Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektarbeit

Andreas Müller

MBC-Ping-Pong Backend

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Müller

MBC-Ping-Pong Backend

Projektarbeit eingereicht im Rahmen der Wahlpflichtfach

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

Betreuender Prüfer: Prof. Dr. Martin Becke

Eingereicht am: 6. März 2017

Andreas Müller

Thema der Arbeit
MBC-Ping-Pong Backend

Stichworte
Ping-Pong, NodeJS, JavaScript, WebRTC

Kurzzusammenfassung
In diesem Dokument wird das Projekt im MBC-Ping-Pong, das im Rahmen des Wahlpflichtfaches Modernebrowserkommunikation an der HAW-Hamburg erstellt wird, behandelt. Dieses Dokument behandelt das Backend und dem damit entstandenen Kommunikationsmodul.
Andreas Müller

Title of the paper
MBC-Ping-Pong Backend

Keywords
Ping-Pong, NodeJS, JavaScript, WebRTC

Abstract
This document is about the Project MBC-Ping-Pong, which is made for the elective course Modernebrowserkommunikation at HAW-Hamburg. This document is about the Backend and the thus implemented communication module.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Backend

1.1 Kommunikation

1.1.1 Zeitkritische Informationen

Als zeitkritisch werden Informationen eingestuft, sofern sie die direkten Eingaben der ControlClients und eventuelles Feedback des OutputClient betreffen. Da es sich um ein Reaktionsspiel handel,t müssen diese Daten zeitnah von Sender zu Empfänger gelangen. Solch eine Relation ist nur über eine Peer-to-Peer Verbindung zu realisieren.

1.1.2 Möglichkeit 1: Predefined Packages 'rtc.io'

Der NodeJS Server wird als Verbindungsserver für die Etablierung einer Peer-to-Peer Verbindung zwischen ControlClients und OutputClient genutzt. Als Technik wird konret WebRTC eingesetzt. Auf dem NodeJS-Server wir das Package "rtc-switchboardëingesetzt, welches eine Grundlage für einen Signalisierungsserver ist. Die Clients nutzen "rtc-quickconnectüm neues Channels anzufordern und eine Peer-to-Peer Verbindung zu etablieren.

Vorteile

- Leichtere Implementierung, da alle Ebenen der Kommunikation bereits abgedeckt wurden und nur noch semantisch auf Informationen eingegangen werden muss.

Nachteile

- Durch Generalisierung ein deutlicher Overhead.
- Status ist offiziell noch 'unstable'.

1.1.3 Möglichkeit 2: Abstrakte WebRTC implementation 'WebRTC'

Wie auch bei Möglichkeit 1 wird der NodeJS-Server als Verbindungsserver genutzt. Jedoch muss auf Client-Seite, also für OutputClient und ControlClient, eine eigene Signalling Implementation stattfinden. Es wird eine viel abtraktere Schicht genutzt.

Vorteile

- Da die genutzen Packages nur eine Grundlage bilden, ist eine spezialisierte Implementierung möglich.

Nachteile

- Implementierungsaufwand deutlich höher.
- Spezialisierung für dieses Projekt eventuell nicht nötig.
- Status ist offiziell noch 'unstable'.

1.1.4 Möglichkeit 3: Socket.io-P2P

Das Package Socket.io bietet auch selber eine Peer-to-Peer lösung auf WebRTC Basis. Hierbei wird eine Spezielle Art eines Sockets genutzt, welche wie ein normaler WebSocket agiert, bis ein Upgrade durchgeführt wird und die Clients nun direkt miteinander Kommunizieren.

Vorteile

- Das Signalling und die P2P Kommunikation können mit einem Package geregelt werden.
- Variabel kann die Kommunikation über den Server, oder P2P ablaufen.
- Signalling events werden von Client-Server Kommunikation direkt zu P2P übernommen.

Nachteile

- Wenig Einfluss auf die Upgrade-Mechanismen.

1.1.5 Möglichkeit 4: Eigenes Kommunikationsmodul

Aus Basis der WebRTC Standartimplementierung von den Browsern, kann eine eigene Schicht entwickelt werden, welche den Kommunikatinsaufbau (das Signalling) und die Kommunikationsabläufe regelt. Dies müsste auf einer generischen Basis geschehen um allen Anforderungen des Programmes gerecht zu werden.

Vorteile

- Volle Kontroller über Signalling und Verwaltungsabläufe der Verbindungen.
- Möglichkeit für schnelle Anpassungen.
- Unabhängigkeit gegenüber anderen Entwicklern.

Nachteile

- Größter Implementierungsaufwand.
- Risikoabschätzung nur schwer möglich.
- Basis-Implementationen verschieden je Browser (gleiches Problem wie bei den anderen Implementationen: rtc.io, wocket.io-P2P, etc.)

1.1.6 Statusinformationen

Für den Austausch von Statusinformationen und Signalen zwischen Peer und Server werden WebSockets genutzt. Für den Austausch von Statusinformationen zwischen Peers wird WebRTC genutzt. Für den Austausch von Signalen zwischen Peers werden WebSockets genutzt, wobei der Server der Verbindungs-Knotenpunkt zwischen den Peers ist.

1.1.7 Fazit nach Test

Nach und auch schon während der Entwicklung des Prototypen hat sich rausgestellt, dass Socket.io-P2P in der Implementierung viele Fehler aufweist. Gleiches gilt für rtc.io. Eine teilweise Abänderung der Module (workarounds) führt näher an das gewünschte Ergebnis, verursacht aber intern wieder mehr Fehler und sorgt für eine inkonsistente Modulversion.

1.1.8 Fazit

Die eigene Implementation auf Basis des WebRTC bietet die meisten Möglichkeiten, birgt jedoch auch die meisten Risiken. RTC.io bietet eine Implementation die sehr gut für Prototypen geeignet ist, aber neben dem Zeitunkritischen Signalling eine eigene Einheit bietet, welche eine Generalisierung des WebRTC darstellt und somit viel Overhead besitzt. Die P2P-Imeplementierung von Socket.io steht von der Implementierungs-Komplexität in der Mitte. Es bietet viele bereits bekannte Mechanismen des Client-Server Signalling über Events, welche mit einem Upgrade nun auch von Client zu Client funktionieren. Dagegen spricht jedoch die instabilität der P2P-Implementation von Socket.io.

Eine eigene Implementation eines Modules auf Basis der von den Browsern gegebenen WebRTC-Schnittstelle ist die Wahl. Sie bietet die größtmögliche Flexibilität im Bezug auf Veränderbarkeit und Anpassung im laufenden Entwicklungszyklus. Als Risiko ist besonders der große Aufwand anzusehen.

1.2 Eigenes Modul Kommunikation

1.2.1 Anforderungen an das eigene Modul für WebRTC

- Es muss möglich sein, unsere ControlPeers nur mit dem DisplayPeer zu verbinden, nicht aber untereinander (Sterntopologie, wobei der DisplayPeer das Zentrum ist).
- Der DisplayPeer muss zu jedem ihm zugeordneten ControlPeer verbunden sein.
- Es Muss auf Serverseite eine Raumlogik existieren, um die Peers zuordnen zu können.
- Das Signalling muss weiterhin über den Server stattfinden.

