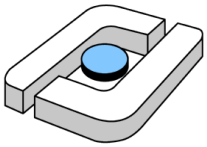


Hochschule Osnabrück

University of Applied Sciences

Fakultät

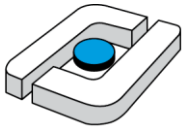
Ingenieurwissenschaften und Informatik

Masterarbeit

über das Thema:

Evaluation des Einsatzes von Apache Kafka als potenziellen Ersatz für bestehende Message Queuing Systeme

Autor:	Christian Blomberg
	christian@blomberg.io
1. Prüfer:	Herr Prof. Dr. Stephan Kleuker
2. Prüfer:	Herr Maciej Barcz
Abgabedatum:	04.03.17



Hochschule Osnabrück
University of Applied Sciences

Themenblatt

Ausgabetermin des Themas für die Abschlussarbeit: 04.10.16

Für den Studenten:

Name: Blomberg **Vorname:** Christian

Matrikel-Nr.: 458695

Studiengang: Informatik - Verteilte und Mobile Anwendungen

Wird folgendes Thema gestellt:

Evaluation des Einsatzes von Apache Kafka als potenziellen Ersatz für bestehende Message Queuing Systeme

Erstprüfer(in): Herr Prof. Dr. Stephan Kleuker

Zweitprüfer(in): Herr Maciej Barcz

Bitte urschriftlich zurück an die

Studierendenverwaltung

Das Thema der Abschlussarbeit wurde ausgegeben am: 04.10.16

Erklärung des Studenten:

Ich habe zur Kenntnis genommen, dass die
Abschlussarbeit spätestens abzugeben ist am: 04.03.17

_____
Unterschrift des Prüflings

I Kurzfassung

In der heutigen Zeit steigen die Anforderungen und die Komplexität von Unternehmensanwendungen rasant an. Hierbei ist häufig eine Vielzahl an Komponenten und Systemen vorhanden. Diese verteilten Anwendungen müssen in einem angemessenen Maße miteinander interagieren können, um eine gemeinsame Aufgabenstellung zu erfüllen.

Message Queuing Systeme tragen einen maßgeblichen Teil dazu bei, diese Aufgabenstellung zuverlässig zu lösen. Aufgrund der wachsenden Anforderungen stellt sich jedoch zunehmend die Frage, wie gut sich die daraus resultierenden komplizierteren Abläufe durch diese Systeme realisieren lassen. Neuartige Ansätze wie jene von *Apache Kafka* bieten komplett neue Möglichkeiten, diese komplexeren Geschäftsprozesse abzubilden.

Diese Arbeit befasst sich mit der Frage, wie gut sich das zuvor angesprochene *Apache Kafka* als Ersatz für klassische Systeme eignet. Aus den komplett neuen Herangehensweisen bei der Auslieferung der Nachrichten leiten sich elementare Herausforderungen, die vor einem produktiven Einsatz entsprechend geprüft werden müssen, ab. Die vorliegende wissenschaftliche Arbeit hat in diesem Zusammenhang das Ziel, die Menge an möglichen Anforderungen, die ein Unternehmen an ein nachrichtenbasiertes System stellen kann, zu betrachten und in einen gemeinsamen Kontext zu stellen.

Abstract

In the current age there is an increasing number of requirements related to business applications of companies. Often these applications are composed out of huge amount of components and systems, which have to interact together in an appropriate manner.

Message Queuing Systems play a mayor role in this process to accomplish these tasks successfully. Due to the growing requirements there are a lot of questions, like if the traditional systems are still good enough, that have to be faced. New systems like *Apache Kafka* offer completely new opportunities to implement these more complex business processes.

This thesis deals with considerations about how qualified *Apache Kafka* is as an replacement for the traditional systems. The new approaches and concepts of this system leads to completely new challanges, which have to be conquered before it can be used in production. The goal of this thesis is to analyse all the common requirements and to bring them in a global context.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Tabellenverzeichnis

V Abkürzungsverzeichnis

ACK	Acknowledgement
AJAX	Asynchronous JavaScript and XML
AMQ	ActiveMQ
API	Application Programming Interface
B2B	Business-to-Business
CPU	Central Processing Unit
DB	Database
DNS	Domain Name System
EDA	Event-driven Architecture
EJB	Enterprise JavaBeans
ESB	Enterprise Service Bus
GB	Gigabyte
GmbH & Co. KG	Gesellschaft mit beschränkter Haftung & Compagnie Kommanditgesellschaft
HTTP(S)	HyperText Transfer Protocol (Secure)
ID	Identifikator
IO	Eingabe und Ausgabe (*input/output*)
IP	Internet Protocol
JAR	Java Archive
JavaEE	Java Enterprise Edition
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JS	JavaScript
KB	Kilobyte
MB	Megabyte
MDB	Message-Driven-Beans
MFT	Managed File Transfer
MOM	Message Oriented Middleware
MQ	Message Queuing
NFR	Non-Functional Requirement

NIO	New I/O
NOA	Neue OTTO Abwicklung
OSI	Open Systems Interconnectio
Pub/Sub	Publish/Subscribe
P2P	Point-2-Point
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOA	Service-Oriented Architecture
SSD	Solid-State Drive
SSL	Secure Sockets Layer
STOMP	Simple Text Oriented Message Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WWW	World Wide Web

1 Einleitung

Die vorliegende Masterarbeit mit dem Titel *Evaluation des Einsatzes von Apache Kafka als potenziellen Ersatz für bestehende Message Queuing Systeme* befasst sich im ersten Kernaspekt mit den Konzepten aus bestehenden Message-Queuing-Systemen, die der JMS-Spezifikation entsprechen.

Im Rahmen des zweiten Kernaspektes befasst sich die Arbeit mit den Konzepten des nicht JMS-konformen *Apache Kafka*, welches eine Alternative und neuartige Form einer Messaging-Plattform darstellt.

Die jeweiligen Konzepte und Paradigmen werden entsprechend gegenübergestellt und anhand von nicht-funktionalen Anforderungen – wie Verfügbarkeit, Skalierbarkeit und Zuverlässigkeit – begutachtet, um eine Aussage darüber treffen zu können, in welchen Ausgangssituationen sich der Einsatz von *Apache Kafka* als Ersatz für JMS-konforme Message-Queuing Systeme eignet.

Im weiteren Verlauf dieses Kapitels wird in Abschnitt 1.1 ein Einblick in die grundsätzliche Thematik von *Message Oriented Middleware* geboten und kurz auf die nennenswerten Rahmenparameter des Projektes eingegangen. Im Abschnitt 1.2 werden der Umfang und die konkreten Ziele dieser Arbeit definiert. Darauffolgend wird in Abschnitt 1.3 Bezug auf verwandte bzw. ähnliche Arbeiten genommen. Zuletzt wird in Abschnitt 1.4 der weitere Aufbau der Arbeit beschrieben.

1.1 Einführung in die Thematik

Eine der Herausforderungen in verteilten Anwendungen besteht darin, die zum Lösen einer Aufgabe notwendigen Daten zwischen den jeweiligen (geographisch) entfernten Anwendungsteilen auszutauschen. Als Beispiel hierfür kann eine Bestellplattform angeführt werden, die sich aus zahlreichen geographisch getrennten Komponenten wie Benutzerverwaltung, Bestellabwicklung, Bezahlungsabwicklung, Versandabwicklung und vielen weiteren zusammensetzt.

Im konkreten Beispiel nimmt ein Web-Server die Bestellung eines Kunden entgegen und leitet diese an die Bestellabwicklung weiter. Diese gleicht ggf. die Kundendaten unter Einbeziehung der Benutzerverwaltung ab. Anschließend wird im Erfolgsfall die Versandabwicklung mit der weiteren Verarbeitung betreut. Im Fehlerfall informiert eine Stornierungs-Komponente den Benutzer – üblicherweise per Email – über das Scheitern seiner Bestellung.

In diesem Konstrukt interagieren unter Umständen sehr viele Gegenstellen miteinander. Aus diesem Sachverhalt ergeben sich grundsätzlich zwei Problemstellungen. Zum einen müssen alle beteiligten Komponenten sich in der Infrastruktur kennen, um sich gegenseitig anzusprechen.

Hieraus resultiert bereits bei verhältnismäßig geringer Komponentenanzahl eine sehr unübersichtliche Kommunikationsstruktur aus sehr vielen individuellen Verbindungen. Diese Situation wird im linken Netzwerk in Abbildung 1 angedeutet. Sofern alle Komponenten miteinander kommunizieren, entspricht dies einem vollständigen Graphen mit insgesamt $\binom{n}{2} = \frac{n(n-1)}{2}$ Kommunikationskanälen. Ausgehend von einer bidirektionalen Kommunikation erhöht sich dieser Wert um den Faktor 2, welches bei 10 Servern bereits eine Anzahl von 90 Kommunikationsverbindungen zur Folge hat.

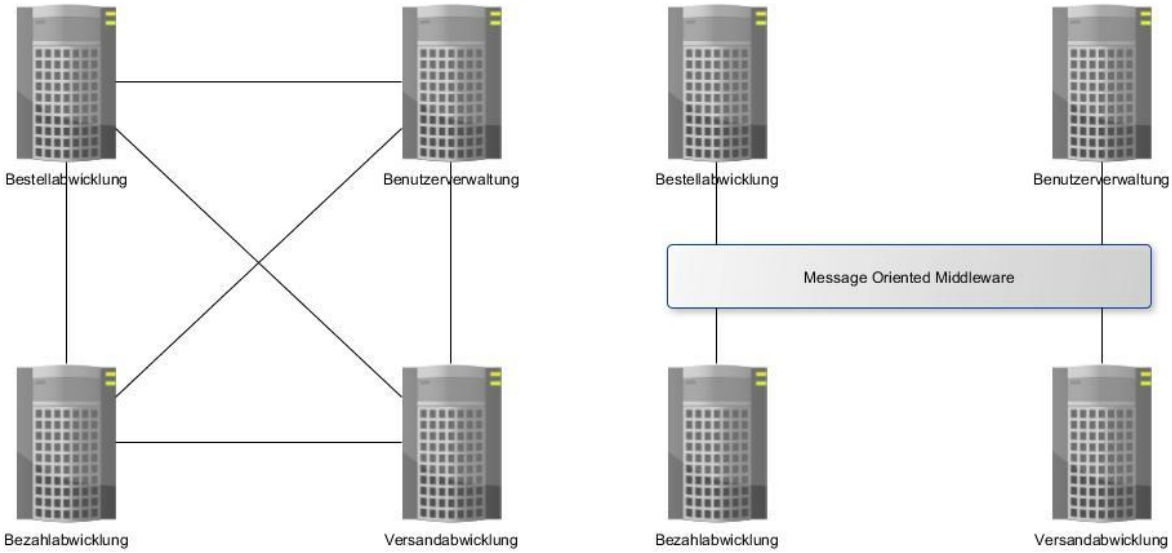


Abbildung 1: Gegenüberstellung Direktverbindung und Message Oriented Middleware

Problematisch an diesem Vorgehen ist die enge Kopplung der einzelnen Komponenten. So kennt jeder Teilnehmer die konkreten Adressinformationen aller anderen Teilnehmer. Dies führt dazu, dass die einzelnen Komponenten nicht mehr ohne Weiteres austauschbar sind, da in allen anderen Komponenten die Adressinformationen angepasst werden müssten. Zwar kann dieses Problem durch den Einsatz eines *Verzeichnisdienstes* wie DNS leicht verbessert werden, allerdings ist und bleibt diese Struktur unflexibel hinsichtlich Anpassungen der Infrastruktur.

Die zweite Problemstellung besteht darin, dass alle Komponenten zu jedem Zeitpunkt erreichbar sein müssen, um die Daten an die andere Komponente zu übertragen. Sofern beispielsweise die Benutzerverwaltung aktuell nicht erreichbar ist, ist die Bestellabwicklung nicht in der Lage, ihren Arbeitsschritt zu beenden. Folglich liegt in diesem Vorgehen ebenfalls eine sehr enge zeitliche Kopplung vor.

Spätestens wenn externe Teilnehmer hinzukommen, die nicht mehr im Einflussbereich des Unternehmens oder der betreibenden Instanz stehen, ist eine möglichst lose Kopplung zwischen den Komponenten besonders erstrebenswert.

Um diesen Problemstellungen vorzubeugen, empfiehlt es sich, eine Abstraktionsschicht zwischen den Komponenten vorzusehen, welche für die Übertragung der Daten zuständig ist und damit als Vermittler (Broker) fungiert. In diesem Aufbau kennen die einzelnen Komponenten nur noch den Vermittler als verantwortliche Gegenstelle und nicht mehr alle Teilnehmer in der gesamten Infrastruktur. Ferner ist für die sendende Komponente nicht relevant, von wem die Nachricht letztendlich konsumiert wird, und ob dieser Teilnehmer zurzeit erreichbar ist. Im konkreten Fall wird diese auf Nachrichten basierende Abstraktionsschicht als Message Oriented Middleware (MOM) bezeichnet, die durch Message-Queuing Systeme realisiert wird.

Eine Anpassung der zuvor thematisierten Beispiel-Infrastruktur hinsichtlich der Verwendung einer Message Oriented Middleware ist in dem rechten Netzwerk der Abbildung 1 dargestellt. Hierbei reduziert sich die Anzahl an Kommunikationsverbindungen um ein Vielfaches, da nur noch Kommunikationsverbindungen zwischen den Komponenten und der zentralen Vermittlungsinstanz existieren.

Da der zentrale Broker in der Lage ist, die Nachrichten für den Weiterversand zwischenzuspeichern, sind die sendende Komponente und die empfangende Komponente darüber hinaus ebenfalls zeitlich entkoppelt. Der Empfänger ist damit in der Lage, die Nachricht zu einem beliebigen Zeitpunkt entgegenzunehmen und muss deshalb zum Zeitpunkt des Nachrichtenversandes selber nicht zwangsläufig verfügbar sein.

1.1.1 Ausgangslage

Die zuvor genannten Vorteile, die sich bei der Verwendung einer Message Oriented Middleware ergeben, führten in der Vergangenheit dazu, dass sich insbesondere in großen Projekten und Unternehmen Message-Queuing Systeme schnell und großflächig etablierten.

Aus Kundensicht entstand auf diese Weise schnell eine Vielzahl an proprietären Lösungen, die alle verschiedenen Konzepten und Vorgehensweisen folgten. Da dies zu einer Unübersichtlichkeit und mangelnder Transparenz führte, wurden im Rahmen der JMS-Spezifikation allgemeine und großflächig nutzbare Vorgehensweisen vereinheitlicht. Die Spezifikation definiert, wie Nachrichten zwischen dem Broker und Java-Clients ausgetauscht werden müssen und welche Rahmenparameter hierbei zu berücksichtigen sind. Hierzu zählen unter anderem die Messaging-Modelle, die zum Datenaustausch verwendet werden können und der grundsätzliche Aufbau einer Nachricht mit ihren entsprechenden Meta-Daten.

