Technische Hochschule Lübeck

BACHELORARBEIT
Websocket Relay Service

Cedric Pump

2019

TECHNISCHE HOCHSCHULE LÜBECK

Bachelor-Thesis

Websocket Relay Service

Vorgelegt von: Cedric Pump

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik / Softwaretechnik

Erstprüfer: Professor Dr. Kratzke

Ausgabedatum: 27. Mai 2019
Abgabedatum: 27. August 2019

A. Hanemann

(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung:

Für einige Browseranwendungen (z.B. Online Games, aber auch Chat-Applikationen, Code-Editoren, etc.) ist es sinnvoll, Echtzeit-artige P2P-Verbindungen zu haben - zum Beispiel um Spielzustände in Multiplayergames abzugleichen. Da Webbrowser meist nur Verbindungen aufbauen, aber (aus Sicherheitsgründen) keine Verbindungen annehmen können, sind P2P-Ansätze im Webclient-Umfeld häufig nicht gangbar.

Daher soll diese Arbeit einen Websocket-basierten Ansatz verfolgen und einen leichgewichtigen Websocket Relay Service realisieren, mit dem sich Clients Message-basiert koppeln lassen. Der Websocket Relay Service soll Messages von einem Client an einen Channel an andere Clients weiterleiten, die sich für diesen Channel angemeldet haben (um bspw. den Spielzustand auf mehreren Geräten abzugleichen).

Die Arbeit soll dabei aktuell existierende WebSocket Relay Lösungen recherchieren, diese überblicksartig darstellen und deren Funktionsumfang miteinander vergleichen. Auf Basis dessen soll ein eigener WebSocket Relay Service realisiert und in Form eines standardierten Container Image (Docker, OCI) bereitgestellt werden.

Der Websocket Relay Service soll Channels für die Messaging-Patterns

- Publish-Subscribe (z.B. für Broadcast Anwendungsfälle)
- Push-Pull (z.B. für Loadbalancing Anwendungsfälle, fan-in/out)

unterstützen. Der Service soll mit der Basis-WebSocket API verwendet werden können. Eine REST-basierte API zum Anlegen, Löschen und Abonnieren von Channels ist vorzusehen.

Im Rahmen der Nachweisführung sind neben der Korrekheit und Nutzbarkeit der Implementierung, auch die Performance und Leistungsgrenzen zu erheben und mit existierenden Lösungen zu vergleichen.

Die Lösung ist nicht nur als Quelltext, sondern auch als standardisierte Deployment Unit in Form eines OCI/Docker-konformen Containers bereitzustellen. Es sind dabei die 12-Factor App Prinzipien zu berücksichtigen.

Es ist ferner zu untersuchen, wie dieser Service horizontal skalierbar (scale out) gestaltet werden kann. Die horizontale Skalierbarkeit ist mindestens prototypisch nachzuweisen.

Die schriftliche Ausarbeitung soll auf den Kontext dieser Arbeit, Websockets im Allgemeinen und deren Einsatzgebiete, existierende Websocket Relay Lösungen, die sich aus dem Kontext ergebenden Anforderungen (Messaging-Pattern, Channels, 12-Factor App, etc.), die Implementierung inkl. Architektur, die Nachweisführung sowie Grenzen und Limitierungen der Lösung eingehen. Ergänzend soll die Arbeit erläutern, wie die Lösung für eigene Projekte deployt und Client-seitig eingesetzt werden kann (Tutorial).

Professor Dr. Kratzke

TECHNISCHE
HOCHSCHULE
LÜBECK
Fachbereich
Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

Erklärung zur Abschlussarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

18.08.2019
(Datum)

Unterschrift

Zusammenfassung der Arbeit
Abstract of Thesis

Fachbereich:
Department: **Elektrotechnik und Informatik**

Studiengang:
University course: **Informatik / Softwaretechnik**

Thema:
Subject: **WebSocket Relay Service**

Zusammenfassung:

Die vorliegende Arbeit recherchiert aktuell existierende WebSocket Relay Lösungen und stellt diese dar. Ein leichtgewichtiger WebSocket Relay Service wird entwickelt, der die Koppelung von Clients mittels WebSocket Protokoll ermöglicht. Die Kommunikation mehrerer Clients erfolgt über Kanäle, die Nachrichten auf Basis spezieller Messaging-Patterns weiterleiten. Die Verwaltung dieser Channel, sowie der Verbindungsaufbau des Clients wird über eine REST-API realisiert. Die entwickelte Software wird auf Korrektheit, Nutzbarkeit, Leistung und Skalierbarkeit untersucht und mit der Featureunterstützung existierender Lösungen verglichen.

Abstract:

The following work researches WebSocket Relay Solutions that currently exist and summaries them clearly. A lightweight WebSocket Relay Service is developed that provides connecting clients via WebSocket Protocol. Clients are communicating on channels witth relay messages according to messaging-patterns. A REST-API is used for channel management and client connection. This software is evaluated in terms of correct behaviour, usability, performance and scalability and is compared to the feature support of existing solutions.

Verfasser:
Author: **Cedric Pump**

Betreuender Professor/in:
Attending Professor: **Prof. Dr. rer. nat. Dipl.-Inform. Nane Kratzke**

WS / SS: **SS 2019**

Inhaltsverzeichnis

1 Einleitung 10
1.1 Hintergrund und Ziele der Arbeit 10
1.2 Gliederung des weiteren Dokumentes 10
2 Grundlagen 11
2.1 Webservices 12
2.2 Verteilte Systeme 12
2.2.1 Horizontale Skalierbarkeit 12
2.2.2 Raft Algorithmus 13
2.3 Messaging-Patterns 15
2.3.1 Publish-Subscribe 15
2.3.2 Request-Reply 16
2.3.3 Push-Pull 16
2.4 REST 17
2.5 WebSockets 18
2.5.1 Alternativen zu WebSockets 18
2.5.2 Technischer Aufbau 19
2.5.3 Verwendung von WebSockets 19
2.6 Existierende Services 20
3 Anforderungsanalyse 22
3.1 Zielgruppe 22
3.2 Nutzeranforderungen 23
3.3 Systemanforderungen 24
3.3.1 Funktionale Systemanforderungen 24
3.3.2 Nicht Funktionale Systemanforderungen 24
3.3.3 Erläuterung der Anforderungen 26
3.3.4 Abdeckung der Nutzeranforderungen 27
3.3.5 Priorisierung der Anforderungen 27
4 Architektur 30
4.1 Aufbau des Service 30
4.1.1 REST-API 31
4.1.2 RelayServer 33
4.1.3 RaftNode 35

5 Implementierung und Entwurfsentscheidungen 38
5.1 Wahl der Programmiersprache 38
5.1.1 Implementierungsaufwand 38
5.1.2 Performanz 38
5.1.3 Entscheidung 39
5.2 Verwendete Module 40
5.3 Anpassung der Messaging-Pattern 42
5.4 Horrizontale Skalierbarkeit 45
5.4.1 Entwurf des Service 46
5.5 Raft commands 49
5.6 Remote-Clients 50
5.7 WebSocket Connection 50
5.8 Offene (bekannte) Probleme 51
5.8.1 Log Snapshot 51
5.8.2 Unregelmäßigkeiten der Lastverteilung 51
5.8.3 API Responses 51
6 Nachweisführung 52
6.1 Nachweis der Korrektheit der Anwendung 52
6.1.1 Testen mit Mocha und Chai 52
6.1.2 Nicht-funktionale Anforderungsabdeckung 56
6.2 Performance 58
6.2.1 Loadtest mit NPM Loadtest 58
6.2.2 Loadtest mit Apache JMeter 60
6.2.3 Performance bei Skalierung 63
6.2.4 Vergleich mit Florkestra 65
7 Fazit 65
7.1 Zusammenfassung 65
7.2 Bewertung des Service 66
7.3 Ausblick 66
Abbildungsverzeichnis 68
Tabellenverzeichnis 69
Literaturverzeichnis 70
Anhang 74

1 Einleitung

1.1 Hintergrund und Ziele der Arbeit

Die Webentwicklung ist mehr denn je, ein wichtiger Aspekt der Softwareentwicklung. Dies liegt nicht nur an der zunehmenden Onlinepräsenz unserer Gesellschaft, und der Verlagerung alltäglicher Prozesse in das World Wide Web (Online Shopping, Online Banking, Video Streaming, Online Anmeldung, Cloudspeicher) sondern auch an der zunehmenden Vernetzung unserer Alltagsgegenstände (Smart Home).
Die Grundlage von Webanwendungen bildet die Markup Language HTML und das Protokoll HTTP der Anwendungsschicht. Letzteres weist jedoch einen Nachteil auf, der für viele Anwendungsfälle unzureichend ist, die Unidirektionalität. Vor allem aus Sicherheitsgründen lässt dieses Protokoll Kommunikation nur in eine Richtung zu, als Anfrage vom Client zum Server. Die Serverseite hat lediglich die Möglichkeit auf Anfragen zu antworten. Möchte man neben dem von HTTP unterstützten Request-Response (auch Request-Reply) Pattern eine bidirektionale Kommunikation ermöglichen, kommt die WebSocket-Technologie ins Spiel. In Multiplayerspielen, Chatprogrammen, Social-Media Plattformen und anderen verteilten Systemen ist es notwendig den Client über Ereignisse informieren zu können. Wie dies mit einem WebSocket Relay Service zu ermöglichen, soll in diesem Dokument gezeigt werden.

Im Rahmen dieser Arbeit werden WebSocket Relay Services betrachtet, die es Anwendungen ermöglichen Nachrichten über einen WebSocket auszutauschen. Dafür werden die Grundlagen der Thematik verdeutlicht und der aktuelle Stand der Technik festgestellt. Letztendlich soll ein leichtgewichteter WebSocket Relay Service entwickelt und verifiziert werden, der nachrichten-basierten Vernetzung von Anwendungen eingesetzt werden kann. Bestandteil der Entwicklung dieses Service ist zu prüfen, wie ein solcher Relay-Service horizontal skaliert werden kann und einen prototypischen Ansatz umzusetzen.

1.2 Gliederung des weiteren Dokumentes

In Kapitel 2 werden zunächst alle Grundlagen vermittelt, die dem Verständnis des weiteren Dokuments dienen. Ein WebSocket Relay Service stellt einen Webservice (Kapitel 2.1) dar, der zu den verteilten Systemen (Kapitel 2.2) gehört. Beide Begriffe werden hier erklärt. Welche Konzepte hinter den Messaging-Patterns stehen, die dieser Service unterstützen soll, wird anschließend in Kapitel 2.3 erläutert. Es folgen die REST Architektur (Kapitel 2.4), die für Webservices verwendet wird, sowie das WebSocket Protokoll (Kapitel 2.5). Anschließend werden die bereits existierenden WebSocket Service Relay Lösungen (Kapitel 2.6) kategorisiert, tabellarisch dargestellt und bewertet.

Die Anforderungsanalyse folgt in Kapitel 3 und ermittelt eine Zielgruppe (Kapitel 3.1), sowie Nutzeranforderungen (Kapitel 3.2) für den zu entwerfenden Service. Aus diesen leiten sich die Systemanforderungen (Kapitel 3.3) ab, die anschließend erläutert und priorisiert werden.
Es folgt eine Beschreibung der Architektur in Kapitel 4, die den Aufbau des Service erläutert. In Kapitel 5 folgt die Beschreibung der Implementierung und der Entwurfsentscheidungen. Dazu gehört die Wahl der Programmiersprache, die Begründung verwendeter Module, sowie die technische Umsetzungen der Anforderungen.
Das fertige Produkt wird dann in Kapitel 6 auf korrekte Funktionalität geprüft (Kapitel 6.1), Performancegrenzen werden ermittelt und mit existierenden Lösungen verglichen (Kapitel 6.2). Abschließend werden im Kapitel 7 die Ergebnisse in einem Fazit reflektiert.

2 Grundlagen

Zunächst werden die Grundlagen des Themas vermittelt. Da der zu betrachtende Service einen Webservice darstellt, muss zunächst geklärt werden, was ein Webservice ist. In diesem Rahmen wird auf das Themengebiet der verteilten Systeme eingegangen, denn ein Message-Relay-Service ist nicht nur Bestandteil eines verteilten Systems, indem es mehrere Clients zu einem System verbindet, durch horizontale Skalierung wird ein solcher Service selbst zu einem verteilten System. Anschließend werden Messaging-Pattern, und Protokolle erklärt, die für das Verständnis zur Funktionsweise eines WebSocket Relay Service hilfreich sind. Unter anderem wird in diesem Kapitel erläutert was WebSockets sind, wie sie funktionieren und wo sie eingesetzt werden. Einige Alternativen zum WebSocket Protokoll werden genannt und mit dem Einsatz von WebSockets verglichen. Im Anschluss werden vorhandene WebSocket Relay Lösungen vorgestellt, verglichen, und untersucht, inwiefern diese das Problem lösen können.

2.1 Webservices

Webservices haben das Ziel Anwendungsfunktionalitäten über das Netzwerk zur Verfügung zu stellen. Hierbei soll der Service möglichst unabhängig von der verwendeten Programmiersprache oder dem verwendeten Betriebssystem angesprochen werden können. Daher werden standardisierte Schnittstellen verwendet, um einen vom System unabhängigen Zugriff zu ermöglichen. Ressourcen werden über Protokolle wie SOAP (Simple Object Access Protokoll) ausgetauscht. Dabei werden für die Repräsentation der Daten standardisierte Formate wie XML oder JSON verwendet [1]. Die Nutzung des Service wird in einer Dienstbeschreibung festgelegt, für die häufig die WSDL (Web Service Description Language) verwendet wird. Heutzutage werden Webservices häufig über eine REST-API (REpresentational State Transfer Application Programeming Interface) angesprochen [2].

2.2 Verteilte Systeme

Webservices stellen in der Regel verteilte System dar [1]. Verteilte Systeme zeichnen sich dadurch aus, dass sie trotz ihrer Verteilung auf mehreren unabhängigen Computern als ein zusammenhängendes System erscheinen. Dadurch lassen sich Vorteile der Skalierbarkeit nutzen. Die Lastverteilung, lokale Verfügbarkeit und Ausfallsicherheit werden verbessert, die Skalierung bringt jedoch auch Herausforderungen mit sich, die darin bestehen einen konsistenten Systemzustand sicher zu stellen [3].

2.2.1 Horizontale Skalierbarkeit

Eine Eigenschaft von verteilten Systemen ist die horizontale Skalierbarkeit. Im Gegensatz zur vertikalen Skalierung, bei der einer Ressource mehr Leistung hinzugefügt wird (Stärkere CPU oder mehr Speicher), besteht die horizontale Skalierung darin, einem System weitere Ressourcen hinzuzufügen. Diese Form der Skalierung kann mehrere Vorteile haben: Durch lokale Verteilung der Systemkomponenten können Ansprechzeiten der Nutzer reduziert werden. Ein verteiltes System mit vielen Knoten, die über den Globus verteilt sind, hat kürzere Latenzzeiten zum nächsten Knoten, als ein System dessen Anfrage gegebenenfalls weite Strecken zurücklegen muss. Die Replikation von Daten und Diensten schafft Ausfallsicherheit. Ein System bleibt funktionsfähig, auch wenn einzelne Komponenten ausfallen. Die Lastverteilung schafft eine höhere Belastungsgrenze des Gesamtsystems. Ein System mit nur einem Server kann schnell an seine Grenzen kommen, jede Leistungssteigerung eines Servers ist an einem gewissen Punkt physikalisch limitiert. In einem verteilten System, indem jeder Knoten dieselben Aufgaben bewältigen kann, können Anfragen auf die einzelnen Knoten verteilt werden. Man spricht vom sogenannten Loadbalancing, bei dem Anfragen nach geeigneten Algorithmen auf die Knoten eines Systems verteilt werden. Kommt ein System

dennoch an seine Grenzen kann es durch das Hinzufügen weiterer Knoten skaliert werden. Bei dieser Eigenschaft spricht man von der horizontalen Skalierbarkeit [3].

Bei allen Vorteilen entstehen durch die horizontale Skalierung eines Systems jedoch auch ganz typische Probleme. Ziel ist es die Verteilungstransparenz des Systems zu gewährleisten, also dem Nutzer zu verbergen, dass Ressourcen über verschiedene Computer verteilt sind. So soll der Nutzer nicht bemerken an welchem Ort die Ressource liegt, ob sie repliziert wurde oder ob andere Nutzer die Resource verwenden. Um dies zu gewährleisten, ist es notwendig das verteilte System in einem konsistenten Zustand zu halten. Diese Konsistenz muss auch dann aufrechterhalten werden, wenn Knoten des verteilten Systems ausfallen oder gegensätzliche Zustände in einem System entstehen. Zudem muss ein solches Cluster, also ein zusammenhängendes Netzwerk aus Computern, verwaltet werden [3].

Für die Verwaltung von Clustern und der Sicherstellung von Konsistenz gibt es verschiedene Algorithmen, darunter der weit verbreitete Paxos Algorithmus [4] und den Raft Algorithmus. Paxos ist ein Protokoll, das verteilten Knoten ermöglicht replizierte Daten über mehrere Knoten zu verteilen und sich auf einen gemeinsamen Zustand zu einigen, auch dann, wenn sich die Zusammensetzung des Clusters ändert. Paxos hat jedoch den Nachteil, dass der es Algorithmus sehr schwer zu verstehen ist und daher schwer praktisch zu implementieren ist. [4] Aus diesen Gründen wurde der Raft-Algorithmus entwickelt. Da Raft somit ein gut verständlicher Konsens Algorithmus ist wird auf diesen Algorithmus im Folgenden näher eingegangen.

2.2.2 Raft Algorithmus

Raft ist ein Konsens Algorithmus, der im Jahre 2014 und der Stanford Universität entwickelt wurde. Die Entwicklung fand unter dem Gesichtspunkt der Verständlichkeit des Algorithmus statt, unter anderem, damit eine Implementierung einfach sei.

Der Raft Algorithmus ermöglicht es ein Log, also eine Liste von Operationen oder "Commands", über mehrere Knoten hinweg zu replizieren. Bei dem Verwalten des Clusters setzt Raft auf das Prinzip des starken Führers oder "Leaders". Dieser Ansatz ist zu Teilen ebenfalls der Verständlichkeit des Algorithmus geschuldet. Der Leader wird aus allen Knoten gewählt. Grundsätzlich kann jeder Knoten einen von drei Zuständen einnehmen. Er kann ein Leader sein, dieser ist für die Verwaltung des replizierten Logs verantwortlich. Er fügt dem Log neue Einträge hinzu und teilt sie mit den anderen Knoten. Erfüllt ein Eintrag gewisse Voraussetzungen wird er als sicher angesehen und als "commited" bezeichnet. Alle anderen Knoten sind "Follower" und ordnen sich dem Leader unter. Über regelmäßige Nachrichten, den Heartbeats, werden die Follower über den Zustand von Cluster und Log informiert. Bleibt dieser Heartbeat für eine gewisse Zeit aus, geht der entsprechende Knoten in den "Candidate" Zustand über und beginnt die Wahl eines neuen Leaders. Die Periode zwischen zwei Wahlen

stellt einen "Term" dar. Der Raft Algorithmus teilt das Problem somit in drei Teilprobleme, die Wahl des Leaders (Leader election), die Verteilung der Logeinträge (Log replication) und den Safety-Nachweis. Letzterer stellt sicher, dass ein Logeintrag, der einmal als sicher gilt, auf alle weiteren Knoten, mit demselben Wert gespeichert werden wird.
Der Raft Algorithmus verwendet Remote Procedure Calls (RPCs) um mit anderen Knoten zu kommunizieren. RPCs gehören in das Feld der verteilten Systeme und stellen Funktions- oder Methodenaufrufe dar, die aus einem Programm heraus meist über ein Netzwerk in einem anderen System ausgeführt werden. Im Folgenden wird auf verschiedene RPCs des Raft Algorithmus eingegangen.

Knoteneigenschaften

Jeder Knoten speichert Variablen, die seine Eigenschaften oder das Cluster betreffen. Darunter befinden sich der Zustand des Knotens (Leader, Follower oder Candidate), eine Liste der Logeinträge, der aktuelle Term, der Eintrag der zuletzt hinzugefügt wurde und der Eintrag, der zuletzt commited wurde.

Leader election

Jeder Knoten beginnt seinen Lebenszyklus als Follower. Erhält der Follower Heartbeats von einem gültigen Leader, so bleibt er Follower. Bleibt der Heartbeat aus, so startet der Follower eine Wahl und somit einen neuen Term und wird zum Candidate.
Die Wahl wird gestartet, indem der Candidate seinen Term erhöht und anschließend alle anderen Knoten auffordert abzustimmen, dabei richtet sich die Stimme der Knoten nach Aktualität des Logs und dem Term des Candidate Knotens. Ein Knoten gewinnt die Wahl durch Mehrheitsentscheid. Gewinnt ein Knoten die Wahl, wird er zum neuen Leader. Gewinnt kein Knoten die Wahl, wird nach einer zufälligen Zeit neu abgestimmt. Die Abstimmung geschieht über den RequestVote RPC der vom Candidate Knoten an alle anderen gesendet wird und dessen Ergebnis die Stimme des Knoten enthält.

Log replication

Der Leader des Raft Clusters nimmt Commands entgegen, die dem replizierten Log hinzugefügt werden. Diese Commands werden ihm mittels ClientRequest RPC übergeben.

Dieser Eintrag wird dem Log hinzugefügt und dann mittels AppendEntries RPC mit allen anderen Knoten geteilt. Der Raft Algorithmus stellt sicher, dass ein Logeintrag, der auf der Mehrheit aller Knoten an derselben Stelle des Logs existiert, später nicht überschrieben oder gelöscht werden kann. Daher gilt ein Eintrag als sicher (commited) sobald die Majorität aller Knoten dem Leader bestätigt, den Eintrag an der richtigen Stelle hinzugefügt zu haben.

Sollte es Differenzen zwischen dem Log des Followers und dem des Leaders geben, so werden diese beseitigt bevor der neue Eintrag eingefügt wird.

Durch geeignete Bedingungen für die Leader election sowie die Handhabung von unterschiedlichen Logeinträgen durch Erreichbarkeitsausfälle wird sichergestellt, dass alle Einträge vor und inklusive des zuletzt commitetem Eintrag auf mindestens der Mehrheit der Knoten identisch sind. Folgende Bedingungen tragen dazu bei:

Es gibt in jedem Term nur einen Leader. Der Leader löscht oder überschreibt niemals Einträge in seinem Log. Ein Logeintrag der commited ist wird im Log jedes zukünftigen Leaders vorhanden sein. Knoten die nicht auf dem aktuellen Stand sind hohlen ihr Defizit automatisch auf. [5] Auf die Einzelheiten und Beweise dieser Eigenschaften wird hier nicht weiter eingegangen.

2.3 Messaging-Patterns

Um die Funktionsweise eines WebSocket Relay Service zu verstehen, ist es sinnvoll auf einige Kommunikationsschemata die für Nachrichtenaustausch verwendet werden können einzugehen. Im Folgenden werden diese Messaging-Pattern, ihre Komponenten und Funktionsweisen und ihre Einsatzgebiete erklärt. Die Auswahl der Messaging-Patterns ist hierbei die Menge an Pattern, die ein zu entwerfender WebSocket Service unterstützen soll.

2.3.1 Publish-Subscribe

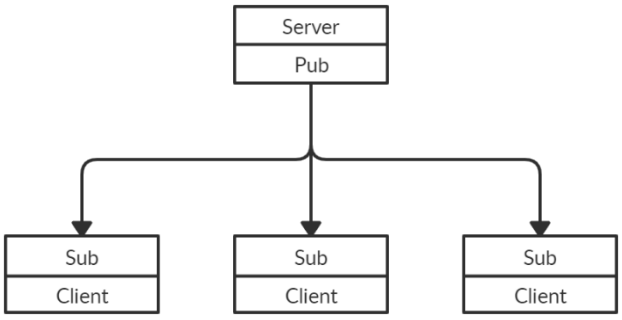

Abbildung 1: DFD Publish-Subscribe

Das erste Messaging-Pattern ist das Publish-Subscribe Pattern. Es kann sowohl für typische Social-Media- oder Chatanwendungen verwendet werden, als auch für die Synchronisierung von Speicherständen oder anderer Daten. Die Publisher senden Nachrichten über einen Kanal, und jeder Subscriber, der diesen Kanal abonniert hat wird über das Update informiert (**Abbildung 1**:).

2.3.2 Request-Reply

Ein weiteres weit verbreitetes Messaging-Pattern ist das Request-Reply Pattern. Der prominenteste Vertreter dieses Patterns ist vermutlich HTTP. Diese Kommunikationsform findet typischerweise zwischen einem Client und einem Service statt. Im Falle von HTTP ist das Request-Reply Pattern unidirektional, das bedeutet, Requests werden immer vom Client an den Server gestellt (**Abbildung 2**). Das Request-Reply Pattern kann nicht nur mehrere Clients beinhalten, die Anfragen an einen Service stellen, sondern auch mehrere Instanzen eines Service, die in der Lage sind, eine Anfrage zu beantworten.

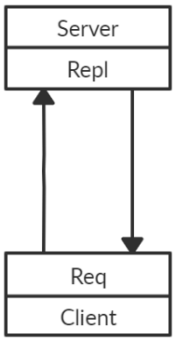

Abbildung 2: DFD Request-Reply

2.3.3 Push-Pull

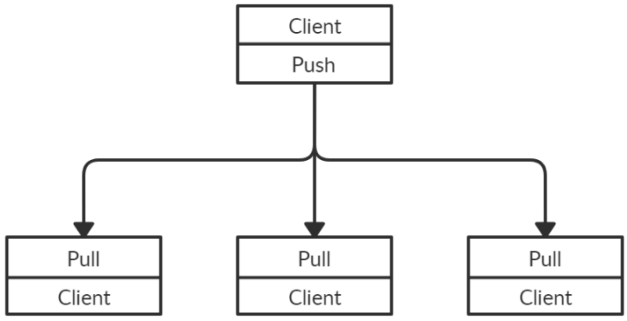

Abbildung 3: DFD Push-Pull

Beim Messaging-Pattern Push-Pull steht die gleichmäßige Verteilung der Nachrichten im Vordergrund. Ein Anwendungsfall für dieses Pattern ist das Loadbalancing, bei der Last gleichmäßig auf verschiedene Knoten verteilt werden soll. Ein Beispiel für diesen Fall ist das Verteilen von HTTP Requests auf verschiedene HTTP Server. Das HTTP (Hyper Text Transfer Protocol) definiert Clients und Server. Clients senden Anfragen (HTTP Requests) an die Server. Diese akzeptieren und bearbeiten die Anfragen und sendet eine Antwort (HTTP Response) zurück an den Client [6]. Gibt es viele Server, die dieselben Anfragetypen bearbeiten können, so werden häufig alle Anfragen nach einem geeigneten Verteilungsalgorithmus an die Server weitergeleitet, die wiederum eigenständig antworten. In diesem Fall spricht man von "fan-out", da sich die Last wie bei einem Fächer aufteilt (siehe **Abbildung 3**).

Bei der Verteilung von Arbeitslast (Workload) auf mehrere Knoten, beispielsweise Server eines Rechenzentrums, wird das "Fan-out Fan-in" Prinzip verwendet. Bei diesem Prinzip wird die Last nach dem Push-Pull Pattern zunächst auf die Knoten verteilt. Nach der Bearbeitung werden die Ergebnisse an eine zentrale Stelle zurückgegeben.

Diese Messaging-Patterns werden später für den Entwurf des Relay-Service als Verteilungsschemata erneut aufgegriffen.

2.4 REST

Ein Webservice stellt Anwendungsfunktionalitäten zur Verfügung. Diese können von Datenspeicherung über Rechenoperationen bis hin zu Infrastruktur reichen. In jedem Fall jedoch muss gewährleistet werden, dass der Service zuverlässig über das Netzwerk angesprochen werden kann. Eine Möglichkeit eine solche Schnittstelle zu schaffen bietet REST.

REST beschreibt den Architekturstil eines Webservice und hat das Ziel für die Bereitstellung von Ressourcen eine einheitliche Schnittstelle zu schaffen. Damit ein Interface als "RESTfull" bezeichnet werden kann, muss es gewisse Anforderungen erfüllen. REST ist Client-Server-basiert und verlangt eine strikte Trennung der Clientseite mit Anwendungslogik und der Serverseite, die Daten oder Services bereitstellt. Zudem sind REST-Operationen zustandslos. Zustandsspeicherungen finden lediglich auf der Clientseite statt. Eine RESTfull API unterstützt sowohl die Auslieferung statischer Repräsentationen einer Ressource, beispielsweise als JSON Objekt, als auch ausführbaren Code [7]. RessoSurcen werden über eindeutige Identifier angesprochen, die in der URI (Uniform Resource Identifier) codiert werden.

