senders vergleichen zu können.
Im Kapitel 6.1 wird untersucht, ob das Datenvolumen, welches bei einer Übertragung der EKG Daten über eine Peer-to-Peer gemessen wird, dem berechneten Datenvolumen und damit den Erwartungen entspricht. Berücksichtigt wird nur das Datenvolumen, welches der EKG Daten Sender sendet, da dies in den Szenarien aus Kapitel 3.1 dem Upload entspricht, den der EKG Daten Sender pro Verbindung zu einer Web-Anwendungsinstanz benötigt.
Anschließend wird im Kapitel 6.1.1 eine Messung des maximalen Datenvolumens präsentiert, welches in Abhängigkeit der Intervalldauer und der Paketgröße gemessen wurde.
Die beiden Lösungen des EKG Daten Sender werden in Kapitel 6.2 verglichen. Dazu wird untersucht, ob die Lösungen ein unterschiedliches Datenvolumen erzeugen wenn EKG Daten gesendet werden. In einer weiteren Messung wird die Auslastung der EKG Daten Sender Lösungen vergleichen, wenn mehrere Instanzen der Web-Anwendung dazu genutzt werden, die EKG Daten eines entfernten EKG Daten Senders zu beobachten. Abschließend wird in Kapitel 6.3 das Verhalten der Web-Anwendungsinstanzen beobachtet, wenn der Upload des EKG Daten Senders ausgelastet wird.

## 6.1 Datenvolumen einer Peer-to-Peer Verbindung

In diesem Kapitel wird das Datenvolumen untersucht, welches beim Senden von EKG Daten über eine Peer-to-Peer Verbindung, anfällt. Es wird berechnet, wie groß das zu erwartende Datenvolumen und wie hoch der Anteil des Overheads dabei ist. Anschließend wird das berechnete Datenvolumen mit einer Messung verglichen, bei der die EKG Daten in einem Netzwerk transportiert werden.
Das Datenvolumen ist in erster Linie vom Volumen der Nutzdaten abhängig. Zusätzlich fällt der Overhead der verwendeten Protokolle an. Wie in Kapitel 5.3.2 beschrieben, sendet der EKG Daten Sender in einem Intervall von 100ms ein Paket mit 40 EKG Samples über den RTCDataChannel, um die erforderlichen 400 Samples pro Sekunde zu übertragen. Mit jedem Sendeaufruf fällt der Overhead der Protokolle an, die zur Datenübertragung genutzt werden.

Im Falle des SCTP basierten RTCDataChannel, der per DTLS verschlüsselt ist und via UDP transportiert wird, welches auf dem IP aufsetzt, ist der Overhead pro Sendeaufruf eine Summe, des jeweiligen Overhead der genannten Protokolle.
Der Overhead des IPv4, in Form eines Headers, beträgt typischerweise 20 Oktetts bzw. Bytes [RFC791]. Der Header vom UDP beträgt feste 8 Oktetts [RFC768]. Der Overhead von DTLS beträgt zwischen 20-40 Bytes [BroNet]. Der Overhead von SCTP beträgt mindestens 28 Bytes, bestehend aus 12 Bytes vom Header und 16 Bytes für jeden Data Chunk, wovon mindestens einer existiert. Zusätzlich fallen 8 Bytes für jeden Control Chunk an [RFC4960].
In Summe muss mindestens mit ein Overhead zwischen 76-96 Bytes pro Sendeaufruf gerechnet werden. Der Sendeaufruf erfolgt in einem Intervall von 100ms und damit zehnmal in der Sekunde, womit das Volumen des anfallenden Overheads auf 760-960 Bytes pro Sekunde anwächst.
Ein EKG Sample ist ein ganzzahliger Wert, der einem Wert zwischen 250 und -150 entsprechen kann. Ein Sample besteht damit aus einer Zeichenkette mit einer Länge zwischen einem und vier Zeichen. Ein Zeichen entspricht einem Byte, womit ein Sample eine Größe zwischen einem und vier Byte besitzt. Die zum Testen verwendete Textdatei, mit 400 generierten EKG Samples, enthält 780 Bytes Nutzdaten. Die Samples werden beim Senden durch Kommata getrennt, wodurch die Nutzdaten/s von 780 Bytes auf 1179 Bytes anwachsen. Analog wächst das Volumen der Nutzdaten pro Sendeaufruf von 78 Bytes auf etwa 118 Bytes an.
Für das Datenvolumen/s sind Overhead/s und Nutzdaten/s zu summieren, so dass ein Volumen von mindestens 1939 Bytes/s bis maximal 2139 Bytes/s zu erwarten ist.

| No. | Time | Source | Destination | Protocol | Length |
|---|---|---|---|---|---|
| 670 | 59.938021000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 671 | 60.039400000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 673 | 60.139830000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 279 |
| 677 | 60.240790000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 231 |
| 679 | 60.341772000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 680 | 60.442833000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 231 |
| 682 | 60.543784000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 683 | 60.644835000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 687 | 60.745816000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 688 | 60.846830000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 231 |
| 690 | 60.947629000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |
| 694 | 61.048901000 | 192.168.1.201 | 192.168.1.202 | DTLSv1.0 | 247 |

**Abbildung 6.1-44.:Wireshark 10 Pakete mit 40 Samples**

Wird das in Abbildung 6.1-44 protokollierte Datenvolumen von 10 Zusendungen, mit jeweils 40 Samples, innerhalb einer Sekunde summiert, entspricht dies einem Volumen von 2454 Bytes/s. Das gemessene Datenvolumen übersteigt den berechneten maximal Wert um 315Bytes/s, was etwa 32 Bytes pro Sendeaufruf ausmacht. Die DTLS Verschlüsselung des SCTP wurde nicht geöffnet, daher bleibt der Einblick auf dessen genutzte Chunks verwehrt, die die gemessene Differenz des Datenvolumens ausmachen können.

Allerdings zeigt die Messung, dass für die Übertragung von 1179 Bytes Nutzdaten mit zehn Senderaufrufen, 1275 Bytes Overhead anfallen, was sehr ineffizient ist. Die Ursache dafür ist aber nicht nur die Summe an Overhead der Protokolle des genutzten WebRTC DataChannel, sondern die kleinen Mengen an Nutzdaten, die jeweils versendet werden. Würden statt zehn nur zwei Sendeaufrufe genutzt werden, würden den 1179 Bytes Nutzdaten etwa 256 Bytes Overhead gegenüberstehen, was deutlich effizienter ist.
Zusätzlich zu dem Datenvolumen, welches durch die Übertragung der EKG Daten anfällt, müssen STUN Nachrichten mit einem Volumen von 150-300 Bytes/s berücksichtigt werden. Somit sollten beim Einsatz des EKG Daten Sender ein Datenvolumen von insgesamt 3 KB/s pro Verbindung zu einer Web-Anwendungsinstanz gerechnet werden.
Abbildung 6.1-45 zeigt einen Screenshot vom zehnminütigem Daten-Monitoring eines Routers über den die EKG Daten eines EKG Daten Senders, an eine Instanz der Web-Anwendung transportiert werden. Die EKG Daten Sender Anwendung und der Web-Browser in dem die Web-Anwendung ausgeführt wurden, befanden sich auf zwei verschiedenen PCs, die sich beide im gleichen Netzwerk befanden. Der in Orange dargestellte durchschnittliche Übertragungswert, lag bei 2,66KB/s, was dem gemessenen Volumen der EKG Daten plus den STUN Nachrichten entspricht.

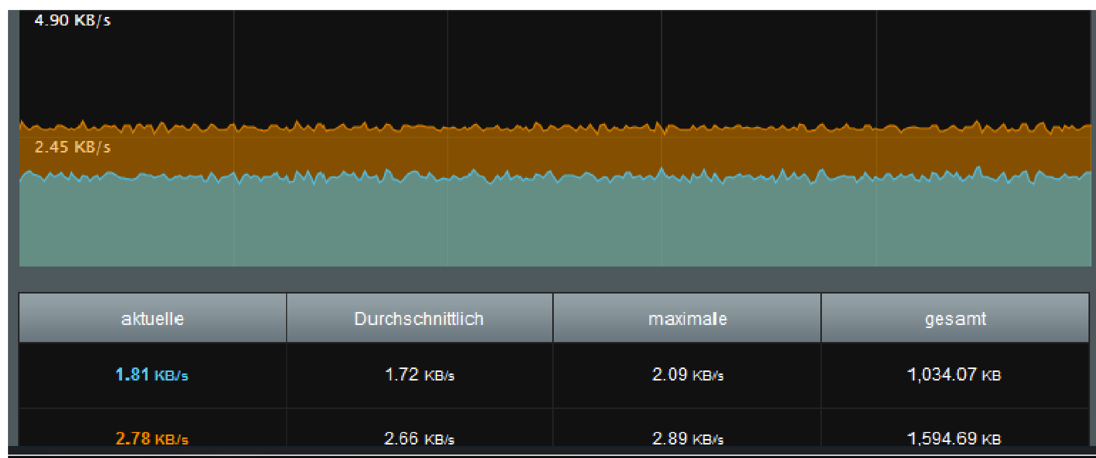


**Abbildung 6.1-45.: Upload Volumen über 10 Minuten**

In Abbildung 6.1-45 ist neben dem versendeten Datenvolumen auch das empfange Datenvolumen zu sehen, welches in Türkis dargestellt ist. Es handelt sich dabei um STUN Nachrichten und um SACK Messages vom SCTP, in denen dem sendenden EKG Daten Sender die empfangenen Sequenznummern bestätigt werden.
Der Datenverkehr vom Empfänger an den Sender wird in dieser Arbeit nicht untersucht, für das gewählte Szenario ist nur das Datenvolumen des Uploads eine kritische Größe ist.

### 6.1.1 Maximales Datenvolumen

Neben der Messung zum Datenvolumen, welches durch die Übertragung der EKG Daten eines EKG Daten Senders entsteht, wurde untersucht wie groß das maximale Datenvolumen der EKG Daten Sender ist. Die Messergebnisse in Abbildung 6.1.1-46 repräsentieren die Messungen beider EKG Daten Sender Lösungen, da diese identisch ausgefallen sind. Die Messung soll demonstrieren, dass beispielsweise die Übertragung der EKG Daten eines 12 Kanaligen EKGs problemlos möglich ist.
Das maximale Datenvolumen wurde in Abhängigkeit der Paketgröße und der Intervalldauer gemessen. Weiterhin bezieht sich die Messung auf eine RTCPeerConnection mit einem RTCDataChannel. Die Messergebnisse wurden bei einer Übertragung über das Internet ermittelt.