Die JMS-Spezifikation stellt sicher, dass die Nachrichten auf vorher definierte und verbindliche Art und Weise ausgetauscht werden. Durch die allgemeingültige Vereinheitlichung lassen sich gezielte Aussagen über die Dienstgüte verschiedener Übertragungsmodi treffen. So garantiert JMS zwar in bestimmten Modi die Übertragung einer Nachricht im Regelbetrieb, toleriert aber

den Verlust des aktuellen Nachrichtenbestandes bei einem Serverausfall. Hierbei wird von nicht-persistentem (*non-persistent*) Messaging gesprochen (siehe Kapitel 3).

Grundsätzlich besteht im JMS-Kontext die Möglichkeit, zwischen *persistent* Messaging und *non-persistent* Messaging zu wählen. Non-Persistent Messaging bietet eine deutliche Performance Steigerung, beinhaltet aber auch das Risiko eines Datenverlustes im Falle eines Serverausfalls. Dies ist darin begründet, dass die Nachrichten nur im flüchtigen Speicher vorgehalten und nicht dauerhaft auf der Festplatte abgelegt werden. Im entgegengesetzten Fall bietet *persistent* Messaging zwar die Sicherheit, dass die Daten auch im Fehlerfall dauerhaft bereitstehen, schränkt jedoch die Performance aufgrund der hohen Schreib- und Lesezugriffe auf die Festplatte deutlich ein [RMC09].

Trotz der weiten Verbreitung von JMS-Konformen Messaging-Systemen und der transparenten Übertragungsmodi, welche klare und verbindliche *Quality of Service* Eigenschaften zur Folge haben, bietet das grundsätzlich genutzte Messaging-Paradigma einige Nachteile, mit denen sich bestimmte Verarbeitungs- und Übertragungsansätze nicht oder nur schwierig umsetzen lassen. Dazu zählt unter anderem, dass Nachrichten im JMS-Umfeld, nachdem sie konsumiert wurden, direkt auf dem Broker gelöscht werden und damit nicht für eine erneute Konsumierung zur Verfügung stehen. Des Weiteren ist – in der Regel – immer nur eine Kopie bzw. Instanz einer Nachricht im jeweiligen Messaging-System vorhanden. Dies ist selbst dann der Fall, wenn ein *Netzwerk von Brokern* verwendet wird, in dem die jeweiligen Broker gemeinschaftlich an der Übertragung der Daten arbeiten. In diesem Fall ist es nicht möglich, mehrere Replikate der Nachricht auf den einzelnen Brokern bereit zu halten, da sonst nicht mehr nachvollzogen werden könnte, welche Nachricht bereits konsumiert wurde.

Eine detaillierte Beschreibung der Konzepte und Limitierungen der herkömmlichen (JMS)-Messaging-Konzepte, sowie eine Gegenüberstellung mit den Konzepten von *Apache Kafka* erfolgt im Kapitel 3.

1.1.2 Problemstellung

Die verteilten Anwendungen, die ihre Kommunikation und ihren Datenaustausch über die klassischen Messaging Systeme realisieren, werden immer komplexer und interaktiver. Dies drückt sich unter anderem in einem steigenden Nutzeraufkommen, einem höherem Datenaufkommen und – damit verbunden – in einer sehr großen Menge an Log-Daten aus. An dieses Datenaufkommen wird, neben einer schnellen und effizienten Übertragung, der Anspruch gestellt, dieses zeitnah von mehreren Instanzen verarbeiten und analysieren zu können [GL13].

Da es sich bei den erzeugten Daten um verschiedenste Themenschwerpunkte wie Benutzerdaten und Fehler-Protokollierung handelt, werden an diese Gruppierungen auch verschiedene An-

forderungen an den Zeitpunkt ihrer Verarbeitung gestellt. Neben der Onlineverarbeitung in nahezu Echtzeit, wird damit an die produzierten Daten auch die Anforderung gestellt, diese ebenfalls in nachgelagerten Offline-Szenarien zu verarbeiten [GL13].

Basierend auf den im vorherigen Abschnitt kurz thematisierten Limitierungen der klassischen Ansätze von Messaging Systemen und den steigenden Anforderungen an interaktive und skalierbare Anwendungen, lässt sich eine Realisierung mit den herkömmlichen Systemen in vielen Fällen nur unzureichend erzielen. Dies kann durch verschiedenste Aspekte begründet sein. Beispielsweise lässt das hohe Datenaufkommen eine zeitnahe Übertragung und Analyse der Daten nicht zu.

1.1.3 Lösungsansatz Apache Kafka

Ein Unternehmen, das sich mit den Anforderungen an die eigene interaktive Anwendung konfrontiert sah, ist das zur Pflege von Geschäftskontakten eingesetzte soziale Netzwerk *LinkedIn* [@lkd]. Da das Unternehmen innerhalb der eignen Infrastruktur zunehmend Probleme damit hatte, der stetig steigenden Datenflut gerecht zu werden und diese in zufriedenstellendem Maße zu analysieren, begann das Unternehmen mit der Entwicklung einer eigenen Lösung, welche die Kritikpunkte an den zuvor eingesetzten Messaging-Systemen aufgriff und entsprechend optimierte.

Das resultierende Messaging-System *Kafka* wurde aufgrund seiner überzeugenden Konzeption, Performance-Charakteristika und Architekturentscheidungen auch für andere Projekte und Unternehmen relevant. Es wurde hierzu im Rahmen der *Apache Software Foundation* [@apa] weiterentwickelt und damit anderen Projekten zur Verfügung gestellt.

Apache Kafka ermöglicht, ähnlich wie klassische Messaging-Systeme, die Übertragung von Nachrichten. Im Gegensatz zu den JMS-konformen Systemen werden die Nachrichten jedoch immer persistent – für einen definierbaren Zeitraum – auf der Festplatte gespeichert. Dabei bietet Kafka jedoch aufgrund seiner effizienten Vorgehensweise im Umgang mit IO-Operationen einen sehr hohen Durchsatz im Vergleich zu klassischen Lösungen.

Auch wenn das Resultat ähnlich zu den bekannten Lösungen ist, unterscheidet sich *Kafka* deutlich von den bisherigen Konzepten und ist damit nicht JMS-konform. *Kafka* verwendet unter anderem ein anderes Messaging-Modell, in dem es einen verteilten Ansatz verfolgt, bei dem die Nachrichten auf allen *Kafka*-Brokern vorrätig gehalten und von den *Consumern* dabei mehrfach konsumiert werden können. Des Weiteren sind die Consumer im *Kafka*-Kontext dafür verantwortlich, sich zu merken, welche Nachrichten bereits konsumiert wurden. Im bisherigen JMS-Kontext stellte sich diese Frage nicht, da Nachrichten, nachdem sie konsumiert wurden, nicht mehr im Messaging-System verfügbar waren. Zuletzt werden die Nachrichten nicht wie

bei JMS direkt an den Consumer übermittelt, sondern dieser fragt kontinuierlich nach, ob neue Nachrichten verfügbar sind [NSP17].

Auf die konkreten JMS- und Kafka-Konzepte wird im Kapitel 3 fundiert eingegangen. Wichtig an dieser Stelle ist, dass sich durch die komplett andere Herangehensweise, die *Apache Kafka* verfolgt, viele Frage- und Problemstellungen im Hinblick auf nicht-funktionale Anforderungen wie die Zuverlässigkeit ergeben, die vor einem produktiven Einsatz zu klären sind.

1.1.4 Vorstellung des Unternehmens

Das im Jahre 1949 gegründete Unternehmen *OTTO (GmbH & Co KG*) mit Sitz in Hamburg ist ein weltweit agierendes Handels- und Dienstleistungsunternehmen, das insgesamt knapp 50.000 Mitarbeiter beschäftigt. Zur *OTTO-Group* gehören neben der Hauptmarke zahlreiche Tochtergesellschaften wie Bonprix, myToys und SportCheck.

Diese Arbeit wird im Auftrag und in Kooperation mit dem oben genannten Unternehmen durchgeführt, das im Rahmen des E-Commerce eine entsprechend große Infrastruktur mit insgesamt 150 Projekten, 600 Systemen und 1300 Servern betreibt. In dieser Infrastruktur müssen die verschiedensten Komponenten und Systeme in der Lage sein, miteinander zu interagieren.

Diese Menge an Systemen und Servern werden nach aktuellem Stand mit klassischen Messaging-System wie *ActiveMQ* [@mq] und *SonicMQ* [@snc] angebunden und interagieren über diese.

Da ein Einsatz von *Apache Kafka* viele neue Möglichkeiten und Potenziale eröffnen würde und damit einen deutlichen Performance-Gewinn zur Folge haben könnte, befasst sich das Unternehmen mit den verbundenen Fragestellungen, die eine generelle Einsetzbarkeit hinterfragen, den Migrationsaufwand beleuchten und wirtschaftliche Gesichtspunkte berücksichtigen.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es, die wesentlichen Kernfragen, welche die Firma *OTTO* an den Einsatz von *Apache Kafka* stellt, zu beantworten und dabei zusätzlich einen wissenschaftlichen und auch für andere Projekte neutralen und allgemeingültigen Blickwinkel zu schaffen.

Der Kern dieser Arbeit beschäftigt sich mit der generellen Machbarkeit bzw. Einsetzbarkeit des neuen Messaging-Systems *Apache Kafka*. Hierbei stehen im Wesentlichen nicht-funktionale Anforderungen wie Zuverlässigkeit, Skalierbarkeit, Verfügbarkeit und viele weitere im Vordergrund. In diesem Rahmen werden die JMS- und Kafka-Konzepte gegenübergestellt und auf die jeweiligen nicht-funktionalen Anforderungen abgebildet.

Im Anschluss an konkrete Ergebnisse, die beinhalten, welche Anforderungen sich für einen Einsatz von *Apache Kafka* eignen, werden diese auf den konkreten Anwendungsfall des betreuenden Unternehmens projiziert.

1.3 Bezug auf ähnliche Arbeiten

Es gibt zahlreiche Arbeiten, die sich mit dem Thema JMS und Messaging als solches befassen. Des Weiteren gibt es schon einige Arbeiten, die sich mit dem Thema *Apache Kafka* auseinandersetzen. Bei letzterem ist anzumerken, dass sich *Apache Kafka* mit der Version *0.10.1.0* noch in einem sehr frühen Entwicklungsstadium befindet. Aus diesem Grund ist davon auszugehen, dass es vor dem ersten Major-Release zu weiteren konzeptionellen Änderungen kommen könnte.

An dieser Stelle wird auf die Nennung der Grundsatzarbeiten über JMS und Kafka verzichtet und nur Bezug auf die Arbeiten genommen, welche in gewissem Zusammenhang mit den Zielen dieser Arbeit stehen.

1.3.1 Kafka – a Distributed Messaging System for Log Processing

Der von den Gründervätern von *Apache Kafka,* Jay Kreps, Neha Narkhede und Jun Rao, im Jahre 2011 veröffentlichte Artikel über die Konzepte von *Apache Kafka* befasst sich mit einer ersten Gegenüberstellung klassischer Messaging Systeme wie *ActiveMQ* mit *Apache Kafka* [KNR11].

Es wird ein Versuchsaufbau beschrieben, in dem die jeweiligen Broker systematisch mit Nachrichten geflutet werden. Das Ergebnis zeigt einen deutlichen Performance-Vorsprung von *Apache Kafka* vor den klassischen Messaging-Systemen. Entgegen dieses sehr knapp beschriebenen Versuchsaufbaus werden in der hier vorliegenden Arbeit neutrale und umfangreichere Performance-Tests durchgeführt, in die alle wesentlichen Tuning-Parameter der Broker sowie alle wesentlichen nicht-funktionalen Anforderungen einfließen.

1.3.2 Behavior and Performance of MOM Systems

Dieses von Phong Tran und Paul Greenfield im Jahre 2002 veröffentlichte IEEE-Paper befasst sich mit dem Verhalten und der Performance von Messaging Systemen [TG02]. Hierbei wird das Messaging System *IBM MQSeries* verwendet.

Das Paper geht im Kern auf Empfangsraten und Laufzeitverhalten bei wachsender Nachrichtenmenge im Broker ein. Die Ergebnisse liefern Anknüpfpunkte über das Verhalten von klassischen Messaging-System. Aufgrund des Alters des Papers und dem damit einhergehenden Fortschritt klassischer Messaging-Systeme sind diese Erkenntnisse jedoch nur in geringem Maße direkt übertragbar.

1.4 Aufbau der Arbeit

Im weiteren Verlauf dieser Arbeit werden im anschließenden Kapitel 2 die notwendigen Grundlagen im Zusammenhang mit verteilten Anwendungen, Messaging Systemen und den damit verbunden Anforderungen beschrieben.

Das darauffolgende Kapitel 3 befasst sich jeweils mit den JMS- sowie *Apache Kafka* Konzepten. Im JMS-Kontext wird hierbei stellvertretend das Messaging-System *Apache ActiveMQ* verwendet, welches der JMS-Spezifikation unterliegt, jedoch noch weit darüber hinausreichende Funktionalitäten bereitstellt.

Anschließend wird im Kapitel 4 der Versuchsaufbau bzw. die Infrastruktur erläutert, die verwendet wurde, um das Verhalten beider Systeme unter möglichst gleichen Voraussetzungen zu analysieren. Die Beschreibung der zugehörigen Testergebnisse und die damit eingehende Evaluation erfolgt im anschließenden Kapitel 5.

Im Rahmen des 6. Kapitels werden die nicht-funktionalen Anforderungen, die im Hinblick des Messagings zu betrachten sind, in Relation mit den beiden Herangehensweisen gestellt.

Im darauffolgenden Kapitel 7 wird, basierend auf den Testergebnissen, ein Leitfaden definiert, der Unternehmen anhand ihrer eigenen nicht-funktionalen Anforderungen als Entscheidungsgrundlage dienen soll. An dieser Stelle wird hinterfragt, ob und inwieweit sich die Ausgangslage für einen Umstieg auf *Apache Kafka* eignet. Anschließend wird in Kapitel 8 im Rahmen der Anforderungen des betreuenden Unternehmens *OTTO* der *Proof-of-Concept* durchgeführt, in dem der zuvor definierte Leitfaden auf die Ausgangslage dieses Unternehmens abgebildet wird.

Zuletzt werden im Rahmen einer Schlussbetrachtung in Kapitel 9 ein Fazit und ein Ausblick der Arbeit geboten.

2 Grundlagen

In diesem Kapitel werden die relevanten Grundlagen beschrieben, die für das weitere Verständnis der Arbeit förderlich sind. In diesem Rahmen wird in Abschnitt 2.1 auf verteilte Systeme im Allgemeinen eingegangen sowie im darauffolgenden Abschnitt 2.2 die damit einhergehenden verteilten Architekturen erläutert. Anschließend wird im Abschnitt 2.3 auf den Begriff des *Enterprise Service Bus* Bezug genommen sowie in Abschnitt 2.4 das Konzept des Clusterings betrachtet. Darauffolgend befasst sich der Abschnitt 2.5 mit den verschiedenen Kommunikationsarten innerhalb verteilter Anwendungen. Anschließend werden im Abschnitt 2.6 die Grundlagen von *Message Oriented Middleware* dargestellt. Zuletzt befasst sich der Abschnitt 2.7 mit funktionalen sowie nicht-funktionalen Anforderungen.