Der Uniform Resource Identifier ist eine Methode Ressourcen als ASCII-Zeichenfolge eindeutig zu identifizieren. Er besteht aus Schema und Pfad, durch einen Doppelpunkt getrennt. hierarchische Struktur der URI wird durch die Trennung von Segmenten mit dem Slash Zeichen "/" gewährleistet. Das Fragezeichen trennt die URI von möglichen Quarry Objekten, die der Ressource zugeordnet werden [8].

Die REST-API wird hierbei häufig mit dem Request-Reply basierten Protokoll HTTP verbunden, dessen Anfragetypen „GET“, „POST“, „DELETE“, „PUT“ und „PATCH“ die Basisoperationen (create, read, update, delete) beschreiben, die auf einer Ressource durchgeführt werden [9].

2.5 WebSockets

Das zu lösende Problem ist der Duplex-Nachrichtenaustausch zwischen mehreren Clients. Im Webkontext kommt das WebSocket Protokoll dort zu tragen, sobald Duplexverbindungen gefordert sind. Die Anfragen des HTTP sind auf eine Richtung beschränkt, vom Client, zum Server. Der Server hat die Möglichkeit auf Anfragen des Clients zu antworten, kann jedoch initiativ keine Nachrichten an einen Client senden [10]. Diese Aufgabe übernimmt das WebSocket Protokoll, das bidirektionale Verbindungen zwischen zwei Endpunkten ermöglicht [11].

2.5.1 Alternativen zu WebSockets

Vor der Standardisierung des Protokolls im Jahre 2011 gab es bereits alternative Methoden Clients mit Nachrichten des Servers zu versorgen. Diese waren jedoch in der Regel nicht sehr performant. Eine Alternative zu WebSockets ist das HTTP Polling. Unter Polling versteht man das regelmäßige Anfragen des Servers, ob neue Nachrichten für den Client vorliegen. Hierfür werden in kurzen Zeitabständen HTTP Requests an den Server gesendet, in dessen Response gegebenenfalls die gewünschten Daten übermittelt werden. Diese Methode hat einige Nachteile. Zum einen müssen Daten auf der Serverseite zwischengespeichert werden bis die nächste Anfrage eingeht, zum anderen ist die Netzwerkauslastung durch viele unnötige HTTP Requests sehr hoch. Eine andere Variante, die mit geringerer Netzwerkauslastung auskommt, ist das HTTP Long-Polling. Hier wird ein HTTP Request an den Server gesendet, der so lange aufrechterhalten wird, bis der Server dem Client neue Daten zukommen lassen möchte. Hier besteht jedoch dauerhaft ein offener HTTP Request, was bedeutet, dass die entsprechenden Netzwerkressourcen dauerhaft gebunden sind. Zudem stellen die HTTP Header, die jede Request und Response beinhaltet bei kleinen Datenmengen einen großen Overhead dar.
Die Latenzzeiten lassen sich noch weiter verbessern durch HTTP-Streaming. Der HTTP Request wird nicht beendet während der Server mehrere Datenpakete als Teil einer HTTP Response sendet. Die Fragmentierung der Response in einzelne Datenpakete kann jedoch zu Problemen im Netzwerk oder auf der Clientseite führen, wenn Pakete gepuffert werden.

Es gibt Mechanismen, die serverseitige Push-Benachrichtigung ermöglichen. Diese bauen auf den oben genannten Mechanismen auf und werden unter dem Sammelbegriff "Comet" zusammengefasst. [10]

Das WebSocket Protokoll nutzt eine einzige TCP-Verbindung für den Nachrichtenaustausch und vermeidet Nachteile wie hohe Latenzzeiten, großen Daten-Overhead durch HTTP Header und komplexe clientseitige Anwendungslogik, um Requests und Responses zu verwalten. WebSockets nutzen dabei die vorhandene Infrastruktur für HTTP-Kommunikation des Internets.

2.5.2 Technischer Aufbau

Die Herstellung der WebSocket Verbindung erfolgt über den sogenannten Opening-Handshake. Vom Client wird ein HTTP GET Request gesendet, der den Server auffordert, zu dem WebSocket Protokoll zu wechseln.

Das WebSocket Protokoll verwendet die URI Schemata "ws" (WebSocket) und "wss" (WebSocket Secure). Die HTTP Header "Upgrade" und "Connection" (siehe **Abbildung 4**) weisen darauf hin auf welches Protokoll gewechselt werden soll. Der Header "Sec-WebSocket-Key" dient der Verifizierung, dass der WebSocket Server die Verbindung angenommen hat. Der enthaltene Schlüssel wird mit einem globalen unique Identifier verbunden und verschlüsselt. Dies stellt sicher, dass beide Kommunikationspartner das WebSocket Protokoll verstehen. Der neue Schlüssel wird dann der Response im Header "Sec-WebSocket-Accept" angehängt. Weitere Header wie "Sec-WebSocket-Version", "Sec-WebSocket-Protocoll" und "Sec-WebSocket-Extensions" geben die verwendete Version, sowie optionale Parameter für das Protokoll auf Anwendungsebene an.

```
GET / HTTP/1.1
Host: foo.com
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Key:
   8wslOz4suMiOgiHAv3ED0g==
Origin: http://bar.com
Sec-WebSocket-Protocol:
   exampleprotocol
Sec-WebSocket-Version: 13
```

Abbildung 4: Beispiel WebSocket Upgrade Request

Der WebSocket Server antwortet bei Erfolg mit Status Code 101 "Switching Protocols" und komplettiert damit den Opening-Handshake (**Abbildung 5**) [12].

Nachrichten können dann über die TCP-Verbindung ausgetauscht werden. Die Nachrichten werden in Frames verpack, die ihrerseits fragmentiert werden können. Neben den Datenframes existieren Kontrollframes, die für die Verbindungverwaltung zuständig sind. Die WebSocket Verbindung wird beendet durch den Closing Handschake, der initiiert wird, indem Client oder Server das Closing-Kontrollframe sendet, woraufhin der andere Partner seinerseits mit dem Closing-Kontrollframe antwortet.

```
HTTP/1.1
   101 Switching Protocols
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept:
0eYMYmoc1gyViISj8xMuh8lcA==
```

Abbildung 5: Beispiel WebSocket Upgrade Response

2.5.3 Verwendung von WebSockets

Die WebSocket API ist eine standardisierte API, die es Webbrowsern und anderen Anwendungen ermöglicht, das WebSocket Protokoll zu nutzen. Die API wurde 2012 vom W3C

spezifiziert. WebSockets werden mittlerweile vom Großteil der gängigen Browser unterstützt (vergleiche **Abbildung 6**).

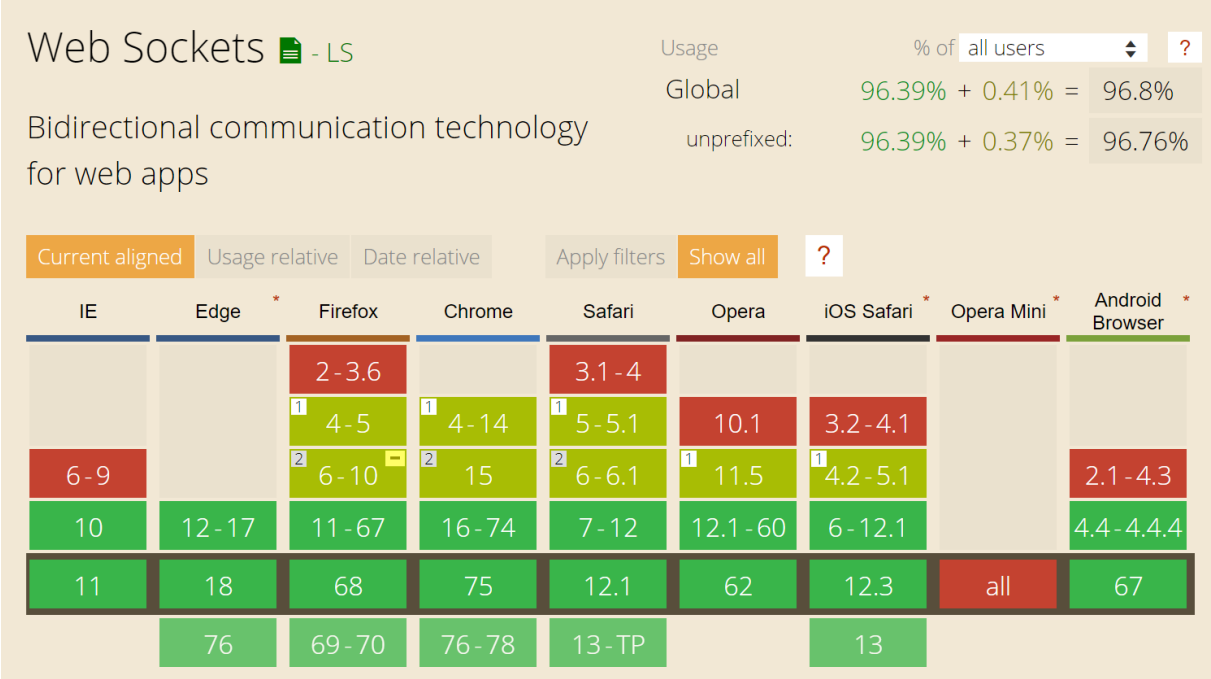

Abbildung 6: Browser Unterstützung WebSockets

Abbildung 6 zeigt eine Analyse der Website caniuse.com die besagt, dass etwa 96% der verwendeten Browser WebSockets unterstützen [13].

Viele verbreitete Webseiten nutzen WebSockets, um echtzeitfähige bidirektionale Funktionalitäten aufzubauen. So verwendet Facebook WebSockets, um die Chatfunktion ihrer Social Media Plattform zu implementieren [14]. GitHub nutzt WebSockets, um dem Nutzer über Änderungen in abonnierten Projekten zu informieren. Das Online Browsergame agar.io implementiert die Kommunikation der Spiele-Logik zwischen dem HTML Client und dem Gameserver über WebSocket-Verbindungen [15].

2.6 Existierende Services

Message-basierte Kommunikation zwischen Clients zu ermöglichen ist keine neue Idee. Es gibt einige Services, großer Firmen oder kleiner Entwicklerteams, die sich dieser Aufgabe annehmen. Betrachtet wurden Message Relay Services, die sich zu den Stichworten WebSocket-Relay bzw. Message-Relay auf Onlineplattformen finden ließen oder zu denen es entsprechende Git-Repositorien gibt.
Der Anspruch dieser Arbeit ist es auf Clientseite lediglich mit dem WebSocket Protokoll auszukommen, ohne seinerseits neue Abhängigkeiten zu erzeugen. Dieser Service soll das dynamische Erstellen von Kommunikationskanälen ermöglichen, verschiedene

Kommunikationspattern unterstützen und über eine standardisierte Schnittstelle ansprechbar sein. Existierende Services weisen unter dieser Betrachtung an verschiedenen Stellen Mängel auf. Die Services, die in Verbindung mit diesem Problem betrachtet wurden, lassen sich allgemein in drei Kategorien einteilen:

- Frameworks zum Erstellen von WebSocket Servern
- Message Relay Services
- Services, die zwischen dem WebSocket Protokoll und anderen Protokollen übersetzen

Eine Übersicht über einige existierende Services die sich mit Message-Relay oder Websocket Relay Funktionen befassen zeigt **Tabelle 1**.

Name / Quelle	**Kurzbeschreibung**	**Features / Defizite**
SocketCluster [16] SocketCluster.io	NodeJS Service und Framework, das skalierbare Client-Server Kommunikation ermöglicht. SocketCluster Clients sind in vielen Programmiersprachen verfügbar	+ skaliert horizontal und vertikal + Publish-Subscribe Pattern + Channels + RPC Pattern + optimiert für Kubernetes + Authentifikation mit Token - Eigener Client benötigt
WebSocket-relay [17] William MacDonald	leichtgewichtiger WebSocket Server in NodeJS der Channel-basiert Nachrichten zwischen Clients austauscht.	+ Publish-Subscribe Pattern + Channels + Authentifikation mit Token - Eigener Client benötigt
SockJS [18] Gitub User: SockJS	JavaScript Framework, für duplex Client-Server Verbindungen. Clients sind in viele Programmier-sprachen verfügbar.	+ HTTP Polling als Backup - Nur Framework
Puscher [19] Pusher Limited	ein Service, der WebSocket basierte Kommunikation zwischen Clients und Servern ermöglicht.	+ Publish-Subscribe Pattern + Ende zu Ende Verschlüsselung + Channels + HTTP Polling als Backup - Eigener Client benötigt
Florkestra [20] Florkestra	Ist ein einfacher WebSocket Relay Server.	+ Publish-Subscribe Pattern + nutzt Websocket Client - viele Features fehlen
axon [21] Jarrett Cruger	Die NPM-Bibliothek liefert die Sockets zum Nachrichtenaustausch auf Basis von TCP Verbindungen.	+ Push-Pull, Pub-Sub, Req-Rep Pattern + Verbindungswiederherstellung - Eigener Client benötigt
ws-tcp-relay [22] Josh Glendenning	Proxy-Server übersetzt der zwischen Nachrichten verbundener WebSocket Clients und TCP-Verbindungen zu Servern.	Relay zwischen WebSocket und TCP Protokoll, keine Verbindung zwischen Clients

Tabelle 1: Übersicht existierender Services

Der Service, der von allen untersuchten Lösungen das Problem am besten löst ist SocketCluster der gleichnamigen Entwicklerfirma. Dies ist ein sehr komfortables Framework, um Anwendungen message-basiert zu verbinden. Die Optimierung für die Verwendung in Kubernetes Clustern und die Skalierbarkeit versprechen eine solide Grundlage zur Lösung des Problems. Jedoch ist für die Verwendung dieses Frameworks ein SocketCluster Client notwendig und eine Verwendung mit nativen WebSockets ist nicht möglich. Zudem unterstützt dieses Framework lediglich das Messaging-Pattern Publish-Subscribe und deckt damit nur einen Teil der Anforderungen ab [23]. Dieses Problem teilen weitere Services wie WebSocket-relay [17], Pusher [19] und axon [21]. Florkestra ist ein sehr simpler WebSocket Relay Service, der auf einem einzigen Port, WebSocket Verbindungen entgegennimmt, und deren Nachrichten an alle weiteren WebSocket Verbindungen weiterleitet. [20] Damit verwendet dieser Service Websocket Clients, unterstützt jedoch lediglich das Publish-Subscribe Pattern, und verfügt weder über ein Channelkonzept noch über Möglichkeiten der Skalierbarkeit [20]. Für solch einfache Relay Services lassen sich weitere Beispiele finden [24] [25]. Andere Anwendungen verstehen Ihre Relay-Aufgabe darin zwischen dem WebSocket Protokoll und einem anderen Protokoll zu übersetzen und Nachrichten weiterzuleiten. Diese erfüllen keine Relayfunktion zwischen Clients im Sinne des hier betrachteten Problems. Ein Beispiel hierfür ist das ws-tcp-relay [22]. Im Anhang finden sich alle untersuchten Lösungen.

Es lässt sich somit sagen, dass es einige Arbeiten gibt, die sich mit der Aufgabe der message-basierten Vernetzung von Clients befassen. Diese unterstützen jedoch nur einen Teil der geforderten Features oder sind so komplex, dass sie in der clientseitigen Anwendung Abhängigkeiten erzeugen und nicht mit dem WebSocket Protokoll auskommen.

3 Anforderungsanalyse

Im Rahmen der Anforderungsanalyse werden Systemanforderungen an die Software aus dem Problem abgeleitet. Diese Systemanforderungen bilden die Grundlage für die Architektur des Service und werden für die Nachweisführung herangezogen.

3.1 Zielgruppe

Zielgruppe dieses Service sind kleine Entwicklergruppen, die mit geringem Implementierungsaufwand Nachrichten zwischen Clientanwendungen austauschen möchten. Viele kleinere Apps oder Webseiten kommen ohne serverseitige Programmierung aus, trotzdem kann es nötig sein Arbeitsstände zwischen Geräten zu synchronisieren, Nachrichten auszutauschen oder Workloads zu verteilen. Einfache Browsergames, die Spielstände in Cookies speichern benötigen keine aufwendige Serverlogik. Solche Spiele könnten von einem solchen Service profitieren, der die Spiele damit mehrspielerfähig macht.

Besonders für Einsteiger der Softwareentwicklung ist es oft hilfreich sich zunächst auf die clientseitige Entwicklung der Applikation zu konzentrieren. Um dennoch die Kommunikation zwischen den Clients zu ermöglichen ist ein leichtgewichtiger WebSocket Relay Service sehr nützlich, zumal für die Administration und Verbindung lediglich die Protokolle HTTP und WebSockets benötigt werden.

Somit besteht die Zielgruppe dieses Service aus Softwareentwickler-Einsteigern, Studenten, und privaten Entwicklern, die kleine verteilte Systeme entwickeln. Man denke sich ein System, das webbasiert auf Smartphones eingesetzt wird und auf Großveranstaltungen den Standort aller Nutzer zu teilen und Alarm schlägt, sollte jemand verloren gehen. Für diese Anwendung ist lediglich ein Nachrichtenservice erforderlich und dieser ließe sich von einem kleinen Entwicklerteam entwerfen.

Für professionelle Software wird vermutlich statt der message-basierten Kommunikation ein komplexes Serverframework verwendet, das Cloud-basierte Datenspeicherung, Nutzerverwaltung und komplexe Authentifizierung uvm. unterstützt. Es ist kaum denkbar, dass ein Produkt wie WhatsApp auf den Service zurückgreift.

3.2 Nutzeranforderungen

Der Großteil der Nutzeranforderungen lassen sich direkt aus der Aufgabenstellung ableiten.
Die Interaktion des Nutzers mit dem Service besteht aus zwei Teilen. Das Verwalten der Channel beinhaltet das Anlegen, Anzeigen und Löschen eines Channels. Zudem kann der Nutzer einen Channel abbonieren, um die Nachrichten des Channels zu erhalten. Dieses Abbonieren entspricht in diesem Fall dem Verbinden mit dem Channel.

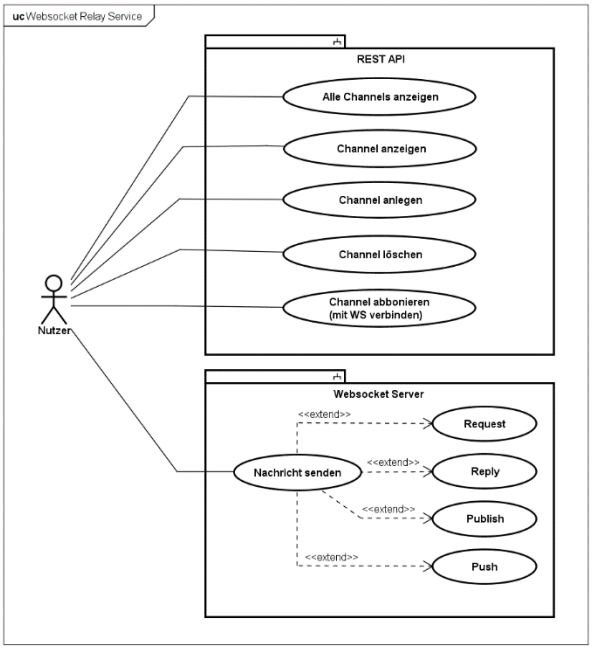

Abbildung 7: Usecase Diagram

Der zweite Teil der Nutzerinteraktion stellt den Nachrichtenaustausch über den WebSocket Server dar. Nutzer sollen in der Lage sein Nachrichten aktiv über einen WebSocket Client zu senden und zu empfangen. **Abbildung 7** zeigt die entsprechenden Anwendungsfälle in einem Use-Case Diagramm. Aus diesen Anwendungsfällen ergeben sich folgende Nutzeranforderungen:

[PF1] Anlegen eines Channels (über REST-API)
[PF2] Löschen eines Channels (über REST-API)
[PF3] Abonnieren eines Channels (über REST-API)
[PF4] Verbinden mit einem Channel über das WebSocket Protokoll
[PF5] Senden von Nachrichten über den Channel
[PF6] Empfangen von Nachrichten über den Channel

3.3 Systemanforderungen

Aus den Nutzeranforderungen leiten sich die Systemanforderungen ab. Zu den Nutzeranforderungen kommen weitere Gesichtspunkte, nichtfunktionaler Natur, die die Benutzbarkeit, Portierbarkeit und Sicherheit des Service betreffen. Im Folgenden werden die Anforderungen an das System aufgelistet und anschließend erläutert und priorisiert.

3.3.1 Funktionale Systemanforderungen

[FR1] POST Request zum Anlegen eines Channels
[FR2] DELETE Request zum Löschen eines Channels
[FR3] Abonnieren des Channels durch WebSocket Verbindung
[FR4] Verbinden mit einem Channel über das WebSocket-Protokoll
[FR5] Weiterleitung der Nachrichten gemäß Pub./Sub. Messaging-Pattern
[FR6] Weiterleitung der Nachrichten gemäß Push/Pull Messaging-Pattern
[FR7] Weiterleitung der Nachrichten gemäß Req./Repl. Messaging-Pattern

3.3.2 Nicht Funktionale Systemanforderungen

[NFR1] Der Service soll mit wenigen Abhängigkeiten auskommen
[NFR2] Der Service soll wartbar sein
[NFR3] Das API soll REST-konform sein
[NFR4] Der Service soll als standardisiertes Container Image bereitgestellt werden
[NFR5] Der Service soll die „12-Factor App Prinzipien“ berücksichtigen
[NFR6] Das REST-API soll SSL-Verschlüsselung unterstützen
[NFR7] Die WebSocket Kommunikation soll SSL-Verschlüsselung unterstützen
[NFR8] Die WebSocketverbindung soll mit einem Sicherheitstoken authentifiziert werden
[NFR9] Ein Channel soll mit einem Timeout versehen werden können
[NFR10] Die Clientzahl eines Channels soll begrenzt werden können
[NFR11] Der Service soll horizontal skalierbar gestaltet werden

Die 12-Factor App Prinzipien beschreiben Methoden, die das Entwickeln von "Software as a Service" erleichtern. Software as a Service beschreibt Software, die von Nutzern On-Demand über das Netzwerk in Anspruch genommen werden. Solche Services laufen in Cloud Plattformen und skalieren mit zunehmender Auslastung. Die zwölf Faktoren befassen

sich mit Methoden der Softwaretechnik, technischer Umsetzung, und Softwarearchitektur, mit dem Ziel WebApps zu entwickeln, die problemlos in modernen Cloud-basierten Umgebungen deployed werden können.
Aus der Systemanforderung [NFR5] leiten sich 12 weitere Systemanforderungen ab [26]:

- [NFR5.1] I. Codebase: Es gibt nur eine Codebase je App, aber viele Deploys
- [NFR5.2] II. Abhängigkeiten: Alle Abhängigkeiten müssen explizit deklariert werden und der Software mitgeliefert werden. Auch spezielle Systemfunktionen dürfen nicht vorausgesetzt werden.
- [NFR5.3] III. Konfiguration: Die Konfiguration des Service ist strikt vom Code zu trennen und wird in Umgebungsvariablen gespeichert
- [NFR5.4] IV. Unterstützende Dienste: Jeder unterstützende Dienst ist eine eigene Ressource, steht unabhängig vom Code und kann ausgetauscht werden
- [NFR5.5] V. Build, release, run: Die Phasen des Deponent Prozesses sind strikt zu trennen und geordnet. Codeänderungen sind ausschließlich in der Build-Phase möglich.
- [NFR5.6] VI. Prozesse: Prozesse sind zustandslos und Shared-Nothing. Daten werden ausschließlich in unterstützenden Diensten abgelegt.
- [NFR5.7] VII. Bindung an Ports: Web-Apps binden sich eigenständig an einen Port, um einen Dienst zur Verfügung zu stellen.
- [NFR5.8] VIII. Nebenläufigkeit: Jeder Aufgabe wird ein Prozesstyp zugewiesen und Skalierung wird durch Nebenläufigkeit dieser zustandslosen Share-Nothing Prozesse erreicht.
- [NFR5.9] IX. Einweggebrauch: Startseiten sind schnell, Anfragen kurz und Prozesse können ohne Probleme gestoppt werden, indem der laufende Job automatisch an die Workqueue zurückgegeben wird.
- [NFR5.10] X. Dev-Prod-Vergleichbarkeit: Die App verwendet Continuous Deployment, indem die Zeit zwischen Entwicklung und deploy klein gehalten wird, und Entwicklungs- und Laufzeitumgebung so ähnlich wie möglich gehalten werden.
- [NFR5.11] XI. Logs: Logs sollen als Streams behandelt werden, die die Arbeitsweise der App über die Zeit auf der Standardausgabe ausgibt. Archivierung und Analyse wird von externen Diensten vorgenommen.
- [NFR5.12] XII. Admin-Prozesse: Einmalige Administarationsprozesse sollen mit möglichst identischer Umgebung, Code und Konfiguration ausgeführt werden wie die der App. Der Code wird mit dem Admin-Code ausgeliefert.

Bei den durch einen Channel aufgebauten Verbindungen handelt es sich um kurz- bis mittelfristige Verbindungen. Eine persistente Speicherung des Systemzustands ist daher nicht notwendig. Des Weiteren wird nicht erwartet, dass der Service Verbindungen nach unerwartetem Beenden wieder aufnimmt. Stattdessen wird nach unerwartetem Verbindungsverlust zum Client die Verfügbarkeit des alten Channels angefragt und gegebenenfalls ein neuer Channel angelegt.

Nachrichten werden weder gepuffert bis Clients bereit sind diese zu empfangen, noch persistent gespeichert.

Es ist keine Nutzerverwaltung mit komplexer Authentifikation, Koto- und Rechteverwaltung gefordert.

Im Folgenden werden die Systemanforderungen erläutert und priorisiert.

3.3.3 Erläuterung der Anforderungen

Nachrichtenkanäle oder Channels sollen einen gesonderten Bereich darstellen, in dem Clients Nachrichten austauschen können. Diese Kanäle haben einen eindeutigen Identifier und gewissen Eigenschaften. Kanäle können erstellt und wieder zerstört werden. Zu den Eigenschaften eines Kanals gehört:

- Messagingpattern
- Clientlimit
- Timeout

Durch das Verbinden mit einem Channel abonniert der Client die Nachrichten, die auf diesem entsprechend dem gewählten Messagingpattern ausgetauscht werden. Im Folgenden wird erklärt welche Bereiche des Service die Anforderungen abdecken. Eine Liste mit der detaillierten Definition jeder Anforderung befindet sich im Anhang.

Die Anforderungen [FR1] bis [FR7] beschreiben die Funktionalen Anforderungen des Service. Zu den Basisfunktionalitäten des Service gehört das Anlegen und Löschen eines Channels über die REST-API, der Verbindungsaufbau zu einem Channel und das Weiterleiten von Machrichten gemäß der unterstützen Messaging Pattern.

Alle weiteren Anforderungen sind nicht funktional. [NFR1] bis [NFR5] stellen Anforderungen an die Architektur des Service, die für Wartbarkeit und Portierbarkeit sorgen und somit die Voraussetzungen schaffen, den Webservice im Server Cluster einzusetzen. Eine besondere Rolle spielen hierbei die 12-Factor App Prinzipien.

Die Anforderungen [NFR6] bis [NFR8] stellen Sicherheitsanforderungen dar und fordern SSL-Verschlüsselung der Kommunikationskanäle, sowie eine einfache Tokenauthentifikation.

[NFR9] und [NFR10] beschreiben Anforderungen an die Eigenschaften der Channels, die Limitierung der Clientzahl, sowie ein Inaktivitätstimeout, der den Channel nach einer gewissen Zeit automatisch beendet. Beide Eigenschaften sollen parametrisierbar sein.

Zuletzt wird die horizontale Skalierbarkeit des Service gefordert [NFR11], die dem Service mindestens prototypisch nachgewiesen werden soll.

3.3.4 Abdeckung der Nutzeranforderungen

Die **Tabelle 2** zeigt die Zuordnung der Systemanforderungen zu den entsprechenden Nutzeranforderungen. Es ist zu erkennen, dass alle Nutzeranforderungen und damit alle Use-Cases abgedeckt sind.

	[PF1]	24[PF2]	[PF3]	[PF4]	[PF5]	[PF6]
[FR1]	X					
[FR2]		X				
[FR3]			X			
[FR4]				X		
[FR5]					X	X
[FR6]					X	X
[FR7]					X	X

Tabelle 2: Abdeckung der Nutzeranforderungen

3.3.5 Priorisierung der Anforderungen

Die Priorisierung der Systemanforderungen wird in die vier Kategorien A, B, C und D vorgenommen.

Kategorie A beschreibt die Basisfunktionalitäten des Service, sowie Anforderungen ohne die die fehlerfreie Funktionsweise des Service nicht garantiert werden kann.

Kategorie B enthält Anforderungen, die den Service mit geltenden Standards kompatibel macht, sowie Sicherheit gewährleistet.

Kategorie C enthält optionale Funktionalitäten, die über den Basisumfang des Service hinaus gehen, sowie Eigenschaften, die die Nutzbarkeit verbessern.

