Redesign einer Chat-Anwendung als verteiltes System.

C. Eidelloth, D. Sautter, F. Stützinger und M. Auch

***Abstract*— Diese Arbeit befasst sich mit der Umstellung einer bislang monolithischen Chat-Anwendung auf ein verteiltes System. Ziel war die Ablösung von TCP-Sockets durch die Einführung von JMS. Für Login und Logout wurden RESTful-Webservices implementiert. Die serverseitigen Verarbeitungsschritte, die u. a. das Persistieren in zwei Datenbanken umfassen, sind Teil einer verteilten XA-Transaktion. Für den Zugriff auf die persistierten Daten wurde unter Verwendung von Angular2 eine administrative Benutzeroberfläche implementiert. Es wurden verschiedene Maßnahmen zur Optimierung der Performance von MariaDB und Wildfly ergriffen. Abschließende Tests mithilfe eines eigenentwickelten Benchmarking-Clients bestätigen die Skalierbarkeit und Zuverlässigkeit der Anwendung.**

***Index Terms*— JMS, Chatapplication, XA-Transaction, RESTful APIs, JavaFX, Wildfly, Performance.**

I. 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 der Realisierung der neuen Chatanwendung ergibt sich eine Reihe von Problemstellungen. Ursache dafür sind gestiegene Anforderungen an die verteilte Strukturierung der Anwendung sowie an die Kommunikation zwischen Server und Clients, die nun anhand einer Message-Service-Architektur implementiert werden soll. Mittels dieser wird zu der bisher einfachen, nachrichtenbasierten Programmierung eine asynchrone Kommunikation eingebracht. Eine weitere Herausforderung liegt in der Umsetzung einer verteilten XA-Transaktion die u.a. die Persistierung von Inhalten in zwei Datenbanken umfasst. Im Vergleich zu dem einfachen synchronen *call-and-return*-Stil auf einem lokalen System, ist unter anderem mit einer erhöhten Anzahl an Fehlversuchen zu rechnen [2]. Diesen ist mit entsprechenden Maßnahmen zu begegnen.

II. Einführung

A. Message-Service-Architektur

Eine Möglichkeit Systemkomponenten zu einem verteilten und nachrichtenbasierten System zusammenzusetzen, ist die Message-Service-Architektur. Diese Architektur und der für die Umsetzung genutzte Standard Java Messaging Service (JMS) sind daher nachfolgend beschrieben.

1) Kriterien für die Anwendung, Vorteile und Herausforderungen

Ein wesentliches Merkmal der anzustrebenden, verteilten Architektur gegenüber dem Ausgangsprojekt ist die asynchrone Verarbeitung. Der Sender einer Nachricht benötigt für die weitere Verarbeitung keine synchrone Antwort, sondern agiert nach dem Prinzip „fire-and-forget“. Hierbei kann ein wesentlich höheres Eintreffen von Nachrichten bei dem Server erfolgen, als dieser verarbeiten kann. Aufgrund dessen ist eine Nachrichten-Queue einzuführen, die Nachrichten puffert. Hierbei sollen Sender und Empfänger soweit entkoppelt sein, dass diese in völlig unterschiedlichen Technologien und Programmiersprachen implementiert werden können. Zudem bietet eine Nachrichten-Queue den Vorteil, dass eine zuverlässigere Zustellung erzielt und damit die Verfügbarkeit und Robustheit von Systemen erheblich gesteigert werden können. Aus diesem Grund werden Message-Queue-Systeme insbesondere im Bereich Finanz- und Kontodaten häufig eingesetzt [2].

Für eine Umsetzung der Message-Service-Architektur steht im Java-Kontext die, unter anderem durch den JSR-343 definierten, JMS API zur Verfügung [3], weshalb diese nachfolgend beschrieben wird.

2) Java Messaging Service

Der als Bestandteil von JavaEE spezifizierte Standard JMS stellt als API eine Möglichkeit der Interaktion von Java-Anwendungen mit einer *Message Oriented Middleware* (MOM) zur Verfügung. Abbildung 1 stellt hierbei den Kommunikationsfluss zwischen MOM und JMS dar. Vom Grundsatz her stellt die *Message Oriented Middleware* eher ein prozessorientiertes Client/Server-Modell dar und bietet damit im Vergleich zu objektorientierten Konzepten ein geringeres Abstraktionsniveau [1].

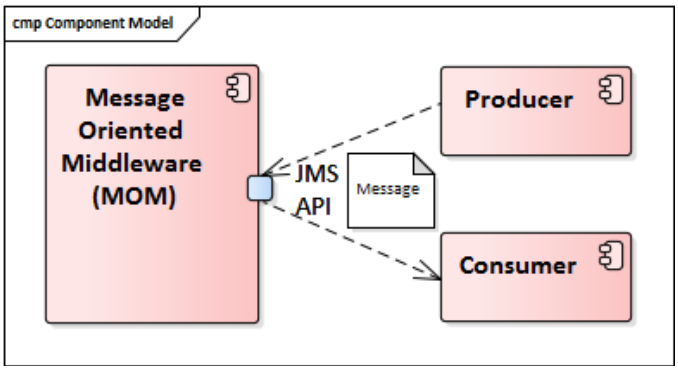


Abbildung 1: Clients in verschiedenen Rollen kommunizieren mit dem JMS Provider (hier eine MOM) über die JMS API [1]

Wenn eine Komponente JMS nutzt, ist diese in Form eines *Producers* oder *Consumers* als JMS-Client zu bezeichnen. Dieser kann mithilfe von JMS senden oder empfangen. Dabei bietet JMS die Möglichkeit eines asynchronen Austausches gleichberechtigter Partner die nicht dem strengen Client-Server-Modell entspricht [4]. Dies bedeutet, dass nicht alle Kommunikationspartner zu einem Austausch gleichzeitig erreichbar sein müssen, um eine verlustfreie Kommunikation zu gewährleisten.

a) JMS-Destinations

JMS bietet zur Übermittlung von Nachrichten zwei Arten von JMS-Destinations an, Queue und Topic. Die Queue dient der asynchronen Point-to-Point Kommunikation. Die Nachrichten werden in der Regel nach dem FIFO-Prinzip durch den Sender in der Queue abgelegt und durch den Empfänger dort abgeholt. Topics werden hingegen eingesetzt, wenn die Nachricht im Rahmen eines Publish-Subscribe-Verfahrens an mehrere Empfänger versendet werden soll. Dabei kann das Topic als eine Art Message-Broker verstanden werden, der Nachrichten sammelt und an die am Topic abonnierten Subscriber verteilt [4].

b) JMS-Provider

Für den Einsatz von JMS wird ein JMS-Provider benötigt, der die genannten JMS-Destinations verwaltet. Der Provider ist eine Instanz, die den JMS-Standard implementiert und unter anderem dafür sorgt, dass die Nachrichtenwarteschlangen bzw. Queues verwaltet werden [5]. Als Beispiel für einen JMS-Provider können HornetMQ und ActiveMQ angeführt werden. Im Falle der Point-to-Point Kommunikation handelt es sich um eine *1:1* Beziehung, in der die Nachricht vom Provider so lange aufbewahrt und in der Queue zu halten ist, bis diese abgeholt wird oder abgelaufen ist [5]. Im Fall des Publish-Subscribe-Verfahrens liegt eine *1:m*-Beziehung vor, in der vom Provider dafür gesorgt wird, eine Liste aller Subscriber zu verwalten und die Nachrichten zu übermitteln [5]. Wird die Eigenschafft Durable für ein Topic gesetzt, besteht die Möglichkeit, die Nachricht wie bei der Point-to-Point Kommunikation aufzubewahren [4].

B. Transaktionen in verteilten Systemen

Eine Transaktion ist eine zusammengefasste Abfolge von Ereignissen, die alle erfolgreich ausgeführt werden müssen, um ein Ergebnis zu erzielen [3]. Verteilte Transaktionen zeichnen sich im Wesentlichen dadurch aus, dass für ihre Ausführung eine Koordination zwischen mehreren, logisch oder physisch voneinander getrennten Knoten erforderlich ist [1].

1) Eigenschaften

Für die Verarbeitung von Transaktionen werden sogenannte Transaktionssysteme eingesetzt. Diese müssen die Einhaltung der ACID-Kriterien gewährleisten. Zu diesen zählen die Unteilbarkeit (Atomicity), Konsistenz (Consistency), Isolation (Isolation) und Dauerhaftigkeit (Durability) [5]. Transaktionen laufen in Phasen ab:

Eine Transaktion wird mit “begin” gestartet. Werden die nachfolgenden Aktionen, die durch die Transaktion zusammengefasst werden korrekt ausgeführt, so erfolgt ein “commit”, der zum Festschreiben der erzielten Ergebnisse auf allen Knoten führt. Verlaufen einzelne Aktionen, die Teil der Transaktion sind, nicht korrekt, so wird ein „rollback“ durchgeführt [1].

2) 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 Koordinationsprotokolle eingesetzt. Ein bekanntes Beispiel ist das Two-Phase-Commit-Protokoll [6]. Wesentlich sind außerdem Logging-Mechanismen, die im Bedarfsfall die notwendigen Informationen für ein Rollback bereitstellen [7].

Eine weitere Herausforderung stellt der nebenläufige Zugriff auf verteilte Objekte dar. Der schreibende Zugriff muss dabei synchronisiert erfolgen, so dass das Objekt zu jedem Zeitpunkt nur durch einen Akteur bearbeitet werden kann. Auf diese Weise können Fehlersituationen wie lost-update vermieden werden [1].

III. 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, ist die bisherige monolithische Struktur der Chatanwendung weitestgehend ungeeignet. Dies ist nicht nur der Tatsache geschuldet, dass die zu ändernde Chatanwendung Server, Client und Benchmarking-Client in einer einzigen Komponente abgebildet. Die Komponenten der zu ändernden Chatanwendung sind zum Teil durch direkte Abhängigkeiten an unterschiedlichen Stellen miteinander gekoppelt. Außerdem wurden einzelne SOLID-Prinzipien[1] verletzt, wodurch 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

[1] 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” [2].

beiliegende Chatanwendung beschrieben.

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

A. *Fachliche Anforderungsbeschreibung*

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 zu dem Senden und Empfangen von Nachrichten ein Login- und Logout-Mechanismus zu implementieren. Der Ablauf ist durch das Modell in Anhang X.F detailliert beschrieben. Dabei soll die Performance der Chatanwendung mithilfe eines Benchmarking-Clients unter Berücksichtigung verschiedener Metriken getestet 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.

B. *Architektur aus der Komponentensicht* [2]

Ausgehend von den Anwendungsszenarien wurde die Architektur der Chatanwendung überdacht und neu entwickelt. Zur Veranschaulichung sind im Anhang X.A entsprechende Komponentendiagramme der anfänglichen und der im Zuge dieser Arbeit neu entwickelten Anwendung gegenübergestellt. Diese sollen den groben Aufbau und die vorgenommenen Veränderungen aufzeigen. Grundsätzlich ist das alte Projekt, das als Ausgangspunkt diente, als eine monolithische Java-Anwendung zu sehen, während die Weiterentwicklung auf modulare Apache Maven Projekte aufbaut. Jede Komponente aus Abbildung 16 stellt hierbei ein eigenes Maven-Projekt dar, das Abhängigkeiten zu anderen Projekten besitzt bzw. diese als JAR-Dateien heranzieht. Diese Abhängigkeiten werden in den nachfolgenden Komponentenbeschreibungen noch einmal genauer betrachtet.