1.2.2 Zuweisungslogik Server

Der Server hat bezüglich der Zuweisungen für Verbindungen die Aufgabe alle Clients zu gruppieren. Dies soll ähnlich wie bei Socket.io über eine Raumlogik geschehen. Es existieren Räume, in welche die Sockets auf dem Server (je Client also einer) eingeordnet werden.

Hierbei unterscheiden sich die Clients in ihrem Kommunikationsmodus:

Single Ein Client mit dem SSingle"Modus pflegt nur eine direkte WebRTC Verbindung.

Multi Ein Client mit dem "Multi"Modus pflegt mehrere direkte WebRTC Vebindungen gleichzeitig und agiert zwischen den ganzen Peers als "Host".

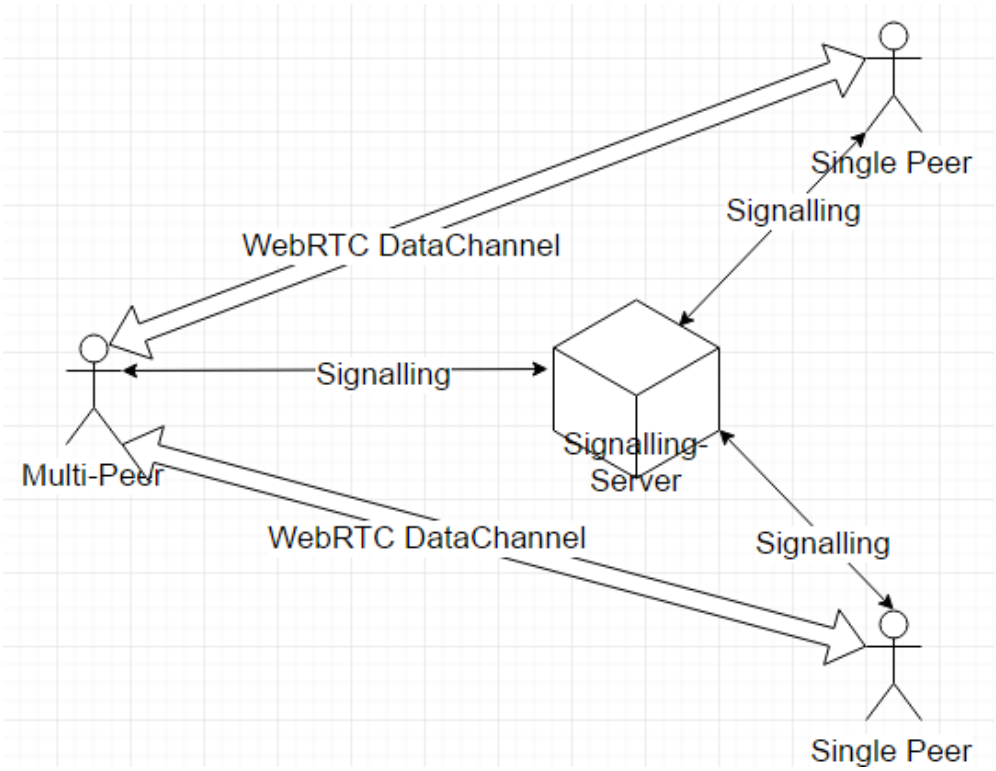

Abbildung 1.1: Verbindungsnetz

In Abbildung 1.1 ist zu sehen, wie ein "Multi-Peer"mit mehreren SSingle-Peer"gleichzeitig über WebRTC (konkret den DataChannel) kommuniziert, jeder einzelne SSingle-Peeräber nur mit dem einen "Multi-Peer". Zu sehen ist außerdem, dass das Signalling weiterhin für alle Peers über den Signalling-Server stattfindet, welcher hier auch gleichzeitig der WebServer ist.

1.2.3 Zuweisungslogik Client (Single)

Von nun an wird der Client der den Single Modus nutzt als SSingle-Peer"bezeichnet.

Der Single-Peer pflegt nur eine WebRTC Verbindung, somit muss keine Zuweisungslogik geschehen.

1.2.4 Zuweisungslogik Client (Multi)

Von nun an wird der Client der den Multi Modus nutzt als "Multi-Peer"bezeichnet.

Der Multi-Peer pflegt mehrere WebRTC Verbindungen zu verschiedenen Single-Peers. Identifiziert werden die einzelnen Single-Peers über ein vom Server beim Signalling gesetztes Attribut in den Signalling Nachrichten: "from". Über dieses Attribut weiß der Multi-Peer von wem diese Signalling-Message kommt und kann so alle ankommenden Informationen diesem Single-Peer zuordnen. Somit wird unter dem Alias des Single-Peer ein eigenes "webRTCConnectionObjekt verwaltet/abgelegt.

1.2.5 Identifizierung der Peers

Die Peers selber erhalten in allen Signalling-Messages ein Attribut, über welches sie den Ursprung ermitteln können, um so Signale Peers zuordnen zu können. Diese ÏD"wird vom Server vergeben und entspricht einfach nur der Socket ID welche die einzelnen Clients in Verbindung zum Server haben. Da nur der Server diese ID vergibt ist diese auch eindeutig.

1.3 Signalling Ablauf

Der Ablauf ist bei den Single-Peers, sowie bei den Multi-Peers nahezu gleich.

1.3.1 Anmeldung beim Server

Multi-Peer Der Multi-Peer meldet sich beim Server mit der Bitte einen Raum zu erstellen. Sollte dieser Raum schon existieren ist dies ein Fehler, da in jedem Raum nur ein Multi-Peer existieren darf. Sollte der Raum noch nicht existieren, wird er erstellt und der Multi-Peer diesem zugewiesen.

Single-Peer Der Single-Peer meldet sich beim Server mit der Bitte einen Raum zu betreten. Sollte dieser Raum noch nicht existieren ist dies ein Fehler, da ein Raum in diesem Projekt ein Spiel darstellt, ein Single-Peer (Control-Peer) aber nur einem bestehenden Spiel beitreten kann. Sollte der Raum bereits existieren und alle weiteren Spielablaufrelevanten Informationen stimmen (das Spiel darf zum Beispiel noch nicht angefangen haben), so tritt der Single-Peer diesem Raum bei. Direkt nach dem Beitreten eines Raumes, wird eine Signal an den Server gesendet, dass wir da sind.

Server Sobald ein Raum existiert und ein Multi-Peer diesem zugewiesen ist, sendet jeder neu dazukommende Single-Peer ein Signal an den Server, dass er da ist. Dieses Signal wird nur an den Multi-Peer weitergeleitet.

1.3.2 Erstes Signal

Multi-Peer Jedes mal, wenn der Multi-Peer ein neues erstes Signal eines Single-Peers über den Server erhält ("hereandready"), ermittelt er seine ICE-Candidates (je nach Browser kann dieses Vorhaben eine erneute Ermittlung der Candidates, oder ein Verwenden der bereits vorher ermittelten veranlassen) und sendet diese an den Signalisierenden Single-Peer.

