[[1]](#footnote-1)

Redesign einer Chat-Anwendung als verteiltes System.

C. Eidelloth, D. Sautter, F. Stützinger und M. Auch, Team <Winner>

*Abstract*—These instructions give you guidelines for preparing papers for IEEE TRANSACTIONS and JOURNALS*.* Use this document as a template if you are using Microsoft *Word* 6.0 or later. Otherwise, use this document as an instruction set. The electronic file of your paper will be formatted further at IEEE. Define all symbols used in the abstract. Do not cite references in the abstract. Do not delete the blank line immediately above the abstract; it sets the footnote at the bottom of this column.

*Index Terms*—About four key words or phrases in alphabetical order, separated by commas. For a list of suggested keywords, send a blank e-mail to [keywords@ieee.org](mailto:keywords@ieee.org) or visit the IEEE web site at [http://www.ieee.org/web/developers/webthes/index.htm](http://www.ieee.org/organizations/pubs/ani_prod/keywrd98.txt).

# Motivation und Problemstellung

Vernetzte Rechnersysteme haben sich in letzter Zeit rasch entwickelt, dazu zählen auch die Verteilten Systeme, welche sich aus verschiedenen unabhängigen Bestandteilen zusammensetzen um ein Vollständiges System zu bilden.[1]

In dieser Arbeit soll der Umbau eines Vorhandenen nachrichtenbasierten Java Programms in ein Verteiltes System realisiert werden. Dabei gilt es im Rahmen der Aufgabenstellung, neue und bereits erlernte Technologien zu nutzen um eine Verteilte Anwendung mit dem Fokus auf Skalierbarkeit, Transaktionssicherheit und Performance zu entwickeln.

Im Zuge einer Umsetzung ergeben sich eine Reihe von Komplikationen aufgrund der gestiegenen Anforderungen, welche auf die Integration bzw. Kopplung von Wildfly und MariaDB der Chatanwendung zurückzuführen sind. Diese Erweiterung soll es ermöglichen die Anwendung, über die Rechnerumgebung hinaus, betreiben zu können.

# Einführung

## Transaktionen in verteilten Systemen

Eine Transaktion ist eine zusammengefasste Abfolge von Ereignissen, die alle erfolgreich ausgeführt werden müssen, um ein Ergebnis zu erzielen.[2] Verteilte Transaktionen zeichnen sich im Wesentlichen dadurch aus, dass sie ihrer Ausführung eine Koordination zwischen mehreren Knoten erforderlich ist.[1]

### Eigenschaften

Um die Erfüllung des Zwecks einer Transaktion sicherzustellen, müssen Transaktionssysteme, die für die Verarbeitung von Transaktionen eingesetzt werden, die Einhaltung der ACID-Prinzipien gewährleisten. Zu diesen Prinzipien zählen die Unteilbarkeit (Atomicity), Konsistenz (Consistency), Isolation (Isolation) und Dauerhaftigkeit (Durability).[3] Wichtig für das Verständnis von Transaktionen sind zudem die Phasen, die während der Verarbeitung einer Transaktion eintreten können.

Eine Transaktion wird mit “begin” gestartet, werden die nachfolgenden Aktionen, die durch die Transaktion zusamengefasst werden korrekt ausgeführt, so erfolgt ein “commit”, der zum Festschreiben der erzielten Ergebnisse auf allen Knoten führt.[1]

Um eine Transaktion im Transaktionssystem eindeutig identifizieren zu können, sind bestimmte Informationen notwendig. Diese werden zusammenfassend als Transaktionskontext bezeichnet.(Quelle Mandl ->)

### Herausforderungen

Die Abwicklung verteilter Transaktionen bringt verschiedene Herausforderungen mit sich.

So müssen die verschiedenen Knoten, auf denen Aktionen ausgeführt werden, miteinander koordiniert werden. Für diesen Zweck werden so genannten Koordinationsprotokolle eingesetzt. Ein bekanntes Beispiel ist das Two-Phase-Commit-Protokoll. Wesentlich sind außerdem Logging-Mechanismen, die im Bedarfsfall die notwendigen Informationen für einen Rollback bereitstellen.

Eine weitere Herausforderung stellt der nebenläufige Zugriff auf verteilte Objekte dar. Der Zugriff muss dabei so erfolgen, als würde das Objekt zu diesem Zeitpunkt nur durch einen Akteur bearbeitet werden und ist anschließend zu synchronisieren um Fehlersituationen wie lost-update oder dirty-read zu verhindern.

## Message-Service-Architektur

### Kriterien für die Anwendung, Vorteile und Herausforderungen

Erzeuger (Sender) von Nachrichten benötigt für die weitere Verarbeitung keine synchrone Antwort (fire-and-forget).