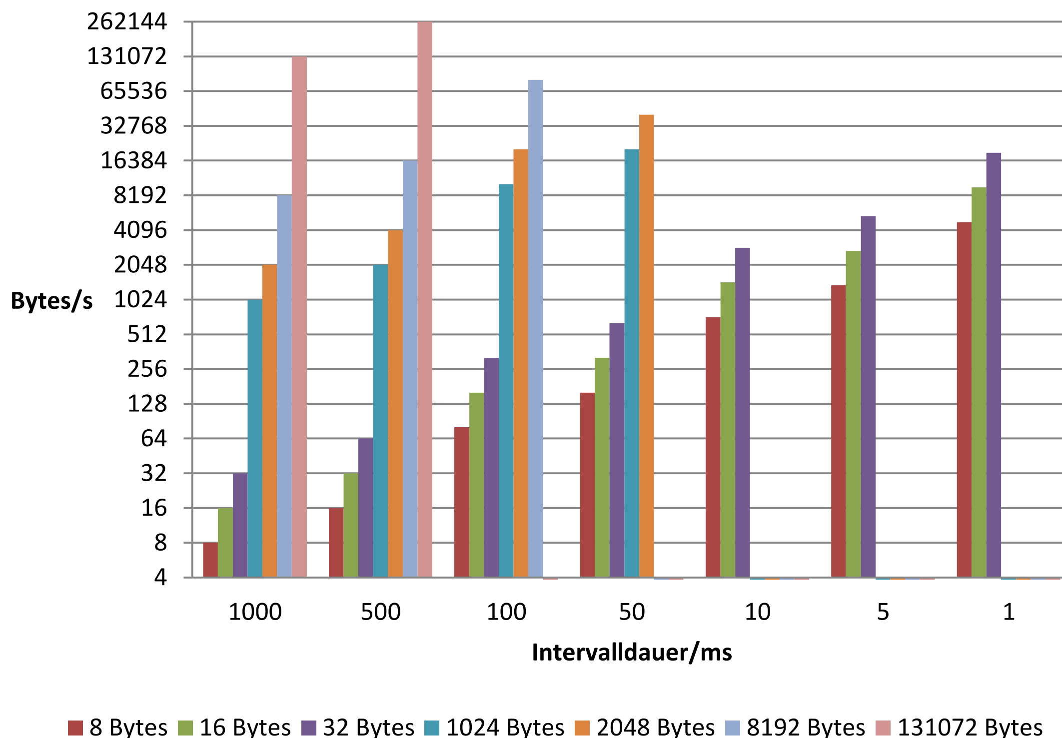



Abbildung 6.1.1-46.:Maximales Datenvolumen

Die Messung in Abbildung 6.1.1-46 zeigt welches Datenvolumen, mit verschiedenen großen Datenpaketen in Abhängigkeit der Intervalldauer, erreicht werden kann.

Eine Verwendung von Intervalldauern unter 50ms ist nicht empfehlenswert. An den Messwerten der Datenpakete mit Paketgrößen zwischen 8 Byte und 32 Byte, ist ein deutlicher Einbruch des gemessenen Datenvolums festzustellen. Der Einbruch ist auf die Anzahl an Datenpaket und dessen Verarbeitung zurückzuführen. Bleibt die hohe Paketanzahl über mehrere Sekunden bestehen, friert der Web-Browser ein und ist nicht mehr zu bedienen. Festgestellt wurde dieser Effekt bei der Verwendung von Chrome und Firefox. Bei Datenpaketen mit Paketgrößen zwischen 1024 und 131072 Bytes kann eine Verzögerung der Paketverarbeitung mit zunehmender Größe früher festgestellt werden.

## 6.2 Simultaner Betrieb von Peer-to-Peer Verbindungen

Im Rahmen eines Feldversuchs wurden die Lösungen des EKG Daten Senders getestet, wobei die Lösung für Web-Browser mit beiden Referenz Web-Browsern Chrome und Firefox getätigt wurden. Bei dem Feldversuch waren der EKG Daten Sender und der Server, der die Web-Seite der Web-Anwendung und den WebSocket-Server hostet, in einem Netzwerk an Standort A.
An Standort B, der in Abbildung 6.2-47 zu sehen ist, wurden an sieben PCs Instanzen der Web-Anwendung geöffnet, wobei bis auf einen PC, auf allen zwei Instanzen des Web-Browser gestartet wurden, so dass bis zu 13 Instanzen der Web-Anwendung parallel ausgeführt wurden.
Der Feldversuch dient der Untersuchung von zwei Punkten. Der erste Punkt dient der Validierung des in Kapitel 6.1 gemessen Datenvolumen einer Peer-to-Peer Verbindung, über die EKG Daten gesendet werden.
Der zweite Punkt ist eine Untersuchung die zeigen soll, wie viele Instanzen der Web-Anwendung parallel von einem EKG Daten Sender mit EKG Daten versorgt werden können. Dabei wurde die Dauer gemessen, die die EKG Daten Sender Anwendung benötigen, um die Funktion zum Senden der EKG Daten abzuarbeiten, die in einem Intervall von 100ms aufgerufen wird.

**Abbildung 6.2-47.: Test im HITeC e.V. 3S Labor**

Dazu wurden schrittweise zusätzliche Instanzen der Web-Anwendung gestartet, die die EKG Daten des EKG Daten Senders empfangen, um das Verhalten der EKG Daten Sender Lösungen zu testen und das benötigte Datenvolumen zu messen.

### 6.2.1 Datenvolumen mehreren Peer-to-Peer Verbindungen

Das in Abbildung 6.2.1-48 dargestellte Diagramm, zeigt den Anstieg des Datenvolumens, mit dem Anstieg der Anzahl von Instanzen der Web-Anwendung, die die EKG Daten eines EKG Daten Senders beobachten.
Die Ergebnisse der drei EKG Daten Sender Variationen sind sehr ähnlich. Die leichten Abweichungen der Ergebnisse sind zu vernachlässigen, da die Messungen mit dem Datenvolumen Monitor eines Routers im Netzwerk der EKG Daten Sender getätigt wurden. Die Ergebnisse zeigen, dass ein Datenvolumen von 3KB/s pro Peer-to-Peer Verbindung einer Web-Anwendungsinstanz ausreichend und realistisch sind. Der Anstieg des Datenvolumens mit der steigenden Anzahl an Instanzen der Web-Anwendung beschreibt ein lineares Verhalten, welches die Unabhängigkeit der einzelnen Peer-to-Peer Verbindungen zeigt.

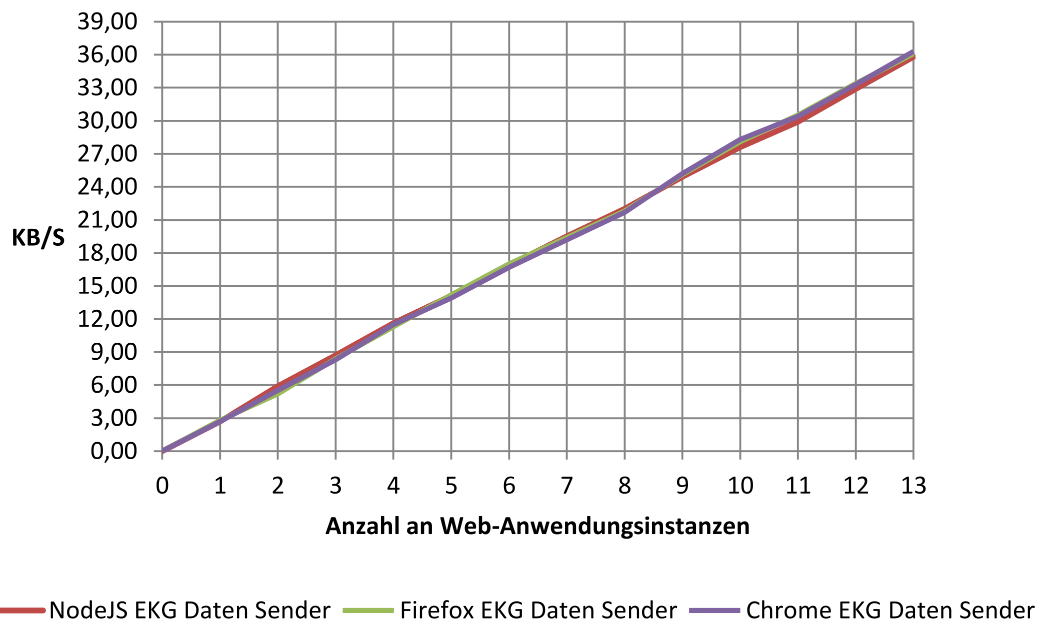



Abbildung 6.2.1-48.: Datenvolumen beim Senden an mehrere Instanzen der Web-Anwendung

### 6.2.2 Auslastung der EKG Daten Sender Lösungen

Im Rahmen des Feldversuchs wurde gemessen, wie mit der steigenden Anzahl an Peer-to-Peer Verbindungen, an die der EKG Daten Sender die EKG Daten sendet, die Auslastung des EKG Daten Senders zunimmt.
Die EKG Daten Sender Anwendung sendet in einem Intervall von 100ms EKG Daten über jeden verbundenen RTCDataChannel. Es wurde untersucht, wie viel Zeit für das Senden an alle RTCDataChannel, von den Lösungen des EKG Daten Senders benötigt wird.
Die Ergebnisse sind in Abbildung 6.2.2-49 dargestellt und zeigen die maximale Dauer, die bei der Abarbeitung des Sendevorgangs gemessen wurde.

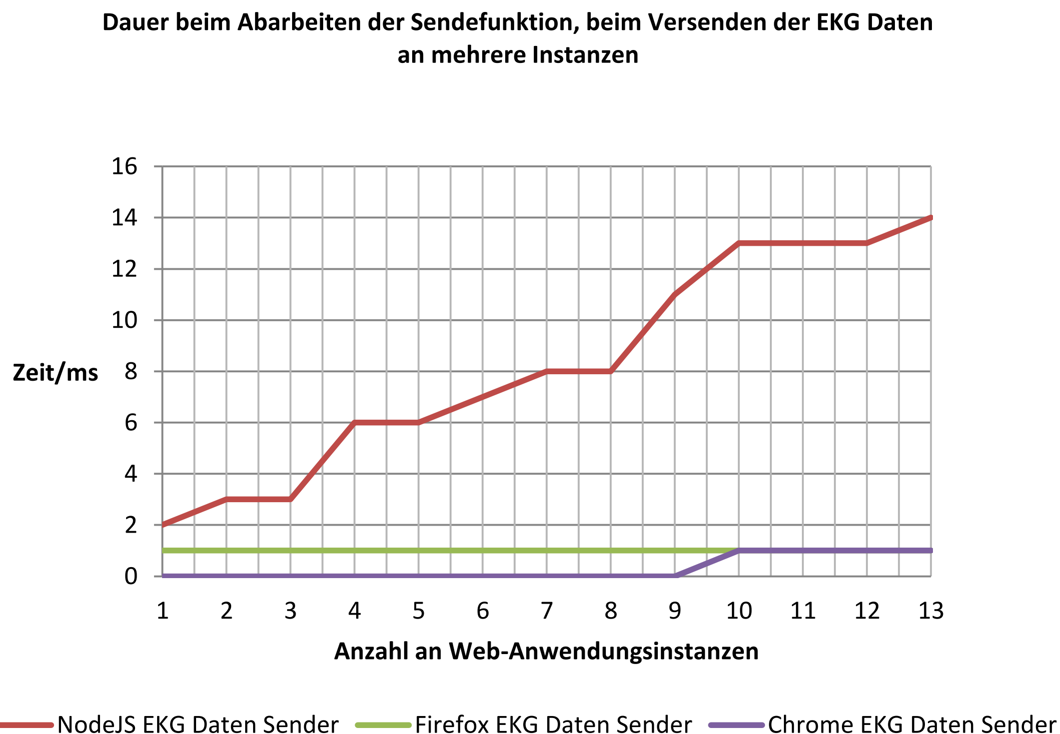



**Abbildung 6.2.2-49.: Zeit beim Versenden der EKG Daten**

Die Lösung für NodeJS hebt sich deutlich von den Lösungen für die Web-Browser Chrome und Firefox ab, wobei die im Diagramm nicht dargestellt durchschnittliche Dauer geringer ist als es die dargestellten Maximalwerte vermuten lassen. Allerdings ist die maximale Laufzeit ausschlaggebend, ob die Abarbeitung innerhalb des Intervalls und damit unkritisch umgesetzt wird oder nicht. Da die Dauer auch vom System abhängt auf dem der EKG Daten Sender betrieben wird, dienen die gemessenen Werte nur als Vergleich zwischen den realisierten Lösungen und geben keine absoluten Werte wieder.