1.3.3 ICE Candidates

Multi-Peer Erhält ein Multi-Peer einen ICE-Candidate wird dieser der Passenden Verbindung hinzugefügt und löst das "negotationneededEvent aus, welches ein neues Offer in Form von SDP Signalisiert.

Single-Peer Erhält ein Single-Peer einen ICE-Candidate wird dieser der akutellen webRTC-COnnection hinzugefügt und löst das "negotationneededEvent, welches ein neues Offer in Form von SDP Signalisiert.

1.3.4 Aufbau über SDP

Multi-Peer Erhält der Multi-Peer ein Offer in Form von SDP erwiedert er dieses mit einer Answer auch in Form von SDP und fügt die "remoteDescription"dieser WebRTCConnection auf Basis der Offer hinzu. Sofern dieser Vorgang Erfolg hatte ist eine direkte Verbindung zwischen Multi-Peer und Single-Peer entstanden.

Single-Peer Erhält der Multi-Peer ein Offer in Form von SDP erwiedert er dieses mit einer Answer auch in Form von SDP und fügt die "remoteDescription"dieser WebRTCConnection

auf Basis der Offer hinzu. Sofern dieser Vorgang erfolg hatte ist eine direkte Verbindung zwischen Multi-Peer und Single-Peer entstanden.

Kommunikationskanal

Da es sich um einfache Steuerungsdaten handelt verwenden beide Modi der Peers einen simplen DataChannel.

1.4 Eigenes Modul: Aufbau

1.4.1 Eigenes Modul: Schnittstelle

```
module.exports.Client = Client;
module.exports.Server = Server;
module.exports.ServerRooms = rooms;
```

Abbildung 1.2: Modul Exporte

Modul.Client Unter Modul.Client ist eine Klassenorientierte Repräsentation des Clients zu finden. Es ist ein Object dieses Typs zu erzeugen.

Modul.Server Unter Modul.Server ist eine Repräsentation des Servers zu finden. Für jeden auf dem Server neu Verbundenen Client muss eine neue (anonyme) Instanz dieses Typs erstellt und dabei der Socket der mit dem Client verbunden ist übergeben werden.

Modul.ServerRooms Unter Modul.ServerRooms ist eine Simple Liste der derzeitigen Räume auf dem Server, inklusive der darin befindlichen Clients zu sehen. Diese Informationen sind jedoch nur auf Serverseite bei Verwendung des Modul.Server einsichtlich, bzw. werden nur unter Verwendung von Modul.Server aktualisiert.

1.4.2 Eigenes Modul: Abhängigkeiten

Das Modul ist derzeit direkt von Socket.io abhängig, von welchem es die WebSockets für das Signalisierungen über den Signalling-Server nutzt.

1.4.3 Eigenes Modul: Struktur

Das Modul implementiert zwei Teile. Erstens die Client-Seite. Zweitens die Server-Seite.

Client Aufgaben

Die Client-Implementation ist für das Frontend gedacht und nutzt alle WebRTC Standarts der Browserimplementationen (Mozilla, webkit und Microsoft überlagert). Es handelt alle Faktoren von Initiierung der Kommunikation, bis Durchführung dieser ab.

Server Aufgaben

Die Server-Implementation kümmert sich auf Server-Seite um die Einhaltung von Kommunikations-Standarts, die Zuweisung von Signalen und die Gruppierung vieler Peers.

1.5 Eigenes Modul: Generischer Ansatz

1.5.1 Generische Anforderung

Als Anfoforderung gilt, dass die Implementierung des Moduls von unserer Nutzung für das Ping-Pong Projekt entkoppelt ist. Dies ist als gegeben anzusehen, wenn die Implementation keine Hinweise auf dieses Projekt hinterlässt und Möglichkeiten zu Umsetzung anderer beliebiger Projekte bietet, ohne Anpassungen vornehmen zu müssen.

1.5.2 Generische Umsetzung

Im folgenden wird die allgemeine generische Umsetzung am Beispiel der Callbacks und Topologien gezeigt.

Generische Umsetzung: Allgemein

Im allgemeinen gibt es zwei Klassen von Peers, die ein Client nutzen kann.

Single-Peer Ein Client der den Modus SSingle-Peer"nutzt gibt damit an, dass er maximal eine direkte Verbindung zu einem anderen Peer pflegen möchte.

Multi-Peer Ein Client der den Modus "Multi-Peer"nutzt gibt damit an, dass er eine oder mehr direkte Verbindungen zu Peers pflegen möchte.

Aus Kombinationen dieser Modi lassen sich verschiedene Topologien bilden. Näheres unter "Generische Umsetzung: Topologien".

Generische Umsetzung: Callbacks

Um jeden Nutzer dieses Moduls selbst die Verarbeitung von Ereignissen zu ermöglichen, ist die Möglichkeit gegeben, dass der Nutzer Callback-Funktionen hinterlegt, die bei gewissen autretenden Ereignissen ausgerufen werden. Diese Ereignisse umfassen z.B:

Neue Verbindung Eine neue Verbindung zu einem Peer wurde erfolgreich hergestellt.

Verbindungsverlust zu Peer Eine bestehende Verbindung zu einem Peer wurde verloren.

Neue Nachricht Es kam eine neue Nachricht über den DataChannel eines Peers an.

Weiteres zu der direkten Verwendung der Callback-Functionen unter Ëigenes Modul Verwendung: Callbacks".

Generische Umsetzung: Topologien

Es lassen sich durch die Verwendung der zwei Modi verschiedene Topologien bilden.

one-to-one Keine konkrete Topologie, aber für direkte Peer-to-Peer Verbindungen oft die gängiste Art. Beide Peers sind SSingle-Peeründ pflegen jeweils nur die Verbindung zu dem anderen. Siehe Abbildung 1.3.

Abbildung 1.3: Topologie one-to-one

Stern Einer der Peers agiert als Host für die anderen. Dieser Host-Peer ist der Mittelpunkt des Sternes und nutzt den Modus "Multi-Peer". Alle anderen Peers nutzen den Modus SSingle-Peeründ pflegen nur die Verbindung zum Mittelpunkt (dem Host). Siehe Abbildung 1.4.

Vollvermascht Alle Peers nutzen den Modus "Multi-Peeründ pflegen zu jedem anderen Peer eine direkte Verbindung. Siehe Abbildung 1.5.