Integrationsszenarien: Es müssen mehrere unterschiedliche Systeme zusammenarbeiten.

Sender erzeugt Nachrichten erheblich schneller, als sie von Empfängern verarbeitet werden können. Eine Message-Queue als Puffer kann hier ausgleichen.

Vorteile

Sender und Empfänger von Nachrichten können in völlig unterschiedliche Technologien und Programmiersprachen erstellt werden.

Message-Queues, insbesondere solche mit zuverlässiger Zustellung (reliable messaging), können die Verfügbarkeit und Rubstheit von Systemen erheblich steigern. Aus diesem Grund werden MQ-Systeme insbesondere im Bereich Finanz- und Kontodaten häufig eingesetzt.

Nachteile

Message-Queues (ob kommerziell oder Open-Source) sind in sich komplexe Systeme mit teilweise hohem Einführungs- und Administrationsaufwand.

Asynchrone und nachrichtenbasierte Programmierung ist signifikant aufwendiger als einfacher call-and-return- Stil. Fehlversuche in asynchronen Systemen kann aufwendig sein.

[QUELLE:ZOTERO: Effektive Softwarearchitekturen Seite 123]

### Java Messaging Service (JMS)

JMS ist eine API, die als Bestandteil von Java EE Interfaces definiert, welche die Interaktion von Java-Anwendungen mit einer Message Oriented Middleware (MOM) ermöglichen. Vom Grundsatz her stellt die Message Oriented Middleware eher ein prozessorientierter Client/Server-Modell dar und bietet damit nicht das gleiche Abstraktionsniveau wie bei objektorientierten Konzepten.[1]

#### JMS-Destinations

JMS bietet zur Übermittlung von Nachrichten zwei JMS-Destinations an, Queue und Topic. Die Queue dient der asynchronen Point-to-Point Kommunikation. Nachrichten werden i.d.R. nach dem FIFO-Prinzip vom Sender in der Queue abgelegt und vom Empfänger dort abgeholt. Topics werden hingegen eingesetzt, wenn die Nachricht im Rahmen eines Publish-Subscribe-Verfahren an mehrere Empfänger versendet werden soll.

#### JMS-Provider

Für den Einsatz von JMS wird ein JMS-Provider benötigt, der die genannten JMS-Destinations verwaltet. Als Beispiel für einen JMS-Provider kann HornetQ (ehemals JBoss Messaging) angeführt werden.

# Konzeption der verteilten Chatanwendung

Ausgehend von der zu Beginn bereitgestellten Chatanwendung, gilt es eine, für ein verteiltes System optimierte Anwendung zu erstellen. Durch diese Änderung der Anforderungen, war die bisherige monolithische Struktur der Chatanwendung weitestgehend ungeeignet. Dies war nicht nur der Tatsache geschuldet, dass die vorhandenen Komponenten Server, Client und Benchmarkclient in einer einzigen Applikation abgebildet wurden. Die Komponenten waren zum Teil durch sehr viele Abhängigkeiten an unterschiedlichen Stellen miteinander gekoppelt. Außerdem wurden einzelne SOLID-Prinzipien[[2]](#footnote-2) verletzt, womit zunächst ein grundlegendes Refactoring durchgeführt und im Zuge dessen die Architektur neu überdacht werden musste. Im nachfolgenden ist daher ein technisches Konzept für die veränderte Umgebung und die grundlegende Architektur für die beiliegende Chatanwendung beschrieben.

Zunächst gilt es allerdings die neuen, zum Ziel gesetzten fachlichen Anforderungen noch einmal zusammenzufassen.

## Struktur des Anwendungsszenarios

Die verteilte Chatanwendung muss mehreren Anwendern die Möglichkeit bieten, Nachrichten an eine Gruppe angemeldeter Benutzer zu versenden und Nachrichten anderer Benutzer zu empfangen. Demnach sind zusätzlich zum Senden und Empfangen von Nachrichten ein Login und Logout zu implementieren.

Dabei soll die Performance der Chatanwendung mithilfe eines Benchmarking-Clients unter Berücksichtigung verschiedener Metriken überwacht werden können.

Über die reine Chat-Funktionalität hinaus sollen verschiedene Informationen bezüglich der versendeten Nachrichten in zwei Datenbanken persistiert werden. Sollte das Persistieren nicht vollständig erfolgen ist ein Rollback durchzuführen.

Auf die in der Datenbank persistierten Daten soll unter Verwendung einer Administrations-Oberfläche zugegriffen werden können.