2.1 Verteilte Systeme

Bei verteilten Systemen handelt es sich um Infrastrukturen, die sich aus einer Vielzahl von einzelnen Komponenten zusammensetzen [SS12]. In der Regel sind diese Systeme geographisch voneinander getrennt und verfügen über keinen gemeinsamen Speicher. Der Nachrichtenaustausch erfolgt hierbei unter anderem über entfernte Methodenaufrufe oder durch nachrichtenbasierte Systeme. Das *World Wide Web* (WWW) zählt zu den größten und bekanntesten Repräsentanten von verteilten Systemen [SS12].

Im Gegensatz dazu handelt es sich bei verteilten Anwendungen um getrennte Software-Komponenten, die in eigenen Prozessen laufen und auf der Grundlage von verteilten Systemen eine gemeinsame Arbeit verrichten. Dem Nutzer erscheint an dieser Stelle die verteilte Anwendung wie eine einzelne Anwendung. Beispiele für verteilte Anwendungen sind Bankensysteme, Buchungssysteme oder soziale Netzwerke.

Ziele von Verteilten Systemen

Eines der Ziele für die Verwendung von verteilten System ist, bestimmte Ressourcen mehreren Komponenten zugänglich zu machen. Hierbei nutzen beispielsweise mehrere Server-Instanzen den gleichen Datenbestand in Form einer Datenbank.

Darüber hinaus ermöglichen verteilte Systeme in vielen Fällen eine Steigerung der Verarbeitungsleistung, in dem bestimmte Verarbeitungsschritte parallel ausgeführt werden können. Eng damit verbunden ist das häufig eingesetzte Konzept der Lastenverteilung. In diesem Zusammenhang werden mehrere Instanzen einer Server-Komponente bereitgestellt. Von diesen Instanzen ist jede einzelne in der Lage, eine eintreffende Anfrage zu bearbeiten. Die Zuteilung der Anfragen erfolgt hierbei durch eine sogenannte Load-Balancing-Komponente, welche die Anfragen auf Basis der aktuellen Last auf die Instanzen abbildet.

Ein wesentliches Bestreben bei der Verwendung von verteilten Systemen liegt in der Realisierung verschiedener nicht-funktionaler Anforderungen. Hierzu gehört beispielsweise die Erhöhung der Verfügbarkeit durch die Bereitstellung mehrerer Instanzen. Auf die relevanten nichtfunktionalen Anforderungen dieser Arbeit wird im Kapitel 6 eingegangen.

2.2 Verteilte Systemarchitekturen

In diesem Abschnitt werden verschiedene Architekturstile, die im Rahmen verteilter Systeme häufig zum Einsatz kommen, beschrieben. Hierzu befasst sich Abschnitt 2.2.1 mit dem Client/Server Modell. Anschließend wird in Abschnitt 2.2.2 das objektorientierte Modell erläutert sowie in Abschnitt 2.2.3 das komponentenbasierte Modell behandelt. Inhalt von Abschnitt 2.2.4 sind mehrschichtige Architekturen.

Als weiterführende und neuartige Konzepte werden anschließend im Abschnitt 2.2.5 der serviceorientierte Ansatz sowie in Abschnitt 2.2.6 der eventbasierte Ansatz beschrieben.

2.2.1 Client/Server-Modell

Der Client/Server-Architekturstil stellt ein Basisvorgehen innerhalb der Interaktion von verteilten Systemen dar. In diesem Fall greift eine Instanz auf eine entfernte Instanz zu, die eine bestimmte Funktionalität bereitstellt. Die anfragende Komponente wird hierbei als Client und die antwortende Komponente als Server bezeichnet. Der Begriff Server ist hierbei von dem Verb *bedienen* (*to serve*) abgeleitet.

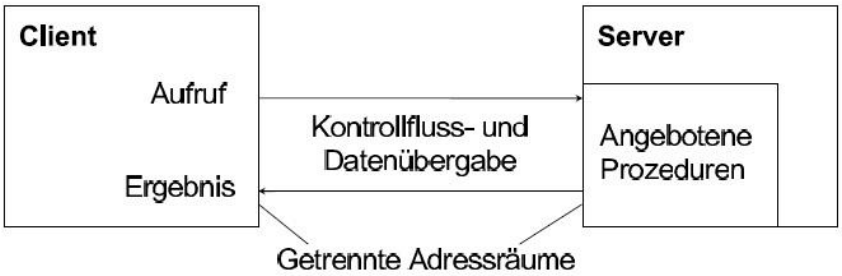


Abbildung 2: [SS12] Client/Server-Modell

Abbildung 2 zeigt den grundsätzlichen Aufbau und Ablauf des Client/Server-Modells. Dementsprechend gibt der Client beim Aufruf einer Server-Funktionalität den Kontrollfluss an den Server ab [SS12]. Üblicherweise blockiert in diesem Zusammenhang der Client, bis er den Kontrollfluss durch die Antwort des Servers zurück erhält. Im Rahmen der Verarbeitung kann der Server selbst wieder die Funktionalität anderer Diensterbringer (Server) einbeziehen und tritt in diesem Zusammenhang ebenfalls als Client (Dienstnutzer) auf [SS12].

Der Datenaustausch erfolgt in der Regel über entsprechende Parameter innerhalb der Anfrage, welche in diesem Rahmen in den Adressraum des Servers übertragen werden [SS12]. Das einfache Vorgehen des Client/Server-Modells führt zu einer handhabbaren Struktur, die herkömmlichen lokalen Programmiermodellen ähnelt. Weitere Informationen zu diesem verteilten Architekturstil sind [SS12] zu entnehmen.

2.2.2 Objektorientiertes Modell

Auf dem Client/Server-Modell aufbauend verfolgt der objektorientierte Architekturstil das Ziel, einzelne Objekte in die Rollen des Clients sowie des Servers zu versetzen [SS12]. Diese lokal strukturierten Objekte kommunizieren in diesem Zusammenhang direkt miteinander und können dabei eine beliebige Granularität aufweisen. Ein Client wird dadurch in die Lage versetzt, ein Objekt zu übertragen, auf das er anschließend eine Referenz erhält. Durch diese Referenz kann der Client unter anderem Statusinformationen vom entfernten Objekt abrufen [SS12].

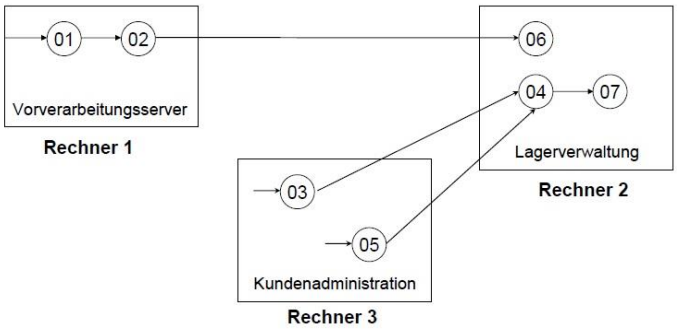


Abbildung 3: [SS12] Objektorientiertes Modell

Abbildung 3 zeigt das grundsätzliche Vorgehen dieses Architekturstils. Die einzelnen, innerhalb der Rechner enthaltenen, nummerierten Objekte kommunizieren dabei mit den entfernten Objekten anderer Rechner [SS12]. Die Realisierung dieses Ansatzes erfolgt weiterhin über entfernte Methodenaufrufe, die beispielsweise in Java durch Java RMI *(Remote Method Invocation)* realisiert werden. Ein weiterer Ansatz für die Realisierung des objektorientierten Modells ist CORBA *(Common Object Request Broker Architecture).*

2.2.3 Komponentenbasiertes Modell

Das komponentenbasierte Modell ermöglicht es, die benötigte Anwendungslogik in einzelne Komponenten zu kapseln und diese weitestgehend von den Eigenschaften des verteilten Systems zu trennen. Im Vergleich zum objektorientieren Ansatz kapseln die zugehörigen Komponenten in der Regel eine Anwendungslogik gröberer Granularität [SS12].

Die jeweilige Komponente wird im Rahmen ihrer Installation mit den notwendigen Eigenschaften, beispielsweise Transaktionen oder Sicherheit, ausgestattet. In diesem Zusammenhang ist

eine spezielle Laufzeitumgebung vorhanden, in deren Verantwortungsbereich es liegt, die Komponente mit den notwendigen Eigenschaften auszustatten, zu starten und zu verwalten [SS12].

Eine Ausprägung dieses komponentenbasierten Ansatzes sind die *Enterprise Java Beans* (EJB). Die zugehörige Laufzeitumgebung wird als EJB-Container bezeichnet [DH03]. EJB unterscheidet grundsätzlich zwischen den Bean-Typen *Session Bean*, *Entity Bean* sowie *Message-Driven Bean* [DH03].

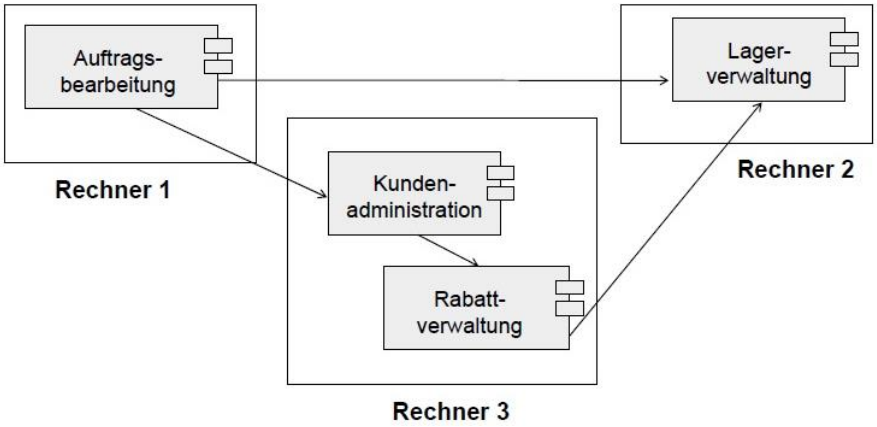


Abbildung 4: [SS12] Komponentenbasiertes-Modell

Die *Session Beans* dienen dazu, entsprechende Geschäftsprozesse abzubilden. In der dargestellten komponentenbasierten Beispielarchitektur in Abbildung 4 repräsentieren die jeweiligen Bausteine, wie beispielsweise die Auftragsbearbeitung, die zuvor angesprochenen *Session-Beans*. Die *Entity Beans* verkörpern persistente Objekte, die im Rahmen der Geschäftsprozesse verwendet werden. Sie ermöglichen einen einheitlichen Zugriff auf den gespeicherten Datenbestand [DH03].

Die Message-Driven Beans (MDB) bieten die Möglichkeit, Daten asynchron (siehe Abschnitt 2.5) in Form von Nachrichten auszutauschen. Sie implementieren hierzu einen *Listener*, der beim Empfang einer Nachricht die *onMessage()*-Methode der Bean aufruft. Message-Driven Beans haben im Hinblick auf diese wissenschaftliche Arbeit eine gesonderte Relevanz, da sie im JMS-Kontext die Möglichkeit bieten, mit sehr einfachen Mitteln eine JMS-Sitzung zu erstellen. In diesem Zusammenhang übernimmt der EJB-Container die notwendigen Schritte zum Aufbau einer JMS-Verbindung. Darüber hinaus wird für die Beans automatisch ein Threadpool verwaltet, der es ermöglicht, bei steigender Last mehrere Instanzen dieser Bean zur Verarbeitung von Nachrichten einzusetzen [Sta15]. Da die MDB jedoch nur im JMS-Kontext und nicht im Kafka-Kontext eingesetzt werden können, wurde im Rahmen dieser Arbeit auf den Einsatz dieser Beans verzichtet. Weiterhin schafft die manuelle Erstellung einer JMS-Sitzung mehr Transparenz und Flexibilität im Hinblick auf die verwendeten Einstellungen und die Menge an genutzten Threads.

2.2.4 Mehrschichtige Architekturen

Ein etablierter Ansatz zur Strukturierung einer Architektur besteht darin, diese in einzelne Schichten (*Layer*) zu unterteilen [Sta15]. Hierbei wird häufig ein dreistufiger Ansatz verfolgt, bei dem eine Unterteilung in Präsentationsschicht, Verarbeitungsschicht sowie Persistenz-Schicht vorgesehen wird. Diese oft gewünschte Unterteilung ermöglicht es, die einzelnen Schichten auf verschiedenen Systemen und entfernt von einander zu betreiben. Beispielsweise befindet sich im, in Abbildung 5 dargestellten, Aufbau die Präsentationsschicht innerhalb der Desktopanwendung eines Clients.

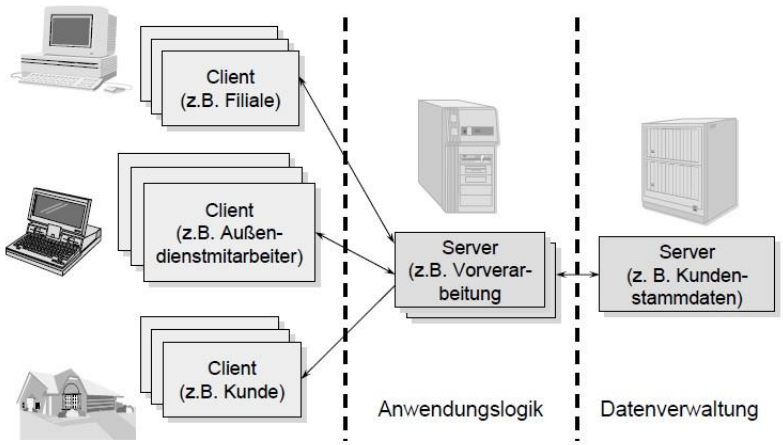


Abbildung 5: [SS12] Dreischichtige Architektur

2.2.5 Serviceorientierte Architektur (SOA)

Der serviceorientierte Architekturstil (SOA) verfolgt das Ziel, die Geschäftslogik und -abläufe modularer und flexibler zu gestalten. Die benötigte Funktionalität wird in diesem Rahmen nicht innerhalb eines einzigen Systems abgebildet, sondern in fachliche Gruppen unterteilt. Diese fachlichen Gruppen werden anschließend durch einzelne Services abgebildet und im Gesamtsystem bereitgestellt [Sta15]. Die Verwendung von SOA ist im Allgemeinen unabhängig von den eingesetzten Technologien.

Serviceorientierte Architekturen ermöglichen es den jeweiligen Unternehmen, flexibler im Umgang mit der eigenen Software zu agieren und diese besser an sich verändernde Marktbedingungen anzupassen [Sta15].

Durch die Trennung der einzelnen Funktionalitäten nach Fachlichkeit und dem Bestreben, jeden Dienst als *Blackbox* zu betrachten, entsteht eine lose Kopplung [Sta15]. Diese ermöglicht es, einen bestehenden Service durch eine neue Implementierung zu ersetzen. In diesem Zusammenhang wird auf die einzelnen Services stets über eine definierte Schnittstelle zugegriffen. Die lose Kopplung ermöglicht außerdem eine Kommunikation über Unternehmensgrenzen hinweg, da hierdurch nur geringe Abhängigkeiten zwischen den Systemen entstehen.