1) *Server*

Die Server-Komponente beinhaltet nach dem Build alle fachlichen Funktionen, die für die serverseitige Anwendung notwendig sind. Dabei ist diese auf den Anwendungsserver 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 Definition und damit ggf. Fehler bei zukünftigen Weiterentwicklungen zu vermeiden. Dieses Model wird bei dem Erzeugen der auszuliefernden WAR-Datei automatisch eingebunden.

Die Serverkomponente besitzt insgesamt vier Schnittstellen. Diese unterteilen sich den angebotenen RESTful Webservice, eine Anbindung zu Queue und Topic des JMS-Service sowie eine Datenbankanbindung zu mehreren MariaDB-Instanzen. Dieser Aufbau wird durch das Komponentenmodell aus Abbildung 2 noch einmal verdeutlicht.

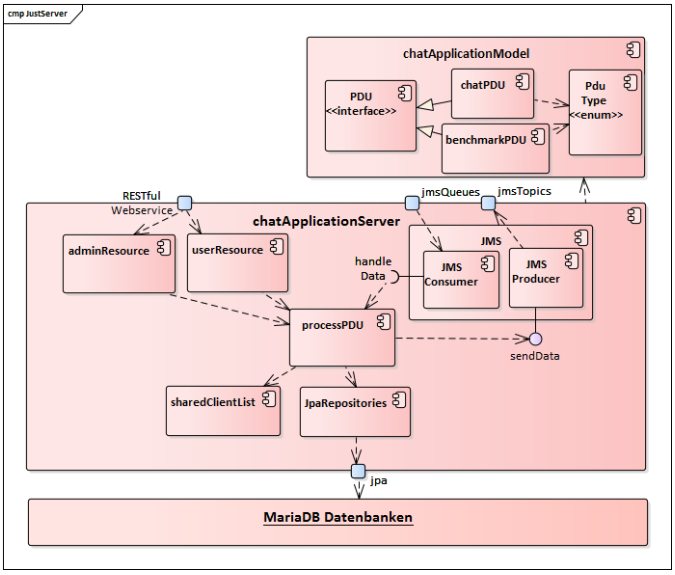


Abbildung 2: Komponentenmodell der serverseitigen Anwendung

2) *Client und Konnektoren*

Anders als die Serverkomponente wurde der Client modularer konzipiert. Aus Sicht der Verteilung handelt es sich hierbei um eine einzelne Client-Komponente, die am Ende eines Build-Prozesses entsteht. Der Code soll dafür in komplett unabhängigen Projekten entwickelt werden. Hiermit konnten durch die striktere Umsetzung des Seperation-of-Concern-Prinzips[3], einige Vorteile gewonnen werden. Dazu zählen die Wiederverwendbarkeit und einfache Austauschbarkeit durch die konsequente Trennung der Verantwortlichkeiten. Die bereits zuvor beschriebene Ausgliederung des Datenmodells ist hierbei nur ein Teil der Modularisierung, welche in Abbildung 3 dargestellt ist. Während die Client-Komponente schlank gehalten ist und hauptsächlich nur die Masken, ein UI-Model, Navigations- und Validierungslogik enthält, ist jegliche Kommunikation den Konnektor-Komponenten überlassen.

Diese Konnektoren sind rein technische Komponenten ohne jegliche Fachlogik[4]. Sie sind generisch und über Schnittstellen für jeden Client individuell konfigurierbar. Dies führt zu einer verbesserten Wiederverwendbarkeit, welche bspw. bei der Implementierung des Benchmarking-Clients oder auch potenzieller zukünftiger neu entwickelter Clients zum Tragen kommt. Außerdem ist somit eine vereinfachte Austauschbarkeit der Kommunikationsmittel gegeben, sollte die Anbindung

[2] 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.

[3] Das Prinzip „Seperation of Concerns“ beschreibt die Trennung zwischen Verantwortlichkeiten, Zuständigkeiten oder Aufgaben [2]. Dies wurde weitestgehend versucht, auf allen Ebenen des Entwurfs und der Implementierung einzuhalten.

[4] Darunter ist die reine Fachlogik zu verstehen. Funktionen, wie eine konsistente Fehlerbehandlung und einem entsprechenden Retry-Mechanismus, sind ebenfalls in diese Komponente ausgelagert.

an den RESTful Webservice oder JMS durch eine andere Kommunikationsart in bestehenden oder neuen Clients ersetzt werden. Dieses Vorgehen kommt ebenfalls der Wart- und Änderbarkeit zugute, da die Kommunikation übergreifend zentral und frei von Redundanzen an den Konnektoren vorgenommen werden kann.

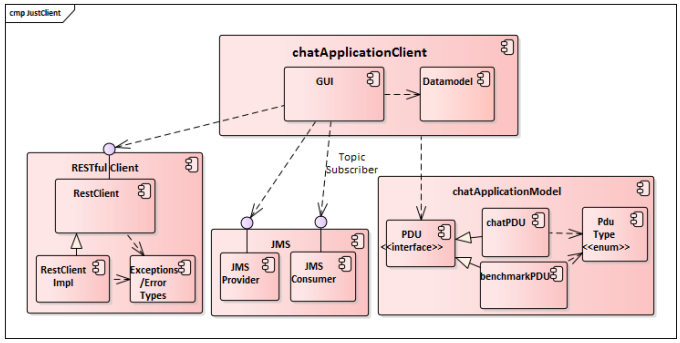


Abbildung 3: Komponentenmodell des Clients mit Konnektoren

3) *Benchmarking-Client*

Für die Durchführung eines Benchmarking-Tests ist ein separater Baustein, der Benchmarking-Client, implementiert. Anders als der anfänglich bereitgestellte Benchmarking-Client ist dieser neu implementiert und besitzt keine Abhängigkeiten auf den zuvor beschriebenen Chat-Client. Wie der Chat-Client, nutzt auch der Client für das Benchmarking lediglich die Konnektoren für die Anbindung an den RESTful Webservice und an JMS.

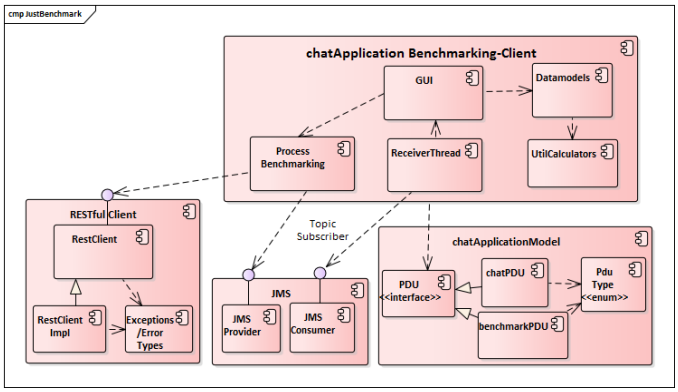


Abbildung 4: Komponentenmodell des Benchmarking-Clients

C. *Datenmodell*

Das umzusetzende Datenmodell ist hierbei sehr simpel. Es handelt sich, wie in Abbildung 5 dargestellt, um zwei Entitäten ohne Beziehungen. Während die TraceDB vor allem auf das Persistieren der Nachrichten ausgelegt ist, speichert die CountDB die Anzahl der Nachrichten gruppiert nach den Chatteilnehmern. Die Komplexität ergibt sich im Rahmen der Arbeit lediglich aus der Umsetzung einer verteilten Transaktion, denn die beiden Tabellen liegen in getrennten Datenbankinstanzen. Eine entsprechende Umsetzung dieser Transaktionsklammer und der entsprechenden Fehlerbehandlung ist nachfolgend im Rahmen der Implementierung beschrieben.

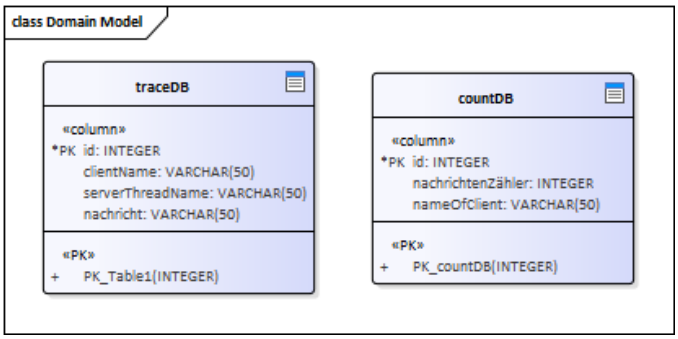


Abbildung 5: Domainmodell der serverseitigen Datenbankschicht

D. *Fehlertolerante Transaktionsverarbeitung für REST-Clients*

Der zuvor in Abschnitt 3 beschriebene REST-Konnektor soll eine konsistente Fehlerbehandlung bieten und fallweise eine Wiederholung die Anfragen automatisiert an den Server senden. Dieser Vorgang mit einer möglichen Wiederholung ist in Abbildung 6 mittels einer Sequenz dargestellt. Die durch eine maximale Anzahl beschränkten Wiederholungen werden hierbei durch bestimmte HTTP-Fehlercodes oder Timeout-Exceptions ausgelöst.

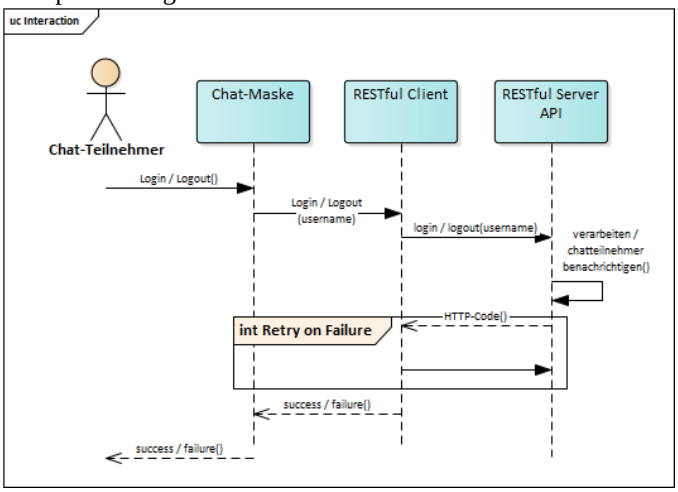


Abbildung 6: Login/Logout mittels REST und Retry-Mechanismus.

IV. Implementierung

Im Folgenden wird auf die Umsetzung der vorab beschriebenen Architektur eingegangen. Hierfür sind zunächst Entwurfsentscheidungen für die Basisarchitektur festgehalten, bevor im Anschluss auf detaillierte Implementierungsdetails eingegangen wird.

A. *Umsetzung der Basisarchitektur*

Bevor die Umsetzung der einzelnen Komponenten beschrieben wird, sind zunächst das verwendete Zielsystem und entsprechende Entwurfsentscheidungen festgehalten.

1) *Das Zielsystem und dessen Vorzüge*

Die ursprüngliche Anforderung an die Chatapplikation definierte den Wildfly 8.2 als zu verwendendes Zielsystem. Unter Absprache wird jedoch im Nachgang der Umstieg auf eine aktuellere Version des Wildfly in Betracht gezogen und

letztendlich umgesetzt. Für einen Umstieg auf die aktuelle Version 10.1.0 gelten vor allem die nachfolgend zusammengefassten Vorzüge nach [8], [9]:

- Performancesteigerung durch den Support des aktuelleren Protokolls HTTP/2.[5]
- 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.
- 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 erleichtert eine mögliche Migration auf Java 9, da diese bereits voll unterstützt wird.
- Es wird die zum Zeitpunkt der Erstellung dieser Arbeit aktuellste Version 5 des ORM-Frameworks Hibernate unterstützt. Dies führt zu einer verbesserten Nutzung und möglichen Performancegewinnen.[6]

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 XML vorgenommen werden.