Kategorie D umfasst Komponenten, die vor allem prototypisch untersucht werden sollen.

Die Priorität der Anforderungen lässt sich beschreiben mit:

Priorität(A) > Priorität(B) > Priorität(C) > Priorität(D)
sehr hoch hoch medium gering

Die Systemanforderungen sind den Kategorien wie folgt zugeordnet:

Systemanforderung	Kategorie
[Req1] POST Request zum Anlegen eines Channels	A
[Req2] DELETE Request zum Löschen eines Channels	A
[Req3] Abonnieren des Channels durch WebSocket Verbindung	A
[Req4] Verbinden mit einem Channel über das WebSocket-Protokoll	A
[Req5] Weiterleitung der Nachrichten gemäß Pub./Sub. Messaging-Pattern	A
[Req6] Weiterleitung der Nachrichten gemäß Push/Pull Messaging-Pattern	A
[Req7] Weiterleitung der Nachrichten gemäß Req./Repl. Messaging-Pattern	A
[NFR1] Der Service soll mit wenigen Abhängigkeiten auskommen	C
[NFR2] Der Service soll wartbar sein	C
[NFR3] Das API soll REST-konform sein	B
[NFR4] Der Service soll als standardisiertes Container Image bereitgestellt werden	B
[NFR5] Der Service soll die „12-Factor App Prinzipien“ berücksichtigen	B
[NFR6] Das REST-API soll SSL-Verschlüsselung unterstützen	B
[NFR7] Die WebSocket Kommunikation soll SSL-Verschlüsselung unterstützen	B
[NFR8] Die WebSocket-Verbindung soll mit einem Sicherheitstoken authentifiziert werden	B
[NFR9] Ein Channel soll mit einem Timeout versehen werden können	C
[NFR10] Die Clientzahl eines Channels soll begrenzt werden können	C
[NFR11] Der Service soll horizontal skalierbar gestaltet werden	D

Tabelle 3: Priorisierung der Anforderungen

Die Interaktion des Clients mit dem Service über die API sowie die WebSocket Kommunikation gehören zum Basis-Anwendungsumfang und hat damit sehr hohe Priorität. Dazu gehört das Verwalten von Channels, das Verbinden zu Channels, sowie der Nachrichtenaustausch über das WebSocket Protokoll.
Die Designvorschrift der API als RESTfull-API schafft eine Standardisierung der Schnittstelle und vereinfacht die Anbindung des Service an andere Anwendungen. Die 12-Factor-App Prinzipien bieten bewährte Methoden die Anwendung so zu entwerfen, dass sie problemlos in Cloudumgebungen, wie beispielsweise einem Kubernetes Cluster, eingesetzt werden kann.

Diese Methoden sind daher von hoher Priorität. Die Möglichkeit zur Verschlüsselung der HTTP- und der WebSocket Kommunikation sowie die Authentifizierung des Channels mit einem Token sind sicherheitsrelevant und damit ebenfalls von hoher Priorität.
Die Begrenzung der maximalen Clientzahl, sowie der Timeout eines Channels stellen Funktionalitäten dar, die über den Basisumfang der Channelnutzung hinaus gehen und ebenso wie die Leichtgewichtigkeit und Wartbarkeit, von mittelstarker Priorität sind.
Die horizontale Skalierbarkeit des Service ist in erster Linie prototypisch zu untersuchen und stellt daher nur geringe Priorität dar.

4 Architektur

Aus der Anforderungsanalyse leitet sich die Architektur des Service ab. Verschiedene Module lösen einzelne Anforderungen und Teilprobleme. Diese Komponenten werden im Einzelnen betrachtet und es wird erklärt wie sie in gewissen Anwendungsfällen zusammenspielen.

4.1 Aufbau des Service

Der WebSocket Relay Service ist grundlegend in drei große Module unterteilt. Diese lassen sich in **Abbildung 8** auf mittlerer Ebenen erkennen. Das UML Diagramm zeigt den Aufbau es Websocket Relay Service stark vereinfacht.

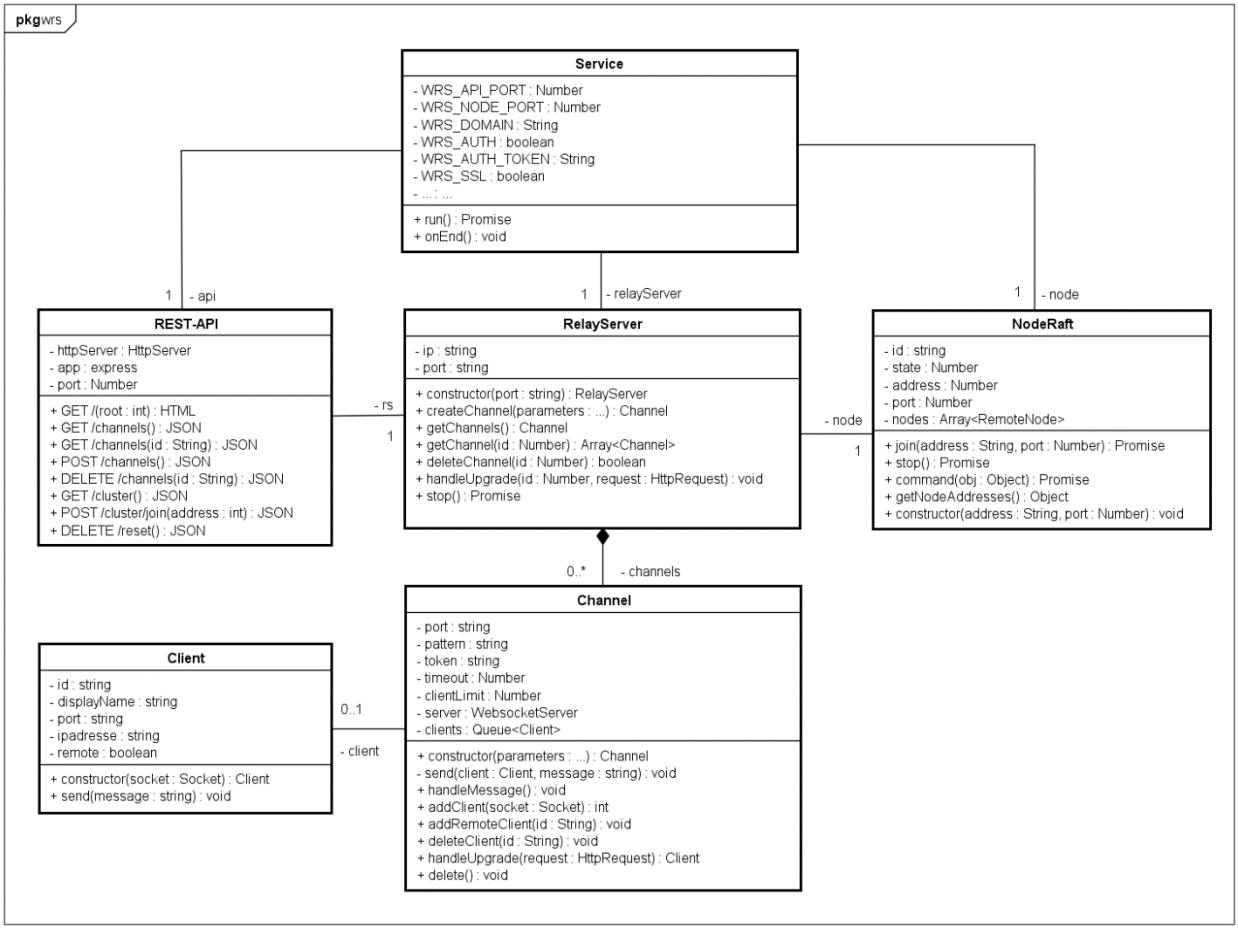

Abbildung 8: Klassendiagramm Service

Ein Bestandteil ist die REST-API. Diese Komponente ist als Schnittstelle für die Kommunikation zu anderen Applikationen mittels REST-konformen HTTP Anfragen zuständig. Die zentrale Komponente des Service stellt der RelayServer dar. Dieser verwaltet alle Channel, nimmt Verbindungsanfragen und leitet sie an die Channel weiter. Um den verteilten Service in persistenten Zustand zu halten kommuniziert der RelayServer mit dem RaftNode. Dieses Modul stellt einen Knoten im Cluster des Service dar, hält die Hierarchie des Clusters aufrecht und kommuniziert Änderungen mit anderen Knoten, um Persistenz zu

erreichen. Für die WebSocket Kommunikation der Clients ist der jeweilige Channel zuständig, der zu diesem Zweck einen WebSocket Server betreibt.

Die Betrachtung der Komponenten erfolgt im Folgenden von der Anwenderseite (in **Abbildung 8** links) zur Clusterseite (rechts) und folgt somit dem typischen verlauf einer Nutzeranfrage durch den Service.

4.1.1 REST-API

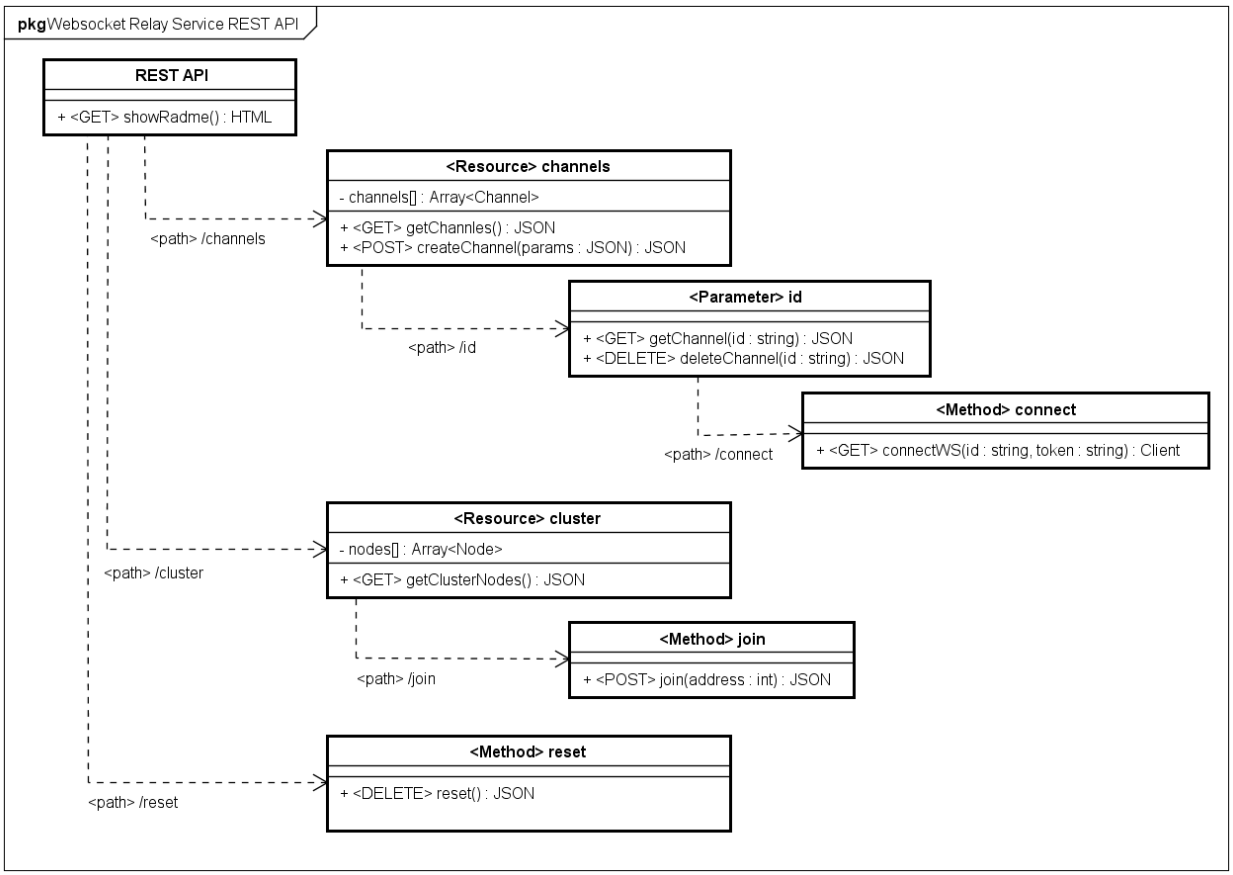

Abbildung 9: REST-API Ressourcen

Der WebSocket Relay Service ist über eine REST-konforme HTTP Schnittstelle für andere Anwendungen ansprechbar. Über einen HTTP Server werden Anfragen entgegengenommen und interpretiert. Je nach angefragter Ressource wird dann der Relay Service oder der RaftNode angesprochen, um den benötigten Service auszuführen. **Abbildung 9** zeigt die Adressierung von Resourcen und ihrer Elemente über Ressourcennamen und ID. Die Interaktionen Create, Read und Delete werden durch die entsprechenden HTTP Methoden POST, GET, und DELETE modelliert. Speziellere Methoden wie Connect, Join oder Reset werden im Pfad spezifizeirt.Über die Ressource */channels* lassen sich Channels anlegen, anzeigen und löschen. Zudem nimmt der HTTP Server unter dieser Ressource HTTP Anfragen entgegen, die von WebSocket Clients gesendet werden, um sich mit einem Channel zu verbinden. Die Ressource

/cluster liefert Informationen zur gegenwärtigen Konfiguration des Clusters. Zu administrativen Zwecken lässt sich der Service über */cluster/join* manuell mit einem bestehenden Cluster verbinden. Die Ressource */reset* erfüllt ebenso administrative Zwecke und ermöglicht das manuelle Zurücksetzen aller Channel.

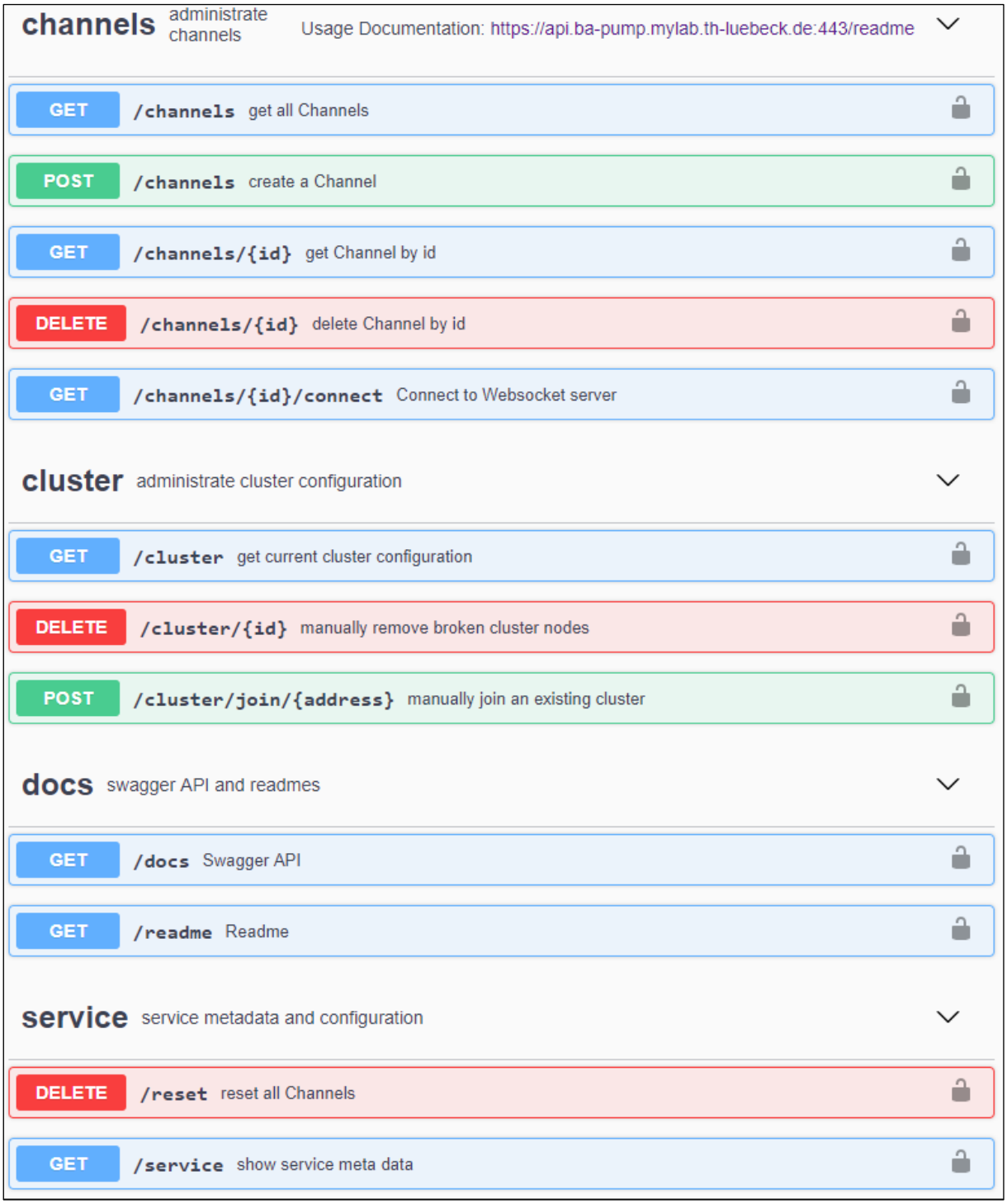

Abbildung 10: HTTP Methoden REST-API

Abbildung 10 zeigt einen Überblick der REST-API und der HTTP Request Methoden, die unterstützt werden. Im Folgenden wird auf einige der Methoden eingegangen und dargestellt welcher Systemanforderung diese entsprechen. Diese Methoden dienen der Administration der Channels und sind damit Kernbestandteil der Funktionalität. Weitere Methoden für optionale Funktionalitäten der Administration der Channels und der Clusterstruktur des Service werden vorerst nicht betrachtet. Im Anhang wird im Rahmen des Administrations- und Nutzerleitfadens auf weitere Methoden eingegangen.

`POST /channels` 24[FR1]

dient dem Anlegen eines Channels. Die Parameter werden im Request Body der Anfrage übergeben. Bei Erfolg erhält der Nutzer Daten zu dem neu angelegten Channel, sowie den Authentifikationstoken.

`DELETE /channels/{id}` 24[FR2]

dient dem Löschen eines Channels. Die Channel ID werden im Pfad übergeben. Bei Erfolg erhält der Nutzer eine entsprechende Benachrichtigung.

`GET /channels/{id}/connect?token={token}` [FR3][FR4]

Die Verbindung mit dem WebSocket Server eines Channels geschieht über einen GET Request mit entsprechenden Upgrade Headern, die den Server auffordern das Verbindungprotokoll von HTTP auf WebSocket zu wechseln. Die Channel ID wird über den Pfad der Anfrage übergeben, der Authentifikationstoken wird als Query Parameter übergeben.

4.1.2 RelayServer

Der RelayServer nimmt Anfragen der REST-API entgegen und verarbeitet diese. Er verwaltet die Channel, die jeweils einen separaten Kommunikationskanal für Clients darstellen. Channel werden über dieses Modul nicht nur angelegt, zerstört, und wiedergegeben, der RelayServer nimmt ebenso HTTP Upgrade Requests von der API entgegen und leitet diese an die entsprechenden Channel weiter.

Ein Channel verwaltet jeweils einen eigenen WebSocket Server. Diesem Server werden eingehende Upgrade Requests übergeben, um eine WebSocket Verbindung zum jeweiligen Client aufzubauen. Um spezielle Message-Verteilungsstrategien umsetzen zu können werden Clientverbindungen in einer Queue gespeichert.

Zudem besitzt ein Channel weitere Attribute, um die Systemanforderungen an einen Channel umzusetzen. Das Messageing-Pattern (*pattern*) stellt das Schema dar, nach dem die Nachrichten innerhalb eines Channels zu anderen Clients weitergeleitet werden

[FR5][FR6][FR7]. Unterstützt werden die in Kapitel 2.3 beschriebenen Pattern. Die konkrete Umsetzung wird im nächsten Kapitel beschrieben.

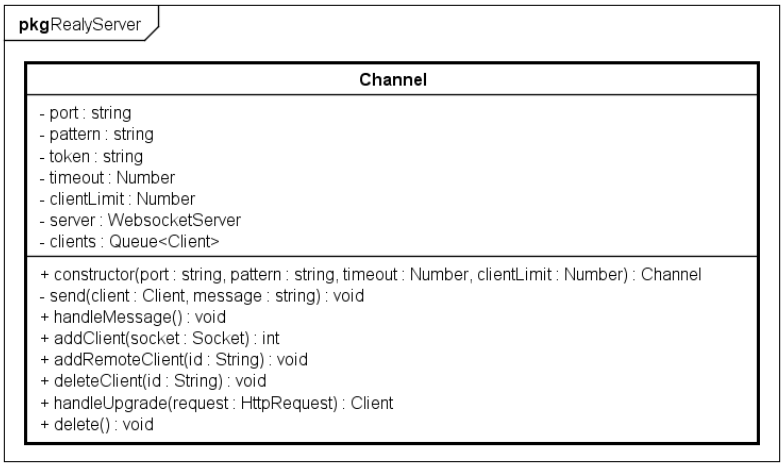

Abbildung 11: Klassendiagramm Channel

Beim Anlegen eines Channels wird neben dem Pattern auch ein Timeout (*timeout*) und ein Clientlimit (*clientLimit*) festgelegt. Der Timeout stellt ein zeitliches Limit dar nach dessen Ablauf sich der Channel bei Inaktivität selbst löscht. Aktivitäten, also das Senden von Nachrichten auf dem Channel setzten dieses Limit zurück [NFR9].

Das Clientlimit begrenzt die maximale Anzahl der mit dem WebSocket Server verbundenen Clients. Ist das Clientlimit erreicht, werden alle weiteren Verbindungsversuche abgewiesen [NFR10].

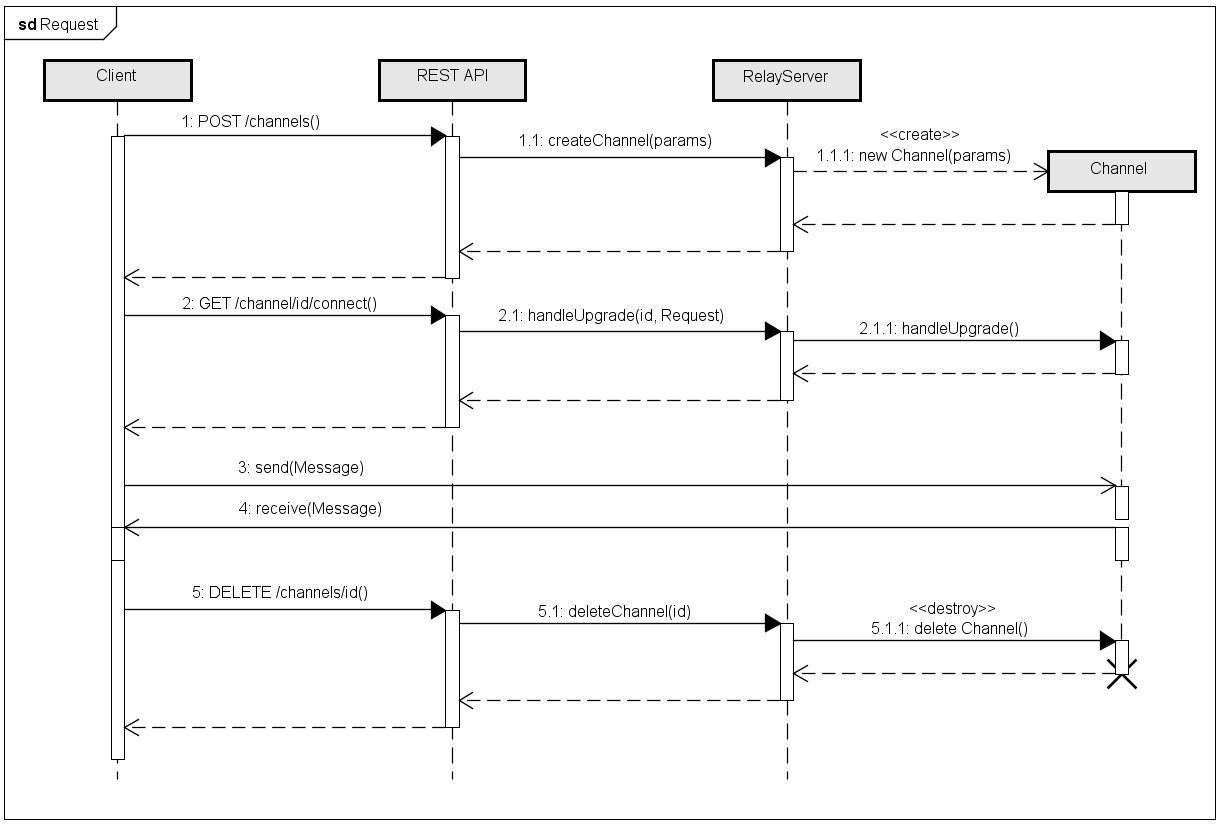

Abbildung 12: Sequenzdiagramm Request

Mit dem Wissen über den Aufbau der ersten Komponenten des Service wird das Zusammenspiel der Rest-API und des RelayServers betrachtet. In **Abbildung 12** ist zu erkennen, dass die REST-API die gewünschte Funktionalität des RelayServers aufruft, der seinerseits den Channel verwaltet. Der WebSocket-basierte Nachrichtenaustausch findet direkt zwischen dem Client und dem WebSocket Server des Channels statt. Die Abbildung zeigt die Aufrufabfolge, die ein HTTP Request an die API im Service auslöst für die Anwendungsfälle des Anlegens eines Channels, dem Verbinden eines Clients und dem löschen eines Channels.

Läuft der Service nicht als alleinstehender monolithischer Knoten, sondern als Teil eines Clusters, so muss bei jeder Aktivität des RelayServers sichergestellt werden, dass die Persistenz erhalten bleibt. Die Architektur dieser Teilkomponente wird im Folgenden betrachtet.

4.1.3 RaftNode

Jede Instanz des Service hat neben der API und dem RelayServer eine dritte Komponente, den RaftNode. Der RaftNode ist ein Knoten nach dem Raft Algorithmus und ist als Teil des Clusters für Konsistenz, Clusterverwaltung und Nachrichtenaustausch zwischen verschiedenen Servern zuständig. Der RaftNode verfügt über eine Liste des Typen RemoteNode. Diese stellen die anderen Knoten des Raft Clusters dar. Über die Funktionen der Klasse RemoteNode werden die RemoteProcedureCalls der jeweiligen Knoten aufgerufen. Die Funktion der RPCs sind in der Klasse RaftNode implementiert.

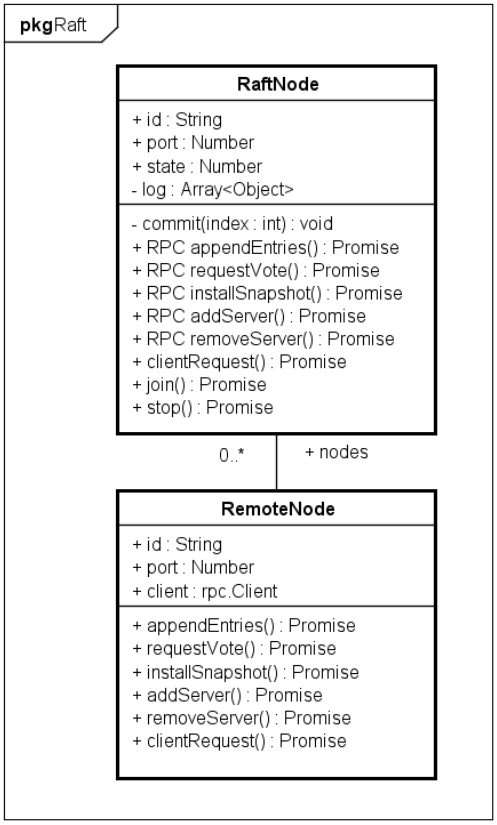

Abbildung 13: Klassendiagramm Raft

Nimmt der RelayServer eine Änderung am Systemzustand vor, legt er beispielsweise einen Channel an, so muss die Zustandsänderung auf alle Knoten (mindestens aber die Majorität aller Knoten) übertragen werden. Hierfür werden dem RaftNode Command-Objekte übergeben, die das Event beschreiben. Diese Objekte haben das Feld "type", das den Command beschreibt, und je nach Typ weitere Attribute. Es gibt die Typen "addChannel", "deleteChannel", "message", "clientConnect" und "clientDisconect". Um einen Command über das Cluster zu teilen wird somit die Funktion command() des eigenen RaftNode aufgerufen und das Command-Objekt übergeben. Wurde der Command durch das Cluster

commited, also auf der Majorität aller Knoten zum Log hinzugefügt, so wird im RaftNode das "commit" Event ausgelöst. Dieses Event enthält das Command Objekt, das commited wurde. Somit kann die gewünschte Änderung vom RelayServer vorgenommen werden. Es wird zudem sichergestellt, dass auch alle weiteren Knoten über den Commit benachrichtigt werden und den gewünschten Befehl ausführen.
Wie die Verteilung des Command Objekts im Cluster abläuft hängt davon ab, ob es sich bei dem Knoten des Servers um den Leader oder einen Follower des Clusters handelt. Ist der Knoten der Leader, so fügt dieser den Command zu seinem Log hinzu und informiert anschließend alle anderen Knoten mittels AppendEntries RPC über den neuen Eintrag. Erhält der Leader von der Majorität aller Knoten die Bestätigung über Erhalt des Eintrags, wird dieser als commited markiert und das "commit" Event erzeugt. **Abbildung 14** zeigt die Aufrufe der Funktionen und RPCs für diesen Fall.