Im Zusammenspiel der einzelnen Services können grundsätzlich zwei Ansätze verfolgt werden. Der erste Ansatz der *Orchestrierung* besteht darin, aus den Basis-Services einen zusammengesetzten und komplexeren Service einer höheren Abstraktionsebene zu erzeugen [Sta15]. Dieser Service zeigt sich von außen als ein gewöhnlicher Service und verbirgt, welche der Basis-Services von ihm aufgerufen werden. Im Gegensatz dazu verfolgt die *Choreographie* das Ziel, die einzelnen Services direkt zu einem Geschäftsablauf zusammenzusetzen, in welchem diese sich gegenseitig aufrufen.

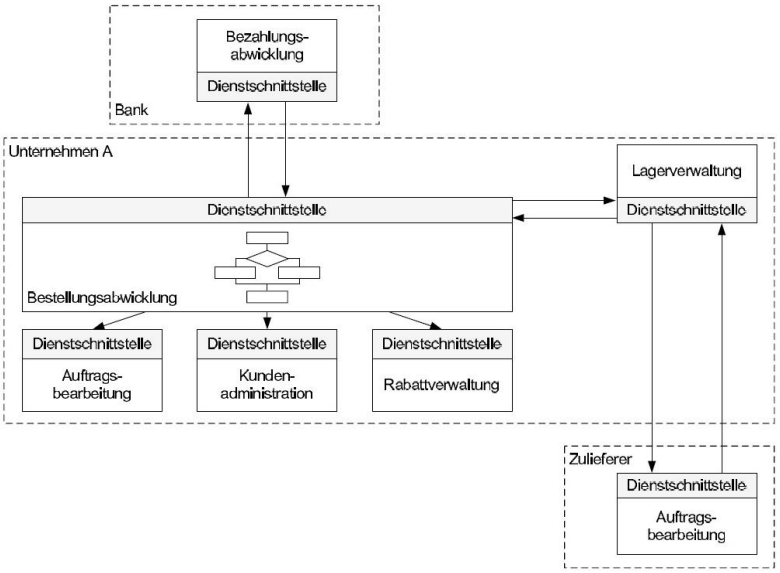


Abbildung 6: [SS12] Serviceorientierter Architekturstil

Abbildung 6 zeigt ein Beispiel für eine *serviceorientierte* Architektur. In dieser Architektur sind grundsätzlich zwei Unternehmensinterne und zwei externe Services zu erkennen. Der interne Service *Bestellabwicklung* nutzt im Rahmen einer *Orchestrierung* jedoch die drei weiteren Services *Auftragsbearbeitung*, *Kundenadministration* sowie die *Rabattverwaltung*. Die anderen Services interagieren in Form einer *Choreographie* miteinander [SS12]. Darüber hinaus ist in der Abbildung zu erkennen, dass alle Services eine Schnittstelle definieren, über welche auf diese zugegriffen werden kann.

Serviceorientierte Architekturen können auf weiteren Architekturstilen aufbauen bzw. mit diesen kombiniert werden. Hierbei kann beispielsweise schnittstellenbasiert in Form von entfernten Methodenaufrufen vorgegangen werden. Darüber hinaus ist es möglich, die Kommunikation asynchron über nachrichtenbasierte Systeme erfolgen zu lassen. Zuletzt kann eine serviceorientierte Architektur ressourcenbasiert über den Architekturstil REST (*Representational state transfer*) aufgebaut werden. REST ist ein Architekturstil, der sich an der Architektur des World Wide Web orientiert. Von den Services werden Ressourcen bereitgestellt, die von anderen Services konsumiert werden können. Hierfür stellt REST die vier Methoden GET, PUT, POST, sowie DELETE zur Verfügung [Sta15].

2.2.6 Eventbasierte Architekturen (EDA)

Der eventbasierte Architekturstil verfolgt das Ziel, innerhalb einer Architektur nicht über einen direkten Methodenaufruf mit einer anderen Instanz zu kommunizieren, sondern indirekt durch die Erzeugung eines Events. Dieses Event informiert eine oder mehrere Instanzen darüber, dass ein Ereignis innerhalb des Systems stattgefunden hat [RMC09]. Es wird grundsätzlich zwischen Eventquellen, die ein Event erzeugen und Eventsenken, an welche das jeweilige Event gerichtet ist, unterschieden. Diese Komponenten kommunizieren hierbei über Ereignisse anstatt über Methodenaufrufe, die fest im Quellcode hinterlegt sind [Sta15].

Innerhalb eines großen Systems, wie beispielsweise dem einer Versicherung, sind viele Vorgänge und Abläufe vorhanden, die ineinander greifen. Beispielsweise führt eine Adressänderung dazu, dass dies Auswirkungen auf viele Aspekte des Gesamtsystems hat. Durch den neuen Wohnort kommt es beispielsweise zu einer Änderung der Versicherungsrichtlinien und damit verbunden auch zu einer Veränderung der monatlichen Kosten [RMC09]. Da es jedoch nicht im Verantwortungsbereich der Adressänderungskomponente liegt, genau zu wissen, welche Vorgänge in diesem Zusammenhang ebenfalls zu erfolgen haben, benachrichtigt sie das gesamte System über dieses Ereignis. Die Komponenten, für die das Ereignis relevant ist, greifen dieses auf und führen auf dessen Grundlage ihre weitere Verarbeitung durch und erzeugen ihrerseits gegebenenfalls neue Events [RMC09].

Das Messaging, das im Fokus dieser wissenschaftlichen Arbeit steht, ist die Grundlage für den eventbasierten Architekturstil [RMC09]. Dabei wird üblicherweise eine zentrale Komponente verwendet, welche als *Message Oriented Middleware* bezeichnet wird und für die Übertragung der Nachrichten bzw. Ereignisse verantwortlich ist. Die Kommunikation erfolgt grundsätzlich asynchron, wodurch keine zeitliche Abhängigkeit zwischen dem Versand und dem Empfang einer Nachricht entsteht. Dies führt dazu, dass verlässliche Messaging-Systeme sich positiv auf die Robustheit und die Verfügbarkeit des Systems auswirken [Sta15]. Abbildung 7 zeigt die Übertragung von einer Eventquelle zur Eventsenke über die Nutzung einer Message-Queue.

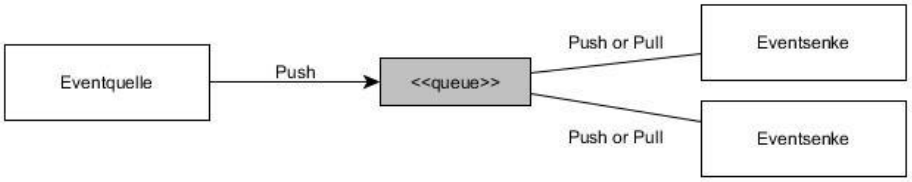


Abbildung 7: Message-Queuing innerhalb einer eventbasierten Architektur

Darüber hinaus bietet die Verwendung einer eventbasierten Architektur gute Integrationsmöglichkeiten, bei denen innerhalb der Eventquellen und Eventsenken komplett andere Technologien zum Einsatz kommen können. Eine tiefergehende Betrachtung von eventbasierten Architekturen in Form von Messaging-Systemen erfolgt in Kapitel 3.

2.3 Enterprise Service Bus

Beim Begriff *Enterprise Service Bus* (ESB) handelt es sich um ein Architekturkonzept, das versucht, die Interaktion und Integration von verschiedenen Anwendungen zu fördern und zu optimieren [Deg05].

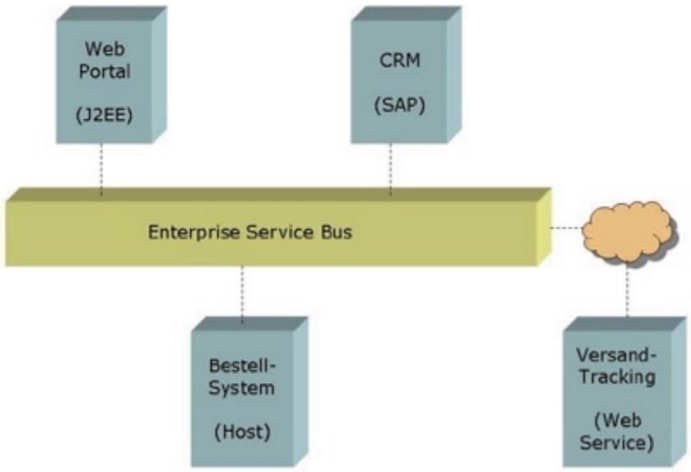


Abbildung 8: [Deg05] Aufbau eines Enterprise Service Bus

Der ESB bildet, wie in Abbildung 8 dargestellt, das Fundament der IT-Struktur innerhalb eines Systems, indem er verschiedene Services und Anwendungen an sich bindet und diese über sich kommunizieren lässt [Deg05]. Es handelt sich bei diesem Ansatz um ein sogenanntes *Compound Integration Pattern*, welches sich aus mehreren feingranulareren Pattern zusammensetzt und dabei das Ziel der Integration einzelner Anwendungen verfolgt [@cip].

In diesem Zusammenhang erfolgt die Verarbeitung ereignisorientiert, wodurch eine *Message Oriented Middleware* einen wesentlichen Bestandteil eines Enterprise Service Bus darstellt. Neben dieser reinen Übertragung der Nachrichten über die zugehörigen Message-Queues und Topics (siehe Kapitel 3) liegen im Verantwortungsbereich des ESB außerdem Aspekte wie die Validierung und Konvertierung der übertragenen Nachrichten [Deg05].

Eine elementare Eigenschaft des ESB besteht darin, dass die Übertragung über ein dokumentenbasiertes Modell erfolgt, in dem jede Nachricht alle für den nächsten Verarbeitungsschritt notwendigen Informationen enthält. Der ESB ist hierbei unabhängig von den eingesetzten Datenformaten und Protokollen [Deg05].

Das Konzept des ESB forciert eine Trennung zwischen fachlichem Code und dem Integrationscode, der benötigt wird, um die Nachrichten über den Bus zu transportieren. Den Entwicklern ermöglicht es, sich primär auf den fachlichen Quellcode zu konzentrieren [Deg05].

Auch wenn es sich beim ESB um einen Architekturstil handelt, gibt es entsprechende Produkte wie *SonicMQ*, welche diesen als vollständiges Produkt anbieten [Deg05].

2.4 Cluster

Das Konzept des Clusterings beschreibt einen Zusammenschluss von redundant vorhandenen Ressourcen innerhalb eines Systems, die von außen als eine Einheit betrachtet werden können [SS12]. Innerhalb eines Clusters sind daher die identisch ausgestatteten Systeme in der Regel über ein lokales Netzwerk miteinanderverbunden und ermöglichen eine Optimierung des Verarbeitungsverhaltens des jeweiligen Teilbereichs des Systems [SS12].

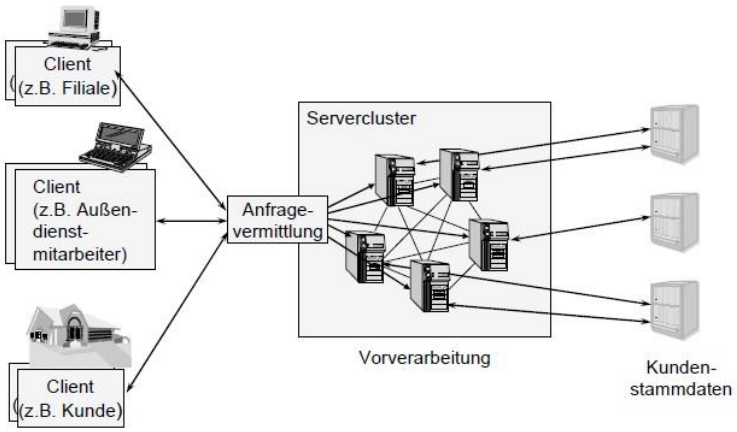


Abbildung 9: [SS12] Aufbau eines Clusters

Abbildung 9 zeigt den beispielhaften Aufbau eines Clusters. Hierbei greifen die verschiedenen Clients über eine Anfragevermittlung auf das Cluster zu. Die Anfragevermittlung weist anschließend die Anfrage einer der Cluster-Instanzen zu.

Die Verwendung eines Clusters hat im Allgemeinen positive Auswirkungen hinsichtlich der ausgeglichenen Verteilung der eintreffenden Last auf die vorhandenen Instanzen sowie auf die Fehlertoleranz und die parallele Verarbeitung von Anfragen [SS12].

2.5 Kommunikationsarten

Im Rahmen der Kommunikation von verteilten Systemen wird allgemein zwischen synchroner und asynchroner Kommunikation unterschieden [Sta15].

Synchrone Kommunikation beschreibt ein stark zeitabhängiges Vorgehen, bei dem die anfragende Instanz nach dem Versand in ihrem Zustand verharrt und auf die notwendige Antwort der empfangenden Instanz wartet [Sta15].

Im Fall der asynchronen Kommunikation erwartet die sendende Instanz keine (direkte) Antwort der empfangenden Instanz, sondern fährt anschließend unmittelbar mit der weiteren Verarbeitung fort [Sta15]. Durch dieses Vorgehen werden beide Komponenten voneinander zeitlich entkoppelt und dabei die Produktivität der sendenden Instanz gesteigert, indem diese im Anschluss an die Nachrichtenübertragung nicht unproduktiv auf die Antwort des Empfängers warten muss

[Sta15]. Bei dieser Kommunikationsart erfolgt die Übertragung grundsätzlich über eine Zwischenkomponente wie eine *Message Oriented Middleware*, welche die Nachrichten für die empfangende Instanz zwischenspeichert [Sta15].

Im Rahmen der Kommunikation sind grundsätzlich die drei verschiedenen Kommunikationsstile *Remote-Procedure Call* (RPC), *Publish/Subscribe* und *Broadcast* vorhanden [Sta15]. Bei einem sogenannten *Remote Procedure Call* handelt es sich um den direkten Aufruf einer entfernten Methode der empfangenden Instanz. Bei diesem Vorgehen weisen die kommunizierenden Instanzen grundsätzlich eine verhältnismäßig hohe Kopplung zueinander auf.

Beim *Publish/Subscribe* Stil meldet die empfangende Instanz ein grundsätzliches Interesse an Nachrichten eines bestimmten Themenschwerpunktes an. Anschließend erhält die Instanz jede erzeugte Nachricht zu diesem Themenschwerpunkt. Durch die indirekte Kommunikation über sogenannte Topics sind die kommunizierenden Instanzen bei diesem Kommunikationsstil grundsätzlich nur lose gekoppelt [Sta15].

Im Fall des *Broadcastings* liegt eine noch deutlich geringere Kopplung zwischen den Kommunikationspartnern vor, da die sendende Instanz die jeweilige Nachricht an alle potenziellen Empfänger überträgt [Sta15]. Dieses Vorgehen sorgt jedoch für einen sehr hohen Overhead, da alle Nachrichten auch an Instanzen übertragen werden, die kein Interesse an der Verarbeitung dieser haben.