## Architektur aus der Komponentensicht[[3]](#footnote-3)

Ausgehend von den Anwendungsszenarien wurde die Architektur der Chatanwendung überdacht und neu entwickelt. Hierfür sind im Anhang VI.A entsprechende Komponentendiagramme der anfänglichen und der, im Zuge dieser Arbeit, neu entwickelten Anwendung gegenübergestellt. Diese sollen den groben Aufbau und die Veränderungen aufzeigen. Grundsätzlich ist das alte Projekt als eine monolithische Java-Anwendung zu sehen, während die Weiterentwicklung auf modulare Apache Maven Projekte aufbaut. Jede Komponente aus Abbildung 4 stellt hierbei ein eigenes Maven-Projekt dar, das Abhängigkeiten zu anderen Projekten besitzt. Diese Abhängigkeiten sind differenziert zu betrachten und werden in den nachfolgenden Komponentenbeschreibungen noch einmal genauer betrachtet.

### Server

Die Server-Komponente beinhaltet alle Funktionen, die für die serverseitige Anwendung notwendig sind. Dabei ist diese auf den JBoss Wildfly optimiert und wird als eine einzelne WAR-Datei ausgeliefert. Es besteht hierbei lediglich eine Abhängigkeit zu der Model-Komponente, welche die, für die Kommunikation server- und clientseitig verwendeten Datenobjekte enthält. Diese sind zentral in einer ausgelagerten Komponente definiert, um eine redundante Auslegung und damit ggf. Fehler bei zukünftigen Weiterentwicklungen zu vermeiden. Dieses Model wird bei dem Bauen der auszuliefernden WAR-Datei automatisch eingebunden.

Die Serverkomponente besitzt insgesamt 4 Schnittstellen. Diese unterteilen sich den angebotenen RESTful Webservice, eine Anbindung zu der Queue und dem Topic des JMS-Service, sowie eine Datenbankanbindung zu mehreren MariaDB-Instanzen. Dieser Aufbau wird durch das Komponentenmodell aus Abbildung 1 noch einmal verdeutlicht.



Abbildung 1: Komponentenmodell der serverseitigen Anwendung

### Client und Konnektoren

Anders als die Serverkomponente wurde der Client viel modularer konzipiert. Aus Verteilungssicht handelt es sich hierbei um eine einzelne Client-Komponente, die am Ende eines Build-Prozesses entsteht. Der Code wurde allerdings in komplett unabhängigen Projekten entwickelt. Hiermit konnte durch die striktere Umsetzung des „seperation of concern“-Prinzips[[4]](#footnote-4), einige Vorteile gewonnen werden. Die bereits zuvor beschriebene Ausgliederung des Datenmodells ist hierbei nur ein Teil der Modularisierung, welche in Abbildung 2 dargestellt ist. Während die Client-Komponente sehr schlank ist und hauptsächlich nur die Masken, ein UI-Model, Navigations- und Validierungslogik enthält, ist jegliche Kommunikation den Konnektoren-Komponenten überlassen.

Diese Konnektoren sind rein technische Komponenten ohne jegliche Fachlogik[[5]](#footnote-5). Sie sind generisch und über Schnittstellen für jeden Client individuell konfigurierbar. Dies führt zu einer verbesserten Wiederverwendbarkeit, welche bspw. bei der Implementierung des Benchmark-Clients oder auch zukünftige neu entwickelte Clients zum Tragen kommt. Außerdem ist somit eine vereinfachte Austauschbarkeit der Kommunikationsmittel gegeben, sollte die Anbindung an den RESTful Webservice oder JMS durch eine andere Kommunikationsart, in bestehenden oder neuen Clients ersetzt werden. Dies ist ebenfalls zum Vorteil der Wart- und Änderbarkeit, da die Kommunikation übergreifend zentral und frei von Redundanzen an den Konnektoren vorgenommen werden können.



Abbildung 2: Komponentenmodell des Clients mit Konnektoren

### Benchmarking-Client

Für die Durchführung eines Benchmarkings ist ein separater Baustein, der Benchmark-Client, implementiert. Anders als der vom Ausgang bereitgestellte Benchmark-Client, ist dieser neu implementiert und besitzt keine Abhängigkeiten auf den zuvor beschriebenen Anwendungsclient. Wie der normale Client, nutzt auch dieser lediglich die Konnektoren für die Anbindung an den RESTful Webservice und an JMS.



Abbildung 3: Komponentenmodell des Benchmarking-Clients

## Datenmodell

Das umzusetzende Datenmodell ist hierbei sehr simpel. Es handelt sich, wie in Abbildung 4 dargestellt, um zwei Entitäten ohne Beziehungen. Während die TraceDB den Fokus vor allem auf das persistieren der Nachrichten ausgelegt ist, soll die CountDB die Anzahl der Nachrichten gruppiert auf die Chatteilnehmer speichern. Die Komplexität ergibt sich im Rahmen der Arbeit lediglich aus der Absicherung einer verteilten Transaktion, denn beide Tabellen liegen in für sich eigene Datenbankinstanzen. Eine entsprechende Umsetzung dieser Transaktionsklammer und entsprechende Fehlerbehandlung ist nachfolgend, im Rahmen der Implementierung beschrieben.