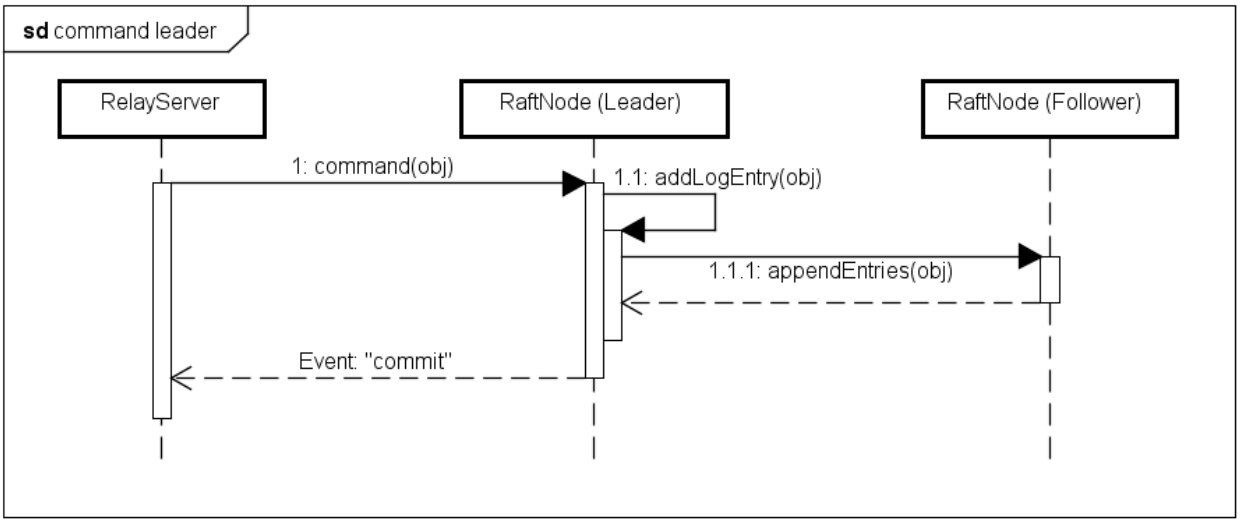

Abbildung 14: Sequenzdiagramm Command mit Leader Node

Ist der eigene Knoten jedoch ein Follower, so wird dieser den Command per ClientRequest RPC an den ihn bekannten Leader weiterreichen.

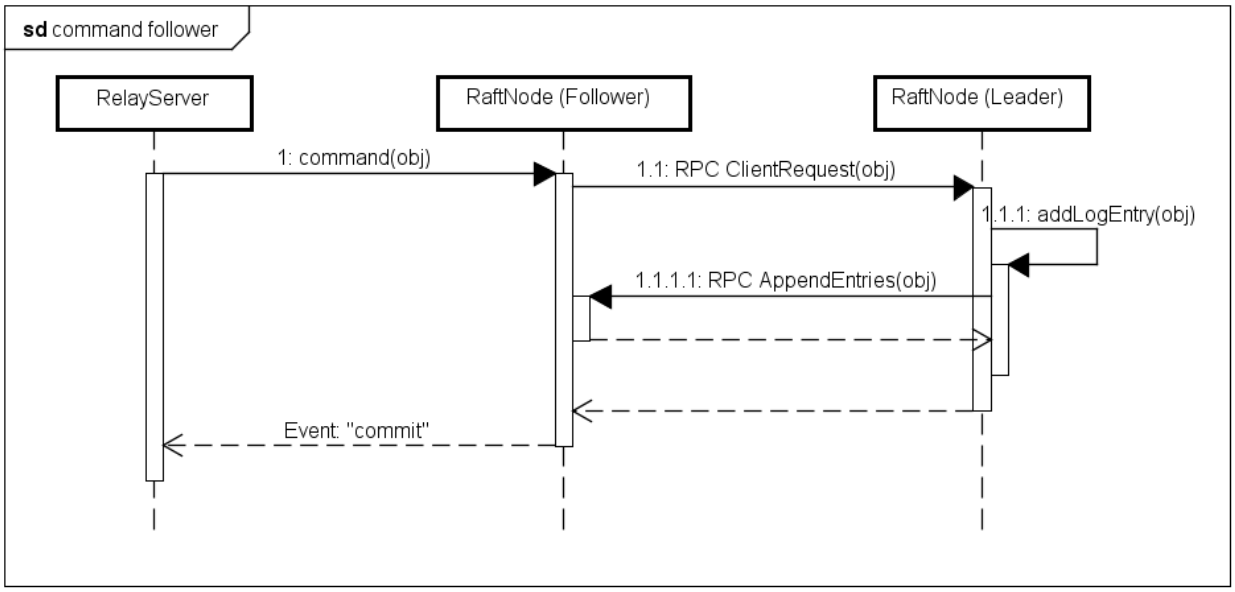

Abbildung 15: Sequenzdiagramm Command mit Follower

Dieser fügt den Command zu seinem Log hinzu und informiert anschließend alle anderen Knoten mittels AppendEntries RPC über den neuen Eintrag. Ist der Command erfolgreich commited wird der eigene Follower Knoten im nächsten Heartbeat über den Commit informiert und erzeugt das "commit" Event. Die Abbildungen zeigen den Vorgang an einem Beispiel mit zwei Knoten. Es können jedoch mehr Knoten beteiligt sein, deren AppendEntries RPC aufgerufen wird. Damit ist der Weg einer Anfrage komplett abgesteckt. Die REST-API nimmt die Anfrage entgegen und interpretiert diese. Die benötigten Funktionalitäten werden dann bei dem RelayServer angefragt. Dieser verteilt die vorzunehmende Änderung als Command über das Cluster und führt die Aktion dann aus. Der RelayService übergibt der API dann das Ergebnis der Aktion und diese sendet eine entsprechende Antwort an den Nutzer.

Auf diese Art und Weise werden nicht nur die Anfragen an die API, sondern auch eingehende Nachrichten an die WebSocket Server über alle Knoten verteilt. Mehr hierzu im **Kapitel 5**.

5 Implementierung und Entwurfsentscheidungen

In diesem Kapitel wird erläutert, wie die zuvor dargestellte Struktur umgesetzt wurde. Im Speziellen wird darauf eingegangen, welche Module verwendet wurden, warum sich für gewisse Implementationsmuster entschieden wurde und welche Vorzüge und Schwachstellen hieraus entstehen.

5.1 Wahl der Programmiersprache

Bei der Betrachtung vorhandener Services fiel auf, dass die Sprache, in der die Projekte, die sich mit der nachrichten-basierten Kopplung befassen, sehr ungleichmäßig verteilt ist. Vertreten sind Programmiersprachen, die typischerweise für die Webentwicklung verwendet werden, allen voran NodeJS, JavaScript, Python und Java. Den größten Anteil machten hierbei die Projekte aus, die in NodeJS implementiert wurden.

Programmiererfahrungen in diesem Gebiet waren lediglich mit der Sprache Python vorhanden.

Eine wichtige Eigenschaft der Programmiersprache ist die Leistungsfähigkeit für Webanwendungen. Um diese zu ermitteln wurden einfache Prototypen implementiert, die einen simplen WebSocket Echoserver realisieren. Die Prototypen wurden in den Programmiersprachen Python, NodeJS und Java umgesetzt.

5.1.1 Implementierungsaufwand

Die Implementierung des Python Prototyps brauchte weniger als 20 Zeilen Code, wobei drei Bibliotheken verwendet wurden. Der Zeitaufwand betrug etwa 4 Stunden inklusive Recherche. Die Dokumentation der Sprache und der verwendeten Module war sehr gut.

Die Implementierung des NodeJS Prototyps brauchte weniger als 20 Zeilen Code, wobei eine Bibliothek verwendet wurde. Der Zeitaufwand betrug weniger als 2 Stunden inklusive Recherche. Die Dokumentation der Sprache und der verwendeten Module war sehr gut.

Der Implementierungsansatz in Java umfasste über 30 Zeilen Code. Es wurden etwa drei Abhängigkeiten sowie ein externer Webserver verwendet. Die Dokumentation war schwierig zu durchblicken. Nach 8 Stunden Zeitaufwand wurde die Implementierung abgebrochen.

5.1.2 Performanz

Die Prototypen, die in Python und NodeJS entworfen wurden, wurden Belastungstests unterzogen. Zunächst wurde durch eine einfache JavaScript Anwendung die Antwortzeit für 300.000 sequenziell gesendete WebSocket Nachrichten getestet. Bei diesem Test schnitt der NodeJS Prototyp mit einer Antwortzeit von durchschnittlich 50,81 Sekunden gegenüber dem

Python Prototypen mit 71,92 Sekunden erkennbar besser ab. Der Python Prototyp stürzte zudem gelegentlich ab.

Anschließend wurde ein Belastbarkeitstest mit dem Testwerkzeug Artillery durchgeführt. Artillery ist ein Testprogramm für HTTP- und WebSocket-Server, das es erlaubt Belastungsszenarien zu konstruieren und auszuführen. Artillery ist in JavaScript geschrieben und unterstützt die Simulation von mehreren Clients, kontinuierlich steigende Last und verschiedene HTTP Methoden. [27]

Es wurden mehrere Artillery Tests durchgeführt, in denen innerhalb eines Zeitintervalls die Clientzahl kontinuierlich gesteigert wurde. Die maximale Clientzahl wurde bei jedem Test erhöht. Alle Tests wurden auf dem lokalen System durchgeführt, sodass Netzwerkeinflüsse ausblieben. Die maximale Clientlast, bei der im Python Prototyp, Nachrichtenverluste auffielen, lag zwischen 120 und 150 Clients. Der Nachrichtenverlust betrug 37% bei 500 Clients und 33% bei 1000 Clients. Der NodeJS Prototyp zeigte bis 1000 Clients keine Nachrichtenverluste. Die Ergebnisse bei 120 Clients zeigt **Tabelle 4**.

Prototyp	max. Clientzahl	Durchschnittliche Antwortzeit	Maximale Antwortzeit
Python	120	0,6 ms	11,8 ms
NodeJS	120	0,3 ms	8,2 ms

Tabelle 4: Loadtest Ergebnisse Prototypen

Es wurden weitere Prototypen für die Implementierung einer einfachen REST-API zum Anlegen, Lesen und Löschen von Channel Objekten entworfen.

Der Python Prototyp umfasst 100 Zeilen Code, benutzt 5 Abhängigkeiten und benötigte etwa 2 Stunden Zeitaufwand.

Der NodeJS Prototyp umfasst ebenfalls 100 Zeilen Code, benutzt 5 Abhängigkeiten und benötigte etwa 2 Stunden Zeitaufwand.

Da die Antwortzeiten der API weniger entscheidend sind wie die des WebSocket Servers, wurden für diese Prototypen keine Belastungstests, sondern nur Funktionstests durchgeführt. Die Antwortzeiten lagen hierbei für wenige Anfragen in dem, für ein lokales Netzwerk, entsprechenden Bereich von wenigen Millisekunden.

5.1.3 Entscheidung

Nach Betrachtung der Prototypen spricht vorerst alles für die Verwendung von NodeJS. Ein weiterer Grund, der für die Verwendung von NodeJS spricht, ist der Event-basierte Aufbau der Sprache. Dies ermöglicht sehr einfach nicht-blockierende Ein- und Ausgabefunktionen, die

einem Webservice mit vielen Nutzern einen Performancevorteil geben. Auch Python Programme können asynchron realisiert werden, dies bedarf jedoch vergleichsweise wesentlich mehr Aufwand.

Nach Betrachtung aller Variablen wird der WebSocket Relay Service in NodeJS implementiert. Die festgestellten Performancevorteile, gute Dokumentation und der nicht-blockierende Aufbau der Sprache sprechen dafür.

5.2 Verwendete Module

Für die Lösung spezieller Probleme wurden NodeJS NPM Module verwendet. Im Folgenden wird darauf eingegangen welche Module genutzt werden, und wofür sie verwendet werden.

NPM-Modul	Quelle	Verwendung
stdio	[28]	bietet Ein-/Ausgabefunktionalitäten um Befehlszeilenparameter zu nutzen
ws	[29]	stellt den WebSocket Server des Channels dar
express	[30]	Framework für Webanwendungen liefert die Funktionalität für die REST-API
http / https		liefert Implementierungen des HTTP bzw. HTTPS Servers
body-parser	[31]	interpretiert den Request Body und stellt diesen als JSON Object zur Verfügung
cors	[32]	handhabt Cross-Origin-Requests und fügt ACESS-CONTROLL Header hinzu
axon-rpc	[33]	ermöglicht RPC Funktionen, die von Raft genutzt werden (verwendet axon)
axon	[21]	ist ein Framework für message-basierte Kommunikation über TCP
markdown-it	[34]	erzeugt HTML aus Markdown Dateien für die Darstellung der REAME.md
ip	[35]	ermöglicht den Zugriff auf die eigene Netzwerkadresse
public-ip	[36]	ermöglicht den Zugriff auf die eigene öffentliche Netzwerkadresse
url	[37]	Wird zum Auflösen von URLs genutzt.
swagger-ui-express	[38]	ermöglicht die API-Dokumentation über die Express-App aus der OpenAPI Spec

Tabelle 5: Übersicht verwendets Module

Auf die Kernmodule, die einen wesentlichen Beitrag zur Funktion des Service leisten wird hier noch einmal genauer eingegangen:

express ist ein Framework, das Funktionalitäten für Webanwendungen bereitstellt. In diesem Fall werden die Werkzeuge zum Bearbeiten von HTTP Requests genutzt, um die REST-API zu implementieren. Die Express-App verwaltet einen HTTP Server und nutzt seinerseits die Module "cors", "swagger-ui-expresss" und "body-parser". cors ist ein NPM Modul, das Cross-Origin-Requests handhabt. Mit Hilfe dieses Moduls lässt sich festlegen, wer auf die API

zugreifen darf, und welche Methoden zur Verfügung stehen. Hierfür fügt CORS weitere ACESS-CONTROLL Header zu der ursprünglichen Anfrage hinzu.

Das Modul ws liefert den WebSocket Server, der die WebSocket Kommunikation innerhalb des Channels ermöglicht. Der WebSocket Server verfügt über keinen eigenen HTTP Server, der die Verbindungsanfragen entgegennimmt, sondern bekommt die Upgrade Requests von der API weitergeleitet.

axon-rpc ist ein Modul, dass den Zugriff auf entfernte Methoden, sogenannte Remote-Procedere-Calls ermöglicht. Diese RPCs werden vom Raft Algorithmus zur Kommunikation zwischen den Knoten verwendet. Das Modul ist eine Erweiterung des Moduls axon, dass Sockets für Message gebundene Kommunikation über das TCP Protokoll liefert. axon selbst unterstützt die Messaging-Pattern push/pull, pub/sub, req/repl, pub-emitter/sub-emitter.

5.3 Anpassung der Messaging-Pattern

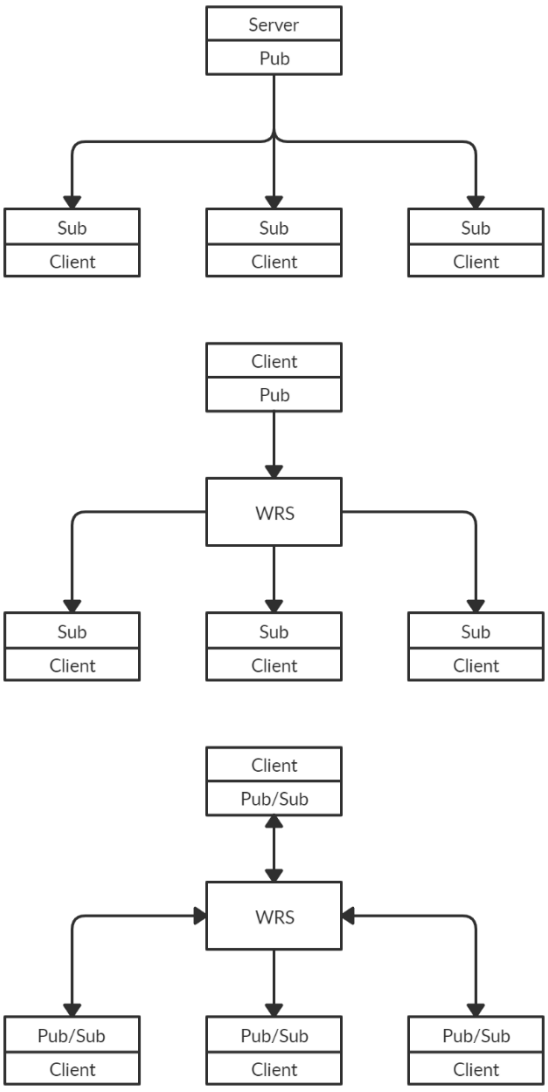

Abbildung 16: DFD Publish-Subscribe mit WRS

Die Anwendung der Messaging-Patterns wie sie in **Kapitel 2.3.1** erklärt werden, im WebSocket Relay Service wird zunächst anhand des Beispiels des Publish-Subscribe Pattern veranschaulicht.

Das Publish-Subscribe Pattern besagt, dass alle Nachrichten eines Publishers an alle verbundenen Clients weitergeleitet werden, die den Kanal abonniert haben (**Abbildung 16** oben).

Der WebSocket Relay Service (WRS) verbindet mehrere Clients untereinander und leitet die Nachrichten weiter. Um zu unterscheiden welcher Client den Channel abonniert hat, muss für jeden Channel diese Eigenschaft verwaltet werden und jeder Client muss seine Absicht bei Verbindungsaubau zu dem Channel mitteilen.

Um diesen zusätzlichen Aufwand zu vermeiden wird angenommen, dass jeder Client den Channel mit Aufbau der WebSocket-Verbindung abonniert (sub). Zudem hat jeder verbundene Client die Möglichkeit Nachrichten zu senden (pub). Jeder verbundene Client ist somit automatisch Publisher und Subscriber. Dies stellt kein Problem dar, unter der Annahme, dass Publisher wissen, dass sie nur Nachrichten senden möchten und eingehende Nachrichten ignorieren. Es muss gegben sein, dass Publisher die Nachrichten anderer Publisher sehen dürfen. Mit dieser Vereinfachung ergibt sich das Messaging-Pattern nach **Abbildung 16** unten. Das Publish-Subscribe Pattern wird damit zu einem simplen Broadcast zu allen verbundenen Clients. **Abbildung 17** zeigt als

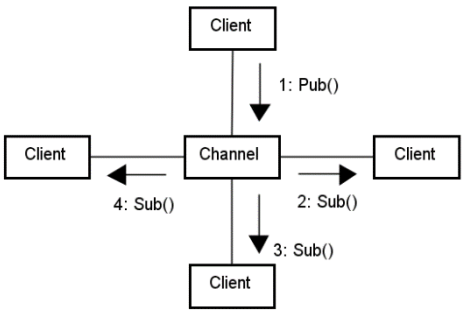

Abbildung 17: CD Pub-Sub

Kommunikationsdiagramm, wie der Channel die Nachrichten nach dem Publish-Subscribe Pattern weiterleitet.

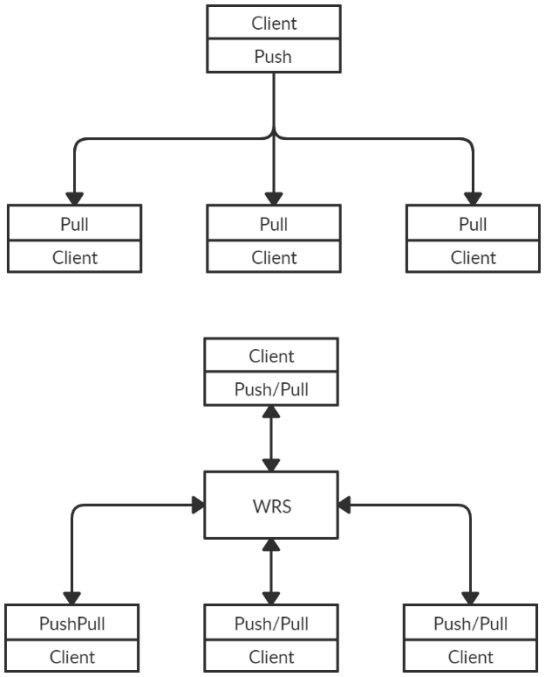

Abbildung 18: DFD Push-Pull mit WRS

Das Push-Pull Pattern hat wie in **Kapitel 2.3.3** erklärt das Ziel, Nachrichten gleichmäßig unter allen Clients zu verteilen. Für dieses Pattern werden dieselben Vereinfachungen vorgenommen wie für das Publish-Subscribe Pattern um spezielle Rollen für verbundene Clients zu vermeiden. Jeder Client darf sowohl Nachrichten senden (push), als auch empfangen (pull) (siehe **Abbildung** 18) Als Verteilungsalgorithmus für die Nachrichten unter den Clients wird ein einfacher Round Robin Algorithmus gewählt. Umgesetzt wird dieser durch eine Client-Queue, aus der der nächste Empfänger der eingehenden Nachricht ausgewählt wird. Der entsprechende Client wird anschließend vom Anfang der Queue entfernt und am Ende angehängt. Der Absender einer Nachricht erhält dabei niemals seine eigene Nachricht, solange weitere Clients verbunden sind. Unter der Annahme, dass dieser eine Aufgabe, die er selbst lösen kann, gar nicht erst abgesendet hätte. Der Absender, der an vorderster Stelle der Queue steht, wird somit an das Ende der Queue rotiert, bevor der nächste Empfänger ausgewählt wird. Befindet sich kein weiterer Client im Channel, so wird dem Absender die eigene Nachricht zurückgesendet.

Diese Vereinfachung verursacht Probleme sobald sich mehr als ein Client im Channel befindet, der Nachrichten sendet. Diese Nachrichten werden zwangsläufig bei den anderen Verteiler Clients ankommen, die im Zweifelsfall keine Worker sind. Dieses Problem lässt sich anwendungsseitig am einfachsten lösen, indem diese Nachrichten erneut gesendet werden. **Abbildung 19** zeigt wie drei aufeinander gesendete Nachrichten auf die verbundenen Clients verteilt werden.

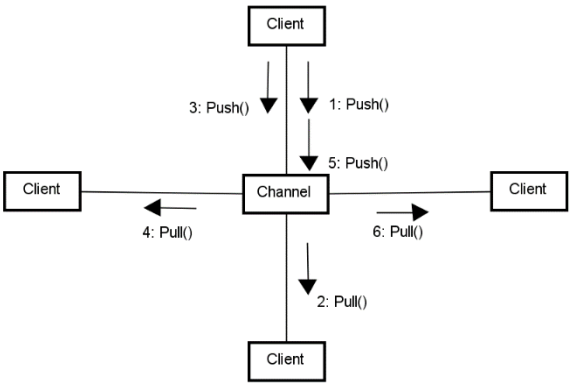

Abbildung 19: CD Push-Pull

Das Request-Reply Pattern befasst sich wie in Kapitel 2.3 beschrieben mit Anfragen (Request) und Antworten (Reply). Die Umsetzung des Request-Reply Messaging-Patterns nimmt zunächst die Selbe Vereinfachung vor wie die oben genannten Pattern. Jeder verbundene Client ist sowohl in der Lage Requests zu senden (req), als auch diese zu beantworten (repl).

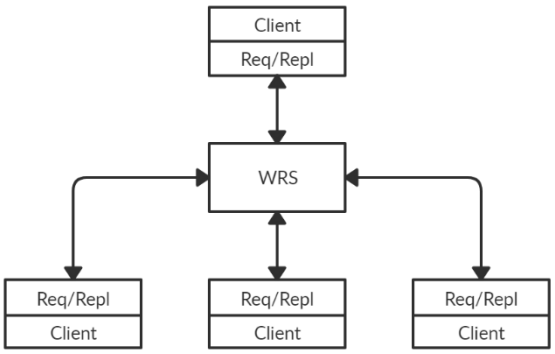

Abbildung 20: DFD Request-Reply mit WRS

Für die Verteilung der Requests unter den Clients gibt es zwei sinnvolle Ansätze. Zum einen ist es möglich jeden Request an alle Clients zu senden. Jedem Client steht es dann frei, je nach Rolle (Service oder Client) den Request zu beantworten. Jede Antwort wird dann über den Channel an den ursprünglichen Client zurückgeleitet. Alternativ kann nur die erste Antwort, also die schnellste Antwort, weitergeleitet werden. Dies nimmt jedoch der Anwendung die Entscheidungsfreiheit, welche Antwort ausgewertet werden soll, denn sie kann es im Zweifelsfall besser beurteilen. **Abbildung 21** zeigt die Verteilung eines Request unter mehreren mit einem Channel verbundenen Clients.

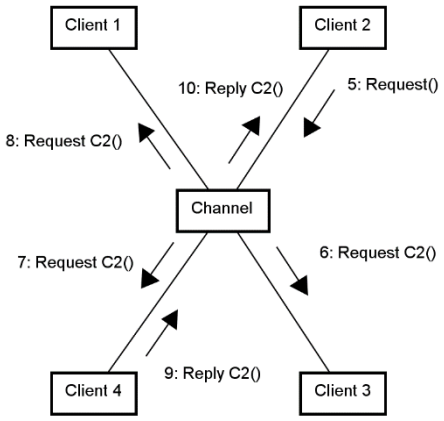

Abbildung 21: CD Request-Reply

Die Alternative besteht darin die Requests wie im Fall des Push-Pull Patterns gleichmäßig auf die Clients zu verteilen. Dies ermöglicht Loadbalancing Anwendungsszenarien sowie das in Kapitel 2.3 beschriebene Fan-out Fan-in Prinzip.

Der WebSocket Relay Service unterstützt beide Ansätze. Das Messaging-Pattern mit dem Namen Request-Reply sendet Requests an alle Clients, während das Messaging-Pattern mit dem Namen Fan-in/out Requests an einzelne Clients verteilt. Letzteres wird im Anschluss betrachtet.

Um die Antwort auf einen Request dem ursprünglichen Client zuordnen zu können ist es notwendig ein einfaches Subprotokoll einzuführen. Dies unterscheidet zum einen, ob es sich bei der an den WebSocket-Server gesendeten Nachricht um eine Request- oder eine Reply-Nachricht handelt und die Herkunft der Request beinhaltet. Dieses Subprotokoll basiert auf JSON und verpackt die eigentliche Nachricht der Anwendung in einem JSON Objekt.

Dieses JSON Objekt muss zwei Felder besitzen, die vom WebSocket Relay Service verwendet werden, um die Nachricht weiter zu leiten. Das Feld "type" unterscheidet die Nachricht in Request und Reply und muss dementsprechend mit den Werten "request" oder "reply" belegt werden. Das Feld "origin" wird vom WebSocket Relay Service gesetzt und beinhaltet die Herkunft des Request. Diese muss beim Senden der Reply-Nachricht mitgeführt werden. Weitere Felder stehen dem Anwender zu Verfügung, um den eigentlichen Inhalt der Nachricht anzufügen.

```
1. {
2.     "type": "reply",
3.     "origin": "ABCD-EFGH",
4.     "payload": "Hello World!"
5. }
```

Abbildung 22: Request-Reply Subprotocoll

Das Messaging-Pattern Fan-in/out kombiniert den Round Robin Verteilungsalgorithmus des Push-Pull Messaging-Pattern mit dem Subprotokoll des Request-Reply Messaging-Pattern um die Verteilung von Workload nach dem Fan-out Fan-in Prinzip zu ermöglichen. **Abbildung 23** stellt das "Auffächern" der Workload auf verschiedene Clients und das anschließende Zusammenführen der Ergebnisse dar. Die Verteilung der Request-Nachrichten erfolgt wie im Push-Pull Fall durch Nutzung der Client-Queue mit Überspringen des Absenders. Für die Zuordnung der Requests wird das oben vorgestellte Protokoll verwendet. Dieses Messaging-Pattern ist darauf ausgelegt, dass ein Client den Workload verteilt. Allen anderen Clients werden die Requests zugeteilt.