Das Verhalten von Chrome und Firefox ist sehr identisch. Auffällig ist, dass bei der Verwendung von Firefox immer einer 1ms zum Senden benötigt wird.
Insgesamt zeigt die Messung, dass die Web-Browser effektiver arbeiten als die Lösung für NodeJS, was mit steigender Last deutlicher wird.

## 6.3 Datenfluss unter Auslastung des Uploads

Abschließend wurde im Feldversuch getestet, wie sich die Auslastung des Upload, des Internetanschlusses über den der EKG Daten Sender angebunden ist, auf den Datenfluss auswirkt. Der folgende Abschnitt zeigt einen Ausschnitt, der Informationen aus dem Konsolenfenster.

| | | |
|---|---|---|
| | 11:40:35:499 | webRTC CONNECTED to Herr Mustermann |
| [0] | 11:40:40:505 | Reduce Buffersize |
| [0] | 11:40:47:516 | Reduce Buffersize |
| | 11:40:49:404 | ICE connection state change: completed |
| [0] | 11:40:52:522 | Reduce Buffersize |
| [1] | 11:43:37:244 | buffering ... |
| [2] | | … |
| [3] | 11:47:58:957 | buffering ... |
| [4] | 11:47:59:519 | buffer overflow: 158 |
| [5] | 11:47:59:520 | Enlarge Buffersize |
| [6] | 11:48:04:156 | Reduce Buffersize |
| [7] | 11:48:18:178 | Reduce Buffersize |
| [8] | 11:48:22:318 | buffering ... |
| | 11:48:31:038 | buffering ... |
| [9] | 11:48:31:621 | buffer overflow: -5 |
| | 11:48:31:622 | Enlarge Buffersize |
| | 11:48:35:204 | Reduce Buffersize |
| | 11:48:46:221 | Reduce Buffersize |
| | 11:48:54:232 | Reduce Buffersize |

Nach dem Aufbau der Verbindung, konnte der Buffer in mehreren Schritten auf eine Größe von 400 Samples reduziert [0] werden. Mit der Auslastung des Uploads wurde die Übertragung des RTCDataChannel für wenige Sekunden blockiert, wodurch der Buffer leer lief [1]. Die Daten hingen in der Warteschlange des Routers fest und sind anschließend schubweise versendet worden. Das Senden in Schüben dauerte einige Minuten an [2]. In dieser Zeit wurde der Buffer durch jeden Datenschub gefüllt und ist anschließend wieder leergelaufen [3]. Danach trat ein Datenschub auf, der den Buffer zum Überlaufen brachte [4], worauf der Buffer vergrößert wurde [5]. Der Buffer konnte im Anschluss wieder reduziert werden [6], was nach der in Kapitel 5.2.2 beschriebenen Arbeitsweise der Bufferregulierung bedeutet, dass die Schübe kleiner und regelmäßiger ankamen. Allerdings gab es weiterhin Verzögerungen und Datenschübe zwischen denen der Buffer leer lief [8] und die den Buffer zum Überlauf [9] gebracht haben.

Der Datenfluss wurde während der Auslastung des Uploads sehr beeinträchtigt, so dass dieser nur noch in Datenschüben ankam, die durch Empfangspausen unterbrochen wurden. Die Bufferregulierung ist für das Szenario nicht passend abgestimmt und hat den Buffer zu

schnell verkleinert, so dass dieser weder die Datenschübe auffangen konnte, noch genug Daten aufgenommen hat, um die Empfangspausen zwischen den Datenschüben auszugleichen.
Datenverlust kann bei Auslastung des Uploads und mit der in Kapitel 5.2.2 beschriebenen Konfiguration des RTCDataChannel nicht ausgeschlossen werden, da nur ein Wiederversenden innerhalb von 200ms erlaubt ist.

# 7 Schlussfolgerung

In Kapitel 7.1 ist zusammengefasst, welche Arbeitsschritte für die Umsetzung dieser Bachelorarbeit durchgeführt wurden. In Kapitel 7.2 wird betrachtet, ob und wie die gesetzten Ziele erreicht wurden. Eine abschließende Bewertung dieser Bachelorarbeit erfolgt im Kapitel 7.3. Abschließend werden in 7.4 mögliche Ausblicke für die in dieser Bachelorarbeit realisierte Web-Anwendung vorgestellt.

## 7.1 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurden der DataChannel des, sich noch in der Spezifikation befindlichen, WebRTC Framework und das moderne WebSocket Protokoll vorgestellt, dabei wurden die Arbeitsweisen beschrieben und Vorteile präsentiert. In Kapitel 2 wurden dazu die Spezifikationen zusammengefasst und beispielgestützt erklärt, so dass ein für die Realisierung nötiges Grundverständnis verinnerlicht werden kann.
Um die gebotenen Möglichkeiten in einer Realisierung nutzen und umsetzen zu können, wurde in Kapitel 3 ein Szenario definiert, für dass eine Web-Anwendung entwickelte werden soll. Mit der Definition des Szenarios wurden Anwendungsfälle vorgestellt, von denen funktionale und nicht funktionale Anforderungen für die zu entwickelnde Realisierung der Web-Anwendung abgeleitet werden konnten.
Unter Berücksichtigung der Zielsetzung, des definierten Szenarios und den abgeleiteten Anforderungen, wurde in Kapitel 4 ein Design präsentiert, welches den Aufbau und das Zusammenspiel der einzelnen systemrelevanten Komponenten beschreibt.
Die Realisierung der im Design präsentierten Komponenten der zu entwickelnden Web-Anwendung erfolgt in Kapitel 5. Dabei werden die entwickelten Komponenten vorgestellt und die Arbeitsweisen weitreichend beschrieben.
Im Anschluss an die Realisierung des Systems, welches es Benutzern ermöglicht über eine Web-Anwendung, Daten von entfernt eingesetzten Datensender simulierenden Anwendungen, über eine verschlüsselte Peer-to-Peer Verbindung in Echtzeit zu beobachten, wurden dieses System in einem Feldversuch getestet. Der Feldversuch in Kapitel 6 wurde genutzt, um das benötigte Datenvolumen zu untersuchen, da dieses für das gewählte Szenario eine wichtige Größe ist, und um die zwei Lösungen der Datensender simulierenden Anwendung zu vergleichen.

## 7.2 Reflexion

Die für diese Bachelorarbeit gewählten Ziele, das WebSocket-Protokoll und das WebRTC Framework vorzustellen, um anschließend eine praktische Entwicklung mit dessen Einsatz zu realisieren, konnte weitgehend erfüllt werden.
Die Realisierung der Web-Anwendung bot eine umfangreiche und interessante Möglichkeit, die Fähigkeiten des WebSocket-Protokolls und des WebRTC Frameworks, abseits einer rein für Web-Browser ausgelegten Implementierung, zu benutzten.
Das breite Themengebiet korreliert mit einem Rechercheaufwand in vielen Quellen, die weniger in Form von klassischer Literatur vorhanden sind, insbesondere trifft dies beim WebRTC Framework zu. Die Breite der Themengebiete ist Grund dafür, dass im Rahmen dieser Bachelorarbeit nicht alle Merkmale zu jedem Thema detailliert behandelt werden konnte. So wurde im Fall des WebRTC Frameworks nur der Einsatz des RTCDataChannel genauer diskutiert, welcher praktische Anwendung in der Realisierung fand.
Es musste leider auch der primäre Ansatz, einen Datensender für einen ARM basierte SOC zu entwickeln, verworfen werden. Folglich wurden auch beim Design und der darauf aufbauenden Realisierung abstriche gemacht. Die Gründe dafür waren vor allem, die begrenzte Zeit die für eine Bachelorarbeit zur Verfügung steht und die vorliegenden Informationen zur Entwicklung nativer WebRTC Implementierungen [WebRTCorg]. Diese waren Ende des Jahres 2014 unübersichtliche und unvollständig, zusätzlich war es nötig entsprechende Bibliotheken aus Googles Chromium Repository zu beziehen, für dessen Inhalt keine ausreichende Dokumentation vorlag und so eine Realisierung deutlich erschwerte:

> *In theory I could build google's WebRTC code drop and cross compile it onto the Pi after removing all the audio and video dependencies and the browser assumptions. Judging from the grumbling I hear from everyone who is going down that route, I wouldn't enjoy it and it would be a lot of work.*
>
> \- Tim Panton [HACKS]

Dabei darf nicht vergessen werden, dass sich das WebRTC Framework auch gegenwärtig noch in der Spezifikation befindet und weiterentwickelt wird, so wurden mittlerweile die Informationen zur Entwicklung nativer WebRTC Implementierungen überarbeitet, ob diese nun ausreichend sind kann an dieser Stelle allerdings nicht beurteilt werden.
Im Nachhinein kann die Arbeit am Thema dieser Bachelorarbeit als intensiv, interessant, lehrreich und vielfältig beschrieben werden, da alle folgenden Schlagwörter Berührungspunkte im Verlauf der Arbeit waren:
WebSocket-Protokoll, WebRTC Framework, Web-Anwendung, Web-Seite, Web- und WebSocket- Server-Hosting, Java, HTML, JavaScript und NodeJS.

## 7.3 Fazit

Die Erkenntnisse, die während der Ausarbeitung dieser Bachelorarbeit und bei der Entwicklung des WebRTC basierten Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung gewonnen wurden, lassen Empfehlungen für das verwendete WebSocket-Protokoll und WebRTC Framework zu.
Die Möglichkeiten die das WebSocket-Protokoll und das WebRTC Framework bieten, wurden in dieser Arbeit nicht ausgeschöpft, trotzdem sind die Möglichkeiten die mit dem entwickelten System erreicht werden beachtlich. So können wie vom Entwicklungsszenario vorgegeben, Benutzer der Web-Anwendung die EKG Daten von einem entfernten EKG Daten Sender über eine verschlüsselte Peer-to-Peer Verbindung in Echtzeit beobachten.

Allerdings gibt es auch Punkte der entwickelten Lösung für das Entwicklungsszenario, die nicht ohne Verbesserungsbedarf auskommen.
Die entwickelten Lösungen des EKG Daten Senders sind nicht für einen Einsatz in der realen Welt einzusetzen. Mit Hilfe der EKG Daten Sender Lösungen lässt sich das System als Ganzes testen aber nicht realistisch betreiben. Es fehlt eine Schnittstelle zu einem EKG, dessen Daten in Echtzeit verarbeitet werden können. Zusätzlich wäre eine Implementierung eines EKG Daten Senders, die auf einem kostengünstigen SOC lauffähig ist erstrebenswert, da die aktuellen Lösungen nur auf x86 Systemen eingesetzt werden können.
Der im Rahmen dieser Bachelorarbeit nicht bereitgestellte TURN-Server verhindert zudem die Nutzung des EKG Daten Senders in den meisten Mobilfunknetzen und anderen Netzen, in denen es aufgrund eines symmetrischen NAT oder einer Firewall nicht möglich ist, eine Peer-to-Peer Verbindung herzustellen.
Auf Seiten der Web-Anwendung ist eine TLS Verschlüsselung der WebSocket-Verbindungen für einen realen Einsatz unumgänglich, da ohne Verschlüsselung alle Messages durch Sniffing mitgelesen werden können, womit auch die EKG Daten Sender gefährdet sind.