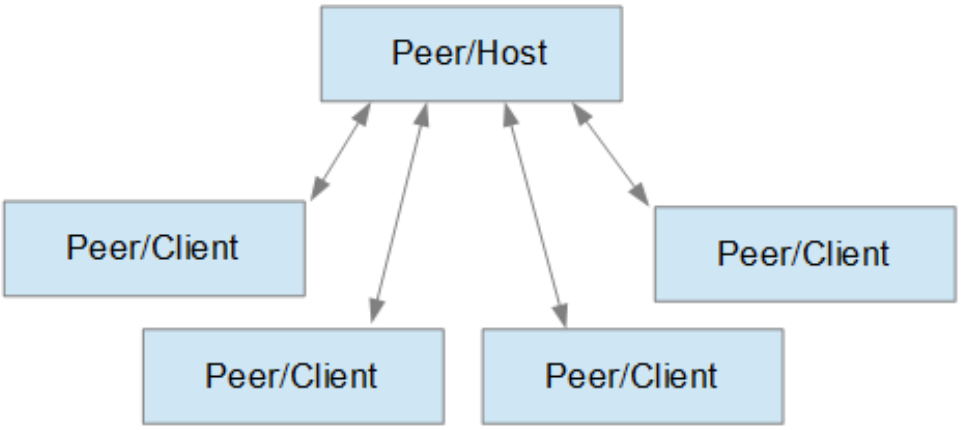

Abbildung 1.4: Topologie Stern

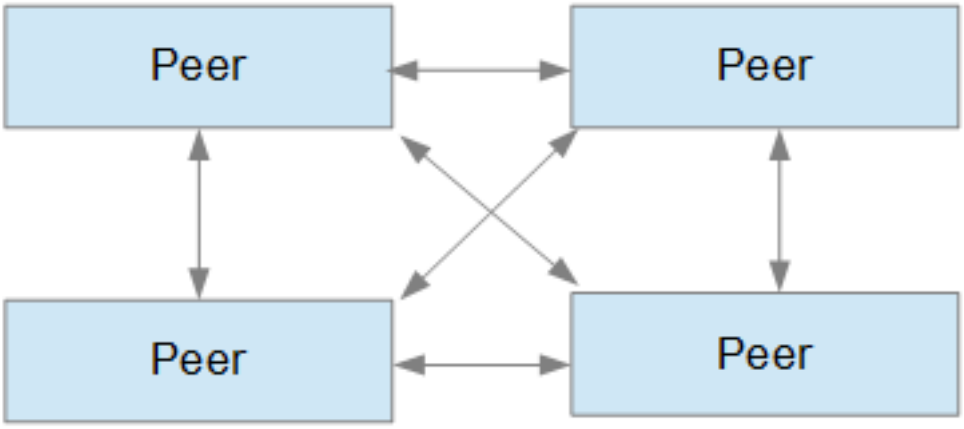

Abbildung 1.5: Topologie Vollvermascht

1.6 Eigenes Modul: Verwendung

1.6.1 Eigenes Modul Verwendung: Client

Abhängigkeiten

```
var io = require('socket.io-client');
var sigClient = require('./../../own_modules/WebRTC-COMM.js').Client;
```

Abbildung 1.6: Dependencies Verwendung Client

In Abbildung 1.6 sind die Clientseitigen Abhängigkeiten für das Modul zu sehen. Die Client-Implementation von socket.io ßocket.io-client"wird für das Signalling benötigt. Zur Nutzung des Moduls muss dieses außerdem "required"werden. Hierzu wird der Pfad zu dem Modul als Parameter an das require übergeben und der Client Export genutzt.

Allgemeine Verwendung

```
iosocket = io.connect();

iosocket.on('connect', function(){
    console.log("Now Connected to the Server.");

    if( !isConnected ) {
        client = new sigClient(iosocket, opts, "example1room", null);
        client.setMessageCallback(getMessage);
    }

    isConnected = true;
});
```

Abbildung 1.7: Verwendung Client

In Abbildung 1.7 ist zu sehen, wie das Modul auf Clientseite genutzt werden kann. Vorbedingung ist, dass wie zu sehen, eine Verbindung des socket.io WebSocket besteht. Sobald diese besteht kann ein neues ßigClientÖbjekt erstellt werden. Diesem müssen vier Parameter übergeben werden:

Socket Der verbundene ßocket.ioSocket.

Opts Etwaiige Optionen, alle optional. Mehr dazu unter 1.6.4.

Roomname Der Name des Raums, dem der Client beitreten will.

Placeholder Ein Platzhalter für mögliche Zweitmodi.

Nachricht Empfangen

```
client.setMessageCallback(getMessage);
```

Abbildung 1.8: Client Nachricht empfangen

In Abbildung 1.8 ist zu sehen, wie eine Callback Funktion für neue Nachrichten gesetzt wird. Diese Funktion wird immer dann aufgerufen, wenn eine neue Nachricht empfangen wurde. Näheres zu der Struktur von Nachrichten unter 1.6.3.

Nachricht Senden

```
client.sendMessage(type, data, from);
```

Abbildung 1.9: Client Nachricht senden

In Abbildung 1.9 ist zu sehen, wie eine Nachricht versendet werden kann. Beim Versenden einer Nachricht sind zwei Parameter von relevanz:

Type Der Typ der Nachricht. Vom Nutzer frei wählbar. Dient der schnellen Zuordnung.

Data Die Nutzlast der Nachricht. Hierbei ist hervorzuheben, dass dies Nutzlast im Format eines JavaScript Objektes sein sollte. Sollte eine Verbindung mit WebRTC nicht hergestellt werden und der Backup-Modus ist aktiviert und unterstützt, kann dem Data-Objekt ein "toÄttribut hinzugefügt werden, welches die ID des Empfängers enthält um dem Server mitzuteilen, für wen diese Nachricht gedacht war.

1.6.2 Eigenes Modul Verwendung: Server

Abhängigkeiten

```
// Socket.io ist eine Dependency des P2P Moduls (-> Signalling)
var io = require('socket.io')(server);

// Als .Server ist die Serverseite des Moduls exportiert
var sigserver = require('./own_modules/WebRTC-COMM.js').Server;

// Unter .ServerRooms findet man alle aktuellen Räume (fals der Nutzer diese wissen will)
var sigserverrooms = require('./own_modules/WebRTC-COMM.js').ServerRooms;
```

Abbildung 1.10: Dependencies Verwendung Server

In Abbildung 1.10 ist zu sehen, welche Abhängigkeiten auf Serverseite bestehen. Für das Signalling nutzt das Modul die Serverseitige Implementation von ßocket.io". Zur Nutzung des Moduls muss dieses außerdem "required"werden. Hierzu wird der Pfad zu dem Modul als Parameter an das require übergeben und der Server Export genutzt. Die Verwendung der ServerRooms ist optional und gibt dem Nutzer die Möglichkeit auf dem Server eine Einsicht in die derzeitigen Räume zu erlangen und über WebSockets die in diesen hinterlegt sind mit den Clients zu Kommunizieren.

Konstruktor

```
function Server(iosocket, config)
```

Abbildung 1.11: Server Konstruktor

In Abbildung 1.11 ist der allgemeine Konstruktor des Moduls für Server zu sehen.

Allgemeine Verwendung

```
io.on('connection', function(socket){

    /*
    new sigserver(socket, {});
});
```

Abbildung 1.12: Nutzung Server

In Abbildung 1.12 ist zu sehen, wie das Modul für Server genutzt wird. Für jeden neuen Client der sich erfolgreich mit dem Server verbunden hat muss ein neues Modul.Server Objekt erstellt werden. An dieses Modul muss der Socket (socket.io) und etwaige optionale Parameter/Optionen übergeben werden. Die Erstellung dieses Objektes führt dazu, dass Eventhandler des WebSocket für das Handling des Signalling erstellt werden und der Nutzer einem Raum zugewisen werden kann. Dies geschieht alles automatisch und intern im Modul. Der in Abbildung 1.12 zu sehende Code stellt die Minimalimplementation dar, wie sie auch im Projekt "Ping-Pong"verwendet wird. In den meisten Fällen reicht diese aus.