2) Das Build- und Deployment-Management

Mittels des Einsatzes eines Dependency- und Build-Management-Tools, wie des verwendeten Apache Maven, bringt die neue Chatanwendung viele Vorteile mit sich. Hierzu zählt neben dem 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.[7]

3) Anbindung der Datenbankinstanzen

Die Anbindung der beiden Datenbanken CountDB und TraceDB wird außerhalb des Codes unter Verwendung der Datei „persistence.xml“ konfiguriert. Dadurch ist die Konfiguration vom restlichen Code entkoppelt und zentralisiert.

[5] Hiermit wird die Komprimierung der Header-Informationen ermöglicht und damit die Latenz verringert [10].

[6] 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 [11].

[7] Dies wird mittels einer Konfiguration in der Maven-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 das Deployment darauf durch.

Für die Umsetzung der Persistenz werden nach dem Modell der Container Managed Persistence (CMP) [12] die Java Persistence API (JPA) [13] und Entity Beans verwendet. Die Anwendung von CMP bietet sich in diesem Falle an, da die zu implementierende XA-Transaktion mehrere Entity-Beans und damit zusammenhängend zwei JTA-Transaktionen umfasst. JTA steht hierbei für Java Transaction API. CMP übernimmt dabei die Koordination der einzelnen JTA-Transaktionen und reduziert somit den Entwicklungsaufwand. Zur Realisierung der CMP kommt der Transaction-scoped Persistence Context zum Einsatz. Demnach ist die Lebensdauer des Kontexts auf eine einzelne JTA-Transaktion beschränkt.

B. Ablösung des TCP-Websocket durch JMS und einen RESTful Webservice

Wie einleitend bereits beschrieben, gilt es das System in eine Message-Service-Architektur zu überführen. Hierfür kommen einige Technologien zum Einsatz, die durch den JavaEE-Stack spezifiziert und durch den Applikationsserver unterstützt werden. Neben JPA und JTA werden ebenfalls der bereits beschriebene JMS sowie die Spezifikation JAX-RS, zur Umsetzung der Anforderungen eingesetzt und werden deshalb in den nachfolgenden Abschnitten beschrieben. Grundlegende JavaEE-Technologien wie die Contexts and Dependency Injection (CDI) sowie die Enterprise-Java-Beans (EJB) werden ebenfalls verwendet und unterstützen unter anderem die Umsetzung der Funktionalitäten durch Container-, Context- und Bean-Management [3]. Mittels Dependency Injection werden die Abhängigkeit des Codes von der Umgebung reduziert [14].

1) Implementierung JMS

Es gilt den TCP-Websocket zu entfernen und mittels Queue und Topic eine asynchrone Kommunikation einzuführen. Dies wurde unter anderem mit dem Einsatz serverseitiger Message-Driven-Beans (MDB) zur Anbindung an die Queue realisiert. Diese reagieren durch einen verwalteten Threadpool auf eingehende Nachrichten in der Queue und bedienen sich eines gemeinsamen Message-Driven-Context [15]. Mittels dieses Kontexts kann die Chatanwendung bei auftretenden Problemen ein mögliches Rollback durchführen.

Zu Beginn erzeugt der Wildfly 10 standardmäßig einen Threadpool mit 20 Threads [16]. Zur Verarbeitung greifen diese Threads wiederum auf Stateless-Session-Beans zu, die die weiterfolgende Verarbeitung übernehmen. Die Größe des Threadpools ist allerdings konfigurierbar und wurde im Rahmen der Performance-Optimierungen angepasst.

2) Implementierung der RESTful Webservices

Die Umsetzung der RESTful Webservices erfolgt nach der *Java API for RESTful Web Services* (JAX-RS) in der Version 2.1, welche in der Spezifikation JSR-339 standardisiert worden ist [17]. Hierbei können per Annotationen Ressourcen definiert, Pfade vergeben und somit Methodenaufrufe an entsprechende URIs gebunden werden. Wie in den Anforderungen genannt, werden hiermit die Schnittstelle für den Admin-Client sowie für den Login-Mechanismus umgesetzt. Dabei nutzt die Chatanwendung die durch den Applikationsserver bereitgestellte Implementierung RESTEasy. Diese setzt ähnlich zu der Referenzimplementierung Jersey alle grundlegen-

den Spezifikationspunkte um [18] und ist für den Anwendungsfall ausreichend.

C. 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 [19].

Je nachdem wo der Ursprung des Aufrufs der Transaktionsmethoden (begin, commit, etc.) liegt, unterscheidet man client-managed, container-managed und bean-managed Transaktionen. In diesem Fall wird eine container-managed Transaktion eingesetzt. Das bedeutet, dass der EJB-Container für das Setzen der Transaktionsgrenzen verantwortlich ist. Für das Anwendungsszenario werden diese Grenzen als Teil des Message-Driven-Context durch das EJB-Transaction-Management verwaltet [20]. Wird während der Verarbeitung einer Nachricht eine Exception geworfen, wird diese unabhängig von ihrem Ursprung bis in die Message-Driven-Bean hochgereicht, von dort aus die Transaktion für ein Rollback zu markieren. Das Rollback hat zur Folge, dass die Nachricht, deren Verarbeitung gescheitert ist, wieder in der Queue abgelegt wird und dort auf eine erneute Verarbeitung wartet. Sämtliche Datenbankaktionen die vor der Exception durchgeführt wurden, werden demnach nicht committet und es kommt zu keiner endgültigen Persistierung. Somit wird mithilfe der verteilten XA-Transaktion die Konsistenz der Datenbestände zwischen TraceDB und CountDB sichergestellt.

Für die Implementierung verteilter Transaktionen mit EJB wird ein Transaktionsmanager benötigt, welcher durch Wildfly bereitgestellt wird. Dessen Schnittstelle wird durch JTA definiert. JTA basiert auf dem XA-Standard, der den Rahmen für die umzusetzende Transaktion bildet [21].

Wichtig ist zudem, dass auch die in die Transaktion eingebundenen Ressourcen XA unterstützen. Das ist für die MariaDB-Instanzen unter Verwendung der Datenbank-Engine InnoDB gegeben [22].

Für die Realisierung der verteilten XA-Transaktionen muss jede der Datenbankinstanzen als XA-Datasource am Wildfly definiert werden. Aus Gründen der Flexibilität werden diese nicht statisch am Wildfly konfiguriert, sondern während des Deployments anhand der Konfigurationen in der mysql-ds.xml angelegt.

D. Umsetzung der Clients mit neuer UI

Aufgrund der neuen Anforderungen an die Chatapplikation und der neuen Architektur, wurden der Chat-Client und der Benchmarking-Client von Grund auf neue geschrieben. Hierfür wurde alter Code beseitigt oder stark modifiziert, um die Clients kompakter und moderner zu gestalten. Es wurden ebenfalls die JavaFX-Masken von Grund auf neu implementiert, um eine erhöhte Benutzerfreundlichkeit durch eine moderne grafische Benutzeroberfläche zu erreichen. Diese sind in Anhang X.C.1) dargestellt. Vor allem der Benchmarking-Client profitiert von dieser Maßnahme, da die alte Maske noch keine Trennung von Code und UI vorsieht. Die neu entwickelte Maske ist durch das Konzept der *fxml*[8] entkoppelt und bietet einige neue Funktionalitäten. Diese umfasst unter anderem die Berechnung und Ausgabe verschiedener Kenngrößen und das Aufbereiten von vordefinierten Diagrammen während des Benchmarkings in Echtzeit, um ein entsprechendes Ergebnis direkt zu visualisieren. Diese Masken sind ebenfalls beispielhaft in Anhang X.C.2) beigefügt.

E. Umsetzung des 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 der erwähnten REST-Schnittstelle kommunizieren und aktuelle Serverdaten aus den Datenbanken auslesen sowie darstellen. Hierzu zählen:

- aktuell angemeldete Chat-Benutzer
- Anzahl der Nachrichten pro Chat-Benutzer
- verschiedene statistische Daten

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

1) Angular 2 als Entwurfsentscheidung

Das Web-Application-Framework Angular wurde am 14. September 2016 in der Version 2.0 veröffentlicht [24]. 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 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 (AngularJS) zu integrieren. Hierbei wurde insbesondere Wert darauf gelegt, dass die Produktivität des Entwicklers erhöht werden kann, indem ihm mehr Unterstützung durch die Entwicklungsumgebung selbst sowie durch die Programmiersprache TypeScript garantiert wird [25]. 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. 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 AngularJS zurück portiert wurde [25].

Diese Vorteile gegenüber anderen modernen Web-Application-Frameworks und weitere Verbesserungen gegenüber AngularJS - insbesondere im Bereich der Performance – haben das Projektteam zu der Entwurfsentscheidung veranlasst, den Admin-Client mit diesem Web-Application-Framework zu entwickeln.

[8] Unter dem Konzept der *FXML* ist eine auf XML beruhende JavaFX-Maskenstrukur zu verstehen. Mittels der Entkopplung vom Code lässt sich die Oberfläche in einer baumartigen Hierarchiestruktur mittels Tools wie bspw. dem Scene Builder von Oracle erstellen [23].

2) Grafische Oberfläche des Admin-Clients

Um die Benutzeroberfläche grafisch ansprechend gestalten zu können, wurde auf das Framework Angular Material 2 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 [26]. Da die für die Entwicklung des Admin-Clients notwendigen GUI-Elemente bereits in der Entwicklungsversion unterstützt sind, hat das Projektteam die Verwendung von Angular Material 2 entschieden.

3) Aufbau und Architektur des Admin-Clients

Die Architektur wurde aufgrund der geringen funktionalen Komplexität des Admin-Clients ebenfalls bewusst einfach gehalten, um keine unnötige Komplexität aufzubauen („Over-Engineering“) [27].

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

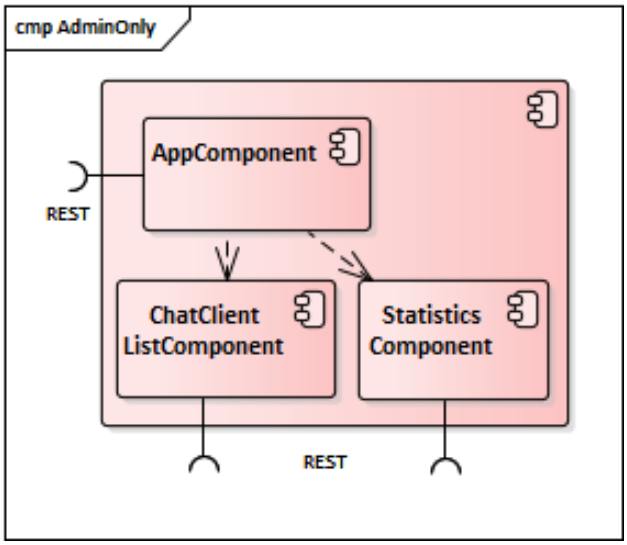


Abbildung 7: Architektur des Admin-Clients. Die obligatorische AppComponent greift für die Darstellung weiterer Daten auf die ChatClientListComponent und die StatisticsComponent zurück.

Aus architektonischer Sicht (s. Abbildung 7) besteht die Applikation aus einer übergreifenden *AppComponent*, welche die Darstellung der Reiter 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 von der Serverschnittstelle ab und stellt sie ebenfalls im entsprechenden Reiter 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 über die REST-Schnittstelle der Serverkomponente 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 Haupteinsatzzweck von Services entspräche [28].

Somit handelt es sich bei dem implementierten Admin-Client um ein einfaches aber funktionelles System, bei welchem jedoch durch die Entwurfsentscheidung Angular 2 die Option besteht, dieses in seiner Komplexität stark zu erweitern und architektonisch entsprechend zu optimieren.