Abseits des gewählten Entwicklungsszenarios kann das entwickelte System durchaus sinnvoll eingesetzt werden, wobei es dann dem neuen Einsatzzweck angepasst werden muss, etwa bei der Visualisierung der übertragenen Daten.
Durch die entwickelte NodeJS EKG Daten Sender Lösung, gibt es eine Datensenderanwendung, die auf Windows, Linux und Mac OS X Systemen ohne einen Web-Browser genutzt werden kann. Mit Hilfe der NodeJS Lösung kann das entwickelte System plattformübergreifend genutzt werden, womit es als bedingt einsatzfähig beschrieben werden kann.

## 7.4 Ausblick

Es ist zu erwarten, dass zukünftig mehr Web-Anwendungen entwickelt werden, die auf das WebSocket-Protokoll anstatt HTML setzen und das WebRTC Framework verwenden, sobald die Spezifikation abgeschlossen ist. Die Vorteile sind zu gewichtig und die Möglichkeiten zu umfangreich, um sie in einer Zeit ignorieren zu können, in der immer mehr Anwendungen ins Web verlagert werden. Anfänge sind unter anderem bei Mozillas Firefox Hello [HELLO] oder auch bei Microsofts Plänen für Skype [SKYPE] erkennbar.

Auch das in dieser Bachelorarbeit entwickelte WebRTC basierte Peer-to-Peer Echtzeitdaten-Managementsystem mit Browser unterstützter Visualisierung bietet ein großes Potential für Verbesserungen und Weiterentwicklungen für das gewählte Anwendungsszenario.
Die Entwicklung eines WebRTC Peer für einen kostengünstigen SOC würde das System deutlich attraktiver machen, da das Einsatzgebiet massiv vergrößert werden würde.
Die Integration eines TURN-Servers würde es ermöglichen, die EKG Daten Sender auch im Netz eines Mobilfunkbetreibers einzusetzen.
Eine WebRTC basierende Implementierung eines Videochats, statt des implementierten Chats auf Basis des WebSocket-Protokolls, wäre in manchen Anwendungsbereichen eine sinnvolle Erweiterung.
Eine Änderung im Verbindungsverhalten, bei dem alle EKG Daten Sender nach dem Start eine Peer-to-Peer Verbindung aufbauen, bringt neue Möglichkeiten mit sich. So könnte die Web-Anwendung viele EKGs in geringerer Auflösung darstellen, auch eine automatisierte Erkennung von Unregelmäßigkeiten der EKG Werte könnte so implementiert werden.
Die Implementierung eines Ärzte-Bereichs in der Web-Anwendung, in dieser eine EKG Kurve nur dann dargestellt wird, wenn dies durch einen anderen Benutzer der Web-Anwendung bei Bedarf ausgelöst wird, bietet interessante Möglichkeiten. Ob die EKG Daten dann über eine separate Peer-to-Peer Verbindung vom EKG Daten Sender oder vom Web-Browser der auslösenden Web-Anwendungsinstanz kommen, könnte beispielsweise vom verbleibenden Volumen des Upload auf Seiten des EKG Daten Senders abhängig gemacht werden.
Abseits des für diese Bachelorarbeit gewählten Anwendungsszenarios könnte das System für andere Arten von zu visualisierenden Daten weiterentwickelt werden.

Es bleiben viele interessante Möglichkeiten, was vor allem daran liegt, dass Entwickler, durch den Einsatz von modernen Frameworks und Protokollen, viele Möglichkeiten geboten bekommen, die nur darauf warten für eine gute Idee verwendet zu werden.

# A. Anhang

## A.1 WebSocket-Protokoll

### A.1.1 Detailliertes Bespiel zum Verbindungsaufbau

Im Folgenden Abschnitt wird der Aufbau einer WebSocket-Verbindung detailliert beschrieben. Header-Felder die im Kapitel 2.1.5 aus Gründen der Übersicht vernachlässigt wurden, werden nun beschrieben und erklärt. Einige Erklärungen wurden bereits in vorherigen Kapiteln verwendet, werden aber für den zusammenhängenden Kontext an dieser Stelle wiederholt.

**Clientseitiger Handshake beim Verbindungsaufbau**

Der Handshake beim Verbindungsaufbau geht vom Client aus. Soll eine sichere WebSocket-Verbindung zum Server hergestellt werden, muss der Client vor dem Senden des WebSocket-Handshakes einen TLS-Handshake [RFC2818] durchführen. Um den WebSocket-Handshake zu initiieren, sendet der Client ein HTTP Upgrade Request. In der Anfrage gibt es nach der ersten Zeile sogenannte Header-Fields bzw. Header-Felder [RFC2616], in denen alle für den Handshake nötigen und optionalen Angaben angegeben werden. Die Reihenfolge der Header-Felder ist unbedeutend und kann abhängig der Implementation variieren. Die Abbildung A.1.1-50 entspricht einem Beispiel für einen clientseitigen Handshake, womit die WebSocket-Verbindung zum Server eingeleitet wird:

1. GET /chat HTTP/1.1
2. Host: server.example.com
3. Upgrade: websocket
4. Connection: Upgrade
5. Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6. Origin: http://example.com
7. Sec-WebSocket-Protocol: chat, superchat
8. Sec-WebSocket-Version: 13
9. Sec-WebSocket-Extensions: private-extension, deflate-stream; mux

**Abbildung A.1.1-50.: Darstellung einer clientseitigen Handshake-Einleitung, aus [RFC6455]**

1. Die erste Zeile der Anfrage, die sogenannte „Leading Line“, entspricht dabei dem Request-Line Format [RFC2616]. Sie beinhaltet die „Request URI“ [RFC2616] mit Angabe des serverseitigen Endpoints, in diesem Fall ist es der Endpoint „/chat“. Die HTTP Version muss mindestens der Version 1.1 entsprechen.
2. In der zweiten Zeile des obigen Beispiels ist das Header-Feld „Host“ angegeben. Darin wird der Hostname des Servers angegeben. Optional kann der Hostname mit der Angabe eines Ports ergänzt werden. Der Server kann so prüfen, ob der empfangene Handshake tatsächlich an ihn gesendet werden sollte.
3. Als dritte Angabe im obigen Beispiel folgt das Feld „Upgrade“, welches das Schlüsselwort „websocket“ enthalten muss. Dem Server wird dadurch mitgeteilt, dass der Client eine WebSocket-Verbindung herstellen möchte.
4. Im obigen Beispiel wird daraufhin als viertes das Header-Feld „Connection“ aufgeführt. Darin ist das „Upgrade“-Token [RFC2817] enthalten. An dieser Stelle wird deutlich, dass eine WebSocket-Verbindung aktuell via HTTP Upgrade Request hergestellt wird.
5. In der fünften Zeile des obigen Beispiels steht das Header-Feld „Sec-WebSocket-Key“. Es enthält einen vom Client zufälligen gewählten Base64 codierten 16-Byte-Wert. Das Header-Feld wird genutzt, um sicherzustellen, dass die Handshake-Anfrage des Clients von einem WebSocket Server beantwortet wird. Dies verhindert, dass der Server durch ein Handshake-Paket korrumpiert wird, welches einen eingebetteten XMLHttpRequest [XMLHttpRequest] oder ein ausführbares Formular beinhaltet. Wie der Server den Inhalt des Header-Felds interpretiert, wird im Kapitel „Verarbeitung des clientseitigen Handshake“ beschrieben.
6. Die sechste Zeile des obigen Beispiels zeigt das optionale Header-Feld „Origin“ [RFC6454]. Es beinhaltet den Hostname des Handshake initiierenden Clients, also die Herkunft des Handshakeinitiators. Dieses Header-Feld wird vor allem dazu genutzt, um den Server vor unerlaubten skriptbasierten „cross-origin“ Anfragen

eines Web-Browsers zu schützen, da Browser dieses Feld im Gegensatz zu nativen Clients immer angeben.

7. Das optionale Header-Feld „Sec-WebSocket-Protocol“ ist im obigen Beispiel in Zeile Sieben angegeben. Es dient zur Auflistung von Unterprotokollen, die vom Client unterstützt werden. Dabei handelt es sich um Unterprotokolle auf der Anwendungsebene zur Kommunikation mit dem angegebenen Endpunkt. Werden mehrere Unterprotokolle angegeben, spiegelt die Reihenfolge die Priorisierung des Client wieder.
8. Das Header-Feld „Sec-WebSocket-Version“ in Zeile Acht des obigen Beispiels wird zur Angabe der genutzten Version des WebSocket-Protokolls genutzt. Der Wert des Header-Feldes muss aktuell „13“ sein. Dies entspricht der aktuell als Standard definierten WebSocket-Protokoll Version.
9. Die letzte Zeile des obigen Beispiels, eines initiierenden clientseitigen Handshake zum Herstellen einer WebSocket-Verbindung, zeigt das optionale Header-Feld „Sec-WebSocket-Extensions“. In diesem Header-Feld kann der Client Protokoll-Erweiterungen angeben, die er für die Kommunikation nutzen will. Wenn, wie im obigen Beispiel mehrere Erweiterungen angegeben werden, ist zu beachten, dass die Reihenfolge die Priorität angibt. Die Erweiterungen können durch Kommata oder Semikolons getrennt oder das Header-Feld mehrfach angegeben werden, mit jeweils nur einem Teil von Erweiterungen.

Im Handshake können noch zusätzliche Header-Felder [RFC6265] genutzt werden, beispielsweise für Verwendung von Cookies [RFC2616] oder zur Authentifizierung.
Nachdem der Client den initiierenden Teil des Handshakes zum Aufbau einer WebSocket-Verbindung an den Server gesendet hat, muss er auf dessen Antwort warten bevor er weitere Daten senden darf.

**Serverseitige Handshake-Antwort beim Verbindungsaufbau**

***Verarbeitung der clientseitigen Handshake Einleitung***

Noch bevor der empfangene Handshake vom WebSocket-Server analysiert und eine Handshake-Antwort erstellt wird, ist es entscheidend, auf welchem Port der clientseitige Handshake empfangen wurde. Ist der Handshake auf einem HTTPS Port, wie etwa Port 443 empfangen worden, dann muss der WebSocket-Server als erstes einen TLS-Handshake ausführen und sämtliche folgende Kommunikation mit dem Client durch den verschlüsselten Tunnel erfolgen [RFC5246], was in dem obigen Beispiel aber nicht berücksichtigt wird.
Wenn ein WebSocket-Server den clientseitigen Handshake aus dem Beispiel des Kapitels „Clientseitiger Handshake beim Verbindungsaufbau“ erhalten hat, muss der WebSocket-Server die Angaben aus den Header-Feldern prüfen. Die Prüfung erfolgt bevor der Server mit einer generierten akzeptierenden Handshake-Antwort antwortet und eine WebSocket-Verbindung hergestellt wird. Da im obigen Beispiel alle nötigen Header-Felder mit

zulässigen Werten angegeben wurden, kommt es nicht zum Abbruch des Handshake bzw. des Verbindungsaufbaus.
Im Folgenden wird die Überprüfung der empfangenen clientseitige Anfrage Zeile für Zeile beschrieben, wie Abbildung A.1.1-50 dargestellt:

1. Bei der Überprüfung der ersten Zeile wird geprüft, ob eine HTTP/1.1 GET Anfrage mit einem URI vorliegt, der interpretiert bzw. einem vorhandenen Endpunkt zugeordnet werden kann.
2. In der zweiten Zeile steht das Header-Feld „Host". Der Server prüft ob die Angabe im Feld seiner eigenen Host Angabe entspricht und die Anfrage somit am vorgesehen Ziel angekommen ist.
3. Anschließend folgt in Zeile drei, das Header-Feld „Upgrade". Der Inhalt des Felds muss den ASCII-Wert [ANSI_X3_4_1986] „websocket" beinhalten, wobei der Wert unempfindlich ist was die Groß- und Kleinschreibung betrifft.
4. Als viertes wird dann das Header-Feld „Connection" geprüft, ob dort der „Upgrade" Token [RFC2817] enthalten ist.
5. In der fünften Zeile steht das Header-Feld „Sec-WebSocket-Key". Der Server prüft, ob wie erwartet ein Base64 [RFC464] codierter Wert enthalten ist, der decodiert 16 Bytes lang ist. Dieses Header-Feld spielt für die serverseitige Handshake-Antwort noch eine wichtige Rolle, welche folgend in diesem Kapitel beschrieben wird.
6. Als nächstes liest der Server das Header-Feld „Origin" [RFC6454]. Eine Überprüfung der Herkunft findet nur statt, wenn der Entwickler dies vorsieht. Beispielsweise um die Herkunft eines Clients anhand einer White List zu verifizieren. Genauere Informationen folgen im Kapitel „Sicherheitsmodell".
7. Anschließend folgt das optionale Header-Feld „Sec-WebSocket-Protocol" in Zeile Sieben. Der Server liest die vom Client vorgeschlagenen Unterprotokolle und prüft, ob er dem Wunsch des Client nachkommen kann und eines der Unterprotokolle unterstützt.
8. Der Server prüft in der achten Zeile, den Inhalt des Header-Feld „Sec-WebSocket-Version", ob dieser dem Wert „13" entspricht.
9. In der neunten und letzten Zeile des empfangenen Handshakes, liest der Server das optionale Header-Feld „Sec-WebSocket-Extensions". Der Server prüft ob eins oder sogar mehrere Protokoll-Erweiterungen zur Kommunikation mit dem Client genutzt werden können.