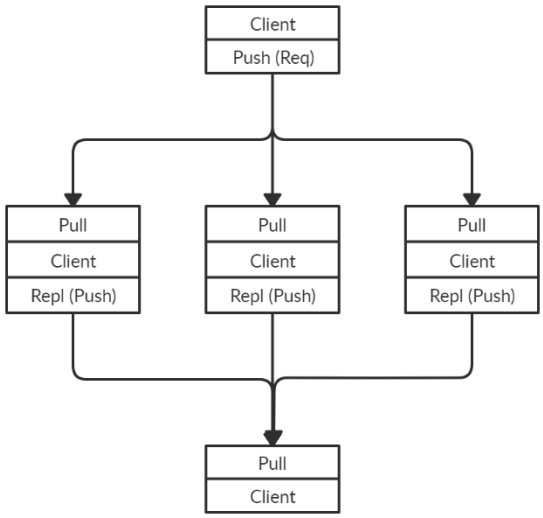

Abbildung 23: DFD Fan-in/out

Das bereits beschriebene Problem, dass Workload-verteilende Clients selbst Requests zugewiesen werden, ließe sich anwenderseitig durch Ablehnung des Requests über geeignete Reply-Nachricht und erneutem senden des ursprünglichem Requests lösen.

Ein weiteres Messaging-Pattern wurde zu Testzwecken implementiert. Der Echo-Channel sendet jede eingehende Nachricht an seinen Absender zurück.

5.4 Horrizontale Skalierbarkeit

Eine typische Eigenschaft, die beim Entwurf verteilter Systeme gewünscht ist, ist die Zustandslosigkeit. Das bedeutet, dass auf der Seite des Service keine Informationen über den Client gespeichert werden. Zustandslose Protokolle bringen somit alle Informationen, die der

Service für die Bearbeitung benötigt, in der Anfrage mit. Diese Eigenschaft ist besonders wichtig für Server Systeme, bei denen mehrere identische Server hinter einem Loadbalancer verborgen sind. In diesem Fall kann nicht sichergestellt werden, dass zwei Anfragen desselben Clients, vom selben Server bearbeitet werden. Dienst des WebSocket Relay Service ist es, feste WebSocket Verbindungen zwischen Clients anzubieten. die Zustandslosigkeit in einem Service zu gewährleisten, ist dadurch nicht unproblematisch. Dieses Kapitel befasst sich mit dem Entwurf des WebSocket Relay Service als horizontal skalierbaren Service.

5.4.1 Entwurf des Service

Betrachtet man den WebSocket Relay Service zunächst als monolithisches System, erscheint das Problem zunächst gut überschaubar. Es gibt einen Service, der HTTP Requests entgegennimmt, Channels erstellt und WebSocket Verbindungen aufbaut. Alle Clients die einen Channel abonnieren, sind mit demselben WebSocket Server verbunden und Nachrichten werden direkt über diesen Server ausgetauscht.

Existieren nun mehrere Instanzen des Service kommt es jedoch zu Problemen, insbesondere wenn die Anfragen zu diesen Instanzen durch einen Loadbalancer verteilt werden. Auf dessen Eigenschaften der Service, je nach Anwendungsumgebung, keinen Einfluss hat. In der Webentwicklung werden zustandsbehaftete Sitzungen oft durch sogenannte Sticky Sessions demselben Server zugeordnet, der die vorherige Anfrage des Clients bearbeitet hat. Solche Sticky Sessions lassen sich über einen Loadbalancer mithilfe von Cookies realisieren. Es kann

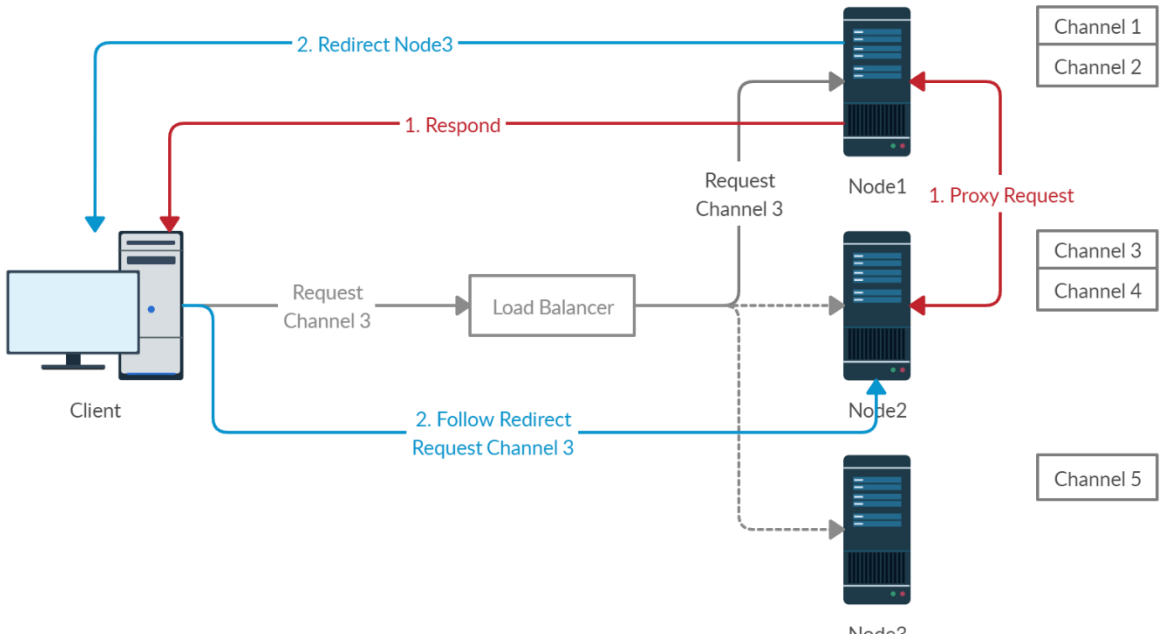

Abbildung 24: Loadbalancing mit Proxy und Redirect

jedoch nicht davon ausgegangen werden, dass der Loadbalancer diese Methode unterstützt.

Zudem lässt sich so nicht das Problem lösen, das zwei verschiedene Clients, die denselben WebSocket Server teilen wollten, auf unterschiedliche Server geleitet werden.

Eine weitere Möglichkeit könnte es sein Anfragen zwischen den Servern umzuleiten. Einfache Lookuptabellen auf den Servern, die Informationen darüber enthalten welcher Server welchen Channel hostet, könnten hier Abhilfe schaffen. **Abbildung 24** zeigt ein Beispiel, in dem drei Instanzen des Service über einem Loadbalancer vom Client angesprochen werden. Dabei hostet jeder Server einen Teil der Channels. Die Anfrage des Clients betrifft Channel 3, wird jedoch zum falschen Server weitergeleitet. Zunächst wird die Möglichkeit betrachtet die Anfrage an den entsprechenden Server weiterzuleiten. Es gibt HTTP Proxy Server, die diese Aufgabe übernehmen könnten. Der Server, der den HTTP Request erhält, würde also die Anfrage nicht selbst bearbeiten, sondern zu dem zuständigen Server weiterleiten, ohne dass der Nutzer es merkt. Die Antwort würde dann ebenfalls über den Proxy zurück an den Nutzer gesendet werden. Dies müsste jedoch nicht nur für alle HTTP Requests an die API, sondern auch für alle Nachrichten über die WebSocket Server geschehen. Dazu kommt die Tatsache, dass die Umleitung der Anfragen mit zunehmender Knotenzahl nicht zum Ausnahme-, sondern zum Regelfall würde. Die Kernaufgabe des Servers ist dann die Proxyfunktion zu anderen Servern zu erfüllen, was zu erheblichen Performanceeinbußen führt.
Eine weitere Möglichkeit Anfragen umzuleiten besteht darin den Client über einen HTTP-Redirect an den richtigen Server zu verweisen. Dies führt jedoch zu erheblichen Sicherheitslücken, da man die Adresse des Servers bekannt gibt. Daher scheidet auch diese

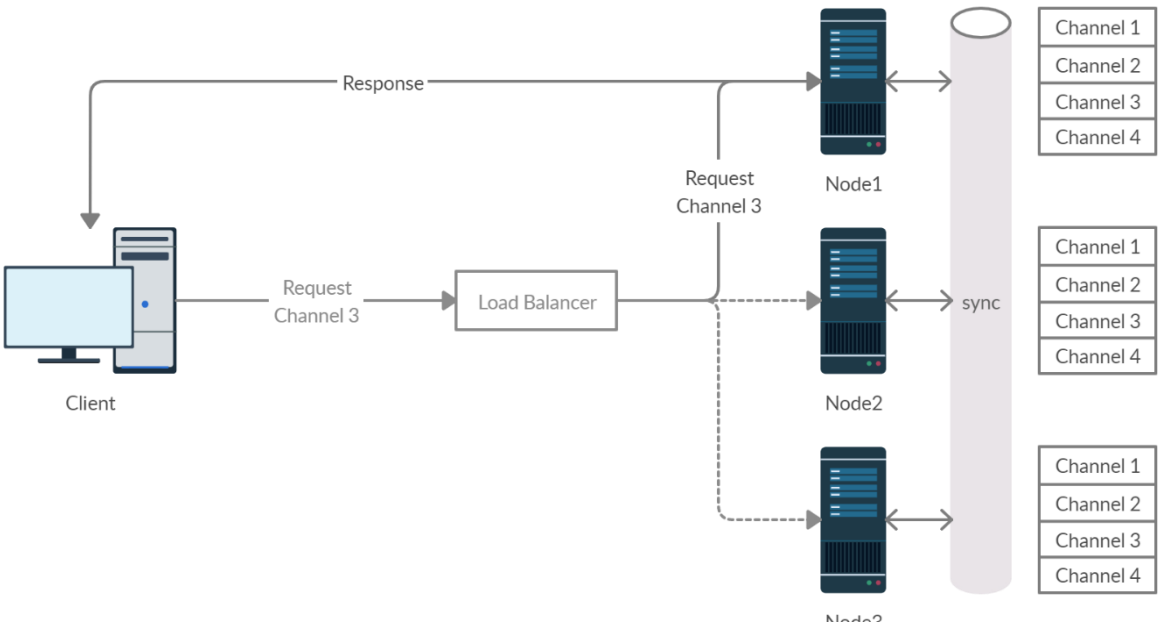

Abbildung 25: Loadbalancing mit Replikation

Möglichkeit aus.

Eine wahrscheinlich sauberere Methode das Problem zu lösen, hat einen anderen Ansatz. Bisher wurde davon ausgegangen, dass jeder Channel einen WebSocket Server hat, zu dem alle Clients eine Verbindung aufbauen. Gestaltet man den Service jedoch so, dass jede Instanz eine replizierte Version jedes Channels besitzt, dann kann jeder Knoten Anfragen und WebSocket Verbindungen der Clients entgegennehmen. Der Service wird als verteiltes System entworfen, das jede Ressource des Systems auf seinen Knoten repliziert. Im Backend der Anwendung tauschen die Knoten über einen Kanal Informationen und Nachrichten aus und streben einen konsistenten Zustand an. Dies schafft hohe Verteilungstransparenz, sorgt jedoch auch für Konsistenz- und Konsensprobleme. **Abbildung 25** zeigt das oben genannte Fallbeispiel in einem System mit replizierten Ressourcen.

Um dieses Konzept umzusetzen ist ein Kanal notwendig, über den die Knoten miteinander kommunizieren. Es werden Methoden benötigt das Cluster zu verwalten und neue Knoten mit dem Cluster zu verbinden. Außerdem ist ein Algorithmus erforderlich, der für die Konsistenz der Daten auf allen Knoten sorgt und mit dem Ausfall einzelner Knoten zurechtkommt.

In **Kapitel 2.2.2** wurde bereits der Raft Algorithmus vorgestellt und einige Vorteile gegenüber der weit verbreiteten Alternative Paxos erläutert. Für dieses Problem bietet sich der Algorithmus an, da er sowohl das Problem der Clusterverwaltung, als auch die Replizierung der Daten ermöglicht. Der Algorithmus wurde entwickelt, um gute Verständlichkeit und Implementierbarkeit zu bieten. Da die horizontale Skalierbarkeit im Rahmen dieser Arbeit in erster Linie prototypisch untersucht werden soll, bietet sich ein leicht zu verstehender Algorithmus an. Der Kern der Untersuchung liegt somit in dem Entwurf des Service als skalierbare Software und nicht in der Studie eines komplexen Konsens Algorithmus. Es existieren zwei Implementierungen des Raft Algorithmus in NodeJS, die für die Anwendung in Frage kommen. Beide werden im Folgenden betrachtet.

LifeRaft

LifeRaft ist eine Implementierung des Raft Algorithmus in NodeJS. Die Implementierung ist jedoch unvollständig. Es fehlt der Nachrichtentransfer zwischen Knoten. Dieser soll vom Anwender hinzugefügt werden. Zudem ist die Logreplikation dieser Implementierung optional und verwendet das NPM-Modul Leveldown [39] und wies starke Einschränkungen bei der Adressierung der Knoten auf. Es konnte kein einfaches funktionsfähiges Cluster mir diesem Modul implementier werden.

Skiff

Skiff ist eine weitere Implementierung des Raft Algorithmus. Die Raft-Implementierung baut auf den NPM Modulen Leveldown und Levelup auf. [40] Beide gehören zu dem

Datenspeichermodul LevelDB.[41][42] Diese Abhängigkeit weist starke Kompatibilitätsprobleme mit neueren NodeJS Versionen auf, die für die Implementierung des Service genutzt werden. Die in Skiff verwendeten Versionen von levedown (1.4.6) und levelup (1.3.2) [43] sind nicht mit neueren NodeJS Versionen kompatibel [44] und somit nicht für die Implementierung geeignet.

Nachdem sich beide Implementierungen als nicht zuverlässig erwiesen haben bleiben zwei Möglichkeiten. Die Programmiersprache gewechselt werden, um andere Abhängigkeiten zu suchen. Im Rahmen dieser Arbeit habe ich mich jedoch dafür entschieden den Raft Algorithmus eigenständig zu implementieren. Diese Implementierung stellt der RaftNode dar, dessen Aufbau im **Kapitel 4.1.3** beschreiben wurde. Im Folgenden werden einige Entwurfsentscheidungen für die Interaktion zwischen dem WebSocket Relay Service und seinem Raft Cluster beschrieben.

5.5 Raft commands

Das Raft Cluster ist in der Lage ein Log aus Commands, also eine Liste von Befehlen, über alle Knoten zu verteilen. Nun gilt es die Funktionalitäten, die im WebSocket Relay Service auftreten, in Befehle zu verpacken, damit diese über das Raft Cluster verteilt und von anderen Knoten interpretiert werden können.

Ein Befehl entsteht durch die Interpretation einer API-Anfrage oder durch Clientinteraktion mit dem WebSocket Server eines Channels. Interaktionen mit der API, die an alle Knoten verteilt werden müssen, sind alle nicht-lesenden Zugriffe auf Ressourcen. Zu diesen Zugriffen gehören das Erstellen eines Channels und das Löschen eines Channels. Zu der Clientinteraktion mit einem Channel gehört das Verbinden zu einem Channel, das Senden von Nachrichten und das Verlassen eines Channels.

Interaktion	**type**	**Parameter**
Channel erstellen	"addChannel"	pattern, timeout, clientLimit, channelId, token
Channel löschen	"deleteChannel"	channelId
eingehende Message	"message"	channelId, data, clients, cycle
neue Clientverbindung	"clientConnect"	channelId, clientId
Client löschen	"clientDisconnect"	channelId, clientId

Tabelle 6: Commands/Events

Die Commands werden als Java-Objekt übergeben und enthalten den Typ des Befehls. Ist ein Command erfolgreich über das Cluster repliziert worden erzeugt der Knoten ein "commit" Event, woraufhin der RelayService den entsprechenden Befehl ausführt. Jeder Befehl

beinhaltet die nötigen Parameter. Welche Typbezeichnungen und welche Parameter den Befehlen zugeordnet sind lässt sich **Tabelle 6** entnehmen. Wie Clientverbindungen und Nachrichten über das Cluster repliziert werden wird im Folgenden genauer betrachtet.

5.6 Remote-Clients

Durch die vollständige Replizierung der Channels auf jeden Server wird die Verwaltung der Clients und die Weiterleitung eingehender Nachrichten erheblich komplexer. Nach welchem Muster Nachrichten unter verbundenen Clients weitergeleitet werden, wurde in **Kapitel 5.3** bereits beschrieben. Diese Methode, inklusive der Verwendung der Clientqueue, bleibt bestehen. Hinzu kommt jedoch, dass Clients auf verschiedenen Servern die weiterzuleitende Nachricht erhalten müssen, wenn sie mit demselben Channel verbunden sind. Genauso müssen diese vom Verteilungsalgorithmus berücksichtigt werden. Um dieses Problem effektiv zu lösen werden Remote-Clients eingeführt. Dies sind Dummyclients, die in der Clientliste, sowie der Clientqueue des Channels vorhanden sind, jedoch keine aktive WebSocket Verbindung haben, da sie stellvertretend für den entfernten Client auf einem anderen Server stehen. Diese Remote-Clients unterscheiden sich von anderen durch eine boolean Variable, die unterscheidet, ob es sich um einen Remote-Client handelt. Remote-Clients haben außerdem keinen WebSocket Clients.

Baut ein WebSocket Client eine Verbindung (über die API) zu dem WebSocket Server eines Channels auf, so wird ein neuer Client erzeugt und dieser erhält eine clientId. Daraufhin sendet der Server, auf dem die Verbindung hergestellt wurde, den "clientConnect" Befehl über das Cluster, woraufhin die entsprechenden replizierten Channels ihrerseits einen Remote-Client erzeugen, der dieselbe clientId besitzt.

Eingehende Nachrichten werden von jedem Channel lokal nach dem entsprechenden Messaging-Pattern bearbeitet. Es wird ermittelt welche Clients demnach die Nachricht erhalten sollen und was mit der Clientqueue geschieht. Dann wird der "message" Befehl erstellt, der eine Liste der Clients beinhaltet, die die Nachricht erhalten sollen, sowie eine Variable mit Queueinformationen. Erst nachdem das "commit" Event die Knoten erreicht, sendet jeder Channel die Nachricht aus dem Befehl an alle Clients, die mit seinem WebSocket Server verbunden sind und rotiert die Queue entsprechend dem Befehl.

5.7 WebSocket Connection

WebSockets nutzen dieselbe Infrastruktur wie HTTP. Somit können WebSocket Verbindungen von HTTP-Relay- oder Proxy-Servern umgeleitet werden. Ein großer Vorteil, den sich der WebSocket Relay Service zu Nutzen macht, ist die Tatsache, dass WebSocket Verbindungen und HTTP Requests vom selben HTTP Server entgegengenommen werden können.

Jeder Channel verfügt über einen eigenen WebSocket Server. Dies hat einerseits Stabilität und Performance Gründe und andererseits sorgt es für Sicherheit durch die logische Trennung der Channel und seiner Clients über die Verteilungsalgorithmen hinaus.
Ein WebSocket Client, baut eine Verbindung auf, indem dieser einen HTTP GET Request mit entsprechendem Upgrade Header sendet. Dieser Request beinhaltet die Channel ID, sowie den Channeltoken. Die Anfrage richtet sich zunächst an die REST-API, wird vom HTTP-Server der API entgegengenommen, interpretiert und anschließend an den WebSocket Server des entsprechenden Channels weitergeleitet. Abschließend führen Server und Client den Opening Handshake durch und öffnen den WebSocket Kanal.

5.8 Offene (bekannte) Probleme

In diesem Kapitel werden bekannte Bugs und Engpässe des Service beschrieben, erklärt wodurch sie entstehen und mögliche Ansätze um sie zu beheben genannt.

5.8.1 Log Snapshot

Während des Betriebs des Raft Cluster wächst das Log des Clusters mit jedem Befehl, der über dieses repliziert wird. Der Raft Algorithmus sieht für die Lösung dieses Problems vor, das Log in regelmäßigen Abständen als Snapshot in den remanenten Speicher zu schreiben. Der Snapshot sorgt dafür, dass der Service auch in Singlenode-Nutzung nicht durch einen Speicherüberlauf abstürzt. In der finalen Version führt dies jedoch noch zu Problemen bei der Verbindungsherstellung von neuen Servern im laufenden Betrieb durch Logreplizierung.

5.8.2 Unregelmäßigkeiten der Lastverteilung

Bei starker Kadenz der Nachrichten auf einem Channel der die Clientqueue nutzt, um Nachrichten zu verteilen kann es dazu kommen, dass Clients mehrfach hintereinander ausgewählt werden. Dieser Effekt entsteht dadurch, dass die nächste Nachricht nach dem Round Robin Verfahren weitergeleitet wird bevor der "message" Befehl der vorherigen Nachricht die Queue bearbeitet hat. Bei hoher Kadenz kann sich dieser Effekt jedoch statistisch aufheben, da er bei jedem Client auftreten kann.

5.8.3 API Responses

Die REST-API erzeugt die HTTP Response sobald die Anfrage lokal bearbeitet und dem Cluster zur Replikation übergeben wurde. Dies geschieht um die Antwortzeit zu minimieren. Um die Antwort nach erfolgreichem replizieren des Befehls zu senden, muss das "commit" Event, das den Befehl enthält, dem HTTP Request zugeordnet werden könnnen. Dies wäre mit fortlaufender Nummerierung der Befehle und einer Map, die Befehle und Requests zuordnet, realisierbar.

6 Nachweisführung

Für die Nachweisführung wird zunächst die Korrektheit der Anwendung überprüft. Im Rahmen dessen wird die Abdeckung der, im Zuge der Anforderungsanalyse festgelegten, Systemanforderungen überprüft. Anschließend wird durch Performancetests die Leistungsfähigkeit des Service geprüft und die Belastungsgrenzen ermittelt.

6.1 Nachweis der Korrektheit der Anwendung

Um die korrekte Funktionsweise der Anwendung und die Abdeckung der funktionalen Systemanforderungen nachzuweisen wird der Service getestet.

6.1.1 Testen mit Mocha und Chai

Um die korrekte Funktion des Service bei der Entwicklung zu testen werden die NodeJS-Module Mocha und Chai verwendet, die in Kombination ein solides Testframework liefern. Mocha ermöglicht das automatische Ausführen von Testfällen und eine detaillierte Auswertung inklusive Codeabdeckung und Ausführungszeiten. Chai ist eine Assertion-Bibliothek, die es erleichtert, Aussagen zur Funktionsweise der zu testenden Anwendung zu formulieren und zu überprüfen. Im Folgenden wird beschrieben, welche Systemanforderungen von den Systemtests abgedeckt werden und welche Aussagen für die jeweiligen Tests formuliert worden sind.

Basiskomponenten

Die Basiskomponenten des Service umfassen das Verwalten der Channel über die API, das Verbinden eines Clients zum WebSocket Server und das Senden und Empfangen von Nachrichten unter den jeweiligen Messaging-Patterns. Die Basisfunktionalitäten des Service werden zunächst mit Standardeinstellungen des Service, ohne SSL Verschlüsselung oder Authentifikationstoken getestet.

Die Testfälle unterteilen sich in:

1. Basiskomponenten
2. SSL Verschlüsselung
3. Authentifizierungstoken
4. Browser-Only-Mode
5. Cluster

In **Tabelle 7** werden alle Testfälle, die mittels des Testtools automatisch ausgeführt werden, aufgelistet und die getestete Anforderung genannt.

Testfall	**Beschreibung** (Testfall testet …)	**Req.**	**test**
1.1.1	Erstellung des Servers		✓
1.1.2	Dass HTTP Server läuft		✓
1.2.1	GET / liefert HTTP		✓
1.2.2	GET /readme liefert HTTP		✓
1.2.3	GET /cluster liefert Clusterconfig als JSON		✓
1.2.4	GET /channels liefert Channels als JSON		✓
1.2.5	POST /channels erstellt Channel, Antwort als JSON	[FR1]	✓
1.2.6	GET /channels/{id} liefert Channel als JSON		✓
1.2.7	DELETE /channels/{id} loscht Channel	[FR2]	✓
1.2.8	GET /channels liefert keine Channel		✓
1.3.1	Verbindung zum WS, senden über WS über Echo Channel	[FR3][FR4]	✓
1.3.2	Senden und empfangen über Pub-Sub Channel	[FR5]	✓
1.3.3	Senden und empfangen über Push-Pull Channel	[FR6]	✓
1.3.4	Senden und empfangen über Req-Repl Channel	[FR7]	✓
1.3.5	Fehlschlag der Verbindung mit falschem Token	[NFR8]	✓
1.4.1	Löschen des Channels nach Timeout	[NFR9]	✓
1.4.2	Verlängerung des Timeouts durch Message	[NFR9]	✓
1.5.1	Limitierung des Channels mit Clientlimit	[NFR10]	✓
2.1.1	SSL Verschlüsselung der REST-API	[NFR6]	✓
2.2.1	SSL Verschlüsselung des Websocket Servers	[NFR7]	✓
3.1.1	Erfolg bei korrektem Auth Token im Header		✓
3.1.2	Erfolg bei korrektem Auth Token im Pfad (Querry)		✓
3.1.3	Erfolg bei korrektem Auth Token im Body		✓
3.1.4	Fehlschlag bei falschem Auth Token im Header		✓
3.1.5	Fehlschlag bei falschem Auth Token im Pfad (Querry)		✓
3.1.6	Fehlschlag bei falschem Auth Token im Body		✓
3.2.1	Websocket Verbindung mit Auth Token im Pfad (Querry)		✓
4.1.1	GET /channel/create erstellt Channel, Antwort als JSON		✓
4.1.2	GET /channels/{id}/delete löscht CHannel		✓
5.1.1	Clusterverbindung schlägt fehl ohne weiter Knoten		✓
5.1.2	Clusterverbindung gelingt manuell über /cluster/join	[NFR11]	✓
5.1.3	Clusterverbindung gelingt automatisch über /cluster	[NFR11]	✓
5.1.4	Knoten melden sich korrekt ab bei Beenden des Servers		(✓)
5.2.1	Channels werden über Knoten repliziert	[NFR11]	✓
5.3.1	Nachrichten werden über Knoten repliziert	[NFR11]	✓

Tabelle 7: Testfälle

Alle Testfälle konnten erfolgreich ausgeführt werden. Einzig Testfall 5.1.4 sorgte für technische Probleme beim automatischen Beenden eines Knotenprozesses und bedurfte manueller Eingriffe. **Abbildung 26** zeigt beispielhaft den Testfall 1.2.5.

```
1.  before(async function (done) {
2.      process.env.WRS_API_PORT = 80;
3.      process.env.WRS_DOMAIN = "localhost";
4.      process.env.WRS_SSL = false;
5.      process.env.WRS_AUTH = false;
6.      [...]
7.      new Service().run().then(ser => {
8.          service = ser;
9.      });
10. });
11. describe("[TF: 1.2] REST-API", function () {
12.     [...]
13.     //create Channel
14.     it('[TF: 1.2.5] [Req1] POST /channels should return JSON [...]',
15.         function (done) {
16.         chai.request(service.api.app).post("/channels").send({
17.             pattern: pattern,
18.             clientLimit: clientLimit,
19.             timeout: timeout
20.         })
21.             .end(function (err, res) {
22.                 chai.expect(err).to.be.null;
23.                 chai.expect(res).to.have.status(201);
24.                 chai.expect(res).to.have.header('content-type',
25.                     /application\/json/);
26.                 res.should.be.json;
27.                 res.body.should.be.a('object');
28.                 [...]
29.                 assert.equal(res.body.data.pattern,pattern);
30.                 assert.equal(res.body.data.clientLimit,clientLimit);
31.                 assert.equal(res.body.data.timeout,timeout);
32.
33.                 channel = res.body.data.id;
34.                 token = res.body.token;
35.                 done();
36.             });
37.     });
38.     [...]
39. });
40. after(async function () {
```

Abbildung 26: Beispiel Testfall Mocha & Chai

Aus dem Code lässt sich erkennen, wie für jeden Testfall vor Ausführung die Testumgebung eingerichtet, also der passende Service gestartet wird. Anschließend wird die gewünschte Interaktion mit dem Service ausgeführt und die Antwort auf erwartete Werte hin überprüft.

In den Testfällen 1.X.X werden alle Basiskomponenten des Service geprüft. Dazu gehört die Servicestruktur (1.1.X) die REST-API (1.2.X), die WebSocket Verbindung (1.3.X), der Channel Timeout (1.4.X) und das Clientlimit (1.5.X). Die Funktion des Timeouts wird durch das Senden zeitlich abgestimmter Nachrichten im Channel getestet. Durch Anfragen an die REST-API wird überprüft, ob sich der Channel bereits beendet hat. Diese Tests werden mit einer Toleranz von 25ms durchgeführt, die dem Channel erlauben den gewünschten Zustand anzunehmen und Verzögerungen durch Netzwerkkommunikation kompensieren. Das Clientlimit wird getestet indem Clients mit einem Channel verbunden werden, bis das Clientlimit erreicht ist.

Die Testfälle 2.X.X testen die SSL Verschlüsselung. Dafür wird der Service mit entsprechenden Parametern neu gestartet. Die Verschlüsselung wird für die REST-API sowie die WebSocket Kommunikation getestet.