Abbildung 4: Domainmodell der serverseitigen Datenbankschicht

## Fehlertolerante Transaktionsverarbeitung für REST-Clients

Der zuvor in Abschnitt B beschriebene REST-Konnektor soll zudem eine konsistente Fehlerbehandlung bieten und fallweise automatisiert Anfragen an den Server wiederholen. Dieser Vorgang mit einer möglichen Wiederholung ist in Abbildung 5 in einer Sequenz dargestellt. Die durch eine maximale Anzahl beschränkten Wiederholungen werden hierbei durch bestimmte http-Fehlercodes oder Timeout-Exceptions getriggert.



# Implementierung

Im Folgenden wird nun auf die Umsetzung der vorab beschrieben Architektur eingegangen. Hierfür sind zunächst Entwurfsentscheidungen für die Basisarchitektur festgehalten, bevor im Anschluss auf detaillierte Umsetzungsdetails, welche als Anforderung an die Anwendung vorgegeben waren, eingegangen wird.

## Umsetzung der Basisarchitektur

Bevor jedoch auf Implementierungsdetails eingegangen wird, sind zunächst das verwendete Zielsystem und entsprechende Entwurfsentscheidungen festgehalten.

### Das Zielsystem und dessen Vorzüge

Die Ursprüngliche Anforderung an die Chatapplikation definierte den Wildfly 8.2 als Zielsystem. Unter Absprache wurde jedoch im Nachgang der Umstieg auf eine aktuellere Version des Wildflys in Betracht gezogen und letztendlich umgesetzt. Für einen Umstieg auf die aktuellste Version 10.1.0 gelten vor allem die nachfolgend zusammengefassten Vorzüge:

* Performancesteigerung durch den Support des aktuelleren Protokoll HTTP/2.[[6]](#footnote-6)
* Einfacheres Mangement der Wildfly-Konfiguration, bspw. durch die Bereitstellung vordefinierter Datenbankkonfigurationen.
* Das in Wildfly 8 für die Umsetzung von JMS standardmäßig genutzte Projekt HornetQ wurde eingestellt. Die Codebasis ist an Apache übergeben worden und in dem neuen Projekt ActiveMQ Artemis aufgegangen. [QUELLE?]
* Die Sicherheit ist in Wildfly 8.2 nicht mehr gewährleistet, da Sicherheitsupdates für Wildfly 9 und 10, jedoch nicht mehr für 8 veröffentlicht wurden.
* Der Wildfly 10 ist zukunftssicherer, da das kommende Java 9 bereits voll unterstützt wird.
* Es wird die aktuellste Version 5 des ORM-Frameworks Hibernate unterstützt. Dies führt zu einer verbesserten Nutzung und möglichen Performancegewinnen.[[7]](#footnote-7)

[QUELLE Doku/Changelog WF9+10]

Es wurde außerdem erreicht, dass die Chatapplikation einen Großteil der Konfiguration während des Deployments selbst übernimmt, wodurch ein entsprechender Aufwand bei dem initialen Aufsetzen des Wildflys zum Teil entfällt. Konfigurative Änderungen können somit in der Applikation selbst mittels einfachem XML vorgenommen werden. Dies umfasst unter anderem die nicht triviale Konfiguration der XA-Datasources, um die man sich nicht weiter zu kümmern braucht.

### Das Build- und Deploymanagement

Mittels des Einsatzes eines Dependency- und Buildmanagement-Tools, wie das verwendete Apache Maven, bringt die neue Chatanwendung viele Vorteile mit sich. Hierzu zählt neben einem vereinfachten und redundanzfreien Verwalten der Abhängigkeiten zu bestimmten Bibliotheken, auch das automatisierte Bauen und Testen der Anwendung. Eine Aufwandsverringerung wurde vor allem mittels des automatisierten Deployments durch Apache Maven bei jedem Build auf den Wildfly erreicht.[[8]](#footnote-8)

### Anbindung der Datenbankinstanzen

* JPA (nicht JDBC)
  + ORM
  + Config außerhalb des Codes über XML
  + Transaction-scoped persistence context

## Ablösung des TCP-Websocket durch JMS und einem RESTful Webservice

* MDB
* EJB Session Beans -> alle stateless

Die Enterprise-Java-Beans-Spezifikation (EJB) ist Bestandteil von JEE und unterstützt die Umssetzung verteilter Komponentensysteme.