Zuletzt ist im Zusammenhang der Kommunikation relevant, über welche Protokolle diese erfolgt. Hierbei kommen üblicherweise Basis-Protokolle wie TCP/IP oder UDP zum Einsatz. Auf diese Basisprotokolle aufbauend werden üblicherweise Protokolle einer höheren Schicht des OSI-Referenzmodells wie HTTP eingesetzt [Sta15].

2.6 Message Oriented Middleware

Die tiefergehende Betrachtung von JMS bzw. einer *Message Oriented Middleware* (MOM) anhand von *ActiveMQ* erfolgt in Kapitel 3. An dieser Stelle werden lediglich die wesentlichen Vorteile betrachtet sowie eine ergänzende Übersicht über vorhandene Messaging Produkte geboten.

Eine *Message Oriented Middleware* ist eine von den sendenden und empfangenden Komponenten getrennte Anwendung, die für die zuverlässige Übertragung der ausgetauschten Daten zuständig ist. Die Übertragung der Daten erfolgt dabei über den Austausch von in sich geschlossenen Nachrichten [Sta15].

Der Einsatz einer *Message Oriented Middleware* besitzt den Vorteil, dass durch die zusätzliche Abstraktionsebene der Middleware sowie den Datenaustausch über Nachrichten eine Kommu-

nikation über Systemgrenzen hinweg ermöglicht wird, welche unabhängig von den eingesetzten Programmiersprachen ist. Darüber hinaus sorgt der Einsatz einer MOM dafür, dass deutlich weniger Abhängigkeiten zwischen den kommunizierenden Instanzen entstehen, da diese über die indirekte Kommunikation voneinander entkoppelt werden [Sta15].

2.6.1 RabbitMQ

RabbitMQ [@rab] ist ein von der Firma *Pivotal Software* entwickelter Open Source Messaging Broker, welcher im Jahre 2007 erstmalig veröffentlicht wurde [Ion15]. Die Brokersoftware ist in der Programmiersprache Erlang geschrieben und ist grundsätzlich nicht JMS-konform. Es wird jedoch ein JMS-Plugin bereitgestellt. [@rjc].

2.6.2 WebSphere MQ

WebSphere MQ ist eine proprietäre Middleware, die auf Nachrichten basiert und im Jahre 1994 von der Firma IBM veröffentlicht wurde [@wsm]. Das Produkt wird kontinuierlich weiterentwickelt und kommt dabei in zahlreichen produktiven Systemen zum Einsatz.

2.6.3 OpenMQ

OpenMQ [@omq] ist eine von *Sun Microsystems* entwickelte Open Source *Message Oriented Middleware*, welche die JMS-API implementiert. *OpenMQ* ist als Standard JMS Provider in den GlassFish Application Server von Oracle integriert [@omq2].

2.6.4 SonicMQ

SonicMQ [@snc] ist eine proprietäre *Message Oriented Middleware*, die vom Unternehmen *Progress-Software* entwickelt wurde. Das Produkt bietet eine skalierbare und verlässliche Infrastruktur zur Übertragung von Nachrichten [@snc2]. Neben einer Kommunikation über JMS bietet *SonicMQ* ähnlich wie *ActiveMQ* weitere über den Standard hinausgehende Features zur Übertragung an [@snc2].

2.6.5 HornetQ

HornetQ ist ein in Java geschriebenes *Message Oriented Middleware* Produkt, welches vom Unternehmen JBoss entwickelt wurde. HornetQ ist die Weiterentwicklung des *JBoss Messaging 2.0* Produktes und kann unabhängig vom zugehörigen JBoss Application Server eingesetzt werden [@com].

2.7 Anforderungen

Dieser Abschnitt befasst sich mit den Anforderungen, die an ein System oder eine Software gestellt werden können. Hierbei wird grundsätzlich zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden [Kle13]. Hierzu befasst sich der Abschnitt 2.7.1 mit den funktionalen und der Abschnitt 2.7.2 mit den nicht-funktionalen Anforderungen.

2.7.1 Funktionale Anforderungen

Funktionale Anforderungen beschreiben eine vom Kunden oder anderen Stakeholdern gewünschte Funktionalität eines Systems. Im Vordergrund steht dabei, was ein System beim Eintritt eines bestimmten Ereignisses oder zu einem bestimmten Zeitpunkt zu erfüllen hat. Stakeholder sind in diesem Zusammenhang alle Personen, die mit dem zu entwickelnden Produkt Berührungspunkte haben und Anforderungen an dieses formulieren können [Kle13].

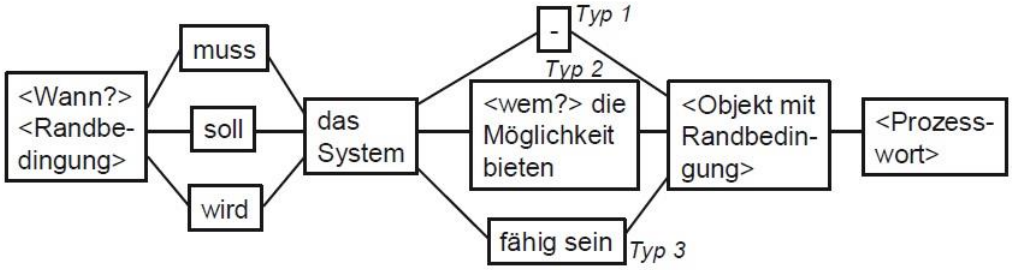


Abbildung 10: [Kle13] Anforderungsschablone nach Rupp

Die funktionalen Anforderungen können im Rahmen der Anforderungsanalyse unter anderem durch die in Abbildung 10 dargestellte Rupp-Schablone definiert werden. Es steht durch eine einheitliche Vorgabe des Satzbaus und den darin enthaltenen Elementen eine vollständige und präzise Beschreibung der jeweiligen Anforderung im Vordergrund.

Im ersten Schritt der Schablone wird hinterfragt, *wann* eine bestimmte Aktion vom System zu erfolgen hat. Anschließend wird durch die Wahl zwischen *muss*, *soll* oder *wird* die Verbindlichkeit der jeweiligen Anforderung festgelegt [Kle13]. Der Wert *muss* gibt an, dass die Aktion zwingend vom Endprodukt zu erfüllen ist. Der Wert *soll* gibt an, dass es aus Kundensicht als sinnvoll betrachtet wird, diese Funktionalität vorzusehen. Zuletzt besagt *wird*, dass der Kunde die Anforderung gerne hätte, diese aber gegebenenfalls Bestandteil eins Folgeprojektes ist [Kle13].

Anschließend wird festgelegt, ob die Anforderung eigenständig durch das System (Typ 1), durch die Interaktion mit einem Benutzer (Typ 2) oder unter Einbeziehung eines Fremdsystems (Typ 3) durchzuführen ist. Darauf folgend wird ein Objekt einschließlich Randbedingung angegeben, für welches die Anforderung greifen soll sowie ein Prozesswort, das definiert, welche Art von Prozess (bspw. *speichern*) durchgeführt werden soll [Rup09].

2.7.2 Nicht-Funktionale Anforderungen

Nicht-funktionale Anforderungen (non-functional requirements; NFR) beschreiben im Gegensatz zu funktionalen Anforderungen keine direkte Funktionalität des Systems, sondern Eigenschaften dieses sowie äußere Rahmenbedingungen. Allgemein lassen sich die NFR in die Bereiche *Qualitätsanforderungen*, *technische Anforderungen*, *sonstige Lieferbestandteile* und *vertragliche Anforderungen* unterteilen [Kle13]. Im Fokus dieser Arbeit stehen im Wesentlichen die Qualitätsanforderungen.

Da die nicht-funktionalen Anforderungen maßgeblichen Einfluss auf das Verhalten und die Qualität des Systems haben, wird sich im Rahmen der Software-Architektur verstärkt mit dieser Art von Anforderungen auseinandergesetzt. Ein Beispiel hierfür ist die Verfügbarkeit des Systems. Diese nicht-funktionale Anforderung ist im Wesentlichen unabhängig von funktionalen Anforderungen wie beispielsweise Bearbeitungsmöglichkeiten eines Benutzer-Accounts. Um jedoch den Verfügbarkeitsanforderungen gerecht zu werden, müssen bei der Definition der zugehörigen Software-Architektur entsprechende Maßnahmen, wie die redundante Auslegung der verschiedenen Komponenten, ergriffen werden. Gleiches gilt beispielsweise für die Skalierbarkeit. Sofern keine Maßnahmen erfolgen, die es beispielsweise ermöglichen, allen Instanzen einen gemeinsamen Datenbestand zugänglich zu machen, ist die Skalierbarkeit entsprechend nicht gegeben.

Die für diese Arbeit relevanten Anforderungen lauten *Verfügbarkeit*, *Skalierbarkeit*, *Zuverlässigkeit*, *Robustheit*, *Wart- und Änderbarkeit*, *Portierbarkeit*, *Performance* sowie *Interoperabilität*. Eine Beschreibung dieser einzelnen Anforderungen erfolgt im Rahmen der Abbildung dieser auf die jeweiligen Messaging-Systeme in Kapitel 6. Vorgreifend wird im Rahmen der zentralen Anforderung der *Zuverlässigkeit* zwischen den in Tabelle 1 dargestellten Zuverlässigkeitsklassen unterschieden [@qos].

Zuverlässigkeitsklasse	**Bedeutung**
At-most-Once	Messaging-System garantiert, dass die Nachrichten maximal einmal ankommen, toleriert jedoch einen Nachrichtenverlust.
At-Least-Once	Messaging-System garantiert, dass die Nachricht ankommen, toleriert jedoch die Erzeugung von Duplikaten.
Exactly-Once	Messaging-System garantiert, dass eine Nachricht exakt einmal ausgeliefert wird.

Tabelle 1: Zuverlässigkeits-Klassen

Weitere Informationen zu nicht-funktionalen Anforderungen bzw. Anforderungen im Allgemeinen können [Kle13] entnommen werden.

3 ActiveMQ & Apache Kafka

In diesem Kapitel werden die beiden in dieser Arbeit betrachteten Systeme *ActiveMQ* und *Apache Kafka* sowie deren Konzepte beschrieben. Hierzu befasst sich Abschnitt 3.1 mit den Konzepten und Eigenschaften von *ActiveMQ* sowie Abschnitt 3.2 mit jenen von *Apache Kafka*.

3.1 Apache ActiveMQ

In diesem Abschnitt werden die Konzepte bzw. Grundlagen des *Java Message Service* (JMS) repräsentativ durch das System *ActiveMQ* dargestellt. In diesem Zusammenhang ist anzumerken, dass *ActiveMQ* die JMS-Spezifikation vollständig abbildet, diese jedoch um eigene Features und Konzepte erweitert. Sofern im Folgenden reine *ActiveMQ*-Features thematisiert werden, wird dieses entsprechend gekennzeichnet.

Allgemein dient die JMS-Spezifikation [@jsr], wie bereits in Kapitel 1 erwähnt, dazu, eine Vielzahl von Messaging-Lösungen zu vereinheitlichen. Dieses Vorgehen bietet allen Herstellern von Message-Brokern entsprechende Regularien bei der Erstellung ihres Messaging-Produktes. Darüber hinaus schafft die Spezifikation durch diese Regularien Transparenz innerhalb der Nachrichtenübertragung. Hierdurch können fundierte Einschätzungen zu Qualitätsanforderungen geboten werden. Grundsätzliche Informationen sind [RMC09] und ergänzend dazu [SBD11] zu entnehmen.

Im weiteren Verlauf dieses Abschnitts werden zu Beginn in Abschnitt 3.1.1 die beiden Messaging-Modelle von JMS bzw. *ActiveMQ* erläutert. Anschließend wird in Abschnitt 3.1.2 näher auf die JMS-API eingegangen. Der darauffolgende Abschnitt 3.1.3 befasst sich mit dem Aufbau einer JMS-Nachricht. Thema des Abschnitts 3.1.4 ist die Filterung von Nachrichten. Die anschließenden Abschnitte 3.1.5 und 3.1.6 befassen sich mit Persistent-Messaging sowie einer verlässlichen Nachrichtenübertragung. Zuletzt wird im Abschnitt 3.1.7 Bezug auf den Einsatz mehrerer Broker genommen.

3.1.1 Messaging Modelle

JMS unterscheidet grundsätzlich zwischen zwei Messaging-Modellen. Diese Modelle lauten *Point-to-Point*- Messaging (P2P) sowie *publish/subcribe*-Messaging (pub/sub). Die Verwendung von zwei Messaging-Modellen ist historisch bedingt, da vor der JMS-Spezifikation die meisten Hersteller einen dieser beiden Ansätze verfolgten. Um eine möglichst große Menge an Systemen zu vereinheitlichen, wurden beide Messaging-Ansätze in die JMS-Spezifikation übernommen [RMC09]. Abbildung 11 stellt die Vorgehensweise der beiden Messaging-Modelle gegenüber. Grundsätzlich werden im Zusammenhang des *P2P*-Messagings die Nachrichten auf dem Broker in einer Queue verwaltet. Beim *pub/sub*-Messaging hingegen werden die Nachrichten einem Topic hinzugefügt.

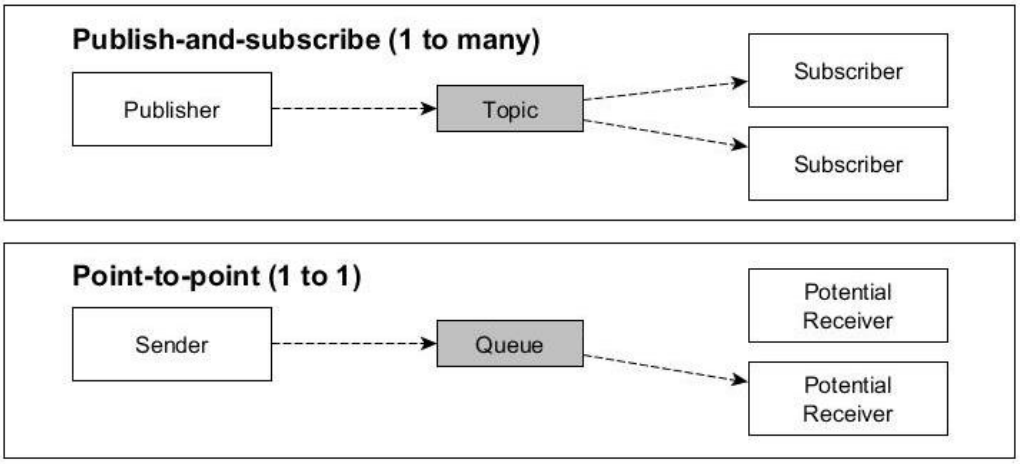


Abbildung 11: JMS Messaging-Modelle

Der wesentliche Unterschied zwischen beiden Messaging-Modellen besteht darin, dass beim Einsatz einer Queue die Nachricht immer nur von einem Consumer auswertet wird, obwohl gegebenenfalls mehrere potenzielle Empfänger vorhanden sind [RMC09]. Auf diese Weise lassen sich beispielsweise Nachrichten gezielt und einmalig an eine bestimmte Server-Instanz übertragen. Dieses Vorgehen ermöglicht darüber hinaus eine entsprechende Lastverteilung, da die Nachrichten gleichmäßig über mehrere identisch ausgelegte Serverinstanzen verteilt werden können [RMC09]. Da in manchen Fällen gefordert wird, dass Folgenachrichten immer von dem gleichen Consumer (bzw. Server) verarbeitet werden, bietet JMS entsprechende Mechanismen, um das zu gewährleisten [RMC09]. Dies kann u. a. durch die in Abschnitt 3.1.4 erläuterte Nachrichtenfilterung erfolgen. Eine Eigenschaft des *P2P* Modells besteht darin, dass die Nachricht so lange auf dem Broker vorliegt, bis diese einmalig an einen Consumer übertragen werden konnte.