Die Funktion des Channel-Tokens wird durch Verbindungsversuche mit korrektem und falschem Token getestet (3.X.X). Die erfolgreiche Verbindungsherstellung mit korrektem Token wurde bereits im Rahmen des Testfalls 1.3.1 erfolgreich getestet.

Letztlich wird die Vernetzung von Knoten zu einem Cluster getestet. Hierzu wird ein Server als Kindprozess gestartet, zu dem sich ein weiterer Server verbindet. Der Verbindungsaufbau wird sowohl manuell (über die REST-API) als auch automatisch getestet. Über dieses Cluster werden dann Channel angelegt und repliziert. Hier wird über API-Anfragen zu beiden Servern getestet, ob die Channel korrekt repliziert wurden. Mittels zweier Websocket Clients wird über diesen Channel dann der Nachrichtenaustausch über mehr als einen Knoten getestet.

Hier wurden nur die Testfälle näher betrachtet, die zur Abdeckung der Systemanforderungen beitragen. Weitere Testfälle zu erweiterten Funktionen des Service wurden geschrieben, werden jedoch hier nicht näher erläutert. Mit den Tests sind alle funktionalen Anforderungen ([FR1] bis [FR7]) sowie alle Features ([NFR6] bis [NFR10]) abgedeckt. Auch die grundlegende Skalierbarkeit wurde für zwei Knoten nachgewiesen.

Die korrekte Funktion des Service wurde durch die beschriebenen Testfälle erfolgreich nachgewiesen. Genaue Beschreibungen zu den Testfällen, die zum Nachweis der Anforderungen erforderlich sind, finden sich in tabellarischer Form im Anhang. Der Quellcode der Testfälle liegt der Arbeit in digitaler Form bei.

6.1.2 Nicht-funktionale Anforderungsabdeckung

Die Systemanforderungen [FR1] bis [FR7], sowie [NFR6] bis [NFR10] wurden bereits im Rahmen der Tests betrachtet. Im Folgenden werden daher die übrigen Anforderungen geprüft, und ermittelt, ob der Service die festgelegten Anforderungen erfüllt.

[NFR3] Das API soll REST-konform sein

Die Anforderungen einer RESTfull API bestehen in der Verwendung einer einheitlichen Schnittstelle, dem Client-Server Prinzip und Zustandsloser Kommunikation. Bei der folgenden Betrachtung wird der Browser-Only-Support außer Acht gelassen, da er die sinnvolle Verwendung der HTTP Methoden bewusst verletzt.

Für die REST-API wird das HTTP Protokoll verwendet. Die HTTP Methoden werden hierbei passend verwendet. Lesende Zugriffe auf Ressourcen erfolgen über GET-Requests, das Erzeugen der Channel und hinzufügen von Servern erfolgt über POST-Requests und das Löschen von Ressourcen, sowie das Zurücksetzen des Service über DELETE Request. Ressourcen werden über eindeutigen Namen angesprochen ("channels", "cluster") und Elemente über ihre ID. Der Verbindungsaufbau zu einem WebSocket Server könnte über die HTTP CONNECT Methode erfolgen. Die Entscheidung dies über einen GET Request umzusetzen hat technische Gründe, die auf dem Aufbau der WebSocket API beruhen.

Auf Serverseite werden keine Sitzungsinformationen der Clients gespeichert, die über die WebSocket Verbindung hinausgehen. Die Verwaltungslogik und Nachrichtenweiterleitung findet ausschließlich auf der Serverseite statt, somit wird das Client-Server Prinzip nicht verletzt.

[NFR5] Der Service soll die „12-Factor App Prinzipien“ berücksichtigen

Im Folgenden wird auf die Themen der "12-Faktor App Prinzipien" eingegangen und beschrieben inwiefern diese in der Entwicklung, Architektur und Implementierung des WebSocket Relay Service berücksichtigt werden.

Für die Entwicklung des Service wird eine einzige Codebasis verwendet, die durch Versionsverwaltung gepflegt wird [NFR5.1]. Während der Entwicklung wurde der Service sowohl lokal, als auch in einer Docker-Umgebung auf Lauffähigkeit getestet. HTTP Anfragen wurden mit dem Tool Postman und über die Swagger API getestet [NFR5.10]. Dieser Service definiert alle Abhängigkeiten in Form von NPM Modulen in der package.json Datei [NFR5.2]. Von hier aus können Anwendungsumgebungen diese automatisch installieren. Alle Konfigurationsvariablen werden als Umgebungsvariablen gespeichert. Standardwerte für

Ports, Pfade andere Konstanten sind ebenfalls in dem Umgebungsvariablen festgelegt [NFR5.3]. Der WebSocket Relay Service stellt Backend-Service für weitere Software dar und ist über eine universelle API ansprechbar. Der Service nutzt seinerseits keine weiteren Ressourcen [NFR5.4]. Für die Verwendung der API betreibt der Service einen eigenen Webserver [NFR5.7]. Der Build-Release-Run Prozess erfolgt über die Kombination des Codes mit den Konfigurationsvariablen zu einem Docker Image. Dieses wird dann in einer Docke oder Kubernetes Umgebung ausgeführt [NFR5.5]. Der Service bietet Nachrichtenaustausch als Dienstleistung. Daher ist es nicht möglich den Service als shared-nothing Architektur zu gestalten, bei der jeder Request bearbeitet wird ohne mit anderen Knoten zu komunizieren. Auf persistente Speicherung in einer Datenback wurde bewusst verzichtet [~~NFR5.6~~]. Der Service läuft als ein einziger eventbasierter Prozess, der jedoch horizontal skalierbar ist [~~NFR5.8~~]. Durch die bestehenden WebSocket Verbindungen ist das spontane Beenden des Service schwierig. Der Abbruch der WebSocket Verbindung zum Client und eine erneute Verbindungsanfrage muss möglicherweise durch das Herunterskalieren des Service in Kauf genommen werden. Die Abmeldung aus dem Cluster erfolgt automatisch [NFR5.9]. Logs und Debugausgaben werden vom Service in die Standardausgabe geschrieben [NFR5.11]. Die Administration des Service erfolgt ebenso wie die Nutzerinteraktion über die API. Eine Anleitung für die Nutzung und Administration liegt dieser Arbeit bei [NFR5.12].

Abschließend lässt sich feststellen, dass der Großteil der 12-Factor App Prinzipien erfüllt wurden. Wenige Aspekte ließen sich wie beschrieben aus technischen Gründen nicht einhalten.

[NFR4] Der Service soll als standardisiertes Container Image bereitgestellt werden

Durch die Orientierung an den 12-Factor App Prinzipien bei der Entwicklung des Service eignet sich dieser gut um als Container Image bereitgestellt zu werden. Die Konfiguration des Service lässt sich über Umgebungsvariablen im Dockerfile festlegen. Alle Abhängigkeiten können beim Build des Container Image automatisch installiert werden und sind somit Teil des Images. Die Interaktion mit dem Service über die API lässt sich über den API-Port an einen beliebigen ServerPort der Anwendungsumgebung binden. Ein Dockerfile für das benutzerdefinierte Erstellen eines Docker Image, sowie eine Anleitung für den Einsatz liegt dieser Arbeit bei.

[NFR11] Der Service soll horizontal skalierbar gestaltet werden

Der WebSocket Relay Service kann durch das hinzufügen weiterer Instanzen des Service horizontal zu einem Servercluster skaliert werden. Die Skalierung des Service erfolgt durch das Replizieren aller Channelressourcen und aller Nachrichten über die Knoten des Clusters. Grundlage für die Replizierung, sowie das Verhalten bei Ausfall einzelner Knoten bietet der

Raft Algorithmus. Die Funktion des Service in Clusterkonfiguration wurde durch Mocha und Chai Tests getestet. Zusätzlich wurden manuelle Tests mit Hilfe von Testclients durchgeführt. Einschränkungen der Funktion des Service durch den prototypischen Entwurf der horizontalen Skalierbarkeit wurden in **Kapitel 5.8** behandelt.

Die Abdeckung aller Anforderungen an den Service konnten somit durch Tests nachgewiesen oder durch Architektur und Eigenschaften der Anwendung belegt werden.

6.2 Performance

Die Performance des WebSocket Relay Service lässt sich für die zwei Nutzerinteraktionskomponenten betrachten: die REST-API und die WebSocket Kommunikation. Für beide gelten ähnlich Kenngrößen, an denen die Belastbarkeit des Service festgestellt werden kann. Die Request pro Sekunde (RPS) beziehungsweise die Nachrichten pro Sekunde, die der Service bewältigen kann, treffen eine gute Aussage über die Belastbarkeit des Service. Ebenfalls betrachtet werden sollte der prozentuale Verlust von Nachrichten beziehungsweise von Requests. Zusätzlich ist die maximale Clientzahl interessant, die der Service bewältigen kann.

6.2.1 Loadtest mit NPM Loadtest

Das Modul loadtest ist ein Softwareframework, das ähnlich dem Artillery Tool, das für die Loadtest der Prototypen verwendet worden ist und ermöglicht Belastbarkeitstests für HTTP- und WebSocket-Server. Als NPM Modul kann dieses in bestehenden Testumgebungen eingebunden und automatisch gestartet werden. Für HTTP Server unterstützt das Framework die Belastung des Service mit einer definierten Zahl von Requests pro Sekunde. Es lassen sich jedoch keine Szenarien beschreiben. Es wird durch die Optionsparameter des Tools nur eine Requestmethode zur Zeit unterstützt. Mit den Folgenden Tests wird zunächst das Verhalten des Service bei verschiedenen Clientzahlen mit Konstantem Durchsatz je Client getestet.

Das Testscript, mit dem die folgenden Ergebnisse ermittelt wurden liegt der Arbeit in digitaler Form bei. Der Service, der belastet wird, ist wie zuvor ein lokaler Server ohne SSL Verschlüsselung und ohne Authentifikation.

Der Test sendet POST-Requests an die API mit definierter RPS je Client.

Clientzahl	Dauer	Ø Antwortzeit	max. Antwortz.	Fehler	RPS
1	60 sek	5 ms	60 ms	0,00%	50
100	300 sek	67 ms	1981 ms	1,14%	400
500	300 sek	23285 ms	82520 ms	64,80%	500
1000	300 sek	978 ms	2218 ms	0,05%	100
1000	300 sek	22430 ms	81839 ms	62.55%	500

Tabelle 8: Ergebnist loadtest API

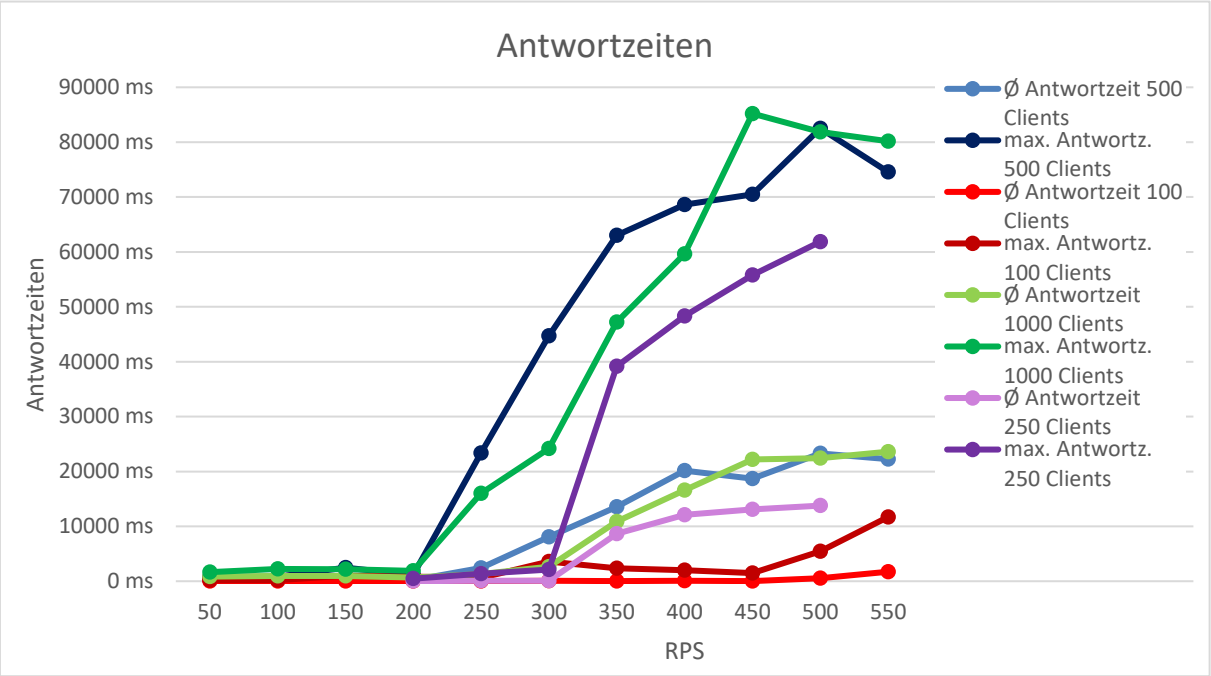

Abbildung 27: Antwortzeiten loadtest API

Die Antwortzeiten und Fehler wurden in Abhängigkeit von Clientzahl und Request pro Sekunde aufgezeichnet. Es ist deutlich zu erkennen, wie die Schwelle, bei der die Antwortzeit deutlich zunimmt, von der Anzahl der verbundenen Clients abhängt. Für 100 Clients beginnt die Antwortzeit ab einem Durchsatz von 500 RPS deutlich zu steigen. Bei >500 Clients steigt die Antwortzeit ab 200 RPS je Client. Ein ähnlicher Trend lässt sich im Verlauf fehlgeschlagener Requests in Abhängigkeit von RPS und Clientzahl feststellen.

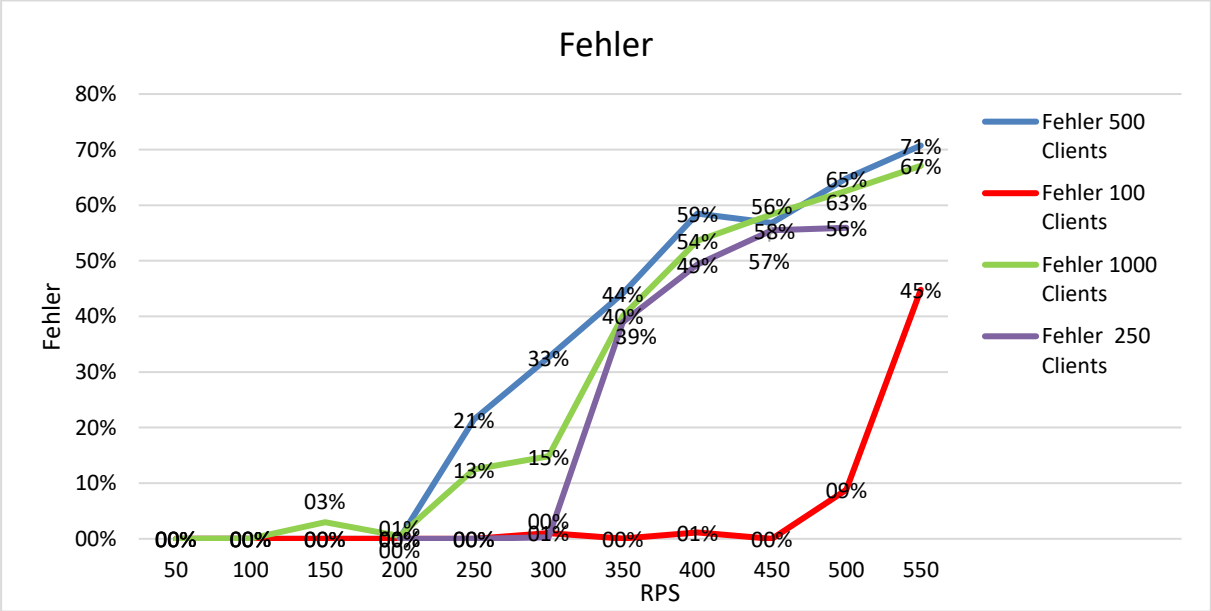

Abbildung 28: Fehler loadtest API

Auch die WebSocket-Server wurden mit dem Loadtest Tool getestet. Hier ließ sich kein Wert für die Nachrichten pro Sekunde einstellen. Der Wert ergibt sich aus der Kombination der Client- und der gesamten Nachrichtenzahl. Mit dem Loadtest Tool war es nicht möglich den Service soweit zu belasten, dass Nachrichten verloren gingen.

MPS	Ø Antwortzeit	max. Antwortz.	Fehler
1000	237 ms	1624 ms	0,0%
1500	138 ms	219 ms	0,0%
2000	254 ms	5696 ms	0,0%

Tabelle 9: Ergebnisee loadtest WS

Es ist zudem nicht auszuschließen, dass die Fehler nicht auf Einschränkungen der Testsoftware beruhen. Um die Testergebnisse zu überprüfen werden die Tests mit einer weiteren Testsoftware überprüft.

6.2.2 Loadtest mit Apache JMeter

Apache JMeter ist ein sehr umfangreiches Werkzeug um Last- und Performance Tests für Verschiedene Webanwendungen durchzuführen. Unter anderem unterstützt JMeter das Testen von HTTP und HTTPS Servern. Der Funktionsumfang von JMeter kann mit Plugins erweitert werden, sodass das Tool auch in der Lage ist Loadtests für Websocket Services durchzuführen. Durch JMeter können individuelle Testpläne erstellt und ausgeführt werden. Grundbestandteile eines Testplans sind die Threadgroups, die parallelisierte Clientinteraktionen ermöglichen, Kontrollstrukturen, mit denen sich der Testablauf steuern lässt, Samplers, über die sich Requests formulieren lassen, und Listeners, die Responses sammeln und auswerten können.

Mit JMeter lässt sich der gesamte Durchsatz für verschiedene Testabschnitte festlegen. Damit ergänzt JMeter das NPM loadtest Framework, das den einstellbaren Durchsatz nur je Client reguliert. Der Aufbau eines Testplans wird beispielhaft an einem kombiniertem API und WebSocket Test beschrieben.

Der Testplan startet 50 Clients. Jeder dieser Clients erzeugt über einen POST Request einen echo-Channel und extrahiert channelId und token. Jeder Client verbindet sich mit dem Channel und sendet Nachrichten über den Channel. Der gesamte Durchsatz kann

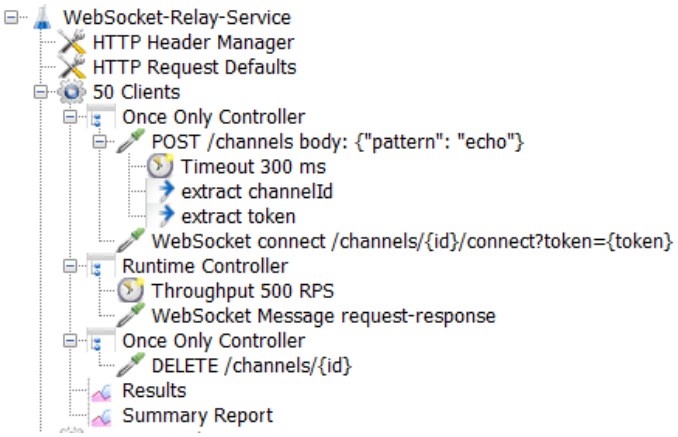

Abbildung 29: Beispiel Testplan JMeter

hierbei eingestellt werden. Zuletzt wird der Channel über einen DELETE Request wieder gelöscht.

Alle Testpläne, die für die folgenden Tests verwendet wurden, liegen der Arbeit in digitaler Form bei.

Zunächst wurde die Belastbarkeit der API des Websocket Relay Service getestet. Der Testplan beschreibt Clients, die zyklisch Channel erstellen, lesen und wieder löschen. Das Erstellen des Channels erzeugt im Server die meiste Last, da es das Erzeugen neuer Websocket Server auslöst. Das Lesen stellt sicher, dass die Channel wirklich angelegt werden. Das Löschen sorgt dafür, dass die Channelliste keinen Speicherüberlauf erzeugt.

Die Antwortzeiten steigen ab 200 RPS nur wenig an. Fehler treten bei diesem Test für 100 Clients ab einer Rate von über 800 RPS auf. Auch die Antwortzeiten steigen hier deutlich an. Ähnliche Ergebnisse bei Einstellung der gesamten RPS ergaben sich für 200 und 500 Clients.

RPS	Ø Antwortzeit	max. Antwortzeit	Fehler
50	5 ms	156 ms	0,0%
100	9 ms	393 ms	0,0%
150	11 ms	246 ms	0,0%
200	31 ms	350 ms	0,0%
250	73 ms	1592 ms	0,0%
300	250 ms	1942 ms	0,0%
350	240 ms	1406 ms	0,0%
400	460 ms	1696 ms	0,0%
550	126 ms	1350 ms	0,0%
750	328 ms	1500 ms	0,0%
900	1660 ms	7329 ms	15,3%
1000	1695 ms	10632 ms	18,7%
1500	1869 ms	11763 ms	34,6%

Tabelle 10: Ergebnisse JMeter API

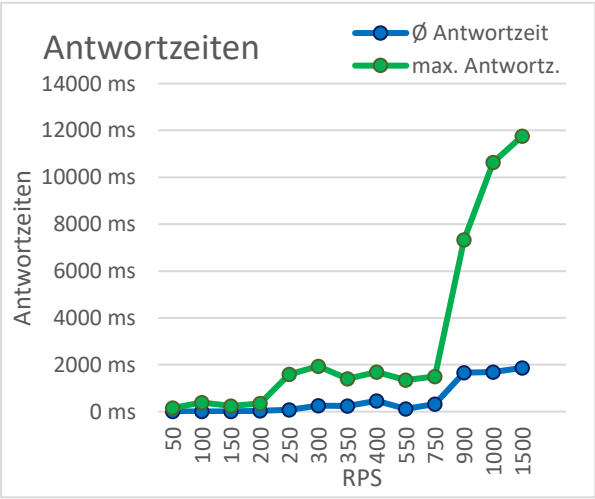

Abbildung 31: Diagramm Antwortzeiten JMeter API

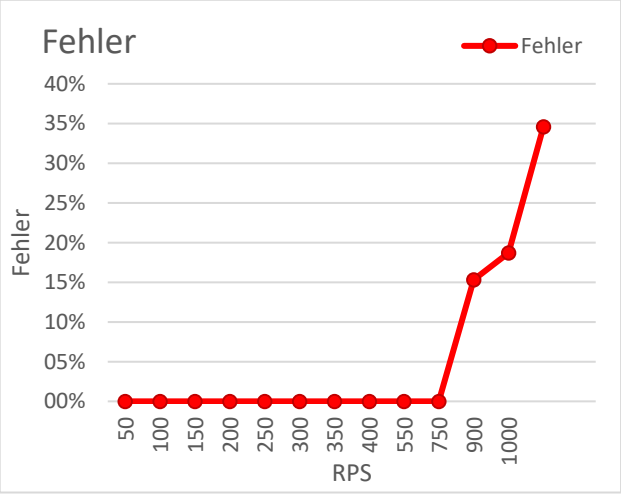

Abbildung 30: Diagramm Fehler JMeter API

Der Verlauf der Fehler und der Antwortzeiten entspricht dem der Tests, die zuvor aufgenommen wurden. Ein Anstieg ist jedoch deutlich später zu erkennen.

Der zeitliche Verlauf der Antwortzeiten während des Tests hat zudem gezeigt, dass die Responsivität des Service mit steigender Channelzahl abnimmt. Server die neu gestartet oder dessen Channel gelöst worden sind zeigen besseres Lastverhalten als Server, die viele Channel bereithalten.
Um den WebSocket Server unter Last zu testen, wurde zunächst ein Testplan angewandt, der wie im Beispiel beschrieben, Clients Echochannel anlegen lässt und mit definierter Zahl von MPS (Message per second). Die maximal Durchsatzrate, die mit den eingehenden Nachrichten weitergeleitet wurden betrugen hierbei 3300 MPS. Bei der Überschreitung dieser Durchsatzrate wird die Auslieferung der Nachrichten verzögert. Es treten hierbei bei geringen Clientzahlen (<250 Clients) Verzögerungen von bis zu 6 Sekunden auf, es kommt jedoch nicht zu Übertragungsfehler.

Bei hohen Clientzahlen treten Verbindungsfehler auf. Diese entstehen durch fehlschlagende Verbindungsversuche neuer Clients zum Websocket Server durch Überlastung. Bereits verbundene Clients zeigen weiterhin keine Übertragungsfehler, sondern lediglich starke Verzögerungen.

Ein weiterer Testplan nutzt den Publish-Subscribe Channel, um mehr Last im Websocket Server zu erzeugen. Jede eingehende Nachricht wird an alle verbundenen Clients weitergeleitet und sorgt so für eine höhere Ausgangslast. Hier beschränkte sich die maximale Antwortrate des Service auf etwa 8700 MPS. Dadurch werden Nachrichten verzögert ausgeliefert. Auch hier führt die Überlastung zu Verbindungsfehlern bei neuen Clients. Die Überlastung des Websocket Servers lässt sich hier jedoch schon mit wenigen Clients erreichen.

1 Client				10 Client				20 Client			
sending				**receiving**				**receiving**			
MPS	**Avg.**	**Max.**	**Err.**	**MPS**	**Avg.**	**Max.**	**Err.**	**MPS**	**Avg.**	**Max.**	**Err.**
500	0 ms	39 ms	0 %	4986	1 ms	222 ms	0 %	8653	3 ms	4751 ms	0 %
600	0 ms	41 ms	0 %	5956	1 ms	221 ms	0 %	8725	3 ms	6001 ms	0 %
700	0 ms	13 ms	0 %	6985	1 ms	229 ms	0 %	8734	3 ms	6142 ms	5 %
800	0 ms	32 ms	0 %	7871	1 ms	278 ms	0 %				
900	0 ms	22 ms	0 %	8442	1 ms	6001 ms	0 %				
1000	0 ms	66 ms	0 %	9552	1 ms	4914 ms	11 %				

Tabelle 11: JMeter Ergebnisse WS Pub-Sub

Bei einem sendenden Client und 10 empfangenden Client wir der Ausgangsdurchsatz im Verglich zu den eingehenden Nachrichten verzehnfacht. Für eine Belastung mit 1000 RPS

überschreitet dies die Grenze von 8700 MPS und sorgt für Verbindungsprobleme eines Clients und einer damit Verbundenen Fehlerrate von 11%.

6.2.3 Performance bei Skalierung

Im Folgenden wird untersucht, wie sich die Leistung des Service bei horizontaler Skalierung verhält. Erwartet wird eine Verbesserung der Antwortzeiten und eine höhere Belastungsgrenze durch die Verteilung der Cientanfragen auf mehrere Knoten. Es wurden 4 Testpläne für die Belastung des Clusters ausgeführt. Jeder Testplan wurde jeweils für die Clusterkonfiguration mit einem Knoten und mit drei Knoten ausgeführt. Der Testplan "API" beinhaltet POST Requests, die Channel anlegen, und GET Requests, die diese lesen. Der Testplan "WebSocket" verbindet Clients zu zuvor angelegten Channels und sendet Nachrichten. Die Belastung der beiden Komponenten wird im "Mixed" Testplan kombiniert, indem Channel erst über die API angelegt und anschließend mit Websocket Clients genutzt werden. Zuletzt wird der Service nur lesend belastet, indem GET Requests an die Channel Ressource gesendet werden.