F. Weitere Umsetzungsmerkmale

Neben den bereits erläuterten Veränderungen weist die beiliegende 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, wodurch sich die Ausführung der Anwendung auf Windows-Rechnern einfacher gestaltet.
- Neben der EXE-Datei wird ein *Fat-Jar* generiert, das alle Abhängigkeiten und Komponenten der jeweiligen Clients enthält und plattformunabhängig mittels Java gestartet werden kann.
- Verschiedene Fremdbibliotheken sind auf die zum Zeitpunkt der Erstellung dieser Studienarbeit aktuellen Versionen geupdated worden.
- Sowohl die JMS- als auch die REST-Kommunikation werden durch automatisierte Integrationstests bei jedem Build-Vorgang qualitätsgesichert.

V. BENCHMARKING

Als Teil der Anforderung sind Performance-Tests vorgesehen. Das Vorgehen, entsprechende Ergebnisse und Auswertungen sind daher nachfolgend zusammengefasst.

A. Testaufbau und Rahmenbedingung

Für einen Performance-Test sollte das Verhalten des Servers bei einem hohen Kommunikationsaufkommen von Chatnachrichten untersucht werden. Dies umfasst neben dem Versenden von Nachrichten über JMS ebenfalls den Login und Logout über die RESTful API. Es wurde daher die im Folgenden beschriebene Infrastruktur gewählt.

1) Verwendete Infrastrukturkomponenten

Im Zuge des Projektes kam es zu Komplikationen mit der ursprünglich angedachten Testumgebung. Die vorgesehenen VMs konnten nach einigen Ausfällen und Fehler nicht als zuverlässige Umgebung genutzt werden[9]. Im Rahmen des Benchmarkings wurde daher eine verteilte Infrastruktur mittels LAN aufgebaut. Diese Verbindung über LAN wird bei jedem Informationsfluss zwischen der Hardware aus Abbildung 8 herangezogen.

[9] Dies wurde durch mehrere Teams festgestellt, weshalb unter Rücksprache nach einer alternativen Testumgebung gesucht wurde.

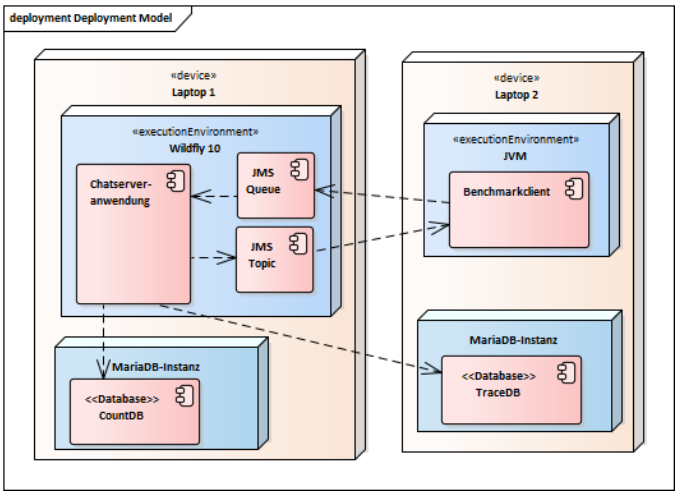


Abbildung 8: Verteilung der Testumgebung

Diese dargestellte Hardware untergliedert sich hierbei wie folgt:

Laptop 1:

- CPU: Intel Core i5 7200U - 2,5GHz
- Arbeitsspeicher: 8 GB DDR4 RAM
- Betriebssystem: Windows 10 Pro
- Prozessorarchitektur: x64
- Festplatte: 500 GB HDD (5400U/min)

Laptop 2:

- CPU: Intel Core i5 4200U - 1,6GHz
- Arbeitsspeicher: 8 GB DDR3 RAM
- Betriebssystem: Windows 7 Pro
- Prozessorarchitektur: x64
- Festplatte: 500 GB HDD (5400U/min)

Die Kommunikation der verteilten Komponenten erfolgte im Rahmen des Tests per LAN, wobei die Verbindung mittels eines CAT.5e-Kabel hergestellt wurde.

2) Maßnahmen zur Performanceoptimierung

Im Rahmen der Arbeit ist die Durchführung eines Benchmarkings gefordert. Aufgrund dessen wurde die Performance als eine der nicht-funktionalen Anforderungen festgelegt und für die Performance-Steigerung entsprechende Optimierungsmaßnahmen angestrebt. Gleichzeitig kann es damit auch zu negativen Auswirkungen auf andere nichtfunktionale Aspekte kommen, weshalb mögliche Vorgehen, wie die Konfiguration von flüchtigen, nicht persistenten und „durable“ Queues, Nachrichten und Topics vermieden wurde. Das Senden von Nachrichten wurde bspw. weiterhin auf "PERSISTENT" belassen und nicht auf "NON_PERSISTENT" gestellt. Dies hätte zur Folge, dass alle Nachrichten zu Gunsten der Performance im Hauptspeicher gehalten werden, diese jedoch bei einem Neustart des Servers verloren gehen. Auf diese Optimierung wurde daher verzichtet, um die Ausfallsicherheit nicht zu verringern [29].

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 „Controlling Message Acknowledgement“-Konfiguration auf die Option „DUPS_OK_ACKNOWLEDGE“ umgestellt. Hiermit werden Bestätigungen nur „*lazy*“ versendet.[10] Dies kann zu möglichen Nachrichtenduplikaten führen. Solche Duplikate konnten allerdings in Tests nicht reproduziert werden und werden daher in möglichen Ausnahmefällen in Kauf genommen [30].
- Statt den einfach zu implementierenden Objektnachrichten wurde nachträglich auf Textnachrichten umgestellt. Hiermit wird eine Verkleinerung der Nachrichtangestrebt. Dies wird mittels der Serialisierung der Transportobjekte in JSON-Strings angestrebt. [11]
- Die, für JMS-Nachrichten automatisch erzeugten Nachrichten-IDs und Zeitstempel, wurden 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-Konfigurationen erhalten diese ebenfalls 2 GB Arbeitsspeicher und eine größere Puffer-Konfiguration, um eine größere Verarbeitung im Hauptspeicher zu erzielen.
- Die Thread-Anzahl der MDB des Wildflys wurde erhöht.
- Wildfly unterstützt ein Bean-Pooling, mittels dessen vorab ein Pool an EJBs erzeugt wird. Dieser Pool wurde erhöht, da gleichzeitig die Zahl der arbeitenden Threads erhöht wurde.

3) Verwendete Testmetriken

Metrik	Beschreibung
ØRTT gesamt	Wie lange braucht eine Nachricht im Durchschnitt, bis sie wieder am Client ankommt?
RTT max	Wie lange braucht die Nachricht mit der größten Round-Trip-Time?
RTT min	Wie lange braucht die Nachricht mit der

[10] Tests mit der Chatanwendung haben bestätigt, dass die Queue weniger ausgelastet ist und damit weniger Timeouts auftreten, wodurch sich erhebliche positive Auswirkungen auf die Performance ergeben.

[11] Statt den Standard von JAX-B (Java Architecture for XML Binding) zu verfolgen und das Objekt nach XML zu serialisieren, wurde auf JSON gesetzt, da dieses Format in der Regel leichtgewichtiger ist und damit mit einer höheren Performance zu rechnen ist [31], [32]. Hierfür kommt das Framework Jackson zum Einsatz [32].

	geringsten Round-Trip-Time?
ØRTT Server	Wie lange braucht der Server im Durchschnitt für die Verarbeitung einer Nachricht?
RTT SD	Wie groß ist die Standardabweichung der Round-Trip-Time?
ØCPU	Wie hoch ist die durchschnittliche CPU-Auslastung des Servers?
Freier Speicher	Wie viel Hauptspeicher ist während der Verarbeitung minimal verfügbar gewesen?

Tabelle 1: Verwendete Testmetriken und Beschreibung

4) Testspezifikation

Die Tests werden mithilfe des Benchmarking-Clients durchgeführt. Durch die Neuimplementierung ergibt sich u.a. eine Änderung im Ablauf der Benchmarking-Tests. Die im Rahmen des Tests gestarteten Clients beginnen nicht wie zuvor unmittelbar nach dem Initialisieren mit dem Versand der Nachrichten. Erst wenn alle für den Test erforderlichen Clients verfügbar sind, wird der Versand der Nachrichten zeitgleich durch alle Clients und eine Messung der Zeit gestartet. Außerdem erzeugt jeder Benchmarking-Client bereits vorab eine PDU, die nach dem simultanen Start für jede Nachricht herangezogen und lediglich mit einer Startzeit (System-Nanotime) befüllt wird. Gleichzeitig wartet der Client nicht nach jeder Nachricht auf eine Bestätigung der Verarbeitung wie es der ursprüngliche Benchmarking-Client handhabt. Das führt im Vergleich zur vorherigen Implementierung zu einer höheren Belastung der Queue sowie der Threads, die die in der Queue befindlichen Nachrichten serverseitig abarbeiten. In Abbildung 9 wird dies nochmal verdeutlicht. Zum Vergleich liegt im Anhang X.D die Abbildung 18, welche das Vorgehen des ursprünglichen Benchmarking-Clients zeigt.

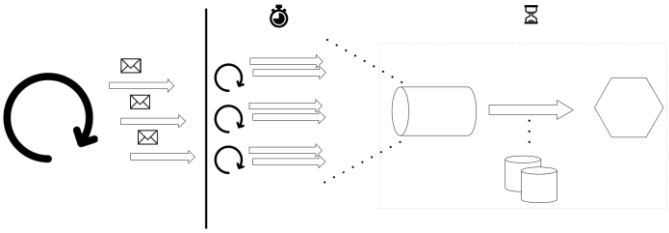


Abbildung 9 Ablauf des neuimplementierten Benchmarking-Tests

Ausgehend vom geforderten Testfall mit 10 Clients, die je 100 Nachrichten mit 50 Byte Länge versenden, wurden vier weitere Tests durchgeführt. Die Anzahl der Nachrichten sowie deren Länge wurden beibehalten, die Anzahl der Clients hingegen schrittweise erhöht. Es wurden Tests mit 10, 30, 50, 75 und 100 Clients durchgeführt.