1.6.3 Eigenes Modul Verwendung: Callbacks

Alle wichtigen Ereignisse, welche den reibungslosen Aufbau und Betrieb der Verbindung betrifft, wird intern geregelt. Dem Nutzer des Moduls ist es jedoch möglich, eigene Callbacks zu definieren, die bei gewissen Ereignissen aufgerufen werden. In Abbildung 1.13 sind die drei

```
Client.prototype.setMessageCallback = function(newCallback) {
    this.messageCallback = newCallback;
};

Client.prototype.setNewConnectionCallback = function(newCallback) {
    this.newConnectionCallback = newCallback;
};

Client.prototype.setDisconnectCallback = function(newCallback) {
    this.disconnectCallback = newCallback;
};
```

Abbildung 1.13: Callbacks

für den Nutzer wichtigsten Funktionen zum Eintragen von Callbacks zu erkennen.

setMessageCallback Mit dieser Funktion trägt man eine Callback Funktion ein, die Falle einer neuen Nachricht aufgerufen wird. Die Callback Funktion wird hierbei mit drei Parametern aufgerufen:

Type Dem Typen den die Nachricht hat, vergeben von dem Sender.

From Der ID des Absenders der Nachricht.

Data Die eigentliche Nutzlast der Nachricht. Format vom Sender bestimmt.

setNewConnectionCallback Mit dieser Funktion trägt man eine CallbackFunktion ein, die im Falle einer neuen erfolgreichen Verbindung zu einem anderen Peer aufgerufen wird. Die Callback Funktion wird hierbei mit einem Parameter aufgerufen:

PeerID Die ID des neuen Peers.

setDisconnectCallback Mit dieser Funktion trägt man eine Callback Funktion ein, die im Falle des Verbindungsverlusts eines Peers aufgerufen wird. Die Callback Funktion wird hierbei mit einem Parameter aufgerufen.

PeerID Die ID des Peers, welcher die Verbindung verloren hat.

1.6.4 Eigenes Modul Verwendung: Optionen

Client Beim Erstellen eines Client-Objektes können gewisse Optionen mit übergeben werden. Hierbei kann eine beliebige Anzahl dieser genutzt werden. Wird eine Option nicht genutzt, tritt ein default in Kraft (default mit = markiert):

mode (=single|multi) Der Modus, den der Peer nutzen soll.

maxpeers (=1|1..2$^{32}-1$ Die Anzahl der maximal möglichen gleichzeitigen Verbindungen. (Relevant für "Multi-Peer")

usebackup (=false|true) Die Möglichkeit der Verwendung von WebSockets als Backup. Diese option muss der Server auch Unterstzützen.

Server Beim Erstellen eines Server-Objektes können gewisse Optionen mit übergeben werden. Hierbei kann eine beliebige Anzahl dieser genutzt werden. Wird eine Option nicht genutzt, tritt ein default in Kraft (default mit = markiert):

allowtopology$_s tar(= true|false)$ Sterntopologie erlauben.

allowtopology$_1 v1(= true|false)$ Eins zu eins "Topologieërlauben.

allowtopology$_a va(= false|true)$ Vollvermaschung erlauben. Default ist false, da hierdurch auch für den Server eine große Last durch die vielen Signale entstehen kann.

usebackup (=false|true) Die Möglichkeit der Verwendung von WebSockets als Backup. Kann sehr Performaceintensiv werden.

1.7 Eigenes Modul: Performance

1.7.1 Testbedingungen

Diese Tests dienen ausschließlich der bewertung und Bildung eines Leistungsschemas für das Modul. Aus diesem Grund wurde der Faktor Netzwerk auf ein minimum begrenzt und nur Computer in einem LAN verwendet. Folge daraus ist, dass die Adressen dem LAN zugeordnet werden konnten und ein Routing außerhalb des LAN nicht nötig war. Eine durchschnittliche Latenz von 1ms war gegeben (laut Anzeige, Messung erfolgte im ms Bereich). Ablauf eines jeden Tests:

1. Es wurde ein Node.JS Server mit Minimalimplementation des Moduls gestartet. (Für Minimalimplementation siehe 1.6.2)

2. Ein einzelner "Multi-Peer"wurde gestartet. Dieser tritt einem Testraum bei.
3. Mit Hilfe eines kleinen Test-Programms wurde eine gewisse Anzahl an SSingle-PeerSSendern gestartet und via WebRTC mit dem "Multi-Peer"Host im gleichen Raum verbunden.
4. Nachdem alle Peers erfolgreich verbunden sind wird eine Startzeit des Tests gespeichert.
5. Die Sender fangen nun an in einer gewissen Frequenz bestimmt oft eine Nachricht an den "Multi-Peerßu senden. Die Payload dieser Nachricht besteht aus:

   ID Die eigene ID. Nötig da mehrere Sender auf einem System befindlich sind und nur so die zuordnung der Ergebnisse möglich ist.

   Time Ein Zeitstempel der direkt zum Sendezeitpunkt generiert wurde.

   Index Ein Index der für ID der versendeten Nachricht steht.

   Aus ID und Index lässt sich eine Nachricht exakt zuweisen.
6. Empfängt der "Multi-Peerëine Nachricht schickt er diese einfach direkt zum Absender zurück.
7. Empfangt einer der SSingle-Peerëine Nachricht, tragen sie in ein Ergebnisobjekt den Sende- und Empfangszeitpunkt ein.
8. Sind alle Sender mit ihren Sende-Iterationen durch und es wurde die gleiche Anzahl an Nachrichten wieder empfangen (Anzahl der Ergebnisobjekte), wird wieder ein Zeitstempel generiert.
9. Nachdem Sende- und Empfangszyklen durch sind werden die Ergebnisobjekte ausgewertet. Dabei wird ermittelt:

   Gesamtlaufzeit Ermittelt aus Start- und Endzeitpunkt.

   Gesamtzahl der Nachrichten Anzahl aller empfangenen Nachrichten.

   Minimale Delay Kleinste bei einer Nachricht ermittelte Verzögerung.

   Maximale Delay Größte bei einer Nachricht ermittelte Verzögerung.

   Durchschnittliche Delay Mittelwert aller ermittelten Verzögerung.

   Zeit pro Nachricht Zeitabstand zwischen Nachrichten.
10. Wiederhole Test noch 9 mal und mittle Ergebnisse.

Erfolgskriterien

Hard Das Harte Kriterium ist, dass das Maximale Delay nicht über der Zeit pro Nachricht liegen darf. Die Nachricht eines jeden Senders wurde also verarbeitet und beantwortet, bevor er eine neue sendet.

Soft Das Weiche Kriterium ist, dass das durchschnittliche Delay nicht über der Zeit pro Nachricht liegen darf. Die Nachricht eines jeden Sendern wurde also in der Regel verarbeitet und beantwortet, bevor er eine neue sendet.