Mögliche weitere Header-Felder [RFC6265] für Cookies oder zu Authentifizierungszwecken sind im Handshake-Beispiel des Kapitels „Clientseitiger Handshake beim Verbindungsaufbau" nicht angegeben.

***Erstellung der serverseitigen Handshake Antwort***

Nach dem der WebSocket-Server den Inhalt des empfangenen clientseitigen Handshake geprüft hat, wird nun eine Handshake-Antwort für den Client erstellt. Eine Handshake-Antwort könnte beispielsweise wie in Abbildung A.1.1-51 aussehen:

1. HTTP/1.1 101 Switching Protocols
2. Upgrade: websocket
3. Connection: Upgrade
4. Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5. Sec-WebSocket-Protocol: chat
6. Sec-WebSocket-Extensions: deflate-stream, mux

**Abbildung A.1.1-51..: Darstellung einer serverseitigen Handshake-Antwort, aus [RFC6455]**

1. Die erste Zeile bzw. „Leading-Line“ ist eine HTTP-Status-Line [RFC2616]. Diese enthält den Statuscode „101“ und dem entsprechenden Hinweis Switching Protocols. Jeder andere Statuscode würde bedeuten, dass der WebSocket-Handshake zur Verbindungsherstellung nicht abgeschlossen ist.
2. In der zweiten Zeile wird in das Header-Feld „Upgrade“, der Wert „websocket“ gesetzt.
3. Gefolgt von der dritten Zeile, in der das Header-Feld „Connection“ den Wert „upgrade“ erhält. Diese beiden Angaben aus Punkt 2 und 3 vervollständigen das HTTP-Upgrade auf das WebSocket-Protokoll.
4. Das Header-Feld „Sec-WebSocket-Accept“ erhält den Wert "s3pPLMBiTxaQ9kYGzzhZRbK+xOo=". Dies bedarf einer genaueren Erläuterung:

   Um dem Client zu beweisen, dass der Handshake von einem WebSocket-Server empfangen wurde, muss der Server zwei Teilinformationen kombinieren und daraus eine Antwort für den Client bilden:

Sec-WebSocket-Key = „dGhlIHNhbXBsZSBub25jZQ==“
GloballyUniqueIdentifier = „258EAFA5-E914-47DA-95CA-C5AB0DC85B11“
Sec-WebSocket-Accept = (Base64( SHA-1(Sec-WebSocket-Key + GloballyUniqueIdentifier)))
Sec-WebSocket-Accept = „s3pPLMBiTxaQ9kYGzzhZRbK+xOo=“

Diese wird vom Server in das Header-Feld „Sec-WebSocket-Accept“ geschrieben. Der erste Wert stammt aus dem Header-Feld „Sec-WebSocket-Key“, des empfangenen clientseitigen Handshake. Dieses beinhaltet den Wert:

"dGhlIHNhbXBsZSBub25jZQ=="

Als Zweite Teilinformation dient der Globally Unique Identifier [RFC4122], mit dem Wert:

"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

Der Server verkettet beide Teilinformationen zu einem neuen String:

"dGhlIHNhbXBsZSBub25jZQ==258EAFA5-E914-47DA-95CA-C5AB0DC85B11"

Der Server erzeugt aus dem kombinierten String, einen 20 Bytes langen SHA-1 Hash [FIPS_180_3], der dann wie folgend aussieht:

„0xb3 0x7a 0x4f 0x2c 0xc0 0x62 0x4f 0x16 0x90 0xf6 0x46 0x06 0xcf 0x38 0x59 0x45 0xb2 0xbe 0xc4 0xea“

Der Server codiert den erzeugten Hash Base64. Daraus resultiert der Wert:

"s3pPLMBiTxaQ9kYGzzhZRbK+xOo="

Diesen Wert kann der Server nun in seiner Handshake-Antwort in das Header-Feld „Sec-WebSocket-Accept“ setzen.

5. Da im Beispiel des clientseitigen Handshakes das Header-Feld „Sec-WebSocket-Protocol“ aufgeführt war, wird es vom Server ausgelesen. Der Server kann in seiner Handshake-Antwort ein vom Client genanntes Unterprotokoll für die Kommunikation bestätigen. Voraussetzung dafür ist, dass der Server eines der genannten Unterprotokolle unterstützt. In diesem Bespiel unterstützt der Server das Unterprotokoll „chat“. In der fünften Zeile gibt er dies entsprechend im Header-Feld „Sec-WebSocket-Protocol“ an.
   Würde der Server das Unterprotokoll nicht unterstützen, gibt es zwei Möglichkeiten, dies an den Client zurückzumelden. Die erste wäre, dass statt dem Wert „chat“, NULL in das Header-Feld eingetragen wird. Die zweite Möglichkeit ist, dass Header-Feld gar nicht in der Handshake-Antwort an den Client aufzuführen.
6. In der sechsten Zeile werden zwei Erweiterungen, getrennt durch ein Komma, vom Server im Header-Feld „Sec-WebSocket-Extensions“ bestätigt. Die Reihenfolge ist, wie auch schon beim clientseitigen Handshake, zu beachten. Da zwei Erweiterungen angegeben werden, werden diese nacheinander genutzt. Die Reihenfolge der Ausführung wird durch die Reihenfolge im Header-Feld angegeben.

Dies bedeutet, dass wenn die Erweiterungen beispielsweise auf Kommunikationsdaten angewandt wird, die Daten dann zuerst durch die Erweiterung „deflate-stream“ verändert, anschließend die veränderten Daten an die Erweiterung „mux“ gelangen und nun von dieser beeinflusst werden. Zur besseren Verständlichkeit, folgendes Bespiel:

Gibt der Server zwei Erweiterungen im Header-Feld an, dann werden Operationen durch die angegebenen Erweiterungen nacheinander umgesetzt.

„Sec-WebSocket-Extensions: foo, bar“

Die Daten sind in folgender Beispielzeile mit „DATA“ bezeichnet. Die Reihenfolge der Operationen wird wie folgend umgesetzt:

bar(foo(DATA))

Damit würde der Server eine gültige Handshake-Antwort an den Client senden, was bedeutet, dass der Server bereit ist die WebSocket-Verbindung herzustellen.

**Clientseitige Prüfung des Handshake**

Nachdem der Server seine Handshake-Antwort an den Client geschickt hat muss der Client die Antwort prüfen. Im einfachsten oder optimalen Fall, wie im obigen Beispiel, erhält der Client eine vollständige und mit gültigen Werten befüllte Handshake-Antwort vom Server zurück. In diesem Fall muss der Client nur den Inhalt des Header- Felds „Sec-WebSocket-Accept“, auf den zu erwartenden Wert, überprüfen. Der vom Server erstellte Wert, ist für den Client vorhersagbar. Der Client kann den Wert gleichermaßen berechnen, wie der Server selbst. Der Client prüft, ob der erwartete Wert und damit ein gültiger Wert eines WebSocket-Server zurückgeschickt wurde.
Die anderen Header-Felder, „Sec-WebSocket-Protocol“ und „Sec-WebSocket-Extensions“, müssen vom Client ausgelesen werden. Der Client kann so, entsprechend vom Server bestätigte Unterprotokolle oder Erweiterungen für die folgende Kommunikation mit dem Server zu nutzen.

### A.1.2 Base Framing Protokoll

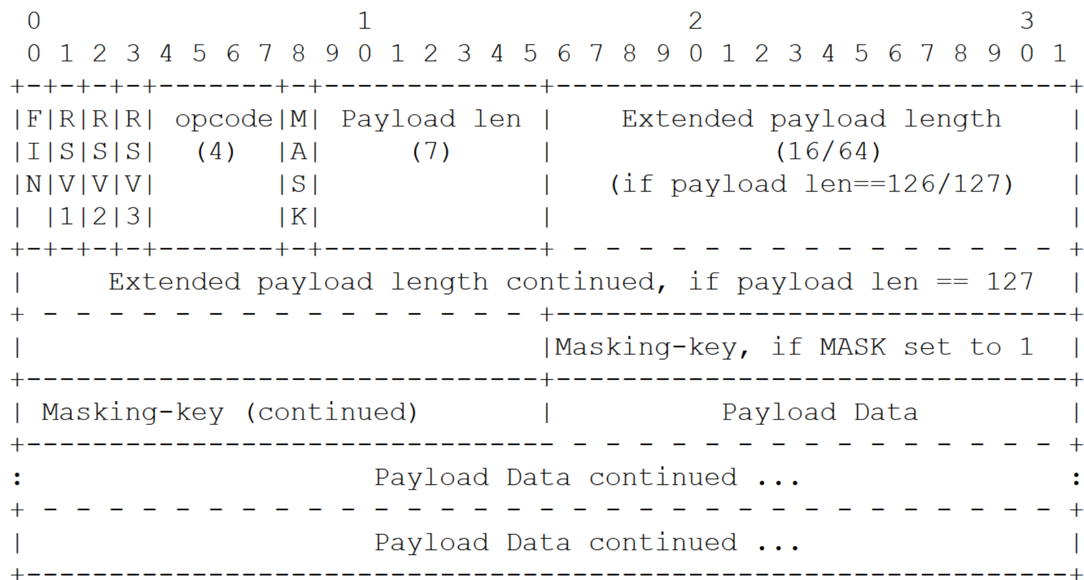
Im Folgenden Abschnitt wird das Kapitel 2.1.6 um Informationen ergänzt, die bislang aus Gründen der Übersicht vernachlässigt wurden. Einige Erklärungen wurden schon in vorherigen Kapiteln verwendet, werden aber für den zusammenhängenden Kontext an dieser Stelle wiederholt.

Das WebSocket-Protokoll nutzt Framing nur aus zwei Gründen: um Frame-basiert statt Stream-basiert zu arbeiten und zur Unterscheidung zwischen Unicode- und Binärdaten. Metadaten sind über die Anwendungsschicht zu transportieren. Das WebSocket-Protokoll setzt mit dem Framing-Mechanismus auf TCP auf, um dessen Segment-Mechanismus zu nutzen, ohne aber von dessen Längenbegrenzung beeinträchtig zu werden.