B. Darstellung der Messergebnisse

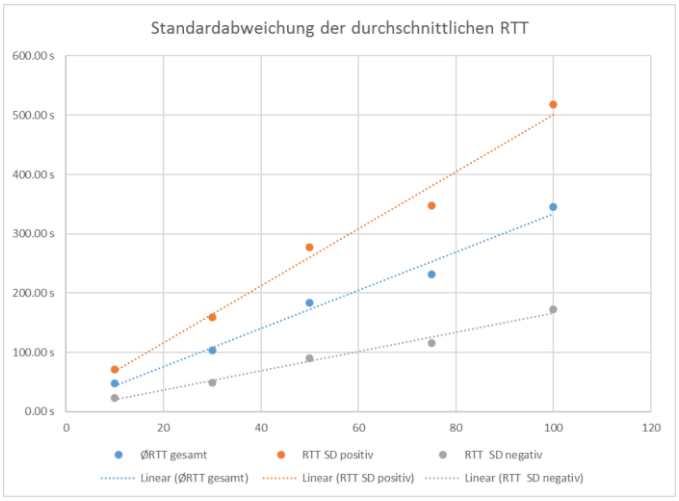


Abbildung 10: Durchschnittliche RTT und Standardabweichung aller durchgeführten Benchmarking-Tests inkl. Regressionslinien; auffälliger linearer Zusammenhang zwischen Anzahl der Chat-Clients und RTT

Abbildung 11 Verlauf der RTT, 10 Clients; linearer Anstieg der RTT mit fortschreitender Nachrichtenverarbeitung

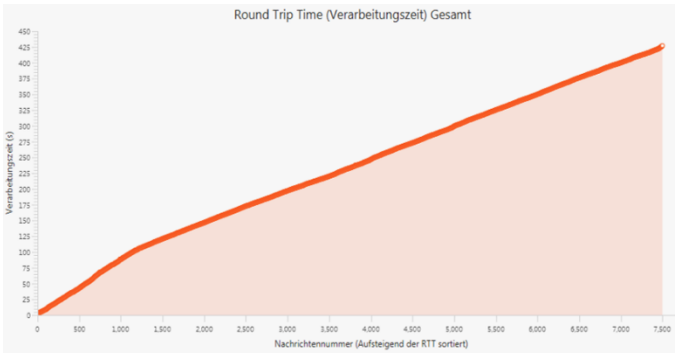


Abbildung 12: Verlauf der RTT, 75 Clients; ebenfalls linearer Anstieg der RTT mit fortschreitender Nachrichtenverarbeitung

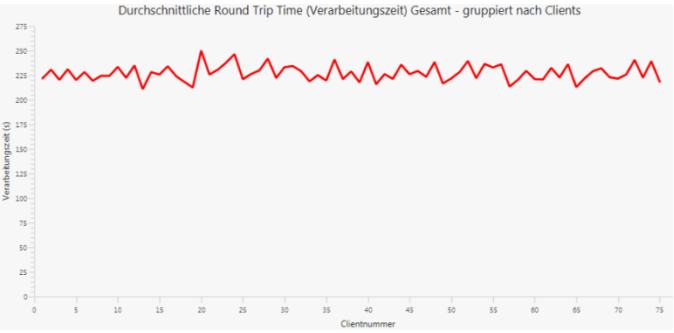


Abbildung 13: Durchschnittliche RTT über alle Nachrichten pro Client, 75 Clients; Linie zwischen diskreten Datenpunkten nur zur Veranschaulichung, keine Interpolation

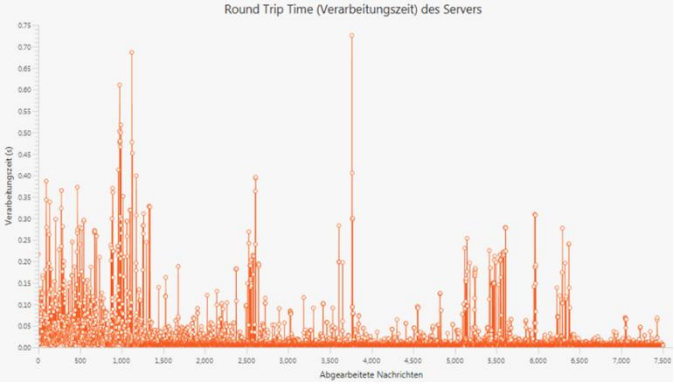


Abbildung 14: Serverseitige Verarbeitungszeit pro Nachricht, 75 Clients; meist gleichbleibend unter 0,05s mit gelegentlichen Ausreißern bis zu ca. 0,75s

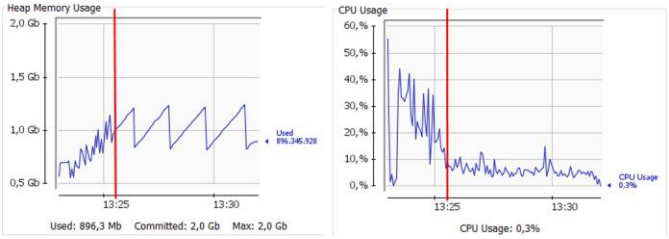


Abbildung 15: Speicher- und CPU-Auslastung durch den Server, 75 Clients; rote Linie unterteilt die beiden Phasen des Testverlaufs (s. Erläuterung)

C. *Interpretation der Testergebnisse*

Basierend auf den dokumentierten Messergebnissen können verschiedene Aussagen getroffen werden.

Unter Verwendung von Microsoft Excel wurde ein Teil der mit dem Benchmarking-Client erhobenen Daten ausgewertet. Diese sind Anhang X.D.2) zu entnehmen. In Abbildung 10 ist für jeden Test die durchschnittliche Round-Trip-Time und deren Standardabweichung dargestellt.

Die Trendlinie ergibt sich mithilfe linearer Regression. Auffallend ist die lineare Entwicklung der durchschnittlichen Round-Trip-Time und deren Standardabweichung über alle Tests. Übertragen auf das gesamte Testszenario kann anhand dessen die Annahme getroffen werden, dass die Anwendung bei steigender Anzahl von Clients und Nachrichten für den untersuchten Bereich linear skaliert.

Diese Annahme wird ebenfalls durch den Verlauf der Round-Trip-Time während der einzelnen Tests bestätigt. Wie aus Abbildung 11 und Abbildung 12 hervorgeht, steigt die RTT linear mit der Anzahl der verarbeiteten Nachrichten im Laufe eines Tests. Hierfür ist im Wesentlichen die Zeit verantwortlich, während der die Nachrichten auf ihre Entnahme aus der Queue durch einen Thread zur Abarbeitung warten. Die Round-Trip-Time der letzten Nachricht ist demnach die höchste. Aus dem nahezu konstanten Anstieg der RTT lässt sich daher eine in etwa konstante Abarbeitungsgeschwindigkeit jeder Nachricht auf dem Server nach Entnahme aus der Queue folgern.

Aus Abbildung 13 geht hervor, dass keiner der Clients auffallend bevor- oder benachteiligt wird, da die Round-Trip-Time für alle Clients selbst über eine hohe Anzahl von Nachrichten in etwa identisch ist. Hieraus lässt sich schlussfolgern, dass für die Abarbeitung der Nachrichten eines jeden Clients in etwa die gleiche Menge von Ressourcen allokiert wird.

Abbildung 14 zeigt die serverseitige Verarbeitungszeit für den Benchmarking-Test mit 75 Clients. Als Erklärung für die Spitzen bei einzelnen Nachrichten könnten zeitgleiche Zugriffe mehrerer Threads auf unterschiedliche Nachrichten desselben Clients herangezogen werden. Beim Persistieren in die Datenbanken werden die zeitlich nachfolgenden Threads zunächst blockiert, bis der aktive Thread seine Änderungen committet hat, um die Konsistenz des Datenbestandes zu gewährleisten. Daraus ergibt sich eine längere Laufzeit der zunächst blockierten Threads und damit verbunden eine höhere serverseitige Verarbeitungszeit.

Abbildung 15 zeigt im Wesentlichen auf, dass sich der Ressourcenverbrauch durch den Server im Testverlauf grob in zwei Phasen unterteilen lässt. In Phase 1 steigt der Speicherverbrauch in etwa linear an, bis er sich bei einem Niveau von 1 GB einpendelt. Die CPU-Auslastung durch den Server ist unstet im Bereich von 0-50% und verringert sich im Laufe der Phase im Mittel. Phase 2 zeigt mehrmals einen nahezu perfekt linearen Anstieg des Speicherverbrauchs mit anschließendem schnellen Abfall um ca. 0,3 GB. Dieses Muster wiederholt sich viermal im Laufe des Tests und lässt sich auf das Wirken des Garbage-Collection-Mechanismus zurückführen, welcher die allokierten Speicherbereiche aufräumt. Gleichzeitig befindet sich in dieser Phase jedoch die CPU-Auslastung meistens im Bereich von 0-10%. Weiterhin zeigt der Windows Ressource Manager, dass die Auslastung der Festplatte beinahe durchgehend bei 100% liegt (nicht abgebildet), wodurch die Schlussfolgerung naheliegt, dass bei diesem Setup die Festplatte hauptursächlich für die Einschränkung der Performance und die CPU unterausgelastet ist.

Zusammenfassend lässt sich die Aussage treffen, dass die Serverkomponente der Chatanwendung imstande ist, ohne sichtbare Performance-Einbußen mit einer Mehrzahl von Chat-Clients gleichzeitig zu interagieren und deren Nachrichten zu verarbeiten.

VI. FAZIT

Ziel dieser Arbeit war es, eine vorhandene, nachrichtenbasierte Java-Anwendung in ein verteiltes System zu überführen. Für die Kommunikation zwischen Clients und Server sollte JMS zum Einsatz kommen. Die serverseitigen Vorgänge sollten in einer XA-Transaktion zusammengefasst werden.

Als Herausforderungen stellten sich, wie zu erwarten, JMS und die XA-Transaktion heraus, da es sich in beiden Fällen

um neu zu erlernende Konzepte handelte. Jedoch stellte die jeweilige Dokumentation bzw. Spezifikation in Zusammenhang mit verschiedenen Tutorials eine geeignete Grundlage für die notwendige Einarbeitung dar.

Somit konnten alle Teile der zugrundeliegenden Aufgabenstellung erfüllt werden. Dementsprechend wurde die Umstellung von TCP auf JMS vorgenommen, RESTful Webservices für die Anmeldung der Benutzer und des Admins implementiert sowie die XA-Transaktion und der Admin-Client umgesetzt. Darüber hinaus konnten weitere Ergebnisse erzielt werden. So wurde beispielsweise die Oberfläche des Benchmarking-Clients überarbeitet, weitere umgesetzte Features und Details sind Abschnitt IV.F zu entnehmen. Zudem wurden, wie in Abschnitt V.A.2) dargestellt, verschiedene Maßnahmen zur Performanceoptimierung ergriffen.

Die abschließenden Benchmarking-Tests bestätigten die Skalierbarkeit und Robustheit der Anwendung.

Als Ergebnis dieser Arbeit stellt die entstandene Anwendung eine gute Grundlage für weitere Entwicklungsschritte dar. Denkbar wären beispielsweise Maßnahmen in Bezug auf die Anwendungssicherheit oder eine weitere serverseitige Modularisierung mithilfe von JBoss Swarm oder entsprechender alternativen Technologien. Möglich wäre darauf aufbauend auch die Umsetzung einer Container-basierten Architektur, beispielsweise unter Verwendung des Container-Managers Docker. Damit könnte ein positiver Einfluss auf die Resilienz, Skalierbarkeit und Verfügbarkeit erzielt werden.

VII. Abkürzungsverzeichnis

ACID	-	Atomicity Consistency Isolation Durability	2
API	-	Application Programming Interface	1
CDI	-	Contexts and Dependency Injection	5
CMP	-	Container Managed Persistence	5
EJB	-	Enterprise Java Beans	5
JAR	-	Java Archive	3
JMS	-	Java Message Service	1
JPA	-	Java Persistence API	5
JTA	-	Java Transaction API	5
JSON	-	JavaScript Object Notation	8
JSR	-	Java Specification Request	1
MDB	-	Message Driven Bean	5
MOM	-	Message Oriented Middleware	1
ORM	-	Object Relational Mapper	5
PDU	-	Protocol Data Unit	9
REST	-	Representational State Transfer	3
RTT	-	Round Trip Time	8
SD	-	Standard Deviation	9
TCP	-	Transmission Control Protocol	5
VM	-	Virtual Machine	7
WAR	-	Web Archive	3
XML	-	Extensible Markup Language	5

VIII. References

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

[2] G. Starke, *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag GmbH \& Company KG, 2015.

[3] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout, und N. Deakin, *The Java Message Service Specification*. Java Specification Request, 2013.

[4] P. Mandl, *Masterkurs Verteilte betriebliche Informationssysteme - Prinzipien, Architekturen und Technologien*, 1. Aufl. VIEWEG+ TEUBNER, 2009.

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

[6] „Two-Phase Commit Mechanism“. [Online]. Verfügbar unter: https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_txns003.htm#ADMIN12222. [Zugegriffen: 17-Dez-2016].