(„The number of MDB threads is equal to 16 (this is the default value)) Es wird automatisch ein Threadpool mit 16 Threads erzeugt, die durch EJB bzw. den Message Driven Beans verwaltet werden [6]. Dies bezüglich sind keine weiteren Anforderungen an die Chatanwendung gegeben, weshalb keine weiteren Konfigurationsarbeiten an dem Pool vorgenommen wurden.

<<Eingehen auf JAX-RS und ggf. Fehlerbehandlung/Retry-Mechanismus>>::David::

## Umsetzung einer gemäß XA verteilten Transaktion

X/OpenXA (XA) ist ein Standard für die Verarbeitung von verteilten Transaktionen. Wesentliches Element dieses Standards ist das Zwei-Phasen-Commit-Protokoll welches der Koordination der verschiedenen Knoten dient.

* EJB

Für die Implementierung verteilter Transaktionen mit EJB wird ein Transaktionsmanager benötigt.

Je nachdem woher der Aufruf der Transaktionsmethoden (begin, commit, etc.) kommt, unterscheidet man client-managed, container-managed und bean-managed.

* XA Datasources
  + Über mysql-ds.xml konfiguriert (Flexibilität)
  + Für jede Datenbank eine XA

### Umsetzung der Clients mit neuer UI

Aufgrund der neuen Anforderungen an die Chatapplikation und die neue Architektur, wurden der Chatclient und der Benchmarkingclient von Grund auf neue geschrieben. Hierfür wurde sehr viel alter Code beseitigt und stark modifiziert, um die Clients schlanker und moderner zu machen. Hierfür wurden ebenfalls die JavaFX-Masken komplett neu implementiert, um eine erhöhte Userexperience durch moderne Masken zu erreichen. Diese sind in Anhang XXX dargestellt. Vor allem der Benchmarkingclient profitiert von dieser Maßnahme, da die alte Maske noch komplett mittels Code erzeugt wurde. Die neu entwickelte Maske ist ebenfalls durch das Konzept der *fxml[[9]](#footnote-9)* entkoppelt bietet viele neue Funktionalitäten. Diese umfassen unter anderem die Live-Berechnung und Ausgabe verschiedener Kenngrößen bereits während des Benchmarkings sowie das ebenfalls Live-Aufbereiten von vordefinierten Diagrammen, um ein entsprechendes Ergebnis direkt zu visualisieren. Diese sind ebenfalls Beispielhaft in Anhang XXX beigefügt.

## Umsetzung eines Admin-Clients

Wie in den Anforderungen festgehalten, ist ebenfalls die Entwicklung eines sogenannten Admin-Clients mittels eines modernen Web-Application-Frameworks vorgesehen. Dieser soll mit einer REST-Schnittstelle kommunizieren und aktuelle Serverdaten aus den Datenbanken auslesen und darstellen. Hierzu zählen:

* Aktuell angemeldete Chat-Benutzer
* Anzahl der Nachrichten pro Chat-Benutzer
* Verschiedene statistische Daten

Ferner soll der Admin-Client die Möglichkeit haben, die Daten der Count- und Trace-Datenbank zu löschen.

### Angular 2 als Entwurfsentscheidung

Das Web-Application-Framework Angular wurde am 14. September 2016 in der Version 2.0 veröffentlicht [7]. Damit ist diese Version des Frameworks zum Zeitpunkt der Erstellung dieser Arbeit in etwa 3 Monate jung und zählt folglich zu den modernen Web-Application-Frameworks. Da hierzu auch einige andere verbreitete Frameworks wie beispielsweise ReactJS zählen, seien im Folgenden einige Gründe aufgezählt, die diese Entwurfsentscheidung herbeigeführt haben.

Bei der Entwicklung von Angular 2 wurde versucht, die Erfahrungen der Community aus ca. 5 Jahren Entwicklung mit Angular 1.x zu integrieren. Hierbei wurde insbesondere Wert darauf gelegt, dass die Produktivität des Entwicklers erhöht werden kann, indem ihm mehr Unterstützung durch die Entwicklungsumgebung, sowie durch das Ergänzen von JavaScript mit TypeScript garantiert wird [8]. Durch die Ausnutzung dieser Faktoren konnte der Admin-Client zügig entwickelt werden.

Bei Angular 2 handelt es sich um einen kompletten Neubau, nicht etwa eine Erweiterung der Version 1.x. Hierdurch konnten viele Konzepte überdacht werden mit dem Ergebnis, dass beispielsweise die Komponentenarchitektur anstelle des vorherigen Konzepts mit Controllern und Direktiven trat. Von diesem Konzept waren die Entwickler schlussendlich so überzeugt, dass es auf Angular 1.x zurück portiert wurde [8].