Im Gegensatz zum *Point-to-Point* Messaging-Modell werden im *pub/sub*-Modell die Nachrichten einem Topic hinzugefügt und anschließend direkt an alle aktiven Consumer ausgeliefert, die sich auf dem Broker für dieses Topic angemeldet haben. In diesem Zusammenhang sind inaktive Consumer nicht in der Lage, nachträglich Nachrichten abzurufen, die innerhalb ihrer Abwesenheit verfügbar waren [RMC09]. Das *pub/sub*-Modell lässt sich grundsätzlich mit dem Vorgehen im Rundfunk vergleichen.

Da in manchen Sonderfällen auch die inaktiven Consumer nachträglich Zugriff auf die Nachrichten erhalten müssen, die während ihrer Abwesenheit übertragen wurden, bietet JMS die Möglichkeit, einen sogenannten *Durable-Subscriber* zu erstellen. In diesem Fall werden die ausgelieferten Nachrichten für alle Durable-Subscriber persistent im Speicher abgelegt [SBD11]. Da dieses Vorgehen den kompletten Nachrichtenverlauf für jeden abwesenden Consumer exklusiv auf der Festplatte ablegt, sollte sorgsam mit dem Einsatz von Durable-Subscribern umgegangen und die clientseitige dynamische Erstellung dieser unterbunden werden.

3.1.2 JMS-API

Die JMS-API ermöglicht es Entwicklern bzw. Java-Anwendungen, mit einem JMS-Broker wie *ActiveMQ* zu interagieren. Die API gibt in diesem Zusammenhang die notwenigen Schritte und Maßnahmen vor, um eine JMS-konforme Nachricht zu erzeugen und an den Broker zu übertragen [RMC09].

3.1.2.1 Versand von Nachrichten

Das exemplarische Vorgehen beim Aufbau einer JMS-Verbindung wird in Abbildung 13 dargestellt. Zu Beginn wird über das Objekt *IntitialContext* eine JNDI-Referenz aus der *jndi.properties*-Datei geladen, welche Zugriff auf die Topics und Queues des Brokers ermöglicht. Diese Datei enthält außerdem die Authentisierungs-Daten zum Broker, die Broker-URL sowie Verweise auf benötigte Destination-Objekte.

```
1   java.naming.factory.initial = org.apache.activemq.jndi.ActiveMQInitialContextFactory
2   java.naming.provider.url = tcp://esbsvr03.ov.otto.de:9093
3   java.naming.security.principal=system
4   java.naming.security.credentials=manager
5   connectionFactoryNames = QueueCF
6   queue.queue1 = queue1
```

Abbildung 12: JNDI-Properties

Abbildung 12 zeigt die enthaltenen Konfigurationsparameter innerhalb dieser *jndi.properties*-Datei. In Zeile 2 wird in diesem Zusammenhang die URL zum Broker angegeben. Anschließend geben die Zeilen 3 und 4 die Zugangsdaten zum Server an, mit denen sich der Client gegenüber dem Broker authentisiert. Zuletzt werden Objekte angegeben, die sich auf dem Broker befinden und während des Verbindungsaufbaus referenziert werden.

```
37  InitialContext ctx = new InitialContext();
38  QueueConnectionFactory queueFactory = (QueueConnectionFactory) ctx.lookup("QueueCF");
39
40  queueConnection = (QueueConnection)queueFactory.createConnection();
41  queueSession = queueConnection.createQueueSession(true, Session.SESSION_TRANSACTED);
42  queue = (Queue) ctx.lookup(_queue);
43  queueSender = (QueueSender) queueSession.createSender(queue);
44  queueConnection.start();
```

Abbildung 13: Aufbau einer JMS-Connection

Weiterführend im Beispiel in Abbildung 13 wird in Zeile 38 ein JNDI-Lookup vollzogen, der eine *connectionFactory* zurückliefert. Das Software-Pattern Factory dient allgemein dazu, gebündelt verschiedene Ausprägungen von Objekten zu erstellen.

Darauffolgend wird in Zeile 40 eine *queueConnection* erzeugt, die eine allgemeine Verbindung zum Broker aufbaut. Basierend auf dieser *connection* wird anschließend in Zeile 41 eine *session* erzeugt, in welcher festgelegt wird, wie die Nachrichtenquittierung (*acknowledgement*) erfolgt und außerdem, ob diese transaktionsbasiert ist.

Anschließend wird mittels JNDI in Zeile 42 eine Referenz auf eine spezifische Queue ermittelt und in Zeile 43 ein *QueueSender* erstellt, dem diese Queue zugewiesen wird. Ein *QueueSender* ist damit immer exklusiv für den Versand von Nachrichten in eine bestimmte Queue zuständig [RMC09]. Zuletzt wird die Connection in Zeile 44 gestartet, die es anschließend ermöglicht, Nachrichten in die definierte Queue zu übertragen.

```
55          TextMessage message = queueSession.createTextMessage(text);
56          queueSender.send(queue, message, DeliveryMode.NON_PERSISTENT, 4, 0);
```

Abbildung 14: Übertragung einer JMS-Nachricht

Der in Abbildung 14 gezeigte Nachrichtenversand beginnt in Schritt eins mit der Erstellung der Nachricht über die *QueueSession*. Im zweiten Schritt überträgt der *QueueSender* die Nachricht über die *send()*-Methode an den Broker. In diesem Rahmen geben die ersten beiden Methodenparameter die Ziel-Queue sowie die zugehörige Nachricht an. Der dritte Parameter legt fest, welche Persistierungsart für diese Nachricht verwendet werden soll. Der Default-Wert gibt die persistente Speicherung der Nachricht auf der Festplatte des Brokers vor. Eine detaillierte Beschreibung der Übertragungsarten *persistent* und *non-persistent* erfolgt in Abschnitt 3.1.5.

3.1.2.2 Empfang von Nachrichten

Auf Seiten des Nachrichtenempfangs zeigt sich ein ähnliches Vorgehen für den Verbindungsaufbau. Hier wird in Abbildung 15 ebenfalls ein JNDI-Lookup durchgeführt, um anschließend über die zurückgelieferte *connectionFactory* eine Verbindung zum Broker herzustellen. Anschließend wird erneut eine *session* erstellt und die jeweilige *queue* ermittelt, aus der Daten abgerufen werden sollen. Im Gegensatz zum Nachrichtenversand wird im nächsten Schritt in Zeile 43 kein *QueueSender,* sondern ein *QueueReceiver* erstellt, dem im Anschluss ein Objekt zugewiesen wird, welches für die Entgegennahme der Nachrichten zuständig ist. Zuletzt wird analog zum Nachrichtenversand die Connection gestartet.

```
37          InitialContext ctx = new InitialContext();
38          QueueConnectionFactory queueFactory = (QueueConnectionFactory) ctx.lookup("QueueCF");
39
40          queueConnection = (QueueConnection)queueFactory.createConnection();
41          queueSession = queueConnection.createQueueSession(true, Session.CLIENT_ACKNOWLEDGE);
42          queue = (Queue) ctx.lookup(_queue);
43          queueReceiver = (QueueReceiver) queueSession.createReceiver(queue);
44          queueReceiver.setMessageListener(this);
45          queueConnection.start();
```

Abbildung 15: Aufbau einer JMS-Connection (Consumer)

Im Anschluss an diesen Verbindungsaufbau erfolgt der Nachrichtenempfang entweder über die Implementierung eines MessageListeners (*onMessage()*-Methode) oder durch den direkten Aufruf der receive()-Methode. Die Auswertung richtet sich anschließend nach dem verwendeten Nachrichten-Typ. Im Fall einer *TextMessage* wird nach einem *parsing* über die Methode *getContent()* der Inhalt ausgelesen. Eine Beschreibung der vorhandenen Message-Typen erfolgt

im Abschnitt 3.1.3.3. Abschließend besteht – der Vollständigkeit halber – die Möglichkeit, die in Kapitel 2 beschriebenen *Message Driven Beans* für den Empfang einzusetzen.

3.1.3 Anatomie einer JMS-Nachricht

Dieser Abschnitt befasst sich mit der Struktur einer JMS-Nachricht, den enthaltenen Meta-Daten sowie den damit verbundenen JMS-Konzepten. Da im JMS-Kontext alle Informationen und Daten im Rahmen von Nachrichten übertragen werden, ist der Aufbau einer Nachricht einer der zentralsten Aspekte der JMS-Spezifikation [RMC09]. Die enthaltenen Meta-Daten geben Aufschluss darüber, wie mit dem Inhalt umgegangen werden soll oder in welcher Form die entsprechende Antwort zu erfolgen hat [RMC09].

Abbildung 16: Anatomie einer JMS-Nachricht

Eine JMS-Nachricht setzt sich, wie in Abbildung 16 dargestellt, grundsätzlich aus den Bereichen Header, Properties sowie dem Payload zusammen. Diese Bereiche werden in den folgenden Abschnitten detailliert beschrieben. In diesem Zusammenhang befasst sich der Abschnitt 3.1.3.1 mit den Attributen des Headerbereichs. Anschließend wird der Einsatzzweck des Properties-Bereiches erläutert (Abschnitt 3.1.3.2). Darauffolgend werden für den Payload-Bereich, in dem sich der eigentliche Nachrichteninhalt befindet, die verschiedenen Nachrichtentypen betrachtet (Abschnitt 3.1.3.3).

3.1.3.1 Header-Bereich

Der Headerbereich enthält eine Menge an Standard-Headerfeldern, die Informationen darüber aufweisen, wie mit der Nachricht während und nach der Übertragung umzugehen ist [SBD11]. Diese Menge an Header-Feldern unterteilt sich in die zwei Bereiche *Automatisch-zugewiesene* und *Entwickler-zugewiesene* Felder [RMC09]. Erstere werden zwar automatisch von der JMS-API bzw. vom verwendeten JMS-Broker gesetzt, der Client hat jedoch indirekt Einfluss auf diese. Zu diesen Header-Feldern gehört unter anderem eine *JMSMessageID*, welche eine eindeutige Nachrichtensequenznummer enthält sowie der *JMSDeliveryMode*, der angibt, ob *persistent* oder *non-persistent* Messaging eingesetzt wird.

Die enthaltenen Felder *JMSTimestamp* und *JMSExpirationdate* geben an, wann die Nachricht erstellt wurde und nach welchem Zeitraum – ausgehend vom Timestamp – die Nachricht veraltet und damit vom Broker gelöscht werden soll.

Das Feld *JMSRedelivered* wird im Broker gesetzt, sofern eine Nachricht erneut an den Consumer übertragen werden musste, da dieser beim ersten Versuch den Erhalt nicht bestätigt hat. Hierdurch wird der Consumer darauf hingewiesen zu prüfen, ob er die Nachricht bereits verarbeitet hat [SBD11].

Zuletzt gibt bei den automatisch zugewiesenen Headerfeldern *JMSDestinationType* an, ob die Nachricht im Rahmen einer Queue oder eines Topics übertragen wurde und wie der Name dieser *Destination* lautet.

Jene Felder, die vom Entwickler gesetzt werden können, lauten *JMSReplyTo*, *JMSCorrelationID* sowie *JMSType*. Das Feld *JMSReplyTo* dient dazu, dem Empfänger der Nachricht mitzuteilen, an wen er die Antwort zu adressieren hat. Hierdurch können komplexere Verarbeitungsschritte durch das Gesamt-System abgebildet werden, in dem die stellenweise verarbeitete Nachricht immer weiter herumgereicht wird. Des Weiteren können hierdurch simple *request/reply*-Mechanismen abgebildet werden, bei denen der Sender synchron auf die Antwort des Empfängers wartet. Eng mit diesem Vorgehen verbunden ist das Headerfeld *JMSCorrelationID*. Dieses ermöglicht es, der antwortenden Instanz in ihrem Response die Sequenznummer der ursprünglichen Nachricht zu hinterlegen, damit der Empfänger prüfen kann, ob es sich um die Antwort seines Requests handelt. Zuletzt gibt das Feld *JMSType* Aufschluss darüber, welcher Nachrichtentyp sich im Payload-Bereich der Nachricht befindet [SBD11].

3.1.3.2 Properties-Bereich

Die Properties im Properties-Bereich agieren wie zusätzliche Headerfelder und ermöglichen es dem Entwickler, ergänzende Informationen zur Nachricht hinzuzufügen. Darüber hinaus werden diese zur Nachrichten-Filterung verwendet, welche im Abschnitt 3.1.4 genauer beschrieben wird. Die Properties werden als Key-Value-Pairs gespeichert und können als String, boolean, byte, double, int, long sowie float abgelegt werden. Die JMS-API stellt für den Zugriff auf die Properties für jeden Datentyp getter()- und setter()-Methoden zur Verfügung. Eine neue String-Property wird damit per *message.setStringProperty(„username“, username);* hinzugefügt.

Neben der Definition dieser anwendungsspezifischen Properties sind zusätzlich noch JMS-Spezifische Properties vorhanden. Diese ermöglichen beispielsweise das Gruppieren von Nachrichten. Weitere Informationen hierzu können [RMC09] entnommen werden.

3.1.3.3 Payload-Bereich (Nachrichten-Typen)

JMS definiert Schnittstellen für 6 verschiedene Nachrichtentypen, die vom Broker-Hersteller implementiert werden müssen [SBD11]. Zu diesen Typen gehören *Message*, *TextMessage*, *ObjectMessage*, *BytesMessage*, *StreamMessage* sowie *MapMessage*.

Der Basis-Typ *Message* stellt die simpelste Form einer Nachricht dar. Eine Nachricht dieses Typs besitzt keinen Payload und wird damit primär als *Event-Notification* eingesetzt. Die Subinterfaces dieses Basistyps stellen entsprechend komplexere Nachrichtenstrukturen zur Verfügung. Der Nachrichtentyp *TextMessage* überträgt beispielsweise eine String-Zeichenkette, die zuvor beliebig formatiert werden kann [RMC09].

Mit dem Nachrichtentyp *ObjectMessage* können serialisierbare Java-Objekte innerhalb einer Nachricht übertragen werden. Dieser Nachrichten-Typ setzt jedoch voraus, dass sowohl der Sender als auch der Empfänger in Java geschrieben und JMS-kompatibel ist [RMC09]. Darüber hinaus muss die serialisierte Klasse auf beiden Seiten verfügbar sein. Dieses Vorgehen sorgt grundsätzlich für eine stärkere Koppelung der Systeme als beispielsweise der Einsatz einer *TextMessage*.