[7] „Distributed Logging for Transaction Processing“. [Online]. Verfügbar unter: http://www.cs.tufts.edu/~nr/cs257/archive/alfred-spector/spector85sigmod.pdf. [Zugegriffen: 17-Dez-2016].

[8] „WildFly 9 Final is released! · WildFly“. [Online]. Verfügbar unter: http://wildfly.org/news/2015/07/02/WildFly9-Final-Released/. [Zugegriffen: 20-Dez-2016].

[9] „WildFly 10 Final is now available! · WildFly“. [Online]. Verfügbar unter: http://wildfly.org/staging/news/2016/01/29/WildFly10-Released/. [Zugegriffen: 20-Dez-2016].

[10] 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].

[11] „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].

[12] „The Java Community Process(SM) Program - communityprocess - final“. [Online]. Verfügbar unter: https://jcp.org/aboutJava/communityprocess/final/jsr220/index.html. [Zugegriffen: 17-Dez-2016].

[13] „The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 338“. [Online]. Verfügbar unter: https://jcp.org/en/jsr/detail?id=338. [Zugegriffen: 17-Dez-2016].

[14] „Latest CDI 2.0 news | Contexts and Dependency Injection“. [Online]. Verfügbar unter: http://www.cdi-spec.org/. [Zugegriffen: 17-Dez-2016].

[15] „What is a Message-Driven Bean?“ [Online]. Verfügbar unter: https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs005.htm#CIHBIHAA. [Zugegriffen: 22-Dez-2016].

[16] „EJB3 subsystem configuration guide - WildFly 10 - Project Documentation Editor“. [Online]. Verfügbar unter: https://docs.jboss.org/author/display/WFLY10/EJB3+subsystem+configuration+guide. [Zugegriffen: 21-Dez-2016].

[17] Oracle Corporation, „JSR 339 - Java Community Process“. [Online]. Verfügbar unter: https://jcp.org/en/jsr/detail?id=339. [Zugegriffen: 16-Dez-2016].

[18] S. Gulabani, *Developing RESTful Web Services with Jersey 2.0*. Packt Publishing Ltd, 2014.

[19] „Distributed Transaction Processing: The XA Specification“. [Online]. Verfügbar unter: http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf. [Zugegriffen: 17-Dez-2016].

[20] „Configuring EJB 3.0 Transaction Management“. [Online]. Verfügbar unter: https://docs.oracle.com/cd/E14101_01/doc.1013/e13981/servtran001.htm. [Zugegriffen: 22-Dez-2016].

[21] „The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 907“. [Online]. Verfügbar unter: https://jcp.org/en/jsr/detail?id=907. [Zugegriffen: 17-Dez-2016].

[22] „XA Transactions“, *MariaDB KnowledgeBase*. [Online]. Verfügbar unter: http://mariadb.com/kb/en/mariadb/xa-transactions/. [Zugegriffen: 21-Dez-2016].

[23] R. Steyer, „Behind the scene – der Aufbau von FXML“, in *Einführung in JavaFX*, Springer Fachmedien Wiesbaden, 2014, S. 123–142.

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

[25] 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].

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

[27] „Code Simplicity » What Is Overengineering?“ [Online]. Verfügbar unter: http://www.codesimplicity.com/post/what-is-overengineering/. [Zugegriffen: 20-Dez-2016].

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

[29] Mastertheboss.com, „Configuring message redelivery on JBoss - WildFly“. [Online]. Verfügbar unter: http://www.mastertheboss.com/jboss-server/jboss-jms/configuring-message-redelivery-on-jboss-wildfly. [Zugegriffen: 18-Dez-2016].

[30] Oracle Corporation, „Controlling Message Acknowledgment (The Java EE 6 Tutorial)“. [Online]. Verfügbar unter: https://docs.oracle.com/cd/E19798-01/821-1841/bncfw/index.html. [Zugegriffen: 18-Dez-2016].

[31] „JAXB und Jackson: Speichern von Java Objekten als JSON“. [Online]. Verfügbar unter: https://www.kompf.de/java/jaxbjson.html. [Zugegriffen: 23-Dez-2016].

[32] „JAXB vs. GSON and Jackson – Thomas Uhrig“. .

IX. EIDESSTATTLICHE ERKLÄRUNG

Erklärung gemäß § 15 Abs. 10 APO i. V. m. § 35 Abs. 7 RaPO. Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Christina Eidelloth _____

David Sautter _____

Felix Stützinger _____

Maximilian Auch _____

MÜNCHEN, 23.12.2016

X. Appendix

A. *Gesamtarchitekturen im Vergleich*

Die in Abbildung 16 dargestellte Gesamtarchitektur der Chatanwendung stellt das Ergebnis der entwickelten Anwendung dar. Die Komponenten sind auf Klassenebene dargestellt bzw. bündeln Klassen in logische 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.

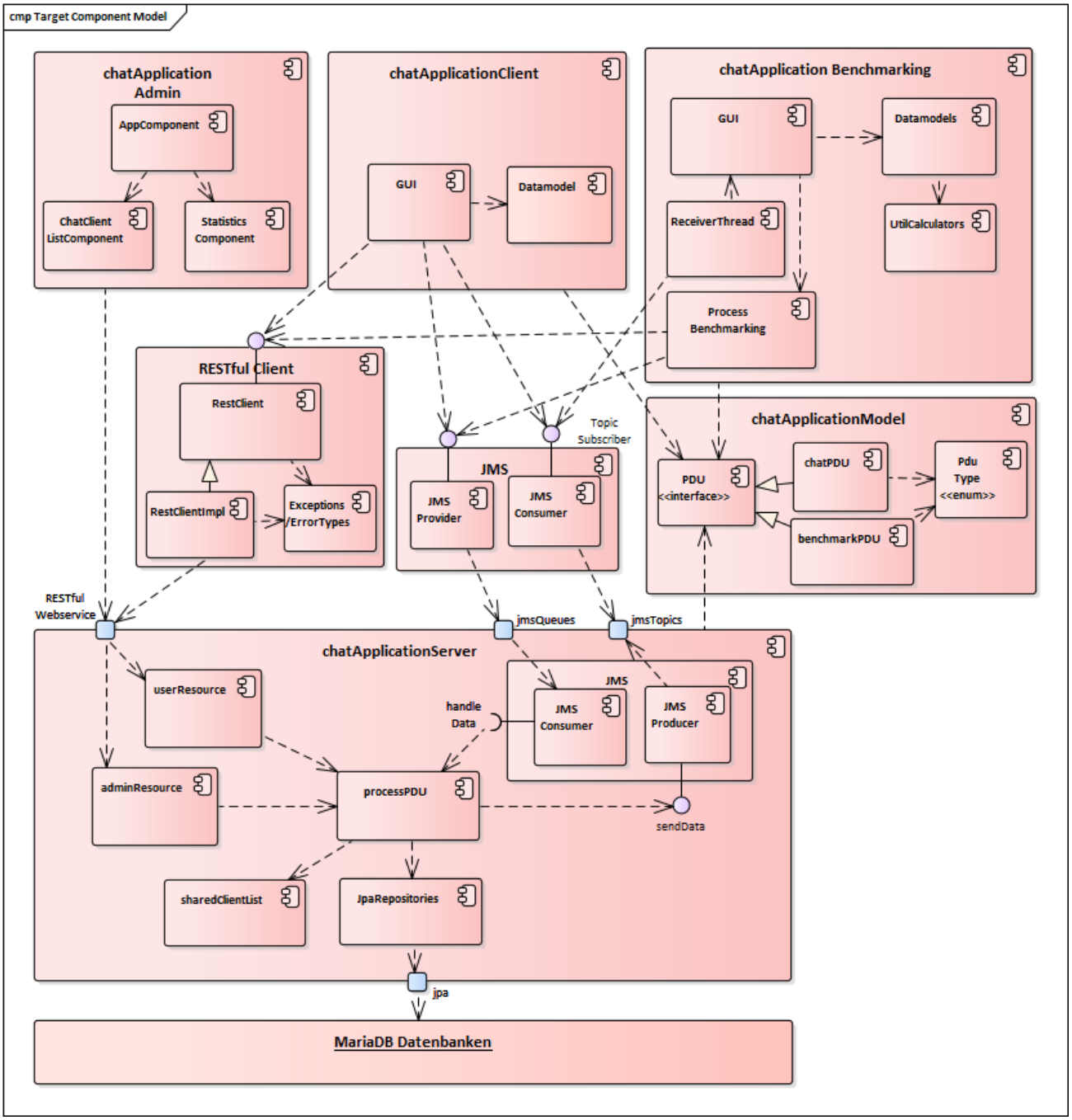


Abbildung 16: Anwendungsüberblick in Form eines Komponentenmodells

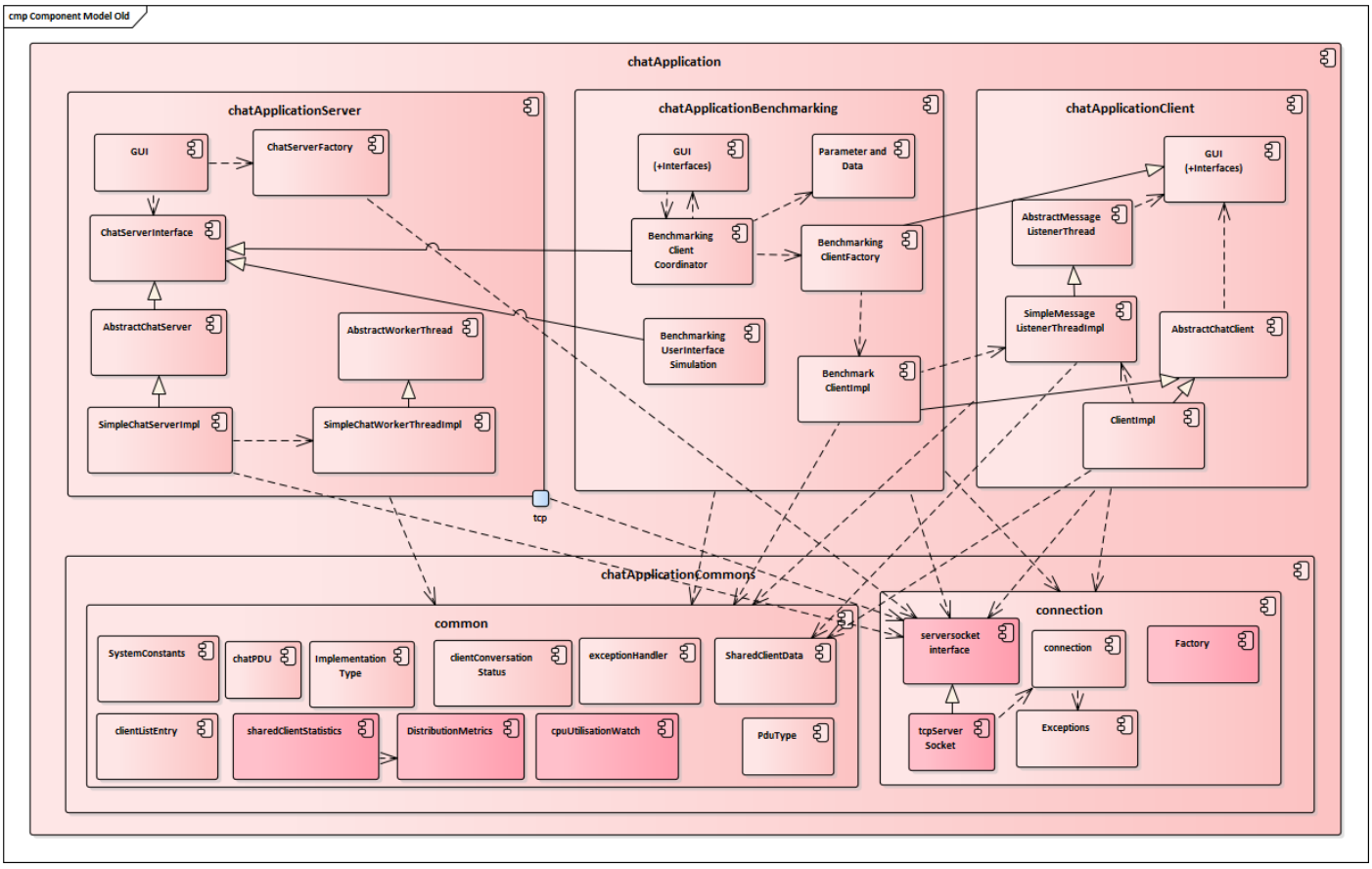


Abbildung 17: Anwendungsüberblick der Ausgangsanwendung in Form eines Komponentenmodells