Diese Vorteile und vielen andere Verbesserungen gegenüber Angular 1.x - insbesondere im Bereich der Performance – veranlassten das Projektteam zur Entwurfsentscheidung, den Admin-Client mit diesem Web-Application-Framework zu entwickeln.

### Grafische Oberfläche des Admin-Clients

Um die Benutzeroberfläche grafisch ansprechend gestalten zu können, wurde auf das Framework Material2 zurückgegriffen, welches sich zum Zeitpunkt der Erstellung dieser Arbeit noch in einer Entwicklungsversion befindet. Es handelt sich um die offizielle Umsetzung der Material-Design-Richtlinien von Google durch das Angular-Team [9]. Da die für die Entwicklung des Admin-Clients notwendigen GUI-Elemente bereits in der Entwicklungsversion unterstützt wurden, entschied das Projektteam die Verwendung von Material2.

### Aufbau und Architektur des Admin-Clients

Die Architektur wurde aufgrund der geringen Komplexität des Admin-Clients ebenfalls bewusst einfach gehalten, um keine unnötige Komplexität aufzubauen („Over-Engineering“).

Die drei Hauptfunktionen „Chat-Benutzer“, „Statistiken“ und „Daten löschen“ werden jeweils optisch durch einen Tab repräsentiert.

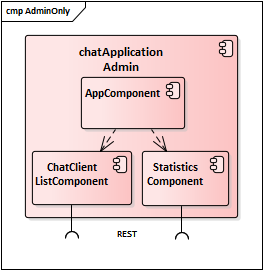


Abbildung 5: Architektur des Admin-Clients. Die obligatorische AppComponent greift für die Darstellung weiterer Daten auf die ChatClientListComponent und die StatisticsComponent zurück.

Aus architektureller Sicht besteht die Applikation aus einer generellen *AppComponent*, welche die Darstellung der Tabs sowie die Funktion zum Löschen der Daten bereitstellt.

Des Weiteren existiert eine *ChatClientListComponent*, welche sich um die Abfrage der aktuellen Chat-Benutzer sowie deren Nachrichtenanzahl kümmert und diese Informationen auf dem Bildschirm darstellt.

Außerdem frägt die *StatisticsComponent* verschiedene statistische Informationen vom Server ab und stellt sie ebenfalls im entsprechenden Tab dar. Hierbei werden die folgenden Server-Informationen ausgegeben:

* Die gesamte Anzahl der Nachrichten
* Die Anzahl der eingeloggten Benutzer
* Die durchschnittliche Nachrichtenzahl pro eingeloggtem Benutzer
* Die durchschnittliche Nachrichtenlänge

Die Komponenten fragen die Daten selbstständig beim Server über die REST-Schnittstelle an. Eine Kapselung dieser Abfragen in Services ergibt für die geringe Komplexität der Applikation noch keinen Sinn, da effektiv keine Logik geteilt werden würde, was jedoch dem Haupteinsatzgebiet von Services entspräche [10].

## Weitere Umsetzungsmerkmale

Neben den vielen Änderungen und neu eingebauten Funktionen, weißt die anhängende Chatanwendung weitere zusätzliche Merkmale auf, die im Folgenden gelistet sind:

* Sowohl Chat-Client, als auch Benchmarking-Client werden durch einen Build-Vorgang zusätzlich in nativen EXE-Dateien bereitgestellt und sind somit typischerweise auf jedem PC einfach ausführbar.
* Der Benchmarking-Client bereitet bereits während der Testdurchführung unterschiedliche Diagramme auf und stellt diese Live in der Maske dar.
* …

System wird zwar optimiert, jedoch wurde nicht total auf lasten der Ausfallsicherheit konfiguriert. Das Senden von Nachrichten wurde bspw. weiterhin auf "Persistent" belassen und nicht auf "NON\_PERSISTENT" gestellt. Dies hätte zur Folge, dass bei einem Neustart des Servers alle Nachrichten verloren gehen (Die Nachrichten könnten zu Gunsten der Performance lediglich im Hauptspeicher gehalten werden.). Ebenfalls würde hiermit die Rollback-Funktion deaktiviert, da somit kein "Redelivery" aus der Queue erfolgen kann. Da der Rollback als funktionale Anforderung an das System gilt, wurde auf diese Optimierung verzichtet [http://www.mastertheboss.com/jboss-server/jboss-jms/configuring-message-redelivery-on-jboss-wildfly].