Testziel	Knoten	Clients	Requests	Avg.	Min.	Max.	Error	Err in %
API	1	500	298947	578 ms	0 ms	1480 ms	0	0
	3	500	300046	773 ms	2 ms	35262 ms	4092	13
WebSocket	1	200	200013	547 ms	0 ms	2943 ms	0	0
	3	200	152105	489 ms	0 ms	6002 ms	40022	26,31
Mixed	1	50	277369	53 ms	3 ms	2871 ms	0	0
	3	50	306933	32 ms	0 ms	1204 ms	231786	75,52
API (read-only)	1	500	300025	347 ms	0 ms	945 ms	0	0
	3	500	300083	214 ms	0 ms	492 ms	0	0

Tabelle 12: Ergebnisse Vergleich Sigle-Node Cluster

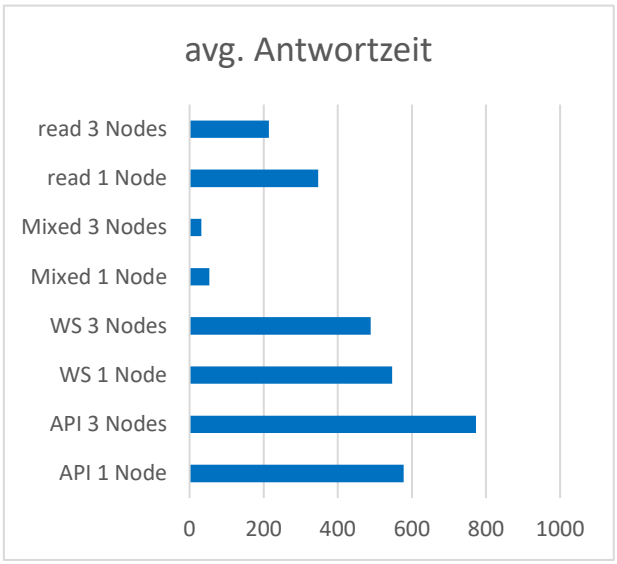

Abbildung 32: Diagram Antwortzeiten Single Node Cluster

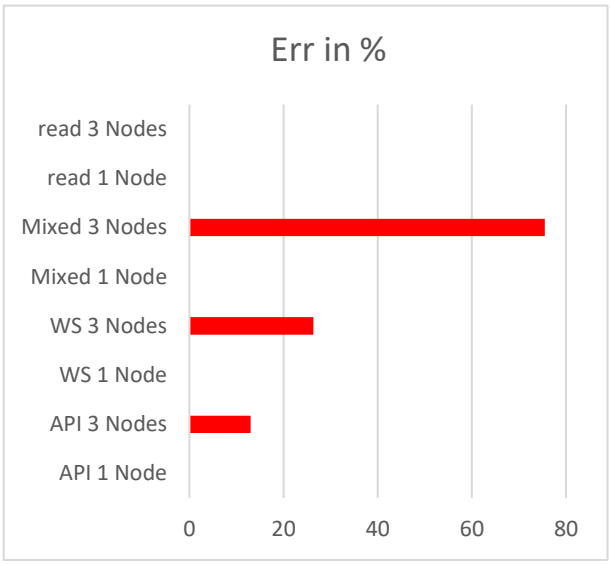

Abbildung 33: Diagram Fehler Single Node Cluster

Anhand der Testergebnisse zeigt sich, dass in der Clusterkonfiguration bei Lasttests, die im Single-Node Fall keine Probleme verursachen, Fehler auftreten. Nach Einsicht der Serverlogs lässt sich sagen, dass diese Fehler vor allem durch Verzögerungen bei der Replizierung der Ressourcen entstehen. Verstärkt wird diese Annahme durch das Ergebnis bei lesendem Zugriff, bei dem sich dieser Effekt nicht zeigt. Besonders stark treten die Fehler beim gemischten Testszenario auf, bei dem sich Replikationsfehler der Channel, die zu WebSocket Verbindungsfehlern führen, mit Nachrichtenreplizierungsfehlern kombinieren.

Performanceprobleme bei der Replizierung von Ressourcen sind in erster Linie auf den Aufbau des Raft Algorithmus zurückzuführen. Der Konsens Algorithmus basiert darauf, dass der Leader Knoten von der Majorität, also n/2+1 von insgesamt n Knoten, eine Bestätigung abwartet, dass ein Befehl im jeweiligen Log hinzugefügt wurde, bevor dieser ausgeführt werden kann. Dies sorgt dafür, dass die Komplexität mit dem Quadrat der gesamten Knotenzahl steigt.

Obwohl auch einzelne Knoten ein Log führen fällt dies für die Performance kaum ins Gewicht. Das Hinzufügen des Befehls zum Log findet lokal statt und beinhaltet keine Netzwerkkommunikation, da ein Knoten jeden Logeintrag sofort commited wenn es keine weiteren Knoten im Cluster gibt.

6.2.4 Vergleich mit Florkestra

Um die Performancewerte, die für den WebSocket Relay Service erfasst wurden einordnen zu können, wird der Service Florkestra mit ähnlichen Testplänen getestet. Der Service unterstützt simples Message Broadcasting. Die Ergebnisse werden daher mit den Tests des Publish-Subscribe Channels vergleichen. Die Ergebnisse aus **Tabelle 13** zeigen ähnliche Merkmale wie die des WebSocket Relay Service. Auch bei dem Florkestra Service treten

1 Client	10 Client				20 Client			
sending	**receiving**				**receiving**			
MPS	**RPS**	**Avg.**	**Max.**	**Err.**	**RPS**	**Avg.**	**Max.**	**Err.**
500	4975	0 ms	230 ms	0 %	8653	3 ms	4751 ms	0 %
1000	9888	1 ms	6001 ms	0 %	11508	2 ms	6002 ms	0 %
1500	10868	1 ms	6000 ms	0 %	12669	1 ms	6002 ms	0 %
2000	9857	1 ms	6001 ms	0 %	10983	3 ms	6002 ms	20 %
3000	9505	1 ms	6000 ms	0 %				

Tabelle 13: Ergebnisse JMeter Florkestra

Fehler durch fehlschlagende Verbindungsversuche zum WebSocket Server auf, jedoch bei höheren Durchsatzraten von 2000 MPS eingehend.

7 Fazit

7.1 Zusammenfassung

Mit dem Ziel Webclients message-basiert zu verbinden wurde ein Service gesucht, der mit wenigen Abhängigkeiten auskommt und in modernen Cloudumgebungen eingesetzt werden kann. WebSockets scheinen für diese Aufgabe gut geeignet. Das duplex-fähige Protokoll kann Clients und Server verbinden, nutzt dabei die vorhandene HTTP Infrastruktur des Web und wird von den meisten gängigen Browsern unterstützt.

Bei der Untersuchung existierender Softwarelösungen fällt auf, dass entweder Features fehlen, die für den flexiblen Einsatz des Service gewünscht sind oder umfangreiche Services mit Nutzeraccounts und eigener Clientsoftware zu komplex sind und für den Nutzer zu neuen Abhängigkeiten führen.

Um einen Websocket Relay Service zu entwickeln wurde im Rahmen der Anforderungsanalyse ermittelt, welchen Funktionsumfang und welche Features ein solcher Service umsetzen soll. Dazu gehörte ein Channelsystem für das Senden von Nachrichten, eine Tokenauthentifikation, eine API für Nutzerinteraktionen und die SSL Verschlüsselung der Kommunikation. Um den Service so zu gestalten, dass er problemlos als Docker Container deployed und im Cluster

eingesetzt werden kann, sind die 12-Factor App Prinzipien sehr hilfreich. Beim Entwurf des Service als horizontal skalierbares und somit verteiltes System gilt es einige Probleme zu lösen.

Es wurden Möglichkeiten untersucht, die Channel auf Knoten zu verteilen. Zwischen den Möglichkeiten jedem Channel einen eindeutigen Knoten zuzuweisen und so Clients zum korrekten Knoten zu leiten und der vollständigen Replizierung aller Channels führte die Entscheidung zu einem verteilten System, das den Raft Algorithmus nutzt um Konsistenz zwischen allen Knoten zu bewahren. Es wurde gezeigt, dass der Raft Algorithmus in der Lage ist den Websocket Relay Service skalierbar zu gestalten und für Konsistenz zu sorgen. Die Entscheidung für diese Architektur hat jedoch schwerwiegende Konsequenzen für die Performance des Service.

7.2 Bewertung des Service

Durch die Replikation aller Channel inklusive aller gesendeter Nachrichten über ein Cluster, entsteht ein großer Overhead, der die Durchsatzzeiten von Nachrichten erheblich einschränkt. Dieser Overhead sorgt dafür, dass der Service als Cluster unter Last schlechtere Ergebnisse erzielt als seine einzelnen Knoten. Damit ist das Ziel der Lastverteilung durch horizontale Skalierung verfehlt worden. Andere Vorteile der Skalierung bleiben jedoch erhalten. Die Replikation der Channel führt zu Ausfallsicherheit, denn der Raft Algorithmus ist mit der Majorität aller Knoten lauffähig.

Die Performance des Service als einzelner Knoten wird für zwei Komponenten separat betrachtet. Die Durchsatzrate der REST-API ist besonders beim Anlegen von Channels gering, wenn bereits viele Channel auf dem Server existieren.
Die Häufigkeit, mit der API-Anfragen zum Erstellen oder Löschen von Channels genutzt werden, wird jedoch verglichen mit der Regelmäßigkeit von Websocket Nachrichtenaustausch als selten eingeschätzt.

Bei der Weiterleitung der Nachrichten über den Websocket Server zeigt das Event-basierte Design von NodeJS seine Wirkung. Der Service hat eine maximale Durchsatzrate, bei der er Nachrichten senden kann. Wird diese überschritten, so gehen Nachrichten jedoch nicht verloren, sondern werden mit Verzögerung ausgeliefert.

7.3 Ausblick

Der Service wurde für die finale Version so implementiert, dass er bei der Nutzung als Einzelknoten stabil läuft. Für die Verwendung als Cluster gibt es einige Stellen an denen Verbesserungen vorgenommen werden können. Das größte Problem ist das Performanceproblem, das der kompletten Replizierung aller Ressourcen geschuldet ist. Die alternative jedem Channel einen dedizierten Knoten zuzuweisen, optimiert die Leistung der

Websocket Server, da alle Clients dann zu demselben Server senden. Für diesen Ansatz müssen Clients jedoch zum richtigen Knoten vermittelt werden, weshalb die Entscheidung gegen diesen Ansatz viel. Die Lösung des Problems könnte eine Hybridlösung sein, bei der zwar Channels über einen Konsens Algorithmus zu allen Knoten verteilt werden, Nachrichten jedoch nur zu den Knoten gesendet werden, ohne die Konsistenz der Nachrichten sicherzustellen.

Auch die Implementierung des Raft Algorithmus ist ein prototypischer Entwurf und Bedarf einiger Verbesserungen in den Punkten Stabilität, Aufholen bei Logdifferenzen, sowie bei der Implementierung von Grenzfällen bei der Leader Vote und Cluster-Konfigurationsänderungen.

Abschließend hat diese Arbeit aufgezeigt, welche Probleme bei Entwurf und Implementierung eines horizontal skalierbaren Service auftreten und welche speziellen Probleme durch die Vernetzung von Clients als Dienst des Service hinzukommen. Es wurde ein Service entworfen, dessen korrekte Funktion als Einzelkonten getestet und als Clusteranwendung protypisch untersucht wurde. Das Thema bietet jedoch Raum für viele weitere Überlegungen.

Abbildungsverzeichnis

Abbildung 1: DFD Publish-Subscribe 15

Abbildung 2: DFD Request-Reply 16

Abbildung 3: DFD Push-Pull 16

Abbildung 4: Beispiel WebSocket Upgrade Request 19

Abbildung 5: Beispiel WebSocket Upgrade Response 19

Abbildung 6: Browser Unterstützung WebSockets 20

Abbildung 7: Usecase Diagram 23

Abbildung 8: Klassendiagramm Service 30

Abbildung 9: REST-API Ressourcen 31

Abbildung 10: HTTP Methoden REST-API 32

Abbildung 11: Klassendiagramm Channel 34

Abbildung 12: Sequenzdiagramm Request 34

Abbildung 13: Klassendiagramm Raft 35

Abbildung 14: Sequenzdiagramm Command mit Leader Node 36

Abbildung 15: Sequenzdiagramm Command mit Follower 37

Abbildung 16: DFD Publish-Subscribe mit WRS 42

Abbildung 17: CD Pub-Sub 42

Abbildung 18: DFD Push-Pull mit WRS 43

Abbildung 19: CD Push-Pull 43

Abbildung 20: DFD Request-Reply mit WRS 44

Abbildung 21: CD Request-Reply 44

Abbildung 22: Request-Reply Subprotocoll 45

Abbildung 23: DFD Fan-in/out 45

Abbildung 24: Loadbalancing mit Proxy und Redirect 46

Abbildung 25: Loadbalancing mit Replikation 47

Abbildung 26: Beispiel Testfall Mocha & Chai 54

Abbildung 27: Antwortzeiten loadtest API 59

Abbildung 28: Fehler loadtest API 59

Abbildung 29: Beispiel Testplan JMeter 60

Abbildung 30: Diagramm Fehler JMeter API 61

Abbildung 31: Diagramm Antwortzeiten JMeter API 61

Abbildung 32: Diagram Antwortzeiten Single Node Cluster 64

Abbildung 33: Diagram Fehler Single Node Cluster 64

Abbildung 34: Dockerfile 75

Abbildung 35: deoply.yml 77

Abbildung 36: ingress.yml 78

Abbildung 37: JSON Request Body 84

Abbildung 38: JSON Response Body 84

Abbildung 39: Beispiel Code - Channel erstellen - JavaScript 85

Abbildung 40: Beispiel Code - Connect WebSocket Client - JavaScript 85

Abbildung 41: Beispiel Code - Channel löschen - JavaScript 85

Tabellenverzeichnis

Tabelle 1: Übersicht existierender Services 21

Tabelle 2: Abdeckung der Nutzeranforderungen 27

Tabelle 3: Priorisierung der Anforderungen 28

Tabelle 4: Loadtest Ergebnisse Prototypen 39

Tabelle 5: Übersicht verwendets Module 40

Tabelle 6: Commands/Events 49

Tabelle 7: Testfälle 53

Tabelle 8: Ergebnist loadtest API 58

Tabelle 9: Ergebnisee loadtest WS 60

Tabelle 10: Ergebnisse JMeter API 61

Tabelle 11: JMeter Ergebnisse WS Pub-Sub 62

Tabelle 12: Ergebnisse Vergleich Sigle-Node Cluster 63

Tabelle 13: Ergebnisse JMeter Florkestra 65

Tabelle 14: Umgebungsvariablen 79

Tabelle 16: Befehlszeilenparameter 80

Tabelle 17: Channel Parameter 84

Literaturverzeichnis

[1] S. F. Andreas Eberhart, *Web Services: Grundlagen und praktische Umsetzung mit J2EE und .NET*. 2003.

[2] S. R. Leonard Richardson, *RESTful Web Services*. 2008.

[3] A. S. Tanenbaum and M. Van Steen, *Verteilte Systeme - Prinzipien und Paradigmen*. 2007.

[4] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consenus Algorithm (Extended Version),” *Proc. USENIX ATC ’14*, 2014.

[5] D. Ongaro, “Consensus: Bridging Theory and Practice,” 2014.

[6] R. Fielding *et al.*, “RFC 2616: Hypertext transfer protocol–HTTP/1.1, June 1999,” *Status Stand. Track*, 1999.

[7] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2013.

[8] T. Berners-Lee, “Universal Resource Identifiers in WWW,” *Internet RFC 1630,* 1994. .

[9] R. T. Fielding, “What is REST – Learn to create timeless RESTful APIs.,” 2000. [Online]. Available: https://restfulapi.net/. [Accessed: 15-Jun-2019].

[10] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins, “RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP,” *Internet Engineering Task Force,* 2011. .

[11] W3C, “The Web Sockets API,” *W3C Working Draft,* 2009. .

[12] P. Saint-Andre, “RFC Standard 6455-- The WebSocket Protocol,” *RFC 6455 (Proposed Stand.,* 2011.

[13] Fyrd, “Can I use... Support tables for HTML5, CSS3, etc,” *Fyrd,* 2013. [Online]. Available: https://caniuse.com/#feat=websockets. [Accessed: 17-Jun-2019].

[14] “(51) Under the hood: Facebook Messenger for Firefox | Facebook.” [Online]. Available: https://www.facebook.com/notes/facebook-engineering/under-the-hood-facebook-messenger-for-firefox/10151175913223920/. [Accessed: 06-Aug-2019].

[15] “Agar.io.” [Online]. Available: https://agar.io/#ffa. [Accessed: 06-Aug-2019].

[16] SocketCluster.io, “SocketCluster/client-drivers: List of SocketCluster clients.” [Online]. Available: https://github.com/SocketCluster/client-drivers. [Accessed: 16-Jun-2019].

[17] W. MacDonald, “wmsmacdonald/websocket-relay: Easily set up communication channels between browser clients.” [Online]. Available: https://github.com/wmsmacdonald/websocket-relay. [Accessed: 17-Jun-2019].

[18] SockJS, “sockjs/sockjs-client: WebSocket emulation - Javascript client.” [Online]. Available: https://github.com/sockjs/sockjs-client. [Accessed: 16-Jun-2019].

[19] Pusher Limited, “Channels overview | Pusher docs.” [Online]. Available: https://pusher.com/docs/channels. [Accessed: 16-Jun-2019].

[20] Github User: Florkestra, “florkestra/websocket-relay-server: A server that relays messages sent amongst a collection of Web Sockets.” [Online]. Available: https://github.com/florkestra/websocket-relay-server. [Accessed: 17-Jun-2019].

[21] tjholowaychuk, gjohnson, and J. Cruger, “axon - npm.” [Online]. Available: https://www.npmjs.com/package/axon. [Accessed: 28-Jul-2019].

[22] J. Glendenning, “isobit/ws-tcp-relay: A simple relay between WebSocket clients and TCP servers.” [Online]. Available: https://github.com/isobit/ws-tcp-relay. [Accessed: 17-Jun-2019].

[23] SocketCluster.io, “SocketCluster.” [Online]. Available: https://socketcluster.io/#!/docs/introduction. [Accessed: 16-Jun-2019].

[24] Unsetbit, “unsetbit/onramp: Web Socket Server for P.” [Online]. Available: https://github.com/unsetbit/onramp. [Accessed: 17-Jun-2019].

[25] K. Sen, “relay/relay.js at master · ksen007/relay.” [Online]. Available: https://github.com/ksen007/relay/blob/master/relay.js. [Accessed: 17-Jun-2019].

[26] Adam Wiggins, “The Twelve-Factor App,” 2017. [Online]. Available: https://12factor.net/. [Accessed: 15-Apr-2019].

[27] “Artillery - a modern load testing toolkit.” [Online]. Available: https://artillery.io/. [Accessed: 09-Aug-2019].

[28] sgmonda, “stdio - npm.” [Online]. Available: https://www.npmjs.com/package/stdio. [Accessed: 28-Jul-2019].

[29] Luigi Pinca, “ws - npm.” [Online]. Available: https://www.npmjs.com/package/ws. [Accessed: 28-Jul-2019].

[30] Douglas Wilson, “express - npm.” [Online]. Available: https://www.npmjs.com/package/express. [Accessed: 28-Jul-2019].

[31] Douglas Wilson, “body-parser - npm.” [Online]. Available: https://www.npmjs.com/package/body-parser. [Accessed: 28-Jul-2019].

[32] Douglas Wilson, “cors - npm.” [Online]. Available: https://www.npmjs.com/package/cors. [Accessed: 28-Jul-2019].

[33] tjholowaychuk and B. Copeland, “axon-rpc - npm.” [Online]. Available: https://www.npmjs.com/package/axon-rpc. [Accessed: 28-Jul-2019].

[34] Vitaly Puzrin, “markdown-it - npm.” [Online]. Available: https://www.npmjs.com/package/markdown-it. [Accessed: 28-Jul-2019].

[35] Fedor Indutny, “ip - npm.” [Online]. Available: https://www.npmjs.com/package/ip. [Accessed: 28-Jul-2019].

[36] Sindre Sorhus and silverwind, “public-ip - npm.” [Online]. Available: https://www.npmjs.com/package/public-ip. [Accessed: 28-Jul-2019].

[37] defunctzombie, “url - npm.” [Online]. Available: https://www.npmjs.com/package/url. [Accessed: 28-Jul-2019].

[38] scottie1984, “swagger-ui-express - npm.” [Online]. Available: https://www.npmjs.com/package/swagger-ui-express. [Accessed: 28-Jul-2019].

[39] J. Cruger, “liferaft - npm.” [Online]. Available: https://www.npmjs.com/package/liferaft#logreplication. [Accessed: 08-Aug-2019].

[40] P. Teixeira, “skiff - npm.” [Online]. Available: https://www.npmjs.com/package/skiff.

[Accessed: 08-Aug-2019].

[41] V. Weevers, “leveldown - npm.” [Online]. Available: https://www.npmjs.com/package/leveldown. [Accessed: 08-Aug-2019].

[42] V. Weevers, “levelup - npm.” [Online]. Available: https://www.npmjs.com/package/levelup. [Accessed: 08-Aug-2019].

[43] P. Teixeira, “skiff/package.json at master · pgte/skiff.” [Online]. Available: https://github.com/pgte/skiff/blob/master/package.json. [Accessed: 08-Aug-2019].

[44] V. Weevers, “leveldown/UPGRADING.md at master · Level/leveldown.” [Online]. Available: https://github.com/Level/leveldown/blob/master/UPGRADING.md#v301. [Accessed: 08-Aug-2019].

[45] “kubectl - Kubernetes.” [Online]. Available: https://kubernetes.io/docs/reference/kubectl/kubectl/. [Accessed: 16-Aug-2019].

Anhang

Erste Schritte

Im Folgenden finden Sie Anleitungen für die Nutzung des WebSocket Relay Service. Zum einen wir aus einem administrativen Blickwinkel erläutert, wie der Service in Betrieb genommen werden kann. Hierzu gehört das Deployment auf verschiedenen Plattformen, sowie die Konfiguration.
Anschließend wird für die Nutzer des Service erklärt, wie dieser aus einer Applikation heraus angesprochen und genutzt werden kann. Die Verwendung des Service wird mit praktischen Beispielen untermalt.

Einrichtungsleitfaden

In diesem Kapitel wird beschrieben, wie der Service in verschiedenen Anwendungsumgebungen eingerichtet werden kann. Anschließend wird erklärt, welche Konfiguration der Service hat.

Manuelles Starten

Aus dem Projektverzeichnis werden die Abhängigkeiten über den Node Package Manager mittels der Datei package.json installiert:

```
$ npm install .
```

oder über die package-lock.jason mittels:

```
$ npm ci --only=production
```

Um den Service manuell in der Konsole auszuführen starten Sie die app.js Datei im Pfad src/app.js mit NodeJS.

```
$ node src/app.js
```

Um die Konfiguration des Service über die Umgebungsvariablen zu ändern können diese direkt über die Konsole gesetzt werden.

```
$ $env:WRS_PUBLIC_API_PORT ="8080" ; node src/app.js
```

Deployment in Docker

Das Projekt enthält eine Dockerfile Datei, die beschreibt wie der Service in einem Dockercontainer verpackt wird.Nun muss ein Docker-Image gebaut werden, das die

```
FROM node:8

RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

COPY package*.json ./
RUN npm ci --only=production

ENV WRS_API_PORT=80 \
    WRS_PUBLIC_API_PORT=443 \
    WRS_SSL=false \
    WRS_PUBLIC_SSL=true \
    WRS_AUTH=false \
    WRS_AUTH_TOKEN=WE_LOVE_MYLAB \
    WRS_DOMAIN=api.ba-pump.mylab.th-luebeck.de \
    WRS_AUTH_RESOURCES=*:/cluster/* \
    WRS_SSL_CERT_PATH=./ssl/wrs.example.com.cert \
    WRS_SSL_CERT_PATH=./ssl/wrs.example.com.de.key \
    WRS_BROWSER_ONLY=true \
    […]
# Bundle app source
COPY . .

EXPOSE 80

CMD [ "node", "src/app.js" ]
```

Abbildung 34: Dockerfile

Applikation enthält. Führen Sie hierzu den Konsolenbefehl im selben Verzeichnis aus, in dem die Dockerfile Datei liegt (Root-Verzeichnis des Prjekts).

```
$ docker build --tag <yourname>/wrs .
```

Anschließend wird der Service in einem Docker-Container gestartet.

```
$ docker run --expose <apiport> --expose <nodeport> --name wrs <yourname>/wrs
```

Um die SSL Verschlüsselung zu aktivieren müssen dem Service die SSL Dateien übergeben werden. Diese liegen in einem Verzeichnis auf dem Host und werden dem Container in einem Docker-Volume zu Verfügung gestellt. Dadurch sind weitere Befehlszeilen Parameter erforderlich:

```
[…] --volume <localpath>:/ssl
    --env WRS_SSL=true
    --env WRS_SSL_CERT_PATH=/ssl/{filename}.cert
    --env WRS_SSL_KEY_PATH=/ssl/{filename}.key
```

Um den Service in einem Docker-Cluster zu starten wird die compose.yml Datei verwendet, die ebenfalls im Root-Verzeichnis des Projekts liegt.

```
$ docker swarm init
$ docker stack deploy -c docker-compose.yml wrs-service
```

Deployment in Kubernetes

Für das Deployment in einem Kubernetes Cluster wird für diese Anleitung das Befehlszeilen Tool kubectl [45] verwendet. Alle hier beschriebenen Operationen könne jedoch auch direkt über die Kubernetes API vorgenommen werden. Das Docker Image wird vor dem Deployment entweder über Docker gebaut oder direkt aus der Registry des Projekts importiert.

Der Deploy kann dann über den kubectl Befehl run gestartet werden. Default Konfigurationseinstellungen sind an diesem Punkt im Image enthalten.

```
$ kubectl create deployment wrs --image=cedricpump/wrs:v1.0
```

Anschließend müssen die Ports für die API und die Clusterkommunikation exportiert werden.

```
$ kubectl expose deployment wrs --port=4000
$ kubectl expose deployment wrs --port=80 --type=NodePort
```

Der Deploy lässt sich dann entsprechend skalieren.

```
$ kubectl scale deployment worker --replicas=3
```

Für fortgeschrittene Konfigurationsmöglichkeiten kann der Service über eine deploy.yml Datei deployed werden. Gegebenenfalls müssen einige Parameter an das vorhandene Kubernetes Umfeld angepasst oder ergänzt werden.

```
$ kubeclt apply -f deploy.yml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wrs-deploy
lables:
  app: wrs
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wrs
      template:
        metadata:
          lables:
            app: wrs
  spec:
    containers:
    - name: wrs
      image: cedricpump/wrs:v1.0
      ports:
      - containerPort: 80
      - containerPort: 4000
      env:
      - name: WRS_API_PORT
        value: 80
      - name: WRS_NODE_PORT
        value: 4000
      - name: WRS_DOMAIN
        value: wrs.example.com
```

Abbildung 35: deoply.yml

Die SSL Verschlüsselung erfolgt in Kubernetes Umgebungen typischerweise über den Ingress Loadbalancer. Ist sichergestellt, dass das verwendete Kubernetes Cluster für Ingress konfiguriert ist, kann der Ingress knoten mit der ingress.yml gestartet werden.

```
$ kubeclt apply -f ingress.yml
```

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: wrs
spec:
  rules:
  - host: wrs.example.com
    http:
      paths:
      - backend:
          serviceName: wrs
          servicePort: 80
  tls:
  - secretName: wrs-example-com-cert
    hosts:
    - wrs.example.com
```

Abbildung 36: ingress.yml

Ist das SSL/TLS Zertifikat nicht im Kubernetes Cluster hinterlegt, kann es über den kubctl Befehl hinzugefügt werden:

```
$ kubectl create secret tls wrs-ingress-tls-secret --key /ssl/{filename}.key --cert
  /ssl/{filename}.cert
```

Konfiguration

Der WebSocket Relay Service kann an verschiedene Anwendungsfälle angepasst werden. Die Konfigurationsmöglichkeiten des Service umfassen:

- das Übergeben eines SSL Zertifikats zur Verschlüsselung der Verbindung
- das Festlegen der Ports für REST-API und Cluster Kommunikation
- das Absichern des Zugriffs auf spezielle Ressourcen mit Authetifikationstoken
- das Ein- und Ausschalten des Browser-Only-Mode
- das Definieren eines DNS für die automatische Clusterbildung
- das Definieren der Läng von generierten Tokens und IDs

Umgebungsvariablen

Der Service nutzt Umgebungsvariablen, um die Konfiguration vorzunehmen. **Tabelle 14**: Umgebungsvariablen

zeigt alle Umgebungsvariablen und ihre Funktion.

Umgebungsvariable	Typ	Beschreibung	Default
WRS_API_PORT	{Number}	Port für die REST-API.	80 bzw. 443
WRS_PUBLIC_API_PORT	{Number}	Öffentlicher API Port durch Loadbalancer oder Proxys	WRS_API_PORT
WRS_TOKEN_LENGTH	{Number}	Länge des Channel-Tokens.	16
WRS_DOMAIN	{String}	IP-Adresse oder Domain, die der Service nutzt.	"127.0.0.1"
WRS_AUTH	{Boolean}	"true" damit Ressourcen mit Token gesichert werden	false
WRS_AUTH_TOKEN	{String}	Authentificationstoken für die API	
WRS_AUTH_RESSOURCES	{String}	Liste der Ressourcen, die geschützt werden	
WRS_SSL	{Boolean}	"true" wenn explizit SSL genutzt werden soll	false
WRS_PUBLIC_SSL	{Boolean}	"true" wen implizit SSL genutzt wird	WRS_SSL
WRS_SSL_CERT_PATH	{String}	Pfad zu SSL Zertifikat	
WRS_SSL_KEY_PATH	{String}	Pfad zu SSL Key	
WRS_BROWSER_ONLY	{Boolean}	Aktiviert Browserser-Only-Support	false
WRS_README_PATH	{String}	Pfad zu README Datei	"./README_API.md"
WRS_NODE_PORT	{Number}	Port für Raft Node Kommunikation	4000
WRS_NODE_HEARTBEAT	{Number}	Raft Node Heartbeat Interval in ms	300
WRS_NODE_MIN_ELECTION_TIMEOUT	{Number}	Raft Node minimalem Election Timeout in ms	2000
WRS_NODE_MAX_ELECTION_TIMEOUT	{Number}	Raft Node maximalem Election Timeout in ms	3000
WRS_NODE_SNAPSHOT_LENGHT	{Number}		10000
WRS_MAX STARTUP_DELAY	{Number}	Max. zufällige Start-Verzögerung Autoconnect	6000
WRS_ID_LENGTH	{Number}	Länge der zufällig generierten IDs	4
WRS_DEBUG	{Boolean}	Debug flag	false

Tabelle 14: Umgebungsvariablen

Technische Parameter zur Nutzung des Service stellen die Ports der API und des Cluster Knotens dar. Diese lassen sich über die Umgebungsvariablen WRS_API_PORT bzw. WRS_NODE_PORT festlegen. Zudem lässt sich die Länge von generierten Channel Token und IDs festlegen. Der Cluster Knoten verfügt über fortgeschrittene Konfigurationsmöglichkeiten, auf die hier nicht weiter eingegangen wird. Es wird empfohlen die voreingestellten Werte beizubehalten, um fehlerfreies Clusterverhalten zu gewährleisten.

Auf die Konfiguration der weitern Parameter wird im Folgenden genauer eingegengen.

Befehlszeilenparameter

Häufig verwendete Konfigurationsvariablen können über Befehlszeilenparameter gesetzt werden und überschreiben somit die Werte der Umgebungsvariablen. Folgende Befehlszeilenparameter stehen zur Verfügung:

key	**Parameter**	**Beschreibung**	**Entsprechende Umgebungsvariable**
-p	--portAPI	Port für die REST-API	WRS_API_PORT
-n	--portNode	Port für Raft Node Kommunikation	WRS_NODE_PORT
-a	--auth	"true" wenn Authtoken verwendet werden sollen	WRS_AUTH
-t	--token	Authentifikationstoken für die API	WRS_AUTH_TOKEN
-s	--ssl	"true" wenn SSL genutzt werden soll	WRS_SSL

Tabelle 15: Befehlszeilenparameter

Die Befehlszeilenparameter sind für schnelle Deployments zu Testzwecken über die Konsole gedacht. SSL Zertifikate werden weiterhin über die Umgebungsvariablen festgelegt, doch die Nutzung von SSL kann so zwischen Deploys gewechselt werden.