Daten werden in den oben genannten Messages transportiert. Messages bestehen aus Frames. Somit werden Daten indirekt in einer Sequenz von Frames übertragen. Zwischen Frames, die vom Client zum Server und vom Server zum Client gesendet werden, besteht ein Unterschied. Frames die vom Client an den Server gesendet werden sind maskiert. Im Kapitel „Clientseitiges Payload Masking“ erfolgt eine genauere Beschreibung. Die Maskierung dient als Verschlüsselung, so dass die Daten nicht mehr ohne einen passenden Schlüssel eingesehen werden können. Diese Verschlüsselung ist unabhängig davon, ob die WebSocket-Verbindung durch einen verschlüsselten TLS-Kanal, also eine WebSocketSecure-Verbindung, genutzt wird oder nicht.

Die im Kapitel 2.1.6 genannten Frame-Typen, werden durch OP-Codes und einer Payload-Längenangabe definiert. Die Payload-Längenangabe ist die Summe von „Extension-Data“ und „Application-Data“.

Folgend die Darstellung eines Frame in Abbildung A.1.2-52, eine Beschreibung der einzelnen Header-Felder und die Voraussetzungen für das Senden und Empfangen von Frames:



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Abbildung A.1.2-52..: Frame des Base-Framing-Protocol, aus [RFC6455] nach [RFC5234]

- FIN: 1 Bit
  Zeigt an, dass dies das letzte Fragment in einer Nachricht ist, wobei das erste Fragment auch das letzte Fragment sein kann.
- RSV1, RSV2, RSV3: jeweils 1 Bit
  Muss jeweils auf „0“ gesetzt sein, außer wenn eine beim Handshake ausgehandelte Erweiterung die Bedeutungen für Nicht-Null-Werte definiert.
- OP-Code: 4 Bits
  Definiert die Interpretation der "Payload data", folgende OP-Codes sind definiert:
  - x0 bezeichnet einen fortsetzenden Frame.
  - x1 bezeichnet einen Text-Frame.
  - x2 bezeichnet einen Binär-Frame.
  - x3-7 sind für weitere Nicht-Control-Frames vorbehalten.
  - x8 bezeichnet einen Close-Frame.
  - x9 bezeichnet einen Ping-Frame.
  - xA bezeichnet einen Pong-Frame.
  - xB-F sind für die weiteren Control-Frames reserviert.
- Mask: 1 Bit
  Legt fest, ob die "payload" maskiert ist. Wenn das Mask-bit auf „1“ gesetzt ist, ist ein „masking-key“ im entsprechenden Feld vorhanden.
- Payload data: (x+y) Bytes:
  Die Nutzdaten bzw. „payload-data“ sind als Verkettung von "Extension Data" und "Application-data" definiert.
- Extension data: x Bytes:

Die "Extension-Data" ist 0 Bytes, sofern keine Erweiterung beim Handshake verhandelt wurde. Jede Erweiterung muss die Länge der "Extension-Data" oder wie diese Länge berechnet werden kann angeben. Falls eine Erweiterung vorhanden, ist die "Extension-Data" in der gesamten Länge der „payload data“ enthalten.

- Application data: y Bytes:
  Beliebige "Application data" die transportiert werden soll. Es kann der Rest des Frames nach den "Extension-Data" genutzt werden.

**Voraussetzung um Daten zu senden**

Um Daten per Message über eine hergestellte WebSocket-Verbindung zu senden, müssen folgende Punkte vom Endpunkt erfüllt werden:

- Daten müssen in Frames gekapselt werden.
- Der erste Frame einer Message muss durch einen entsprechenden OP-Code den Typ der Nachricht angeben.
- Im letzten Frame einer Message muss das FIN Bit auf „1“ gesetzt sein.
- Sendet ein Client Daten, müssen die Frames maskiert werden.
- Werden Erweiterungen genutzt, ist deren Spezifikation zu beachten.
- Erstellte Frames müssen über die darunterliegende TCP-Verbindung übertragen werden.

**Voraussetzung um Daten zu empfangen**

Um Daten zu empfangen, muss ein Endpunkt auf der darunterliegenden TCP-Verbindung horchen und folgende Punkte erfüllen:

- Empfangene Frames müssen zwischen Control-Frames und non-Control-Frames differenziert werden, um diese entsprechend verarbeiten zu können.
- Wenn ein Frame, Teil einer fragmentierten Message ist, müssen die Anwendungsdaten aller Fragmente der Message Zusammengesetzt werden, um die Daten auslesen zu können.
- Das Ende einer fragmentierten Message signalisiert das FIN Bit, folgende empfangene Daten müssen als neue Message interpretiert werden.
- Beim Einsatz von Erweiterungen ist dessen Auswirkung auf die Daten und den Umgang mit diesen zu berücksichtigen.

### A.1.3 „Polling“ und „Long Polling“

Um die Anforderung von modernen Web-Anwendungen, mit dem Einsatz von HTTP erfüllen zu können, wurden Techniken wie „Polling“ oder „Long Polling“ entwickelt [RFC6202].

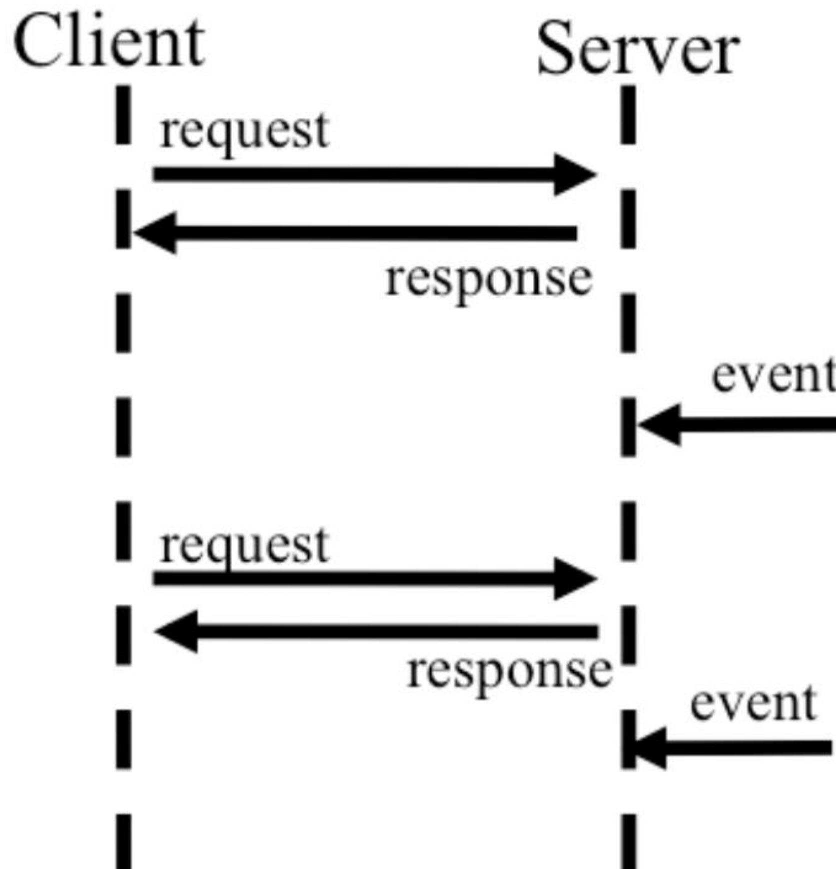



Abbildung A.1.3-53..: Beispielsequenz bei der Verwendung von Polling, aus [WSAaEiW]

„Polling“ wie in Abbildung A.1.3-53 dargestellt, ermöglichen die automatische Aktualisierung von Web-Seiten durch den Web-Browser. Der Web-Browser fragt dabei den Server in periodischen Abständen, ob es aktuelle Daten gibt. Gibt es neue Daten, sendet der Server diese in seiner Antwort. Gibt es keine neuen Daten, antwortet der Server ohne dass neue Daten übermittelt werden. Dem Benutzer des Web-Browsers wird ein asynchrones Verhalten simuliert, bei dem der Server ohne vorherige Anfrage eine Reaktion zeigt.

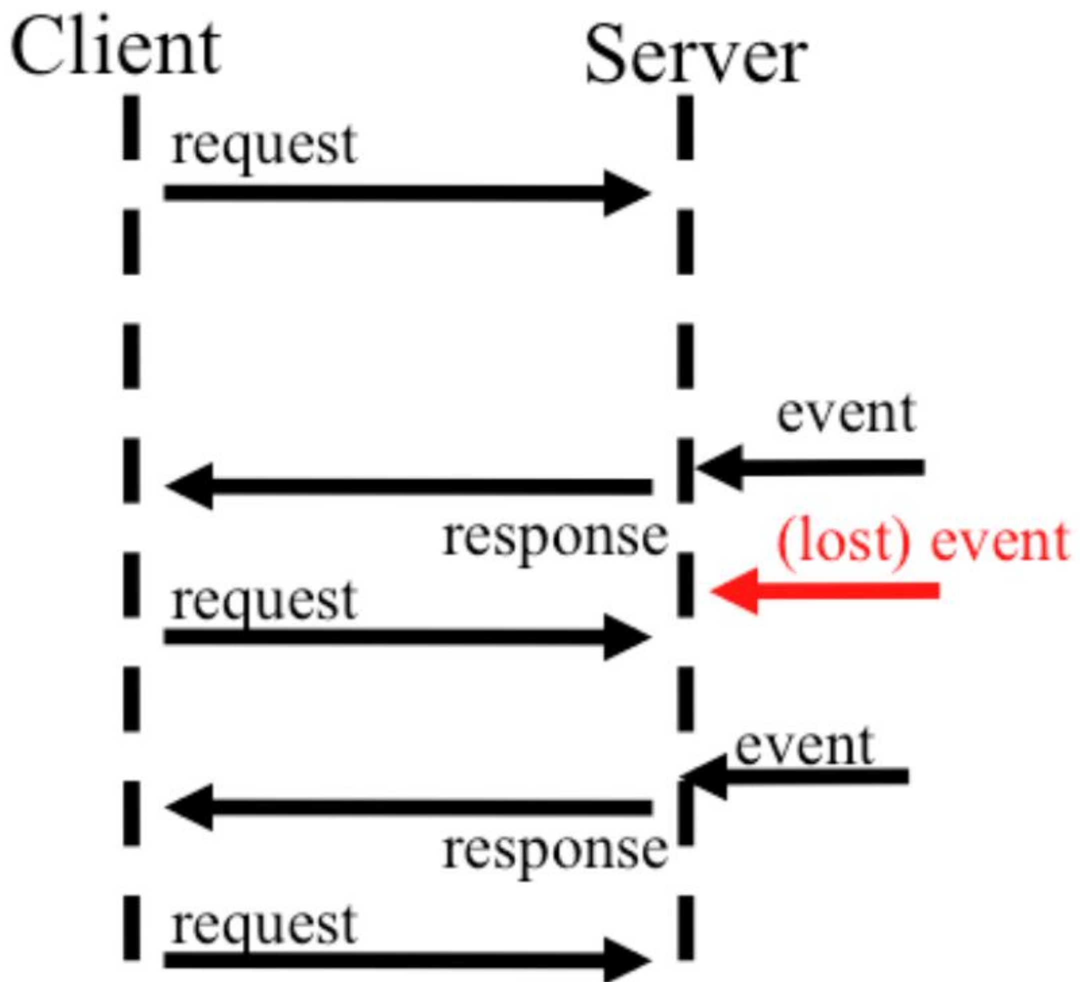



Abbildung A.1.3-54..: Beispielsequenz bei der Verwendung von Long-Polling, aus [WSAaEiW]