B. Anhang Software

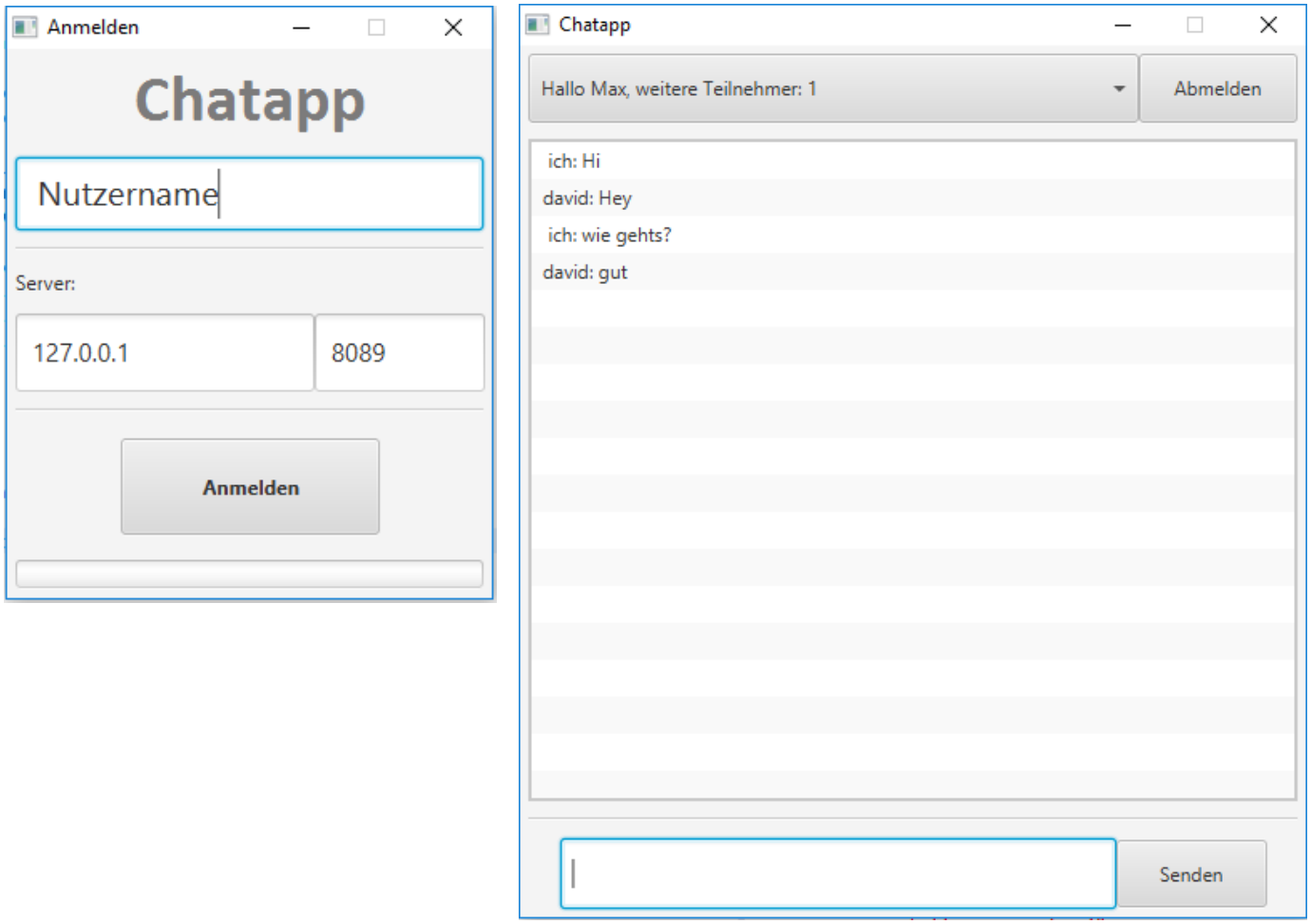
Neben den in der Arbeit beschriebenen Grundprinzipien, der Architektur und Implementierungen sowie den Testergebnissen sind ihr der Quellcode und Konfigurationsdateien angehängt. Der Anhang umfasst im Detail die folgenden Punkte:

- Alle Maven-Projekte der einzelnen Komponenten
- Alle Konfigurationsdateien des Wildfly 10
    - Standalone.xml
    - JAR-Datei des Mariadb-Java-Clients in der Version 1.5.4
- Alle Konfigurationsdateien der MariaDB
    - Exportierte Verbindungseinstellungen aus HeidiSQL
    - Initialisierungsdatei
- Angular2 Komponenten des Admin-Clients

C. Masken der Clients

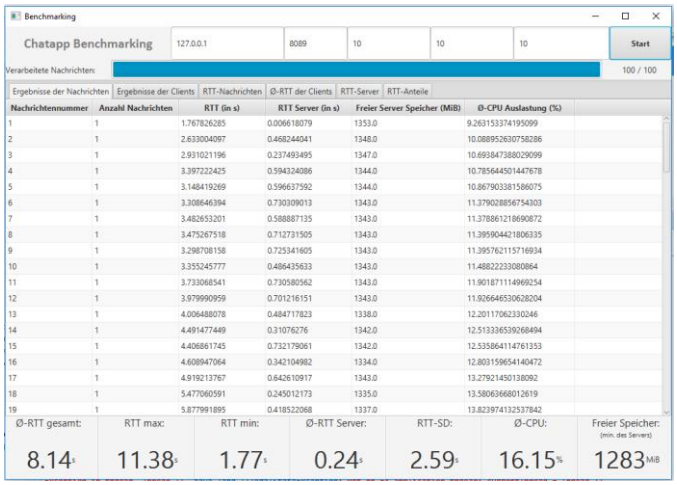
1) Chat-Client Masken

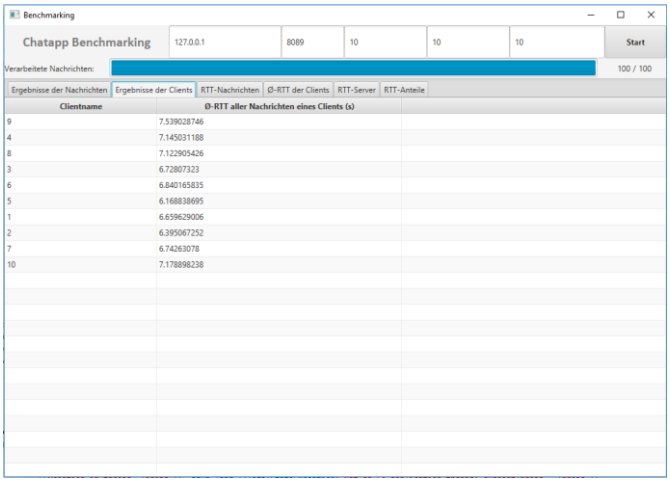
Die Masken des Chat-Clients wurden auf Benutzerfreundlichkeit vereinfacht und sehen wie folgt aus:

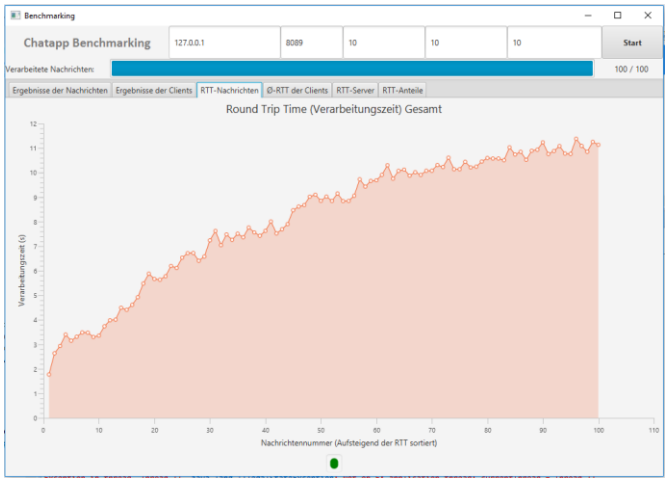


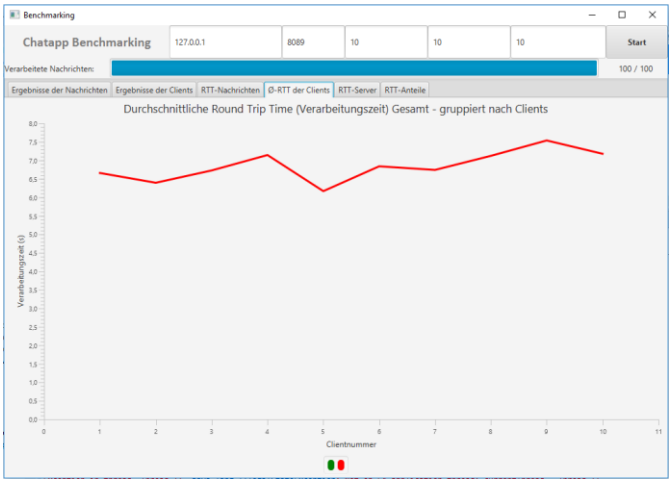
2) Benchmarking-Client Masken

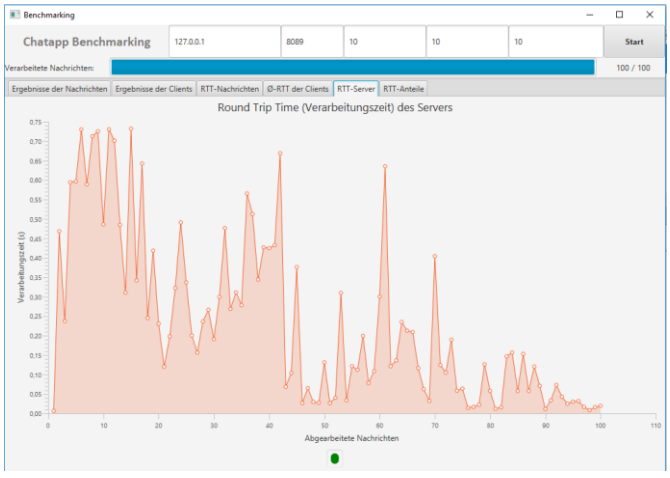
Die hier aufgeführten Abbildungen zeigen die verschiedenen Sichten des Benchmarking-Clients, welche in verschiedene Reiter unterteilt sind. Die Testergebnisse sind hierbei beispielhaft aufgeführt und sind nicht relevant für die in der Arbeit durchgeführten Performance-Tests.