Failed Sobal das weiche Kriterium nicht mehr erfüllt ist, ist ein Erfolg im allgemeinen nicht mehr gegeben. Je nach Anwendungsfall kann auch ein Failed noch ausreichend sein und es müssten speziellere Kriterien herangezogen werden.

1.7.2 Allgemeiner Performanceaspekt

Bei diesen Tests sollte berücksichtigt werden, dass der Verarbeitungsaufwand nach erhalt der Nachrichten sich lediglich auf das richtige weiterleiten der Nachricht beschränkt. Dabei wird die Nachricht an eine Callback Funktion weitergereicht und dann eine neue Nachricht an einen Peer versendet. Tragend für die Performance ist hier die Verwaltung der Verbindungen. Alle Verbindungen werden in ässoziativenÄrrays verwaltet. Da ein ässoziativesÄrray in JavaScript eigentlich nicht existiert, sind dies Objekte mit Attributen.

Lookup

Je nach Browser kann das "lookup"verschiedene Aufwände annehmen. Bei modernen Browsern kann von einem Worst-Case von O($log_2\,(N)$) $ausgegangenwerden.BeimodernenBrowsernkannvoneinem$
$CasevonO\,(1)\,ausgegangenwerden.BeilterenBrowsernkannvoneinemWorst-CasevonO\,(N)\,ausgega$

Zuweisung

Zuweisung von Werten auf bereits bestehende Attribute von Objekten fällt unter gleichen Aufwand wie "lookup". Zuweisung von Werten auf neue Attribute hat einen Aufwand von O(1) $imBest-CasebisO\,(N)\,imWorst-CasejeBrowserimplementation.InmodernenBrowsernistvon$

Concurrency

Concurrency ist mit JavaScript unter den hier genutzten Techniken nicht möglich, somit müssen keine Synchronisationsmechanismen untersucht werden.

1.7.3 Eigenes Modul Performance: Beispiel am Projekt

Beschreibung

Dieser Test soll die Bedingungen des Projektes wiederspiegeln.

Durchführung

Anzahl Sender Als Anzahl der Sender für die Versuche wurde festgelegt: 2, 3, 4, 6, 10, 15, 25, 50, 75, 100, 125, 150.

Frequenz der Nachrichten Jeder Sender sendete 20 Nachrichten pro Sekunde, also alle 50ms eine Nachricht.

Iterationsgröße Jeder Sender sendete 600 Nachrichten.

Folge: Nachrichtendurchsatz Aus Anzahl der Sender und Frequenz entstand folgender Durchsatz(in Nachrichten/s): 40, 60, 80, 120, 200, 300, 500, 1000, 1500, 2000, 2500, 3000.

Ergebnis

In Abbildung 1.14 ist das Ergebnis der Tests zu erkennen. Auf der Y-Achse ist die Verzögerung logarithmisch skaliert in ms dargestellt. Auf der X-Achse ist die Anzahl der Sender dargestellt. Die Balken stellen die maximale Verzögerung dar. Die gelbe Linie stellt die durchschnittliche Verzögerung dar. Die rote Linie stellt die Sende-Verzögerung pro Nachricht dar. Befindet sich ein Balken über der roten Linie ist das "HardKriterium nicht mehr erfüllt. Befindet sich die gelbe Linie über der roten Linie ist das SSoftKriterium nicht mehr erfüllt.

Fazit

In Abbilding 1.15 ist das Ergebnis noch einmal Tabellarisch dargestellt. Zu erkennen ist, dass das "HardKriterium noch bis 100 Peers erfüllt ist. Dies entsrpicht einem Nachrichtendurchsatz von 2000 Nachrichten pro Sekunde. Das SSoftKriterium ist noch bis 125 Peers erfüllt. Dies entspricht einem Nachrichtendurchsatz von 2500 Nachrichten pro Sekunde. Markante Punkte:

25 Sender Bei der Steigerung von 15 auf 25 Sender steigt die maximale Verzögerung nicht, jedoch die durchschnittliche Verzögerung um 53

75 Sender Bei 75 Sendern erreicht die maximale Verzögerung zum ersten mal mehr als 50% des SSoftKriterium.

125 Sender Bei 125 Sendern ist das "HardKriterium nicht mehr erfüllt.

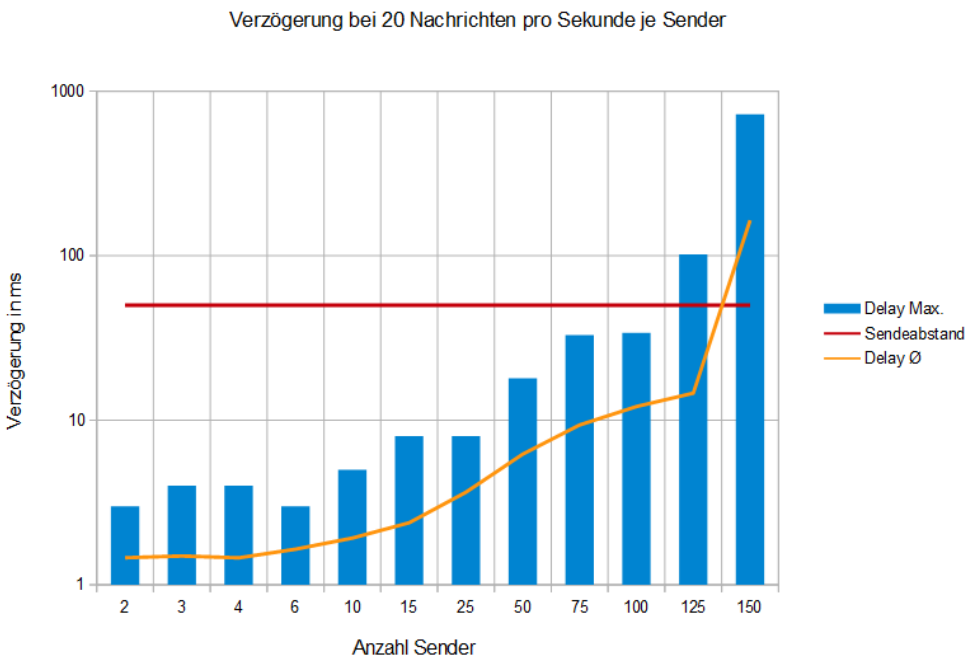

Abbildung 1.14: Diagramm Performancetest Normal

150 Sender Bei 150 Sendern ist das SSoftKriterium nicht mehr erfüllt.

Bezogen auf das Ping-Pong Projekt, welches Spieler-/Peerzahlen unter 10 nutzt, sind keine Auffälligkeiten zu erkennen und sowohl SSoft als auch "HardKriterium sind erfüllt.

1.7.4 Eigenes Modul Performance: Beispiel für Chat

Beschreibung

Folgender Test nimmt sich eine Chat-Anwendung als Beispiel.

Durchführung

Anzahl Sender Als Anzahl der Sender für die Versuche wurde festgelegt: 50, 100, 150, 200, 250.