# Benchmarking

## Testaufbau und Rahmenbedingung

### Verwendete Infrastrukturkomponenten

### Maßnahmen zur Performanceoptimierung

Aufgrund der Tatsache, dass die Performance als eine der nicht-funktionalen Anforderungen gilt, wurden Optimierungsmaßnahmen angestrebt, um eine Performance-Steigerung zu erreichen. Hierbei wurden viele Maßnahmen mit deren Wirkung und entsprechend negativen Folgen, wie den Verlust einer Ausfallsicherheit abgewogen. Aufgrund dessen wurden mögliche Vorgehen, wie die Konfiguration von flüchtigen, nicht persistenten und „durable“ Queues, Nachrichten und Topics vermieden. Die folgenden Maßnahmen wurden dagegen vorgenommen, um ein optimales Testergebnis zu erzielen:

* Das serverseitige Entgegennehmen und Verarbeiten von JMS-Nachrichten wird im Anschluss mittels einer gesendeten Bestätigung an die Queue beendet. Um die Queue zu entlasten, wurde daher mittels der einem „Controlling Message Acknowledgment“-Konfiguration auf die Option „DUPS\_OK\_ACKNOWLEDGE“ umgestellt. Hiermit werden Bestätigungen nur „lazy“ versendet und somit Overhead reduziert. Dies führt zu möglichen Nachrichtenduplikaten, die wir in Tests nicht reproduzieren und in möglichen Ausnahmefällen in Kauf nehmen. [QUELLE: <https://docs.oracle.com/cd/E19798-01/821-1841/bncfw/index.html>]
* Statt den bequemen Objektnachrichten wurde nachträglich auf Textnachrichten umgestellt. Hiermit soll die Nachricht verkleinert werden. Dies wird mittels der Serialisierung der Transportobjekte in JSON-Strings angestrebt. [[10]](#footnote-10)
* Für die JMS-Nachricht werden automatisch durch JMS erzeugte Nachrichten-IDs und Zeitstempel explizit entfernt. Hiermit soll die Nachricht verkleinert werden.
* Dem Wildfly wurden konfigurativ 2 GB Arbeitsspeicher hinzugefügt, um eine erhöhte Abarbeitung der ankommenden Nachrichten zu gewährleisten.
* Durch eine Optimierung der MariaDB-Datenbanken erhalten diese ebenfalls 2 GB Arbeitsspeicher und eine größere Puffer-Konfiguration, um eine größere Verarbeitung im Hauptspeicher zu erzielen.

### Verwendete Testmetriken

### Testspezifikation

## Dokumentation der Messergebnisse

## Evaluation der Messergebnisse

# FAZIT

Glossar

CMT - Container Managed Transaction X

JMS - Java Message Service X

JSON - JavaScript Object Notation X

MDB - Message Driven Bean X

References

[1] A. Schill und T. Springer, *Verteilte Systeme*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[2] H. Balzert, *Lehrbuch der Softwaretechnik, Entwurf, Implementierung, Installation und Betrieb*, 3. Aufl. Spektrum, Akademischer Verlag.

[3] U. Hammerschall, *Verteilte Systeme und Anwendungen*. Pearson Education.

[4] M. Belshe, M. Thomson, und R. Peon, „Hypertext Transfer Protocol Version 2 (HTTP/2)“. [Online]. Verfügbar unter: https://tools.ietf.org/html/rfc7540. [Zugegriffen: 05-Dez-2016].

[5] „Hibernate ORM 5.0 User Guide“. [Online]. Verfügbar unter: https://docs.jboss.org/hibernate/orm/5.0/userguide/html\_single/Hibernate\_User\_Guide.html. [Zugegriffen: 05-Dez-2016].

[6] „Message-Driven EJBs“. [Online]. Verfügbar unter: http://docs.oracle.com/cd/E11035\_01/wls100/ejb/message\_beans.html. [Zugegriffen: 17-Nov-2016].

[7] J. Kremer, „Angular, version 2: proprioception-reinforcement“. [Online]. Verfügbar unter: http://angularjs.blogspot.com/2016/09/angular2-final.html. [Zugegriffen: 13-Dez-2016].

[8] Rangle.io, „Why Angular 2? · Rangle.io : Angular 2 Training“. [Online]. Verfügbar unter: https://angular-2-training-book.rangle.io/handout/why\_angular\_2.html. [Zugegriffen: 13-Dez-2016].

[9] Google Inc., „angular/material2“, *GitHub*. [Online]. Verfügbar unter: https://github.com/angular/material2. [Zugegriffen: 13-Dez-2016].

[10] Google Inc., „Angular Services“. [Online]. Verfügbar unter: https://angular.io/docs/ts/latest/guide/architecture.html#!#services. [Zugegriffen: 16-Dez-2016].

[7] J. Kremer, „Angular, version 2: proprioception-reinforcement“. [Online]. Verfügbar unter: http://angularjs.blogspot.com/2016/09/angular2-final.html. [Zugegriffen: 13-Dez-2016].