```
$ node src/app.js --ssl false -p 80 -n 5000
$ node src/app.js --ssl false -p 8080 -n 4000
```

Diese Befehle starten zwei WRS Server mit unterschiedlichen Ports, aber derselben Konfiguration über Umgebungsvariablen.

Authentifikation

Die API des WebSocket Relay Service kann mittels Authentifikationstoken abgesichert werden. Dabei kann entweder die gesamte API oder einzelne Ressourcen gesichert werden. Die Umgebungsvariable WRS_AUTH bzw. der Kommandozeilenparameter --auth schaltet die Authentifizierung global ein oder aus. Über die Umgebungsvariable WRS_AUTH_TOKEN kann der eigentliche Token übergeben werden.
Die Umgebungsvariable WRS_AUTH_RESSOURCES ermöglicht das Sichern einzelner Ressourcen komplett oder teilweise. Der Variable wird eine leerzeichen-separierte Liste übergehen, die die zu sperrenden Ressourcen und Methoden angibt. Jeder Eintrag ist formatiert nach dem Muster: Methode + ":" + Pfad. Die Methode kann jede HTTP Request Methode sein ("GET", "POST", "DELETE", …) oder "*" um alle Methoden zu sichern. Der Pfad wird beschrieben durch einen regulären Ausdruck. Ist die Umgebungsvariable WRS_AUTH_RESSOURCES leer oder undefiniert, so wird die API global mit Token gesichert. Dies entspricht dem Eintrag "*:*" Im Folgenden wird die Verwendung des Parameters anhand einiger Beispiele Verdeutlicht:

```
WRS_AUTH_RESOURCES=POST:/channels DELETE:/channels
```

sichert alle POST und DELETE Requests auf die Ressource /channels.

```
WRS_AUTH_RESOURCES=*:/cluster */reset
```

sichert alle Requests auf die Resourcen /cluster und /reset.

```
WRS_AUTH_RESOURCES=POST:/cluster/join\w+
```

sichert die manuelle Clusterverbindung.

```
WRS_AUTH_RESOURCES=GET:/channel/w+/connect
```

Sichert alle WebSocket Verbindungsanfragen.

Verbinden zu einem Cluster

Die Verbindung eines neu gestarteten Knotens zu einem bestehenden Cluster kann automatisch erfolgen. Hierzu wird dem Service über die Umgebungsvariable WRS_DOMAIN die Domain oder IP-Adresse und über WRS_PUBLIC_API_PORT der Port übergeben, unter der der Service erreichbar sein wird. Beim Start ermittelt der Knoten dann per HTTP Request ob bereits ein Cluster besteht und tritt diesem automatisch bei.

Sollte dieser Mechanismus fehlschlagen, kann die Verbindung zu einem Cluster manuell initiiert werden. Mittels Request an:

```
POST /cluster/join/{address}
```

wird dem Service die Adresse des Cluster-Leaders (IP-Address:Port) übergeben, zu dem der Knoten einen Verbindungsversuch sendet. Die gegenwärtige Clusterkonfiguration lässt sich ausgeben mit einem GET Request:

```
GET /cluster
```

Trennen vom Cluster

Bei regulärem Beenden eines Knoten-Prozesses wird der Knoten automatisch vom Cluster getrennt. Sollte ein Knoten jedoch abstürzen, kann es nötig sein den Knoten manuell aus dem Cluster zu entfernen, da der Cluster Algorithmus nicht zwischen abgestürzten und zeitweise nicht erreichbaren Knoten unterscheiden kann. Das manuelle Löschen eines Knotens aus einem Cluster erfolgt über:

```
DELETE /cluster/{id}
```

Die NodeID des Knotens wird dabei im Pfad übergeben.

SSL Verschlüsselung

Der Service ist in der Lage sowohl die HTTP als auch die WebSocket Kommunikation zu verschlüsseln. Die SSL Verschlüsselung wird über die Umgebungsvariable WRS_SSL oder die Kommandozeilenparameter --ssl aktiviert. Das benötigte Zertifikat, sowie der Schlüssel, können über die Umgebungsvariablen WRS_SSL_CERT_PATH bzw. WRS_SSL_KEY_PATH als Pfad zur jeweiligen Datei übergeben werden.

Browser-Only Unterstützung

Zu Test- oder Demonstrationszwecken kann es hilfreich sein den Service aus dem Browser heraus ansprechen zu können. Dazu verfügt der Service über eine Browser-Only Unterstützung der alternative GET Requests bereitstellt, um die Verwendung von POST und DELETE Methoden zu umgehen. Es sei angemerkt, dass die API unter Verwendung dieses Modus nicht mehr REST-konform ist. Der Service ist damit vollständig, jedoch nicht ausschließlich mit dem Browser ansprechbar. Der Browser-Only-Support lässt sich über die Umgebungsvariable WRS_BROWSER_ONLY aktivieren.

Nutzungsleitfaden

Dieser Service ermöglicht es Anwendern Nachrichten über Kanäle (Channels) auszutauschen. Diese Channels lassen sich über eine REST-API erstellen und löschen. Für den Nachrichtenaustausch wird das WebSocket-Protokoll verwendet. Der Service unterstützt für

den Nachrichtenaustausch die Messaging-Pattern Echo, Publish-Subscribe, Push-Pull und Publish-Subscribe (siehe Kapitel 0).

Für die Verwendung dieses Service ist es notwendig HTTP Anfragen zu senden und dessen Antworten auszuwerten, sowie WebSocket Verbindungen mittels eines WebSocket Clients aufzubauen.

Die Daten, die innerhalb der HTTP Antworten übergeben werden, sind in JSON formatiert.

Ein Channel ist gesichert mit einem Authentifizierungstoken (*token*). Dieses wird beim Erstellen des Channels generiert und muss bei der Verbindung übergeben werden. Zudem hat ein Channel drei weitere Eigenschaften: das Messagingpattern (*pattern*), das die Form der Nachrichtenverteilung bestimmt, ein Clientlimit (*clientLimit*), das die maximale Anzahl gleichzeitig verbundener Clients begrenzt, und das Timeout (*timeout*), das dem Channel nach einer Zeit der Inaktivität löscht.

Der Arbeitsablauf mit diesem Service unterteilt sich in vier grundlegende Abschnitte:

- Erstellen des Channels
- Verbinden zu Channel
- Senden / Empfangen von Nachrichten
- Löschen des Channels

Etwas feiner unterteilt ergeben sich die Schritte:

- Channel erstellen mittels HTTP POST Request mit gewünschten Parametern
- Speichern der Felder *id* und *token*
- Teilen der ID und des Tokens mit anderen Clients
- Erstellen eines WebSocket Clients und Verbinden zu gewünschtem Channel
- Senden und Empfangen von Nachrichten mittels WebSockets
- WebSocket Verbindung trennen
- Channel löschen mittels HTTP DELETE Request oder warten auf Timeout

Im Folgenden werden diese Schritte erklärt und anhand von Beispielen verdeutlicht. Für die Beispiele wird in diesem Fall auf die Scriptsprache JavaScript zurückgegriffen, der Service kann jedoch in jeder Programmiersprach angesprochen werden, die HTTP Anfragen und WebSocket Clients unterstützt. Für den Beispielcode wird ferner angenommen ,dass ein WebSocket Relay Service unter der Domain "wrs.net" auf Port 80 läuft.

Channel erstellen

Um einen Channel zu erstellen wird ein HTTP POST Request an den Pfad "/channel" des Service gesendet. Diesem Request werden die Parameter zum Erzeugen des Channels als JSON Object im Request Body übergeben. Das Feld *pattern* gibt hierbei das Messagingpattern an. *clientLimit* entspricht der Anzahl der maximalen Clients und *timeout* entspricht dem gewünschten Timeout in Millisekunden. Die Felder *clientLimit* und *timeout* sind hier optional.

```
{
    "pattern":      "pub-sub"
    "clientLimit":  10
    "timeout":      3600000
}
```

Abbildung 37: JSON Request Body

Name	Typ	Bereich	Default
pattern	String	["echo", "pub-sub", "push-pull", "req-repl", "fan-in-out"]	keine
clientLimit	Number	0 <= clientLimit <= 2^{53} -1 = kein Limit	-1
timeout	Number	0 <= timeout <= 2^{53} -1 = kein Timeout	86400000

Tabelle 16: Channel Parameter

Wenn die Anfrage erfolgreich ist, wird eine HTTP Response mit Status Code: 201 (Created) zurückgegeben.
Der Response Body enthält dann ein JSON Object, mit den Felder, *message*, *token* und *data*. Das token Feld enthält das Authentifizierungstoken. Das data Feld enthält Informationen zu dem erstellten Channel. Für die Verwendung des Channels ist es in jedem Fall notwendig den Token, sowie die Channel-ID zu speichern.

```
{
    "message": "Success […]",
    "token": "85ee97e8",
    "data": {
        "id": "D679-05F5",
        "port": 80,
        "pattern": "echo",
        "timeout": 200,
        "countdown": 198,
        "clientLimit": -1,
        "clients": 0
    }
}
```

Abbildung 38: JSON Response Body

```javascript
var id = "";
var token = "";
var xhr = new XMLHttpRequest();
// create POST Request
xhr.open("POST", `http://wrs.example.com:80/channels`, true);
// set Headers
xhr.setRequestHeader("Content-Type", "application/json");
let body = JSON.stringify({
    "pattern": "pub-sub" ,
    "timeout": 3600000,
    "clientLimit": -1
});
// save Response
xhr.onreadystatechange = function()  {
    if (this.readyState !== 4) return;
    let data = JSON.parse(this.responseText);
    id = data.data.id;
    token = data.token;
};
// send
xhr.send(body);
```

Abbildung 39: Beispiel Code - Channel erstellen - JavaScript

Client verbinden

Um den Client mit dem Server zu verbinden wird ebenfalls ein HTTP Request an die API des Service gesendet. Dies übernimmt der WebSocket Client, der einen HTTP GET Request verwendet, um mit entsprechenden Upgrade Headern ein Upgrade zum WebSocket Protokoll einzuleiten.
Die REST-API nimmt diese unter dem Pfad "/channel/id/connect" entgegen und leitet diese an den WebSocket Server weiter. Da viele WebSocket Clients keine Benutzerdefinierten Header und Bodys unterstützen, wird der Token als Quary Parameter übergeben.

```
"ws://wrs.example.com/channel/{id}/connect?token={token}"
```

Die URL, die dem WebSocket Client zum Verbinden zu einem Channels mit der ID "ABC" und dem Token "DEF" übergeben wird, ist somit:

```
"ws://wrs.example.com/channel/ABC/connect?token=DEF"
```

Folgender Beispielcode beschreibt den Verbindungsaufbau in JavaScript zu dem zuvor erstellten Channel.

```
1.  let ip = `ws:// wrs.example.com:80/channels/${id}/connect?token=${token}`;
2.  WebSocket = new WebSocket(ip);
3.  // send messages with button
4.  WebSocket.onopen = function () {
5.      myButton.onclick = function () {
6.          let msg = "some text";
7.          WebSocket.send(`${msg}`);
8.      };
9.  };
10. // print incoming messages
11. WebSocket.onmessage = function (event) {
12.     console.log(`${event.data}`);
13. };
14. WebSocket.onclose = function () {};
```

Abbildung 40: Beispiel Code - Connect WebSocket Client - JavaScript

Dieser Programmausschnitt erzeugt einen neuen WebSocket Client und verbindet diesen mit dem WebSocket Relay Service. Wird ein Button geklickt, so wird eine Nachricht gesendet. Eingehende Nachrichten werden in der Konsole ausgegeben.

Channel löschen

Um einen Channel zu löschen wird ein HTTP DELETE Request an die API gesendet, indem der Channel über seine ID identifiziert wird. Der Pfad ist somit "/channel/{*id*}". Ist das Löschen

```
1.  var xhr = new XMLHttpRequest();
2.  // create DELETE Request
3.  xhr.open("DELETE", `http://wrs.example.com:80/channels/${id}`, true);
4.  xhr.setRequestHeader("Content-Type", "application/json");
5.  // send
6.  xhr.send();
```

Abbildung 41: Beispiel Code - Channel löschen - JavaScript

des Channels erfolgreich, so antwortet die API mit dem Statuscode: 200 (Success).

Dieser Codeausschnitt zeigt, wie ein DELETE Request gesendet werden kann. Die Antwort wird hier vernachlässigt.

SSL-Verschlüsselung

Nutzt der WebSocket Relay Service SSL Verschlüsselung, so ist für HTTP Request und WebSocket Verbindungen zu beachten, in der URL das korrekte Protokoll anzugeben: "https://" bzw. "wss://".

Authentifizierung

Der WebSocket Relay Service unterstützt eine Token Authentifikation für den Zugriff der API. Einzelne Ressourcen oder die gesamte API können somit, je nach Konfiguration des Services, die Authentifikation mittels eines Tokens erfordern. Diese ist nicht zu verwechseln mit dem individuellen Token, der den Verbindungzugriff der einzelnen Channels schützt. Der Authentifizierungstoken kann auf verschiedenen Wegen übergeben werden.

Der Token kann als *auth* Parameter direkt im URL Querry übergeben werden:

```
GET "/channel/{id}?auth={token}"
```

Der Token kann dem Request als Authentifikation Header hinzugefügt werden:

```
"authentification": "Bearer {token}"
```

Der Token kann als *auth* Datenfeld im JSON Object des Request Bodys übergeben werden:

```
{"auth": {token}}
```

Browser-Only Unterstützung

Um mit den Funktionalitäten des Service warm zu werden, ist es hilfreich diese testen zu können, ohne weitere Tools installieren zu müssen. Ist die Browser-Only Unterstützung aktiviert, so lassen sich Channel auch mit HTTP GET Requests aus dem Browser heraus erstellen und löschen. Dieses Feature verletzt jedoch die Grundsätze einer RESTfull API und sollte daher ausschließlich für erste Versuche und zu Demonstrationszwecken verwendet werden.

Channel können mittels GET Request über den Pfad "/channel/create" erstellt werden. Die Konfiguration des Channels wird hier nicht im Request Body als JSON Object, sondern als URL Querry Parameter übergeben. Die Eigenschaften und die Verwendung der Parameter unterscheiden sich jedoch nicht von denen, die beim POST Request Verwendung finden.

```
GET "/channel/create?pattern={pattern}&clientlimit={clientlimit}&timeout={timeout}"
```

Das Löschen eines Channels mittels GET Request geschieht über den Pfad "/channels/{id}/delete". Hier sind keine Parameter erforderlich.

```
GET "/channel/{id}/delete"
```

Test Client

Um den WebSocket Relay Service zu testen oder sich initial mit den Funktionen vertraut zu machen ist es hilfreich einen WebSocket Client zu haben, um sich mit dem Service zu verbinden. Daher liefert der Service einen einfachen Client, der über /client als HTML Dokument ausgeliefert wird.

Erläuterung der Anforderungen

[FR1] POST Request zum Anlegen eines Channels

Das Anlegen eines Channels soll über die REST-API mittels HTTP POST Request erfolgen. Beim Erstellen des Channels sollten gewünschte Eigenschaften des Channels als Parameter übergeben werden können. Nach dem Erstellen des Channels erhält der Nutzer eine HTTP Response, die über Erfolg des Vorgangs informiert, sowie Informationen zu dem neu erstellten Channel liefert.

[FR2] DELETE Request zum Löschen eines Channels

Das Löschen eines Channels soll über die REST-API mittels HTTP DELETE Request erfolgen. Nach dem Löschen des Channels erhält der Nutzer eine HTTP Response, die über Erfolg des Vorgangs informiert.

[FR3] Abonnieren des Channels durch WebSocket Verbindung

Die WebSocket Verbindung soll den Client an den Channel und dessen Nachrichtenaustausch binden. Ein verbundener Client hat diesen Channel abonniert und erhält alle Nachrichten entsprechend dem gewählten Messaging-Pattern.

[FR4] Verbinden mit einem Channel über das WebSocket-Protokoll

Der Client verbindet sich mit dem Channel über das WebSocket-Protokoll. Den HTTP GET Request, der den Verbindungsaufbau imitiert, nimmt die REST-API entgegen.

[FR5] Weiterleitung der Nachrichten gemäß Pub./Sub. Messaging-Pattern

Der Service soll eine Weiterleitung der Nachrichten nach dem Publish-Subscribe Messaging-Pattern unterstützen.

[FR6] Weiterleitung der Nachrichten gemäß Push/Pull Messaging-Pattern

Der Service soll eine Weiterleitung der Nachrichten nach dem Push-Pull Messaging-Pattern unterstützen.

[FR7] Weiterleitung der Nachrichten gemäß Req./Repl. Messaging-Pattern

Der Service soll eine Weiterleitung der Nachrichten nach dem Request-Reply Messaging-Pattern unterstützen.

[NFR1] Der Service soll mit wenigen Abhängigkeiten auskommen

Der Service soll clientseitig allein mit dem HTTP und mit dem WebSocket Protokoll verwendet werden. Es sind keine komplexen Konto- und Verifikationsmechanismen gefordert.

[NFR2] Der Service soll wartbar sein

Der Service soll modular aufgebaut sein, sodass fehlerhafte Komponenten ausgetauscht werden können.

[NFR3] Das API soll REST-konform sein

Die API des Service soll REST-konform sein.

[NFR4] Der Service soll als standardisiertes Container Image bereitgestellt werden

Der Service soll als standardisierte Deploymentunit in Form eines OCI/Docker-konformen Containers bereitgestellt werden, sodass der Service in ein Kubernetes Cluster oder andere Docker-basierten Serverumgebungen eingebunden werden kann.

[NFR5] Der Service soll die „12-Factor App Prinzipien“ berücksichtigen

Um Portierbarkeit und Elastizität zu gewährleisten sind die „12-Factor App Prinzipien zu beachten.

[NFR6] Das REST-API soll SSL-Verschlüsselung unterstützen

Die HTTP Kommunikation zwischen Client und Server soll mittels SSL-Verschlüsselung geschützt werden. Der Service wird hierzu das SSL-Zertifikat zur Verfügung gestellt.

[NFR7] Die WebSocket Kommunikation soll SSL-Verschlüsselung unterstützen

Die WebSocket Kommunikation zwischen Client und Server soll mittels SSL-Verschlüsselung geschützt werden. Der Service wird hierzu das SSL-Zertifikat zur Verfügung gestellt.

[NFR8] Die WebSocket Verbindung soll mit einem Sicherheitstoken authentifiziert werden

Jeder Channel soll mit einem Token gesichert sein, den ein Client beim Verbindungsaufbau der WebSocket Verbindung übergeben muss, um sich zu authentifizieren. Dieses Token wird beim Erstellen des Channels zufällig generiert und dem Client mit der Bestätigung des Erstellens mitgeteilt. Der Token kann anschließend nicht mehr eingesehen werden. Für die Verbreitung des Tokens an weitere Clients ist der Anwender zuständig.

[NFR9] Ein Channel soll mit einem Timeout versehen werden können

Der Channel kann mit einem Timeout versehen werden. Das Timeout wird mit jeder Aktivität in einem Channel (also jeder gesendeten Nachricht) zurückgesetzt. Nach Ablauf des Timeouts wird der Channel gelöscht und alle offenen Verbindungen zu Clients werden beendet.

[NFR10] Die Clientzahl eines Channels soll begrenzt werden können

Die Anzahl der Clients, die sich gleichzeitig mit einem Channel verbinden können soll begrenzt werden. Jeder weitere Verbindungsversuch, der die maximale Anzahl der Clients überschreiten würde, wird abgehlehnt.

[NFR11] Der Service soll horizontal skalierbar gestaltet werden

Der Service soll in der Lage sein als Cluster mehrerer Knoten angelegt zu werden. Unter Last sollen weitere Knoten hinzugefügt oder entfernt werden können. Ein konsistenter Zustand aller Knoten soll angestrebt werden.

Eine automatische Erkennung großer Lastzustände und eine automatische Skalierung ist nicht gefordert.

Mocha & Chai Testberichte

Testumgebung:

Von Testumgebung gestarteter WebSocket Relay Service mit Default Parametern (Auth = false, SSL= false) mit lokaler Adresse.

TF: 1.2.5	
Systemanforderung:	[FR1] POST Request zum Anlegen eines Channels
Aussage: POST Request an /channels gibt neu angelegten Channel als JSON zurück	
Testmethode: Sendet HTTP POST Request an Service API und wertet Response Header sowie Body aus Sendet HTTP GET Request und prüft dass Channel existiert	

TF: 1.2.7	
Systemanforderung:	[FR2] DELETE Request zum Löschen eines Channels
Aussage: Delete Request an /channels löscht Channel	
Testmethode: Sendet HTTP DELETE Request an Service API und wertet Response Code sowie Body aus Sendet HTTP GET Request und prüft dass Channel nicht existiert	

TF: 1.3.1	
Systemanforderung:	[FR3] Abonnieren des Channels durch WebSocket Verbindung [FR4] Verbinden mit einem Channel über das WebSocket-Protokoll
Aussage: über WebSocket gesendete Nachrichten werden empfangen (echo Channel)	
Testmethode: Erstellen eines echo Channels über HTTP POST Request Verbindungsaufbau zu Channel über WebSocket Client Senden und Empfangen einer Nachricht über WebSocket Client	

TF: 1.3.2	
Systemanforderung:	[FR5] Weiterleitung der Nachrichten gemäß Pub./Sub. Messaging-Pattern
Aussage: über WebSocket gesendete Nachrichten werden empfangen (pub-sub Channel)	
Testmethode: Erstellen eines pub-sub Channels über HTTP POST Request Verbindungsaufbau zu Channel von zwei Clients über WebSocket Client Senden und Empfangen von Nachrichten über WebSocket Client zwischen beiden Clients	

TF: 1.3.3	
Systemanforderung:	[FR6] Weiterleitung der Nachrichten gemäß Push/Pull Messaging-Pattern
Aussage: über WebSocket gesendete Nachrichten werden empfangen (push-pull Channel)	
Testmethode: Erstellen eines push-pull Channels über HTTP POST Request Verbindungsaufbau zu Channel von drei Clients über WebSocket Client Senden von Nachrichten über WebSocket Client Auswertung der Gleichverteilung der empfangenen Nachrichten	

TF: 1.3.4	
Systemanforderung:	[FR7] Weiterleitung der Nachrichten gemäß Req./Repl. Messaging-Pattern
Aussage: über WebSocket gesendete Nachrichten werden empfangen (req-repl Channel)	
Testmethode: Erstellen eines req-repl Channels über HTTP POST Request Verbindungsaufbau zu Channel von zwei Clients über WebSocket Client Senden von Request über WebSocket Client und Reply von zweitem Client Auswertung der Empfangenen Reply Nachricht	

TF: 1.3.5	
Systemanforderung:	[NFR8] Die WebSocketverbindung soll mit einem Sicherheitstoken authentifiziert werden
Aussage: Die WebSocket Verbindung schlägt fehl mit falschem Token	
Testmethode: Erstellen eines echo Channels über HTTP POST Request Verbindungsaufbau zu Channel über WebSocket Client mit falschem Token Auswertung der Fehlermeldung	
Testumgebung:	

Von Testumgebung gestarteter WebSocket Relay Service mit Default Parametern mit lokaler Adresse.

Testumgebung:

Von Testumgebung gestarteter WebSocket Relay Service mit aktivierter SSL-Verschlüsselung und selbstsignierendem Zertifikat mit lokaler Adresse.

TF: 2.1.1	
Systemanforderung:	[NFR6] Das REST-API soll SSL-Verschlüsselung unterstützen
Aussage: Die Rest-API ist SSL-Verschlüsselt	
Testmethode: HTTP GET Request an /channels unter Verwendung von "https" Auswertung des Response Status Code	

TF: 2.2.1	
Systemanforderung:	[NFR7] Die WebSocket Kommunikation soll SSL-Verschlüsselung unterstützen
Aussage: Die WebSocket Verbindung ist SSL-Verschlüsselt	
Testmethode: Erstellen eines echo Channels über HTTP POST Request Verbindungsaufbau zu Channel über WebSocket Client Verwendung von "wss" Senden und Empfangen einer Nachricht über WebSocket Client	

TF: 1.4.1, 1.4.2	
Systemanforderung:	[NFR9] Ein Channel soll mit einem Timeout versehen werden können
Aussage: Ein Channel soll sich nach Ablauf des Timeouts selbst löschen	
Testmethode: Erstellen eines echo Channels über HTTP POST Request mit definiertem Timeout Sendet GET Request vor Ablauf des Timeouts und prüft dass Channel existiert Sendet GET Request nach Ablauf des Timeouts und prüft dass Channel nicht existiert	
Aussage: Der Timeout soll durch das Senden von Nachrichten zurückgesetzt werden	
Testmethode: Erstellen eines echo Channels über HTTP POST Request mit definiertem Timeout Verbindet Client und sendet Nachricht nach definiertem Offset (< Timeout) Sendet GET Request nach Ablauf des Timeouts und prüft dass Channel existiert Sendet GET Request nach Ablauf des Timeouts + Offset und prüft dass Channel nicht existiert	
Testumgebung: Von Testumgebung gestarteter WebSocket Relay Service mit Default Parametern mit lokaler Adresse.	

TF: 1.5.1	
Systemanforderung:	[NFR10] Die Clientzahl eines Channels soll begrenzt werden können
Aussage: Es sollen nicht mehr Clientverbindungen als Clientlimit akzeptiert werden	
Testmethode: Erstellen eines echo Channels über HTTP POST Request mit definiertem Clientlimit Verbindungsaufbau zu Channel über WebSocket Client bis Clientlimit überschritten Auswertung der Fehlermeldung	
Testumgebung: Von Testumgebung gestarteter WebSocket Relay Service mit Default Parametern mit lokaler Adresse.	

TF: 5.1.1 – 5.3.1	
Systemanforderung:	[NFR11] Der Service soll horizontal skalierbar gestaltet werden
Aussage: Service Knoten sollen sich zu Cluster verbinden lassen.	
Testmethode: Erstellen eines Knotens als Kindprozess des Testprozesses Starten des Service und Testen dass verbindungsaufbau manuell und automatisch funktioniert Knotenzahl im Cluster prüfen	
Aussage: Channels werden zwischen Knoten repliziert	
Testmethode: Erstellen eines Knotens als Kindprozess des Testprozesses Starten des Service und Verbindungsaufbau zum Cluster Channel auf erstem Knoten anlegen Prüfen ob Knoten auf zweitem Knoten repliziert wurde	
Aussage: Messages werden zwischen Knoten repliziert	
Testmethode: Erstellen eines Knotens als Kindprozess des Testprozesses Starten des Service und Verbindungsaufbau zum Cluster je einen WebSocket Client zu jedem Servern verbinden Nachrichten senden und empfangen über beide Knoten zwischen den Clients	
Testumgebung: Von Testumgebung gestarteter WebSocket Relay Service mit Default Parametern mit lokaler Adresse. Weiterer Knoten wird über Kindprozess gestartet.	