Der Nachrichtentyp *BytesMessage* überträgt ein Array mit primitiven Bytes. Dieser Nachrichtentyp eignet sich für die reine Übertragung von Anwendungsdaten im Rohformat [RMC09]. Im Gegensatz dazu überträgt die *StreamMessage* eine Abfolge von primitiven Java-Typen. Gespeicherte Werte werden dabei in der gleichen Reihenfolge ausgelesen, wie diese gespeichert wurden [RMC09]. Zuletzt bietet der Nachrichten-Typ *MapMessage* die Möglichkeit, eine *Map* aus *Key-Value-Pairs* in einer Nachricht zu übertragen [RMC09].

3.1.4 Message Filterung

ActiveMQ bietet den Consumern die Möglichkeit, Nachrichten vor dem Empfang zu selektieren. Hierbei wird ein so genannter *MessageSelector* einem *QueueReceiver* oder *TopicSubscriber* bei der Erstellung hinzugefügt [RMC09]. Anschließend werden nur noch Nachrichten an die zugehörigen Comsumer übertragen, die der – im Selektor enthaltenen – Bedingung entsprechen.

Ein *MessageSelector* setzt sich aus Bezeichnern (*indentifiers*), Werten (*literals*) sowie Vergleichsoperatoren (*comparison operators*) zusammen. Die Bezeichner dienen dazu, eine bestimmte *Property* oder ein Header-Feld innerhalb der Nachricht zu identifizieren. Der darin enthaltene Inhalt wird anschließend über den Vergleichsoperator mit einem vorher definierten Wert verglichen. In diesem Zusammenhang ist es nicht möglich, den eigentlichen Nachrichteninhalt in den Verarbeitungsprozess mit einzubeziehen [RMC09]. Die sendende Instanz muss die benötigten Werte, vor dem Versand der Nachricht, entsprechend in den *Properties* hinterlegen.

```
44            String selector = "price <= '999'";
45            queueReceiver = (QueueReceiver) queueSession.createReceiver(queue, selector);
```

Abbildung 17: Erstellung und Anwendung eines Message-Selektors

Abbildung 17 zeigt in Zeile 44 die Erstellung eines *MessageSelectors* sowie die anschließende Nutzung dieses bei der Erzeugung eines *QueueReceivers*. Im Anschluss an diesen Vorgang werden dem Consumer nur noch Nachrichten übertragen, bei denen der in den Properties gespeicherte Preis unter 1000 (Euro) liegt.

Durch eine sehr strikte Filterung innerhalb aller Consumer kann es dazu kommen, dass Nachrichten dauerhaft im Broker verbleiben und damit von keinem geeigneten Consumer verarbeitet werden. In diesem Fall empfiehlt es sich allgemein, jeder Nachricht ein Ablaufdatum hinzufügen, damit diese nicht auf unbestimmte Zeit im Broker verbleibt.

Darüber hinaus sollte diese Problemstellung bereits bei der Definition einer queue-Struktur berücksichtigt werden. In diesem Zusammenhang ist ein möglicher Ansatz, jeden Consumer mit einem Selektor auszustatten oder für jeden Anwendungsfall eine spezifische Queue zu nutzen. Weitere Informationen hierzu sind [RMC09] zu entnehmen.

3.1.5 (Non) Persistent Messaging

Im JMS-Kontext kann ein Broker grundsätzlich auf zwei verschiedene Arten bei der Übertragung von Nachrichten verfahren. Im Fall des *non-persistent*-Messagings werden die empfangen Nachrichten bis zu ihrer Auslieferung im flüchtigen Arbeitsspeicher des Brokers hinterlegt. Dieses Vorgehen birgt jedoch das Risiko eines Datenverlustes, sofern es zu einem Ausfall des Brokers kommt [SBD11].

Bei der zweiten Art, wie mit der Übertragung von Nachrichten umgegangen werden kann, handelt es sich um das *persistent*-Messaging. Hier werden die Nachrichten, nachdem sie vom Broker entgegen genommen wurden, direkt auf der Festplatte gespeichert und stehen damit auch nach einem Brokerausfall weiterhin zur Verfügung [SBD11].

Die Speicherung und Verwaltung der enthaltenen Nachrichten erfolgt durch sogenannte *Persistence-Stores*. Im Fall von *ActiveMQ* sind unter anderem die Persistence-Stores *KahaDB*, *AMQ Message-Store* sowie der *JDBC Message-Store* verfügbar [SBD11]. In diesem Zusammenhang arbeiten die beiden erstgenannten nach einem Datei-basierten Prinzip. Der *JDBC-Message-Store* dient hingegen dazu, die gespeicherten Nachrichten in einer relationalen Datenbank abzulegen [SBD11]. Im Folgenden wird der *KahaDB*-Message Store näher beschrieben, da es sich bei diesem um einen sehr verbreiteten und schnellen Persistence-Store handelt [SBD11]. Darüber hinaus ist das Vorgehen von *KahaDB* ähnlich zu den Speicherungsmechanismen von *Apache Kafka*.

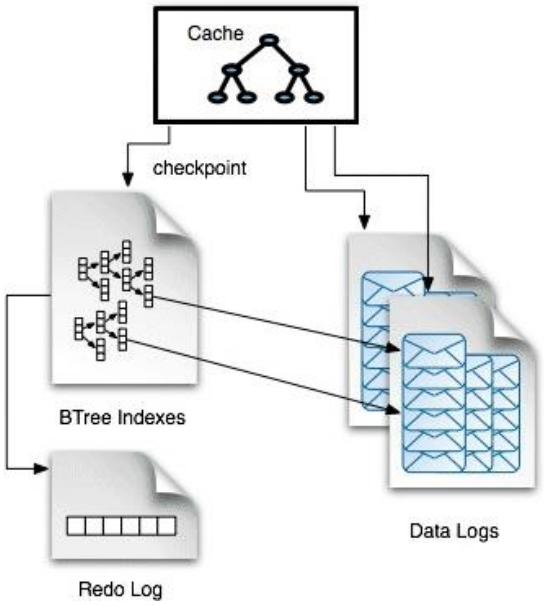


Abbildung 18: [SBD11] Aufbau des KahaDB Persistence-Stores

KahaDB ist ein Persistence-Store, bei dem die jeweiligen Nachrichten in Dateien abgelegt werden. Abbildung 18 zeigt in diesem Zusammenhang den Aufbau bzw. die Struktur des *KahaDB* Persistence-Stores, der sich in drei Bereiche unterteilt. Der Bereich *Data-Logs* enthält Textdateien, in denen die zugehörigen Nachrichten gespeichert werden [SBD11]. Die Dateien besitzen dabei eine einheitliche maximale Größe. Sofern diese maximale Größe erreicht ist, wird die aktuelle Datei geschlossen und für die anschließenden Nachrichten eine neue Datei erzeugt [SBD11]. Auf diese Weise entsteht eine Vielzahl an Dateien, welche wiederum eine große Menge an Nachrichten enthalten.

Der zweite Bereich *cache* beinhaltet temporäre Kopien der in den *Data Logs* gespeicherten Nachrichten, die von aktiven Consumern aktuell oder zeitnah benötigt werden [SBD11]. Dieses Vorgehen dient der schnelleren Auslieferung von persistenten Nachrichten.

Der letzte Bereich *BTree* enthält Referenzen zu den Nachrichten, die innerhalb der *Data-Logs* gespeichert sind und während der Speicherung mit einem *Index* versehen wurden. Dieses Vorgehen führt dazu, Nachrichten möglichst zeiteffizient auffinden zu können, sofern sich diese nicht im Cache befinden [SBD11]. Der zugehörige *redo-log* dient dazu, die gespeicherten Indexe wiederherstellen zu können, sofern der Broker unerwartet ausgefallen ist [SBD11].

```
40        <persistenceAdapter>
41            <kahaDB directory="${activemq.data}/kahadb" indexWriteBatchSize="1"/>
42        </persistenceAdapter>
```

Abbildung 19: Einbindung von KahaDB

Die Nutzung und Konfiguration von *KahaDB* erfolgt, wie in Abbildung 19 dargestellt, über die Konfigurationsdatei *activemq.xml* des Brokers. Hierzu wird innerhalb des *persistenceAdapter-*

Tags das in Zeile 41 dargestellte <*kahaDB ... />*-Tag hinzugefügt, in welchem wieder weitere Konfigurationsparameter definiert werden können. Dazu gehört beispielsweise der Parameter *directory*, der angibt, in welchem Verzeichnis die Nachrichten abgelegt werden sollen. Der zweite Parameter gibt an, wie viele Nachrichten in einem Schreibvorgang auf der Festplatte gespeichert werden sollen. Der enthaltene Wert hat maßgeblichen Einfluss auf die Performance des Persistence-Stores und damit transitiv auf den Broker [SBD11]. Der Wert 1 gibt an, dass eine eintreffende Nachricht immer direkt gespeichert und nicht auf Folgenachrichten gewartet wird. Dieses Vorgehen verhindert, einen Nachrichtenverlust während des Wartevorganges, sofern der Broker unvorhergesehen ausfallen sollte.

Weitere Informationen zum *persistent*-Messaging sowie zu den einzelnen Persistence-Stores können [SBD11] entnommen werden.

3.1.6 Garantierte Nachrichten-Übertragung

Die Mechanismen für eine verlässliche und korrekte Nachrichten-Übertragung mit einer *Exactly-Once* Zustellgarantie befassen sich mit den drei Kernaspekten *Nachrichten-Struktur*, *Store-and-Forward*-Messaging und *Quittierungsmechanismen* [RMC09].

3.1.6.1 Nachrichten-Struktur

Der Aspekt der Nachrichtenstruktur ist dahingehend von Bedeutung, dass es sich bei Nachrichten im Messaging-Kontext um autonome und in sich geschlossene Einheiten handelt [RMC09]. Eine Nachricht wird unter Umständen über viele Stationen innerhalb des Systems weitergereicht. Diese Zwischenstationen analysieren den Nachrichteninhalt und modifizieren diesen. Anschließend senden sie die angepasste Nachricht weiter oder ersetzen diese durch eine neue Nachricht [RMC09].

In diesem Zusammenhang gehen die einzelnen Komponenten (JMS-Clients und JMS-Broker) eine Art Vertrag ein, durch den der sendenden Komponente gewährleistet wird, dass in angemessenem Maße die Übertragung an den Empfänger durchgeführt wird. Nachdem der JMS-Client den Versand der Nachricht abgeschlossen hat, ist die Angelegenheit damit für ihn beendet und der Messaging-Server versichert ihm, dass die entsprechenden Gegenstellen die Nachrichten erhalten werden [RMC09].

3.1.6.2 Store-And-Forward Messaging

Sofern im Nachrichten-Header hinterlegt ist, dass *persistent*-Messaging zum Einsatz kommen soll, ist der Broker dafür verantwortlich, die Nachricht unverzüglich auf der Festplatte zu hinterlegen, um die Vereinbarung mit dem JMS-Client zu erfüllen [SBD11]. Nachdem die Nachricht in einem gesicherten Medium enthalten ist, versucht der Broker diese Nachricht weiter in die Richtung des Empfängers zu leiten. In vielen Fällen ist der Empfänger direkt über eine

Sitzung mit dem Broker verbunden. Dieser *Forwarding*-Mechanismus ist dafür verantwortlich, die zuvor gespeicherte Nachricht aus dem Speicher zu laden, diese anschließend zu *routen* und auszuliefern. Dieses gesamte Vorgehen stellt sicher, dass Nachrichten im Anschluss an einen Broker-Ausfall weiterhin auf diesem verfügbar sind [SBD11].

3.1.6.3 Nachrichten-Quittierung

Im Rahmen der zuvor angesprochenen Nachrichtenauslieferung können verschiedene Quittierungsmechanismen eingesetzt werden. Diese Mechanismen sind ein wesentlicher Bestandteil einer verlässlichen Nachrichtenübertragung. In diesem Rahmen bestätigen zum einen die Broker, den sendenden Producern, den korrekten Erhalt der gesendeten Nachricht. Zum anderen bestätigen die Consumer dem Broker den Erhalt der empfangenen Nachricht [SBD11].

Der verwendete Quittierungsmechanismus ermöglicht es dem Broker, den aktuellen Verarbeitungsstand einer Nachrichtenübertragung zu überwachen und gewährleistet damit eine verlässliche Nachrichtenübertragung [SBD11]. Im Gegensatz zum *Store-And-Forward* Mechanismus ist die Nachrichten-Quittierung nicht dafür zuständig, einem Nachrichtenverlust auf dem Broker vorzubeugen, sondern sicherzustellen, dass die Nachrichten nicht im Rahmen der Übertragung verloren gehen [SBD11].

JMS bietet in diesem Zusammenhang drei verschiedene Übertragungsarten, die AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE und DUPS_OK_ACKNOWLEDGE lauten. Die wesentlichen Unterschiede zwischen diesen Quittierungsarten bestehen darin, dass DUPS_OK_ACKNOWLEDGE eine mehrfache Übertragung einer Nachricht an den gleichen Client akzeptiert und damit den Empfang von Duplikaten hinnimmt [RMC09]. Dieser Modus führt zwar zu einer deutlichen Performance-Steigerung, verhindert jedoch allgemein eine *Exactly-Once* Zustellgarantie. Der wesentliche Unterschied zwischen CLIENT_ACKNOWLEDGE und AUTO_ACKNOWLEDGE besteht darin, dass bei Ersterem die Nachrichten nicht automatisch, sondern manuell vom Client quittiert werden.

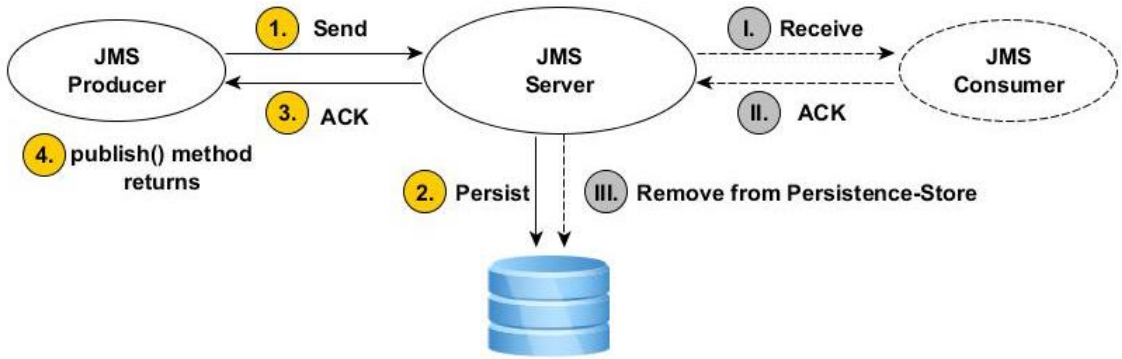


Abbildung 20: Vorgehen bei der Quittierung einer Nachricht

Abbildung 20 zeigt den grundsätzlichen Ablauf einer Nachrichtenübertragung an den Broker mit anschließender Quittierung der erhaltenen Nachricht. In diesem Zusammenhang sendet der JMS-Producer im ersten Schritt die Nachricht an den Broker. Nachdem dieser im zweiten Schritt die Nachricht auf der Festplatte gespeichert hat, bestätigt er dem Producer im dritten Schritt den korrekten Empfang der Nachricht. Dies führt dazu, dass abschließend im Schritt 4 die sendende Methode im Client einen Rückgabewert liefert und der Client mit weiteren Arbeitsschritten fortfahren kann [RMC09].