Frequenz der Nachrichten Jeder Sender sendete 1 Nachricht pro Sekunde, also alle 1000ms eine Nachricht.

Iterationsgröße Jeder Sender sendete 30 Nachrichten.

Anzahl Sender	Delay Max.	Delay Ø
2	3	1,46
3	4	1,5
4	4	1,46
6	3	1,65
10	5	1,92
15	8	2,38
25	8	3,65
50	18	6,23
75	33	9,36
100	34	12,12
125	102	14,6
150	724	164,45

Abbildung 1.15: Tabelle Performancetest Normal

Folge: Nachrichtendurchsatz Aus Anzahl der Sender und Frequenz entstand folgender Durchsatz(in Nachrichten/s): 50, 100, 150, 200, 250.

Ergebnis

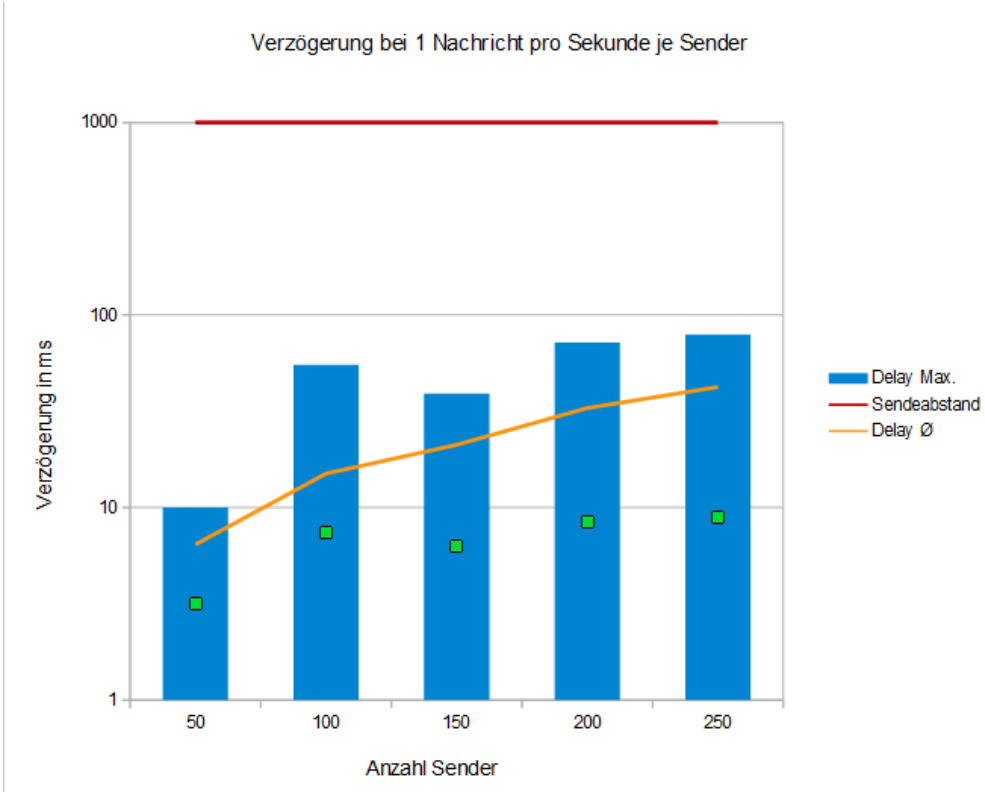

Abbildung 1.16: Diagramm Performancetest Chat

In Abbildung 1.16 ist das Ergebnis der Tests zu erkennen. Auf der Y-Achse ist die Verzögerung logarithmisch skaliert in ms dargestellt. Auf der X-Achse ist die Anzahl der Sender dargestellt. Die Balken stellen die maximale Verzögerung dar. Die gelbe Linie stellt die durchschnittliche Verzögerung dar. Die rote Linie stellt die Sende-Verzögerung pro Nachricht dar. Befindet sich ein Balken über der roten Linie ist das "HardKriterium nicht mehr erfüllt. Befindet sich die gelbe Linie über der roten Linie ist das SSoftKriterium nicht mehr erfüllt.

Fazit

In Abbildung 1.17 ist das Ergebnis noch einmal Tabellarisch dargestellt. Zu erkennen ist, dass auch bei 250 Sendern noch die SSoft und "HardKriterien erfüllt sind. Ein Test mit mehr Peers

Anzahl Sender	Delay Max.	Delay Ø
50	10	6,43
100	55	15,02
150	39	21,20
200	72	32,89
250	79	42,28

Abbildung 1.17: Tabelle Performancetest Chat

stellte sich als problematisch da, da der Browser je nach Implementation bei einer großen Anzahl gleichzeitiger Verbindungen anfängt einzelne Verbindungen, die eine Weile nicht genutzt wurden, zu schließen. Dies trat auch auf, da die Initialisierungsphase des Tests diese Inaktivitätsschwälle überschritt. Gleichmäßige Testdurchläufe waren nicht mehr möglich. Bester Einzeldurchlauf: 950 Peers. Inkonsistent.

Chatanwendungen auf dieser Basis stellen theoretisch keine Probleme dar.

1.7.5 Eigenes Modul Performance: Beispiel extreme

Beschreibung

Das extreme Beispiel soll zeigen, wie sich eine hohe Sendefrequenz auswirkt. Beispiel hierfür soll sein, dass Steuerdaten zu jedem Bild verarbeitet werden sollen. Hierbei markant, es soll sich um einen 144Hz Monitor handeln. (Aufgrund Rundung: 142.85Hz)

Durchführung

Anzahl Sender Als Anzahl der Sender für die Versuche wurde festgelegt: 20, 25, 30, 35, 40.

Frequenz der Nachrichten Jeder Sender sendete 142.85 Nachricht pro Sekunde, also alle 7ms eine Nachricht.

Iterationsgröße Jeder Sender sendete 4320 Nachrichten.

Folge: Nachrichtendurchsatz Aus Anzahl der Sender und Frequenz entstand folgender Durchsatz(in Nachrichten/s): 2857, 3571.25, 4285.5, 4999.75, 5714.