Beim „Long Polling“, wie in Abbildung A.1.3-54 dargestellt, wurde die Effizienz gegenüber „Polling“ in vielen Fällen gesteigert. Die Effizienzsteigerung wird damit erreicht, dass der Client keine Anfragen nach einer bestimmten Zeitperiode sendet, sondern immer erst wenn der Server eine Anfrage beantwortet hat. Empfängt der Server eine Anfrage vom Client, hält er diese solange, bis entweder neue Daten an den Client gesendet werden können oder ein Timeout auftritt. Anschließend antwortet der Server, was wiederrum den Client zu einer neuen Anfrage anregt. Der Austausch zwischen Client und Server ist so dynamisch und die Anfragen seitens des Clients in vielen Fällen deutlich reduziert. Dadurch kann der Einsatz der Technik den Server deutlich von clientseitigen Anfragen entlasten. Wie in Abbildung A.1.3-54 zu sehen ist, können neue Daten aber auch verloren gehen. Dies geschieht dann, wenn sich Daten in der Zeit ändern, die zwischen einer serverseitigen Antwort und einer erneuten clientseitigen Anfrage vergeht. Die „Long Polling“ Technik ist

daher kein guter und vollwertiger Ersatz für eine einfache asynchrone bidirektionale Verbindung.
Dieses und weitere entwickelte Techniken, die einen asynchrone bidirektionale Datenaustausch im Web ermöglichen, haben viele Web-Seiten deutlich sinnvoller gemacht und dynamisch erscheinen lassen. Ein gutes Beispiel sind Web-Seiten, die Nachrichten oder Börsenwerte anzeigen. Der Nutzer erwartet, dass die Daten aktuell sind und dass sie auch nach dem Aufruf der Web-Seite aktuell gehalten werden und so dem aktuellen Stand entsprechen.
Allerdings bringt der Einsatz dieser Techniken auch folgende Probleme mit sich:

**Probleme durch den Einsatz von „Polling“ und „Long Polling“**

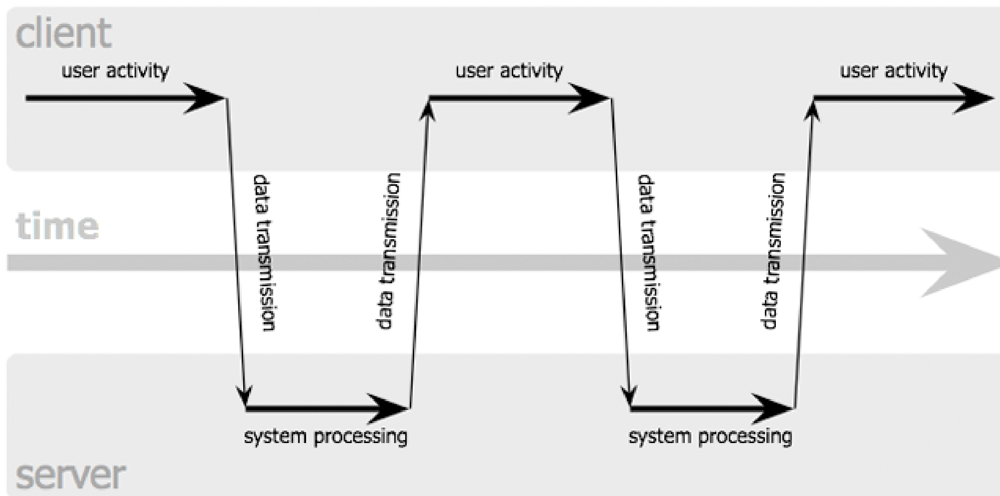
- Der Server ist gezwungen, für jeden Web-Browser mehrere TCP Verbindungen zu verwalten. Eine, um Daten an den jeweiligen Web-Browser zu senden, und jeweils ein für jede ankommende Anfrage vom Web-Browser.
- Das auch für die periodischen Anfragen verwendete HTTP hat einen Header, der bei jeder Anfrage mit übertragen werden muss. So kommen viele Daten zusammen, welche als „Overhead“ bezeichnet werden, da sie keine Nutzinformation beinhalten die kommuniziert werden müssen. Die Menge des Overheads hängt sehr von der Dauer einer Anfrageperiode ab und davon wie oft neue Daten auf Seiten des Servers bestehen. Gibt es häufig neue Daten, so verliert „Long Polling“ seine Effizienz gegenüber „Polling“, da eventuell sogar häufiger zwischen Client und Server kommuniziert wird.
- Auf Seiten des Web-Browsers müssen Abbildungen zwischen den vielen ausgehenden Verbindungen bei Anfragen an den Server und der eingehenden Verbindung erstellt werden, um eingehende Informationen der vorherigen Anfrage zuordnen zu können.
- Die Latenz zwischen Web-Browser und Server, begrenzt die möglichen Anfragen und Antworten in einem bestimmten Zeitraum. Daher hängt auch das dynamische Verhalten von der Latenz ab. Allerdings je nach verwendeter Technik mehr oder weniger.

Um ein dynamisches Web zu erzeugen und stets aktuellen Web-Seiten im Web-Browser betrachten zu können, ist ergo ein erheblicher Aufwand nötig.
Es gibt auch andere Ansätze, um die Anfrage-Flut an Servern nicht ausufern zu lassen und trotzdem Web-Anwendungen zu entwickeln die den herkömmlichen Desktop-Anwendungen äquivalent oder überlegen sind.

### A.1.4 Auf dem Weg zum Web 2.0 mit AJAX

Sehr bekannt und auch stark vertreten ist die AJAX-Technologie (Asynchronous JavaScript and XML). Die AJAX-Technologie verwendet eine Engine, die die asynchrone Datenübertragung innerhalb einer Web-Seite ermöglicht.
Das Konzept sieht vor, dass nicht jede Eingabe eines Benutzers in eine HTTP Anfrage umgesetzt wird. Die Abbildung A.1.4-55 verdeutlicht das Ziel des Konzepts. Die AJAX-Engine kann Eingaben des Benutzers oft direkt clientseitig umsetzen, ohne dass mit dem Server kommuniziert werden muss. Wenn Daten vom Server benötigt werden, können diese einzeln nachgeladen werden, was bisher mit reinen HTTP Anfragen nicht möglich war. Die Eingaben des Benutzers können durch die Engine gesammelt werden, etwa beim Ausfüllen eines Formulars mit mehreren Eingabefeldern. Jede bestätigte Eingabe wird zunächst nur lokal umgesetzt und dargestellt. Hat der Benutzer das komplette Formular ausgefüllt, sendet die Engine die Nutzereingaben gesammelt an den Server.
Dieses Verhalten bringt Vorteile mit sich:

- Durch den Einsatz von AJAX entsteht eine asynchrone Kommunikation zum Server. Die Eingaben des Benutzers sind unabhängiger von der Kommunikation mit dem Server. Dadurch kann eine Web-Anwendung deutlich reaktionsfreudiger auftreten. Dies verdeutlicht Abbildung 3 „Vergleich der Kommunikation zwischen HTTP und Ajax“, in der der Benutzer deutlich mehr Eingaben in der gleichen Zeit tätigen kann.
- Zusätzlich ist durch die asynchrone Kommunikation, die Latenz zwischen Web-Browser und Server weniger kritisch.
- Die Möglichkeit, gezielt einzelne benötigte Daten nachzuladen, reduziert die benötigte Menge an Daten die der Server senden muss.
- Die Anzahl von Anfragen an den Server werden reduziert und damit auch der „Overhead“.
- Die AJAX-Engine erlaubt es neben dem Client auch dem Server Änderungen auszulösen.



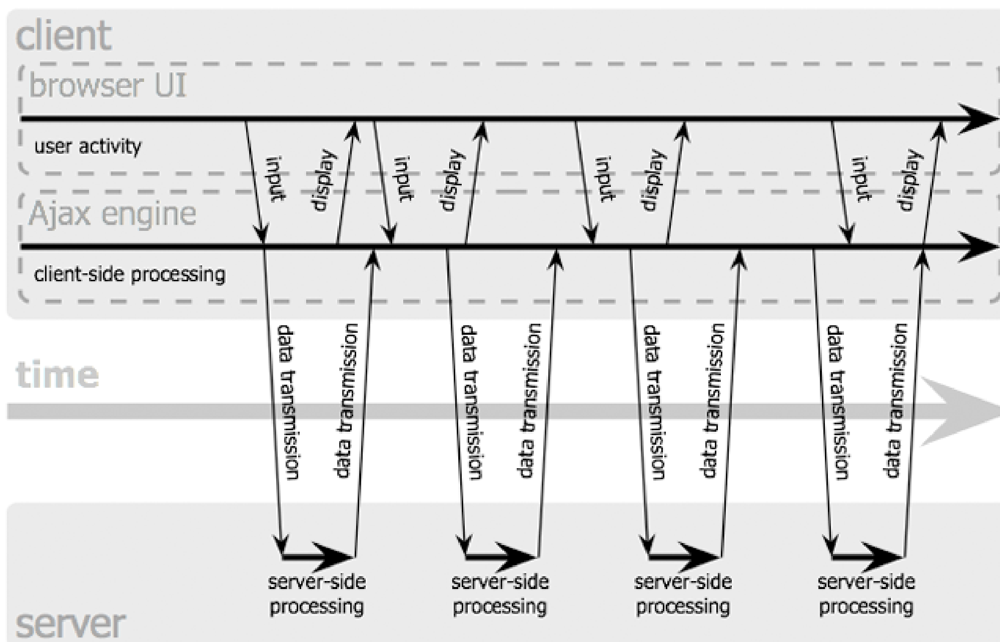




Abbildung A.1.4-55..: Vergleich der Kommunikation zwischen HTTP und Ajax, aus [ANAtWA]

Allerdings bleiben auch zuvor genannten Probleme aus Kapitel 2.1.4 bestehen, auch wenn diese durch den Einsatz der AJAX-Technologie reduziert werden können. Um diese

Probleme anzugehen und so Server von Anfragen zu entlasten, Overhead bei der Übertragung einzusparen und Skripte von Web-Seiten zu vereinfachen, hat man mit der Entwicklung des WebSocket-Protokolls begonnen.

Eine einfache Lösung wäre es, für die Kommunikation zwischen Server und Web-Browser nur eine TCP Verbindung zu nutzen und über diese asynchron in beide Richtungen zu kommunizieren. Nutzt man nur eine dauerhafte TCP Verbindung bidirektional, muss der Server keinen hohen Aufwand bei der Verwaltung der Verbindungen betreiben. Gleichzeitig entfällt fast der komplette „Overhead“. Da nicht per HTTP kommuniziert wird, wird der Overhead der Header eingespart. Eine direkte asynchrone Kommunikation zwischen Client und Server, erspart Anfragen an den Server. Der Server kann neue Daten direkt an den Client senden. Genau diese Ansätze wurde bei der Entwicklung des WebSocket-Protokolls verfolgt und so eine passende Technik für Web-Anwendungen geschaffen.