[8] „Why Angular 2? · Rangle.io : Angular 2 Training“. [Online]. Verfügbar unter: https://angular-2-training-book.rangle.io/handout/why\_angular\_2.html. [Zugegriffen: 13-Dez-2016].

Appendix

## Gesamtarchitekturen im Vergleich

Die in Abbildung 4 dargestellte Gesamtarchitektur der Chatanwendung stellt das Ergebnis der beigefügten Anwendung dar. Die Komponenten sind auf Klassenebene dargestellt bzw. bündeln Klassen in logischen Komponenten. Ein Beispiel hierfür ist sind die Komponenten „GUI“, die verschiedene Masken und Controller umfassen. Im Vergleich zu dieser Zielanwendung steht die Architektur der, zu Beginn bereitgestellten Chatanwendung. Diese besitzt keine Verteilung und ist ggf. nicht vollständig, zeigt allerdings den Unterschied.

Abbildung 6: Anwendungsüberblick in Form eines Komponentenmodells





Abbildung 7: Anwendungsüberblick der Ausgangsanwendung in Form eines Komponentenmodells

## Anhang Software

Neben den in der Arbeit beschriebenen Grundprinzipien, der Architektur und Implementierungen sowie den Testergebnissen wurde der Arbeit der Quellcode und Konfigurationsdateien angehängt. Der Anhang umfasst im Detail die folgenden Punkte:

* Alle Maven-Projekte der einzelnen Komponenten
* Alle Konfigurationsdateien des Wildfly 10
* Alle Konfigurationsdateien der MariaDB

## Masken der Clients

### Chatclient Masken

<Grafik login>

<Grafik chatmaske>

### Benchmarkingclient Masken

<<Grafiken der einzelnen Tabs>>

### Adminclient Masken

<<Grafiken der Sichten>>

1. [↑](#footnote-ref-1)
2. Das Akronym SOLID umfasst die, im Allgemeinen für die Umsetzung einer sauberen Anwendungssoftware, häufig verwendeten Prinzipien „Single responsibility principle“, „Open/closed principle“, „Liskov substitution principle“, „Interface segregation principle“ und „Dependency inversion principle” [QUELLE]. [↑](#footnote-ref-2)
3. An dieser Stelle sei festgehalten, dass eine grobe Verteilung sowie entsprechende Sequenzdiagramme bereits durch das Anforderungsdokument gegeben sind und damit nachfolgend lediglich bei einer Abweichung dieser spezifizierten Logik entsprechende Diagramme für die Beschreibung zum Einsatz kommen. [↑](#footnote-ref-3)
4. Das Prinzip „Seperation of concern“ beschreibt die Trennung zwischen Verantwortlichkeiten, Zuständigkeiten oder Aufgaben [QUELLE:ZOTERO: Effektive Softwarearchitekturen Seite 67]. Dies wurde weitestgehend versucht, auf allen Ebenen des Entwurfs und der Implementierung einzuhalten. [↑](#footnote-ref-4)
5. Darunter ist die reine Fachlogik zu verstehen. Funktionen, wie eine konsistente Fehlerbehandlung und einem entsprechenden Retry-Mechanismus, sind ebenfalls in diese Komponente ausgelagert. [↑](#footnote-ref-5)
6. Hiermit wird die Komprimierung der Header-Informationen ermöglicht und damit die Latenz verringert [4]. [↑](#footnote-ref-6)
7. Hiermit kommen Vorteile wie die erleichterte Nutzung von Java 8 Typen (bspw. Date, Time, Timestamp), einer neuen Validierungskomponente mittels derer fachliche Validierungen durchgeführt werden können und das Deklarieren einzelner Felder als LAZY, damit diese nur bei einem tatsächlichen Zugriff genutzt werden [5]. [↑](#footnote-ref-7)
8. Dies wird mittels einer Konfiguration der POM aus der Server-Komponente erreicht. Das frei verfügbare *wildfly-maven-plugin* kommt hierfür zum Einsatz. Dieses findet automatisch die entsprechende Wildfly-Instanz und führt bei einem erfolgreichen Maven-Build der Serverkomponente ein deployment aus. [↑](#footnote-ref-8)
9. Unter dem Konzept der FXML ist… blabla [QUELLE?] [↑](#footnote-ref-9)
10. Statt den Standard von JAX-B zu verfolgen und das Objekt nach XML zu serialisieren, wurde auf JSON gesetzt, da dieses Format in der Regel leichtgewichtiger ist und damit einer höheren Performance zu rechnen ist. Hierfür kommt das Framework Jackson zum Einsatz. [↑](#footnote-ref-10)