Ergebnis

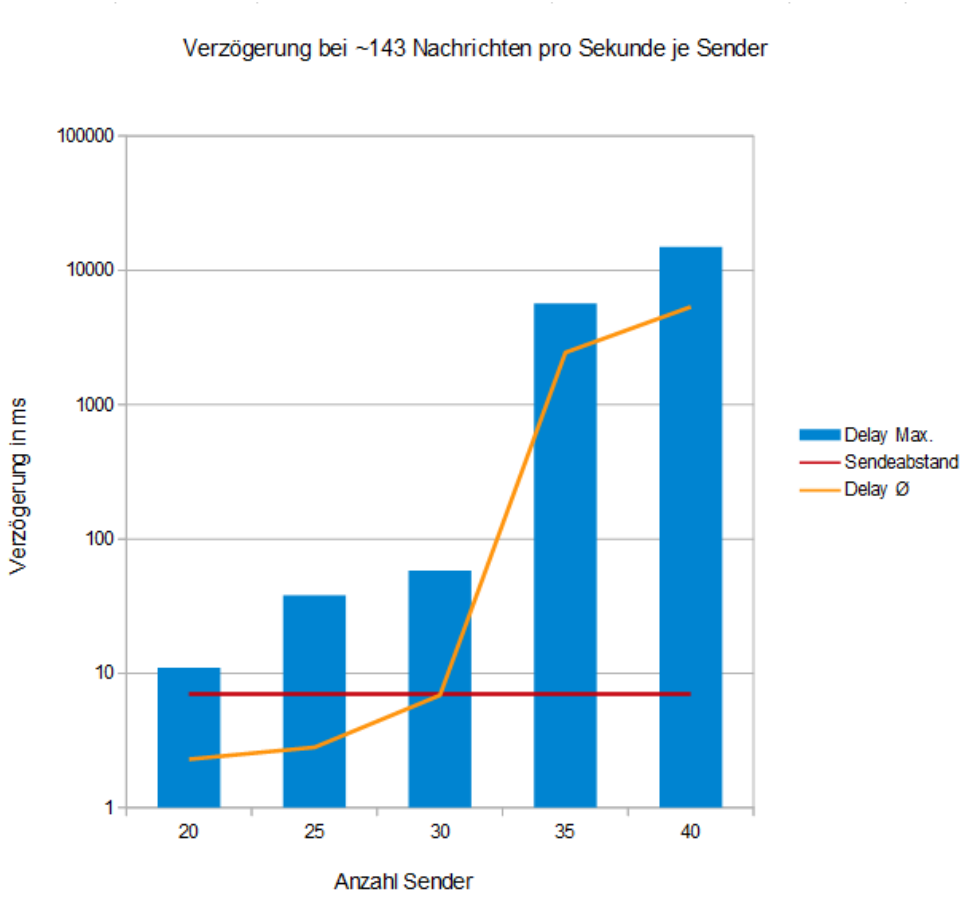

Abbildung 1.18: Diagramm Performancetest Chat

In Abbildung 1.16 ist das Ergebnis der Tests zu erkennen. Auf der Y-Achse ist die Verzögerung logarithmisch skaliert in ms dargestellt. Auf der X-Achse ist die Anzahl der Sender dargestellt. Die Balken stellen die maximale Verzögerung dar. Die gelbe Linie stellt die durchschnittliche Verzögerung dar. Die rote Linie stellt die Sende-Verzögerung pro Nachricht dar. Befindet sich ein Balken über der roten Linie ist das "HardKriterium nicht mehr erfüllt. Befindet sich die gelbe Linie über der roten Linie ist das SSoftKriterium nicht mehr erfüllt.

Anzahl Sender	Delay Max.	Delay Ø
20	11	2,29
25	38	2,81
30	58	6,85
35	5648	2443,33
40	14916	5349,20

Abbildung 1.19: Tabelle Performancetest Chat

Fazit

In Abbildung 1.19 sind die Ergebnisse noch einmal Tabellarisch dargestellt. Zu erkennen ist, dass das "HardKriterium schon bei 20 Sendern nicht mehr erfüllt ist. Außerdem zu erkennen ist, dass ab 35 Sendern auch das SSoftKriterium nicht mehr erfüllt ist.

Markante Punkte:

20 Sender Bei 20 Sendern ist das "HardKriterium nicht mehr erfüllt, die durchschnittliche Verögerung aber noch sehr gut.

25 Sender Im Gegensatz zu 20 Sendern, hat sich bei 25 Sendern die Maximale Verzögerung um 245% auf 38ms erhöht. Anzeichen für einen zeitweisen Nachrichtenstau.

30 Sender Im Gegensatz zu 25 Sendern, hat sich bei 30 Sendern die durchschnittliche Verzögerung um 143% auf 6.85ms erhöht. Ein Zeichen dafür, dass immer öfter Nachrichtenstaus auftreten.

35 Sender Im Gegensatz zu 30 Sendern, hat sich bei 35 Sendern die maximale Verzögerung um 9637% auf 5648ms und die durchschnittliche Verzögrung um 35569% auf 2443.33ms erhöht. Hierbei ist von einem Durchgängig anwachsendem Nachrichtenstau auszugehen.

40 Sender Noch einmal hat sich im Gegensatz zu 35 Sendern bei 40 Sendern die Verzögerung noch einmal vervielfacht.

Sofern das "HardKriterium kein muss ist, können solche Mengen an Nachrichten noch bis 30 Sender verarbeitet werden.

1.8 Eigenes Modul: Möglichkeiten

1.8.1 Eigenes Modul Möglichkeiten: Verwendbar

Das Modul ist wie im Projekt genutzt getestet und verwendbar. Es ist generisch nutzbar. Einige zusätzliche Features, welche auch für das Projekt nicht von Relevanz waren, sind nur teilweise implementiert oder nicht getestet. Dies ist auf den großen Umfang eines solchen Modules zurückzuführen.

Bei diesen teilweise implementierten Features handelt es sich um:

1. Interaktiven Raumwechsel.
2. Benutzerspezifische Sendekanäle.
3. Echtzeit Verbindungsanalysen.

Bei den nicht ausführlich getesteten Features handelt es sich um:

- Topologie Eins-zu-Eins und Vollvermascht. Grund ist, dass hierfür weitere Testanwendungen nötig wären.
- Backup funktionalität über WebSocket. Grund ist, dass ein Test in verschiedenen Netzen noch erfolgen muss.

1.8.2 Eigenes Modul Möglichkeiten: Erweiterbar

- Angesichts der verschiedenen möglichen Topologien gibt es für die Peers nur zwei verschiedene Verhalten. Entweder sie pflegen nur eine Verbindune, oder mehrere. Da dieses Verhalten bereits modelliert ist, ist die implementation neuer Topologien auf Serverseite zu geschehen, welche die Räume verwaltet.
- Als Verbindungsobjekte werden unveränderte RTCPeerConnections verwendet, wodurch das hinzufügen neuer Datenkanäle ein neues Verwalungsobjekt auf Clientseite benötigen würde. Außerdem müssten die verschiedenen Datenkanäle entweder auf mehrere Callback Funktionen gemapped werden, oder ein Parameter für die eine Callback Funktion definiert werden, welche die Datenkanäle logisch trennt.
- Für die Implementation von Speziellen Datenkanälen, wie z.B. Medienkanälen, muss ähnlich wie bei den normalen Datenkanälen, ein Verwaltungsobjekt geschaffen werden, und etwaiige Callback Funktionen bestimmt.

- Die Implementation eines Senderelays (A, B, C und D sind Peers. A sendet an D über B und C: A->B->C->D) liegt in Nutzerhand. Es müsste eine Implementation stattfinden bei dem ein Client (B und C) zwei Verbindungen herstellt. Dabei pflegt B jeweils eine zu A und C, C jeweils zu B und D. Ankommende Nachrichten müssten dann an die andere Verbindung weitergereicht werden. Diese Nutzer-Implementation könnte dann auch die Teilvermaschung ermöglichen.

1.9 Fazit der Arbeit am Modul

See also ?.

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

Hamburg, 6. März 2017 Andreas Müller