# Quellen- & Literaturverweise


[RFC6455] I.Fette, A.Melnikov “The WebSocket Protocol“, RFC 6455, December 2011, URL: https://tools.ietf.org/html/rfc6455

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, URL: https://www.ietf.org/rfc/rfc2616.txt

[wsDEMO] WebSocket.org, WebSocketDemos, URL: https://www.websocket.org/demos.html

[ANSI_X3_4_1986] American National Standards Institute, "Coded Character Set - 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986, URL: http://sliderule.mraiow.com/w/images/7/73/ASCII.pdf

[FIPS_180_3] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-3, October 2008, URL: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[RFC2817] Khare, R. and S. Lawrence, "Upgrading to TLS Within HTTP/1.1", RFC 2817, May 2000, URL: https://www.ietf.org/rfc/rfc2817.txt

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000, URL: http://tools.ietf.org/html/rfc2818

[TtYfFaP] Huang, L-S., Chen, E., Barth, A., Rescorla, E., and C. Jackson, "Talking to Yourself for Fun and Profit", 2010, URL: http://w2spconf.com/2011/papers/websocket.pdf

[XMLHttpRequest] van Kesteren, A., Ed., "XMLHttpRequest", W3C Candidate Recommendation CR-XMLHttpRequest-20100803, August 2010, http://www.w3.org/TR/2010/CR-XMLHttpRequest-20100803
Latest version available at:
http://www.w3.org/TR/XMLHttpRequest/

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005, URL:
http://www.ietf.org/rfc/rfc4122.txt

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006, URL:
http://www.ietf.org/rfc/rfc4648.txt

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, URL:
http://www.ietf.org/rfc/rfc3986.txt

[WSC_UIG] Roessler, T. and A. Saldhana, "Web Security Context: User Interface Guidelines", World Wide Web Consortium Recommendation REC-wsc-ui-20100812, August 2010, URL:
http://www.w3.org/TR/2010/REC-wsc-ui-20100812
Latest version available at:
http://www.w3.org/TR/wsc-ui/

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011, URL:
http://tools.ietf.org/html/rfc6265

[WSAPI] Hickson, I., "The WebSocket API", W3C Working Draft WD-websockets-20110929, September 2011, URL:
http://www.w3.org/TR/2011/WD-websockets-20110929/
Latest version available at:
http://www.w3.org/TR/websockets/

[RFC6202] Loreto, S., Saint-Andre, P., Salsano, S., and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", RFC 6202, April 2011, URL:
https://tools.ietf.org/html/rfc6202

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454,December 2011, URL:
https://www.ietf.org/rfc/rfc6454.txt

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008, URL:
https://tools.ietf.org/html/rfc5246

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, URL:
https://tools.ietf.org/html/rfc5234

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003, URL:
https://tools.ietf.org/html/rfc3629

[IETF] Internet Engineering Task Force, URL:
https://www.ietf.org/

[IESG] Internet Engineering Steering Group, URL:
https://www.ietf.org/iesg/

[W3C] World Wide Web Consortium, URL:
http://www.w3.org/

[RFC793] Information Sciences Institute University of Southern California, Transmission Control Protocol, RFC 793, September 1981, URL:
https://tools.ietf.org/html/rfc793

[HTML5WS] Robin Berjon, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, „HTML5 A vocabulary and associated APIs for HTML and XHTML“, W3C Working Draft 25 October 2012, URL:
http://www.w3.org/TR/2012/WD-html5-20121025/

[Web2_0] Tim O'Reilly, "What Is Web 2.0", URL:
http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html

[BroNet] Ilya Grigorik, „High Performance Browser Networking“, September 2013, Print ISBN: 978-1-4493-4476-4, ISBN 10: 1-4493-4476-3, Ebook ISBN: 978-1-4493-4471-9, ISBN 10: 1-4493-4471-2, URL:
http://chimera.labs.oreilly.com/books/1230000000545

[RTCWEB_JSEP] J. Uberti, C. Jennings. Javascript Session Establishment Protocol, October 2014, URL: http://datatracker.ietf.org/doc/draft-ietf-rtcweb-jsep/

[RFC5245] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. April 2010. RFC. URL: http://www.ietf.org/rfc/rfc5245.txt

[HTML5PUB] Robin Berjon; Steve Faulkner; Travis Leithead; Erika Doyle Navara; Edward O'Connor; Silvia Pfeiffer. HTML5. 6 August 2013. W3C Candidate Recommendation. URL: http://www.w3.org/TR/html5/

[RFC3264] J. Rosenberg; H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). June 2002. RFC 3264. URL: http://tools.ietf.org/html/rfc3264

[STUN_URI] S. Nandakumar; G. Salgueiro; P. Jones; and M. Petit-Huguenin. URI Scheme for Session Traversal Utilities for NAT (STUN) Protocol. 12 March 2012. Internet Draft (work in progress). URL: http://tools.ietf.org/html/draft-nandakumar-rtcweb-stun-uri

[TURN_URI] M. Petit-Huguenin; S. Nandakumar; G. Salgueiro; and P. Jones. Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers. 12 March 2012. Internet Draft (work in progress). URL: http://tools.ietf.org/html/draft-petithuguenin-behave-turn-uris

[RFC5389] J. Rosenberg; R. Mahy; P. Matthews; D. Wing. Session Traversal Utilities for NAT (STUN). October 2008. RFC 5389. URL: http://tools.ietf.org/html/rfc5389

[RFC5766] P. Mahy; P. Matthews; J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). April 2010. RFC 5766. URL: http://tools.ietf.org/html/rfc5766

[WebRTCpub] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, WebRTC 1.0: Real-time Communication Between Browsers, W3C Working Draft 10 September 2013, URL: http://www.w3.org/TR/webrtc/

[WebRTCedi] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, Anant Narayanan, WebRTC 1.0: Real-time Communication Between Browsers, W3C Editor's Draft 02 February 2015, URL: http://w3c.github.io/webrtc-pc/

[RFC4566] M. Handley, V. Jacobson, C. Perkins, SDP: Session Description Protocol, July 2006, URL: http://www.ietf.org/rfc/rfc4566.txt

[RFC791] Information Sciences Institute University of Southern California, INTERNET PROTOCOL, September 1981, URL: https://tools.ietf.org/html/rfc791

[RFC2460] S. Deering, R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, December 1998, URL: https://www.ietf.org/rfc/rfc2460.txt

[RFC6347] E. Rescorla, N. Modadugu, Datagram Transport Layer Security Version 1.2, January 2012, URL: https://tools.ietf.org/html/rfc6347

[RFC3711] M. Baugher, D. McGrew, M. Naslund, E. Carrara, K. Norrman, The Secure Real-time Transport Protocol (SRTP), March 2004, URL: http://www.ietf.org/rfc/rfc3711.txt

[RFC4960] R. Stewart, Stream Control Transmission Protocol, September 2007, URL: https://tools.ietf.org/html/rfc4960

[RFC3550] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, RTP: A Transport Protocol for Real-Time Applications, July 2003, URL: https://www.ietf.org/rfc/rfc3550.txt

[RFC3758] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, P. Conrad, Stream Control Transmission Protocol (SCTP) Partial Reliability Extension, May 2004, URL: http://www.ietf.org/rfc/rfc3758.txt

[RFC768] J. Postel, User Datagram Protocol, August 1980, URL: https://www.ietf.org/rfc/rfc768.txt

[KAAZING] „About HTTPS and WSS“, KAAZING, Stand: 11.01.15, URL: http://developer.kaazing.com/documentation/jms/4.0/security/c_sec_https_wss.html

[WebRTCorg] Copyright Google Inc., webrtc.org, Stand: 02.02.2015, URL: http://www.webrtc.org/

[WSAaEiW] Matthias Weßendorf, „WebSocket: Annäherung an Echtzeit im Web“, Stand: 11.01.15, URL: http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html

[ANAtWA] Jesse James Garrett, Ajax: A New Approach to Web Applications, Stand: 11.01.15, URL: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/

[JavaEE7Tut] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, William Markito, Java Platform, Enterprise Edition The Java EE Tutorial, Release 7, Stand 10.03.15, URL: https://docs.oracle.com/javaee/7/tutorial/index.html

[JSR356] Johan Vos, JSR 356, Java API for WebSocket, Stand 10.03.15, URL: http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html

[Tyrus] Project Tyrus, JSR 356: Java API for WebSocket – Refrence Implementation, Stand 10.03.15, URL: https://tyrus.java.net/

[POJO] Martin Fowler, POJO An acronym for: Plain Old Java Object, Stand 10.03.15, URL: http://www.martinfowler.com/bliki/POJO.html

[JavaEE7API] Danny Coward, Annotation Type ServerEndpoint, Java EE 7 Specification APIs, Stand 16.03.15, URL: https://docs.oracle.com/javaee/7/api/javax/websocket/server/ServerEndpoint.html

[FS] File System API, Stand 20.03.2015, URL: https://nodejs.org/api/fs.html

[FILEAPI] Arun Ranganathan, Jonas Sickling, File API, Stand 20.03.15, URL: http://www.w3.org/TR/file-upload/

[WRTC] Alan K, Eric Rescorla, Pasquale Boemio, Damon Oehlman, Benjamin Byholm, Alex Londeree, Jesús Leganés Combarro "piranna", Dario Andrei, Matt Porritt, Mark Andrus Roberts, Arno Klein, Manu Raghavan, js-platform, node-webrtc, WebRTC stack for node.js, Stand 20.10.2014 URL: https://github.com/js-platform/node-webrtc

[WS] Einar Otto Stangvik, websockets, ws, `ws`: The fastest RFC-6455 WebSocket implementation for Node.js, Stand 20.10.2014 URL: https://github.com/websockets/ws

[NODEJS] Node.js, Joyent Inc., Stand: 19.10.2014, URL: https://nodejs.org/

[CBUF] Trevor Norris , trevnorris, cbuffer, JavaScript Circular Buffer Utility, Stand: 21.03.2015, URL: https://github.com/trevnorris/cbuffer

[ADAP] Google Inc., Mozilla Foundation, Intel Corporation, Vonage Holdings Corp., MIPS Technologies, Ben Strong, Martin Storsjo, Jie Mao, Anil Kumar, Opera Software, Silviu Caragea , adapter.js, Stand: 21.03.15, URL: https://code.google.com/p/webrtc/source/browse/trunk/?r=3905#trunk%2Fsamples%2Fjs%2Fbase

[HASH] Ariel Flesler, Hashmap, Hashmap JavaScript class for NodeJS and the browser. The keys can be anything and won't be stringified, Stand: 21.03.2015, URL: https://github.com/flesler/hashmap

[ECGSYN] Mauricio Villarroel, Patrick McSharry, Gari Clifford, ECGSYN: A REALISTIC ECG WAVEFORM GENERATOR, Stand: 21.03.2015, URL: http://www.physionet.org/physiotools/ecgsyn/

[SOFTRT] J. Richling, Weiche Echtzeit, Stand 21.03.2015, URL: https://www2.informatik.hu-berlin.de/~richling/emes2003/07-softrt.pdf

[HACKS] Victor Pascual, webrtchacks.com, webrtcH4cKS: ~ SDP: The worst of all worlds or why compromise can be a bad idea (Tim Panton), 20.01.2014, Stand 22.10.2014, URL: https://webrtchacks.com/tim-rant/

[SKYPE] Senthil Velayutham, Bringing Interoperable Real-Time Communications to the Web, Garage & Updates, blogs.skype.com, 27.10.2014, Stand: 06.04.2015, URL: http://blogs.skype.com/2014/10/27/bringing-interoperable-real-time-communications-to-the-web/

[HELLO] Chad Weiner, Test the new Firefox Hello WebRTC feature in Firefox Beta, Future Releases, blog.mozilla.org, 16.10.2014, Stand: 06.04.2015, URL: https://blog.mozilla.org/futurereleases/2014/10/16/test-the-new-firefox-hello-webrtc-feature-in-firefox-beta/

# Versicherung über Selbstständigkeit

*Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.*

*Hamburg, den* ________________ ____________________________