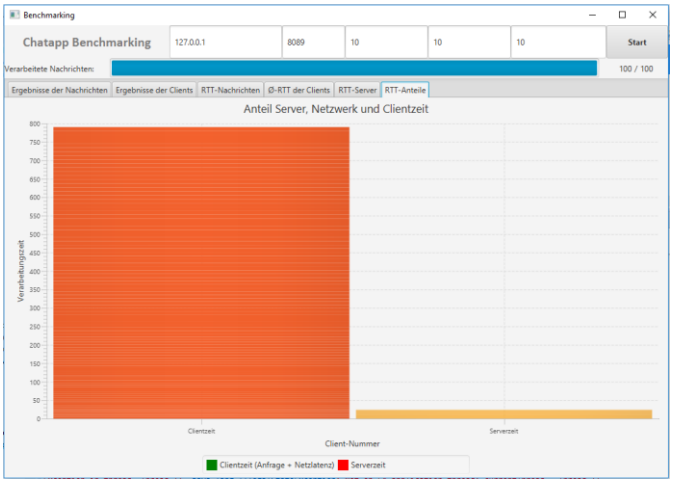






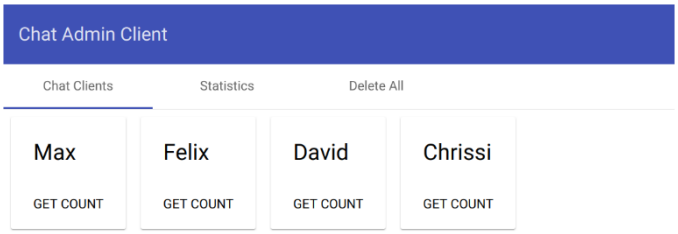


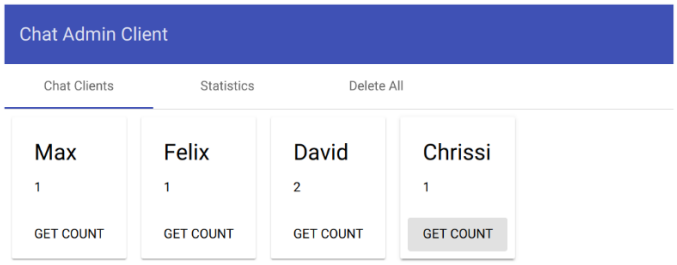


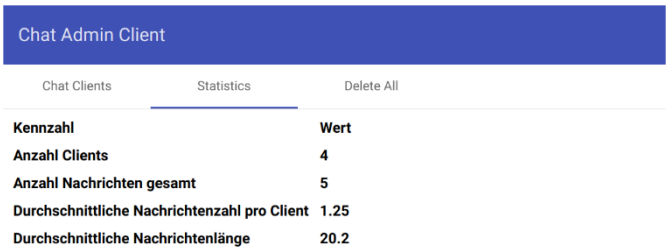


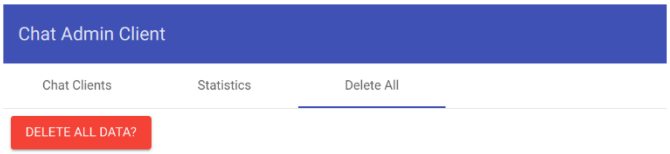
3) Admin-Client Masken

Die hier abgebildeten Benutzeroberflächen stellen die verschiedenen Funktionen des Admin-Clients dar.









D. Benchmarking-Tests

1) Ursprünglicher Ablauf der Benachmarking-Tests

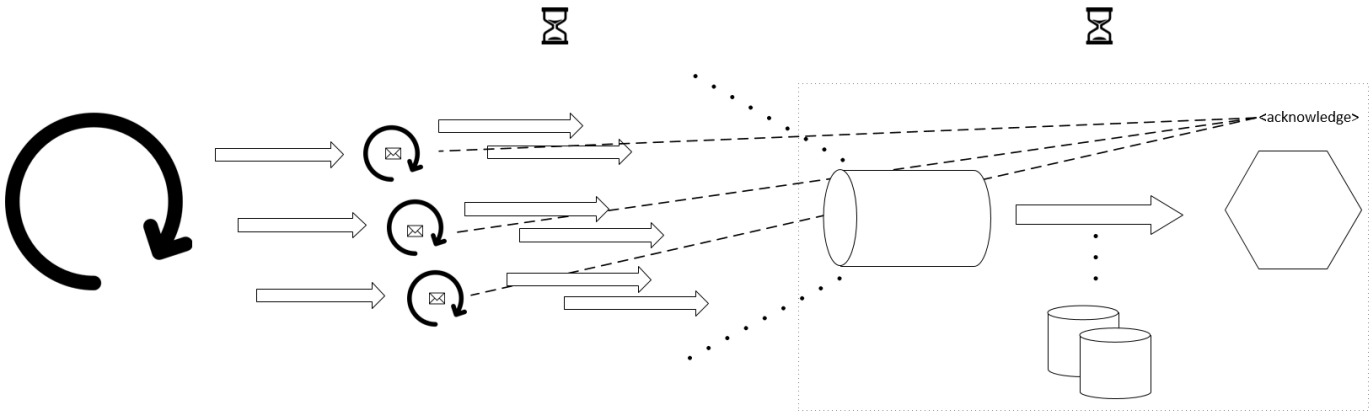


Abbildung 18 Ablauf des ursprünglichen Benachmarking-Tests

Die Clients starten nicht wie in der Neuimplementierung simultan mit dem Senden der Nachrichten, sondern senden sobald sie initialisiert sind. Zudem wird die ChatPDU für jede zu sendende Nachricht neu erzeugt. Nach dem Versand einer Nachricht wartet der Client auf ein Acknowledgement bevor er wieder sendet. Insgesamt ergibt sich durch den langsameren Sendeprozess eine deutlich geringere Belastung der Queue und der abarbeitenden Threads, da die Nachrichten mit einem größeren zeitlichen Versatz an der Queue eintreffen.

2) Testdaten in Excel

Anzahl Clients	ØRTT gesamt	RTT max	RTT min	ØRTT Server	RTT SD	ØCPU	Freier Speicher
10	47.44 s	87.80 s	1.83 s	0.01 s	24.19 s	32.93%	1016 MB
30	103.97 s	198.82 s	1.79 s	0.02 s	54.93 s	31.36%	969 MB
50	184.08 s	351.98 s	2.23 s	0.02 s	93.62 s	34.39%	882 MB
75	231.74 s	427.07 s	2.38 s	0.02 s	115.93 s	36.52%	820 MB
100	345.74 s	640.59 s	3.63 s	0.02 s	172.89 s	44.83%	727 MB

Tabelle 2: In Excel ausgewertete Testdaten

E. Deployment Description

Für das Testen und Weiterentwickeln der beschriebenen Chatanwendung, soll die nachfolgende Anleitung als Unterstützung dienen.

1) Deployment der Chatanwendung

a. Die MariaDB-Instanzen starten und die Datenbanken **tracedb** und **countdb** anlegen.
b. Den Wildfly mit der bereitgestellten **standalone.xml** starten und den MariaDB Java Client manuell deployen.
c. Das Projekt entpacken und mittels Apache Maven und dem JDK in der Version 8 bauen. Hierfür kann der Mavenbefehl **mvn clean install** verwendet werden.
    i. Die Serverkomponente wird automatisiert in Form einer WAR-Datei auf den Wildfly deployed. Das Projekt enthält bei einer erfolgreichen Maven-Operation einen neuen Ordner ./target, in welchem die gleiche WAR-Datei liegt. Diese enthält neben der Backendkomponente selbst auch alle notwendigen Abhängigkeiten.
    ii. Die Client- und Benchmarking-Komponenten werden als Fat-Jars und als native Exe-Dateien in den ./target-Ordnern des jeweiligen Maven-Projekts abgelegt.

2) Deployment des Admin-Clients

a. Voraussetzungen sind Node.js in der Version 4 oder höher und NPM in der Version 3 oder höher.
b. Installation der Angular-CLI mittels **npm install -g angular-cli**
c. Im Verzeichnis des Frontend-Projekts **ng serve** ausführen und mit dem Browser auf angezeigte Adresse (http://localhost:4200) navigieren. Im Browser (Google Chrome) mittels der Taste **F12** die Entwicklertools öffnen und die Emulation einer mobilen Ansicht aktivieren (**Strg + Shift + M**).

F. Fachliches Ablaufdiagramm

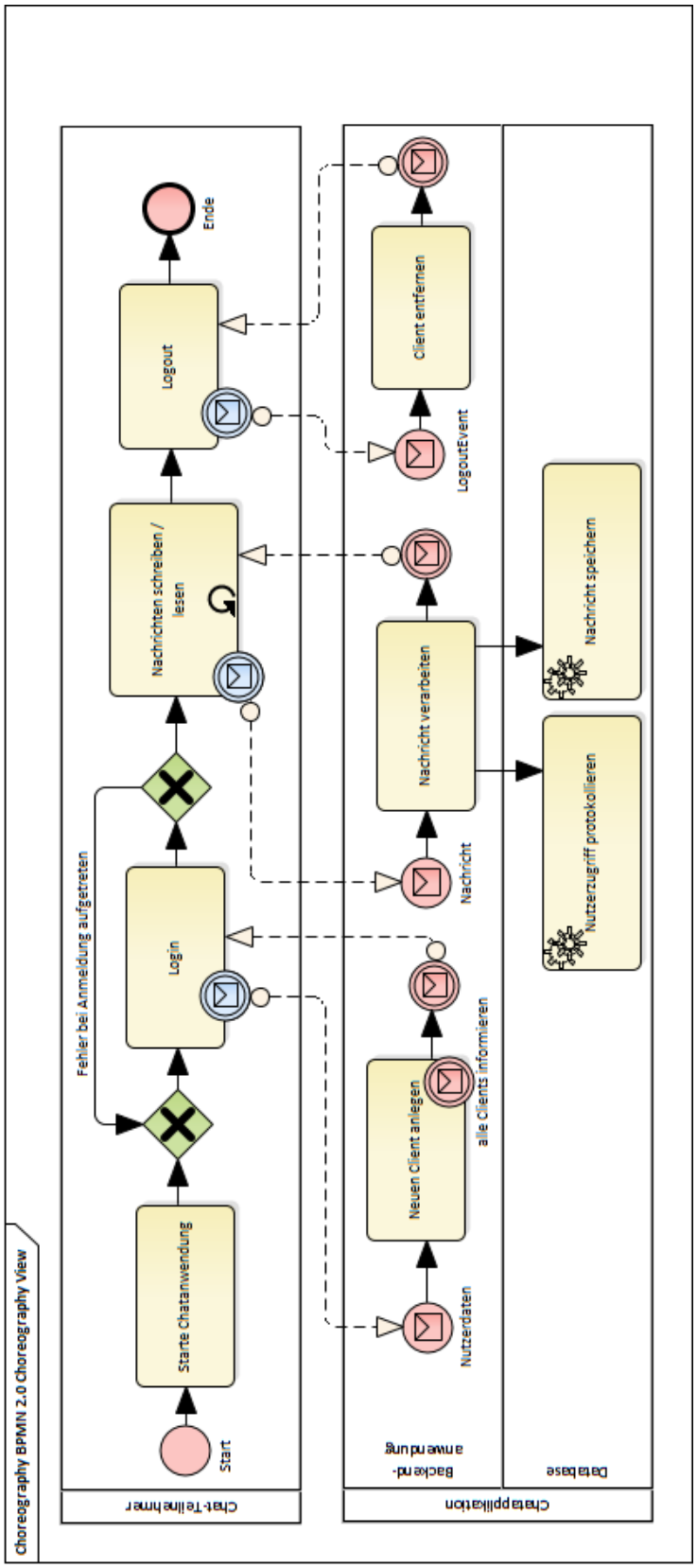


Abbildung 19: Fachliches Ablaufdiagramm der geplanten Chatanwendung mittels BPMN beschrieben.