Im rechten Teil der Abbildung 20 wird angedeutet, wie bei der Auslieferung der Nachricht an den Consumer vorgegangen wird. Es erfolgt zunächst die Auslieferung der Nachricht, deren Erhalt anschließend vom Consumer bestätigt wird. Diese Bestätigung gewährleistet, dass für den Broker die Pflicht zur verlässlichen Nachrichtenübertragung erfüllt ist und dieser die Nachricht abschließend aus dem Persistence-Store entfernen kann.

Dieses Vorgehen gewährleistet eine verlässliche Nachrichtenübertragung, beinhaltet jedoch einen sehr hohen Overhead bei der Nachrichtenübertragung zwischen den Systemen, da jede Nachricht auf dem Übertragungsweg mindestens zweimal quittiert werden muss. Da dieser Overhead zu entsprechenden Durchsatzeinbußen führt, kann diese Ausgangslage durch den Einsatz von Transaktionen deutlich verbessert werden. Weitere Informationen zum Thema *Quittierung* können [RMC09] entnommen werden.

3.1.6.4 Transaktionsbasierte Nachrichtenübertragung

Um den zusätzlichen Overhead für die Quittierung der Nachrichten zu minimieren, bietet *ActiveMQ* bzw. JMS die Möglichkeit, Nachrichten innerhalb eine JMS-Transaktion zu übertragen. In diesem Zusammenhang erfolgt lediglich beim Abschluss der Transaktion durch den Aufruf der *commit()*-Methode eine synchrone Kommunikation zwischen Broker und Client. Die in der Transaktion übertragenen Nachrichten werden jedoch asynchron bestätigt, wodurch die sendende Instanz nicht auf die Quittierung des Empfängers warten muss [RMC09].

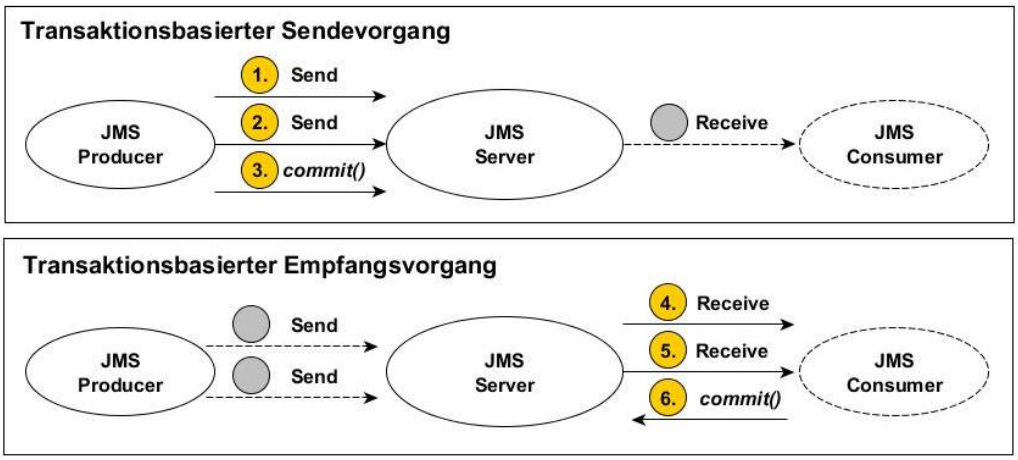


Abbildung 21: Transaktionsbasierte Nachrichtenübertragung

Die obere Grafik in Abbildung 21 zeigt den Ablauf einer Transaktion aus der Perspektive des Producers. Im ersten und zweiten Schritt überträgt der Producer im Rahmen einer *Transacted*-Session zwei Nachrichten an den Broker. Die Nachrichten sind zwar anschließend auf dem Broker vorhanden, jedoch noch nicht für die Consumer sichtbar [RMC09]. Im dritten Schritt wird die Transaktion durch den Aufruf der *commit()*-Methode abgeschlossen und die Nachrichten sind vollständig für die Consumer sichtbar [RMC09]. Die in einer Transaktion gesendeten Nachrichten werden damit komplett oder gar nicht vom Broker akzeptiert. Sofern, statt eines Commits, ein Rollback erfolgt oder die Verbindung abbricht, entfernt der Broker die bereits übertragenen Nachrichten, die zu keinem Zeitpunkt für die Consumer sichtbar waren [RMC09].

Die untere Grafik in Abbildung 21 zeigt den Ablauf einer Transaktion aus der Perspektive des Consumers. Dieser empfängt im 4. und 5. Schritt die Nachrichten und schließt im folgenden Schritt 5 die Transaktion ab. Erst im Anschluss löscht der Broker die ausgelieferten Nachrichten vollständig aus dem Persistence-Store [RMC09]. Die Verwendung von Transaktionen erfolgt durch entsprechende Parameter bei der Erstellung der JMS-Session sowie durch den Aufruf der *commit()*-Methode. Weitere Informationen hierzu sind [RMC09] zu entnehmen.

3.1.7 Einsatz mehrere ActiveMQ-Broker

ActiveMQ ermöglicht eine parallele Verwendung von mehreren Brokern durch zwei verschiedene Ansätze. Im Abschnitt 3.1.7.2 wird hierzu der *Master/Slave* Ansatz beschrieben. Anschließend erfolgt in Abschnitt 3.1.7.3 die Beschreibung des *Network-of-Broker* Ansatzes.

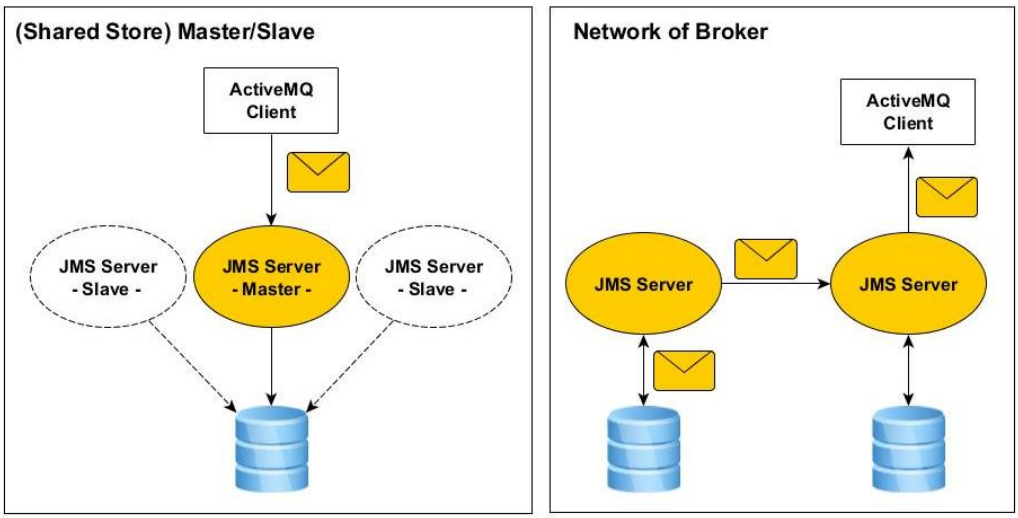


Abbildung 22: Master/Slave (links) vs. Network-of-Brokers (rechts)

3.1.7.1 Konnektoren

ActiveMQ ermöglicht es im Rahmen der Konfiguration über die *activemq.xml* Konnektoren anzugeben, über die mit dem Broker kommuniziert werden kann. Hierbei wird zwischen Trans-

port- und Netzwerk-Konnektoren unterschieden [SBD11]. Erstere dienen dazu, mit den Producern und Consumern zu kommunizieren. Netzwerk-Konnektoren hingegen dienen der Kommunikation zwischen den Brokern untereinander [SBD11].

```
60        <transportConnectors>
61            <transportConnector name="openwire" uri="failover:(tcp://192.168.2.100:61616, tcp://192.168.2.101:61616)?key=value"/>
62        </transportConnectors>
```

Abbildung 23: Definition eines transportConnectors

Abbildung 23 zeigt die Definition eines Konnektors. In dem dargestellten Beispiel wird in Zeile 61 ein Transport-Connector erstellt, der die beiden Broker-URIs enthält. Mit diesen beiden Brokern wird über das TCP-Protokoll kommuniziert. Das umschließende *failover()* gibt an, dass der Client, sofern die Verbindung zum Broker abbricht, sich automatisch mit dem anderen Broker verbinden soll. Detaillierte Informationen zu Konnektoren können [SBD11] entnommen werden.

3.1.7.2 (Shared Storage) Master/Slave

Im Rahmen des Master/Slave Clustering-Ansatzes werden, wie in Abbildung 22 dargestellt, mehrere identische Broker parallel nebeneinander betrieben. Von diesen Brokern ist jedoch zu einem bestimmten Zeitpunkt nur einer für die Entgegenahme und Auslieferung von Nachrichten zuständig (*Master*). Die anderen Broker dienen dazu, einzuspringen, sofern der *Master* ausfällt. Das Besondere an diesem Vorgehen ist, dass alle Broker auf den gleichen Datenbestand zugreifen. Dabei wird das Konzept verfolgt, dass der erste Broker, der auf den Datenbestand zugreift, einen *File-Lock* verursacht und sich damit selbst zum *Master* ernennt. Die *Slaves* können solange nicht auf die Daten zugreifen, bis der *Master* dieses *File-Lock* beispielsweise durch einen Serverausfall wieder freigibt. Anschließend wird jener *Slave* zum *Master*, der als Erster den *File-Lock* für sich in Anspruch nehmen kann [SBD11].

Dieses Vorgehen dient dazu, die System-Verfügbarkeit zu erhöhen und damit die Ausfallwahrscheinlichkeit des Systems in Richtung *Null* zu senken. In diesem Zusammenhang stellt das Speichermedium jedoch einen *Single-Point-of-Failure* dar und muss ebenfalls entsprechend gegen einen Ausfall abgesichert werden.

3.1.7.3 Network of Brokers

Im Gegensatz zum *Master/Slave* Ansatz verfolgt der *Network-of-Brokers* Ansatz das Ziel einer horizontalen Skalierung, bei der mehrere Broker für die Anfragen von Clients zur Verfügung stehen [SBD11].

Da sich ein Client jedoch immer nur mit einem Broker verbindet, beide Broker jedoch gemeinsam den Inhalt einer *queue* oder eines *topics* verwalten, müssen die Daten in vielen Fällen zwischen den Brokern transportiert werden. In diesem Fall kommen Network-Konnektoren zum

Einsatz, über die eine sogenannte *Demand Forwarding Bridge* konfiguriert wird. Über diese Bridge teilt ein Broker dem anderen Broker mit, dass er eine aktive Verbindung zu einem oder mehreren Consumern besitzt, die aus der jeweiligen *queue* konsumieren möchten. Anschließend überträgt der andere Broker, sofern er keine aktiven Consumer für die *queue* besitzt, die Nachrichten an den anderen Broker.

Dieses Vorgehen verdeutlicht die konzeptionelle Problematik von JMS, dass immer nur eine Instanz einer Nachricht innerhalb des Clusters vorhanden sein darf [RMC09]. Sofern diese Nachricht am anderen Ende des Netzwerkes benötigt wird, muss diese über alle sich auf dem Weg befindenden Broker transportiert werden, um sie an den jeweiligen Client auszuliefern. Die Konzepte von JMS lassen in diesem Zusammenhang eine redundante Speicherung der Nachrichten auf mehreren Brokern nicht zu, da in diesem Fall nicht mehr nachvollzogen werden könnte, welche Nachricht bereits konsumiert wurde.

3.2 Apache Kafka

In diesem Abschnitt werden die Konzepte und Vorgehensweisen von *Apache Kafka* beschrieben. Bei diesem System handelt es sich um ein nicht JMS-konformes System, welches andere Konzepte hinsichtlich der Speicherung und Übertragung der Nachrichten verfolgt.

Apache Kafka entstand anfänglich im Rahmen einer Eigenentwicklung des Unternehmens *LinkedIn* [@lkd]. Dieses strebte an, die vorhandenen Probleme, die bei der Übertragung des stetig steigenden Datenaufkommens innerhalb des bestehenden Systems entstanden, zu beheben. Das Ziel bestand darin, ein Messaging System mit einer hohen Performance zu entwickeln, welches mit den verschiedensten Typen von Daten umgehen kann und dabei die Bereitstellung von Nutzeraktivitäten und System-Metriken in nahezu Echtzeit ermöglicht [GL13]. Die erzeugten Daten sollten dabei verschiedenen Systemteilen einheitlich zugänglich gemacht werden. Da das resultierende Produkt darüber hinaus Relevanz für andere Projekte besaß, wurde es anschließend durch *Apache Software Foundation* weiterentwickelt [@apa].

Der Name *Kafka* wurde von Jay Kreps, einem der Gründer von *Apache Kafka,* gewählt, da er den Namen eines Schriftstellers für ein optimiertes System zum Schreiben von Daten als passend erachtete. Ferner hatte er in seiner Schulzeit Berührungspunkte mit Werken von Franz Kafka und empfand, dass sich dieser Name für ein Open Source Projekt eignet [NSP17].

Im weiteren Verlauf wird in Abschnitt 3.2.1 das *Kafka* zugrunde liegende *Apache ZooKeeper* beschrieben. Anschließend erfolgt in Abschnitt 3.2.2 die Beschreibung des zugehörigen Messaging-Modells. Der Abschnitt 3.2.3 befasst sich mit dem Clustering-Konzept dieses Systems. In Abschnitt 3.2.4 erfolgt eine Betrachtung der Basis-Konfiguration eines Kafka-Brokers. Darauffolgend werden im Rahmen des Abschnitts 3.2.5 die zugehörigen Producer und Consumer-Aspekte beschrieben. Der Abschnitt 3.2.6 befasst sich mit den internen Eigenschaften und Charakteristika von *Apache Kafka*. Zuletzt wird im Abschnitt 3.2.7 Bezug auf die verlässliche Datenübertragung genommen. Grundsätzliche Informationen zu *Apache Kafka* sind [NSP17] und ergänzend dazu [Gar15] zu entnehmen.

3.2.1 ZooKeeper

Bei *Apache ZooKeeper* [@zoo] handelt es sich um ein Produkt, welches dazu dient, Aktionen und Vorgänge innerhalb von verteilten System zu koordinieren [JR16]. Als ein robuster Dienst, ermöglicht ZooKeeper den Entwicklern, sich primär auf die Entwickelung der Business-Logik zu konzentrieren [JR16]. In diesem Zusammenhang kommt es in vielen komplexen und bekannten Systemen wie dem *Facebook Messanger*, dem *Yahoo! Fetching Service* sowie *Apache Solr* zum Einsatz [JR16].

In *Apache Kafka* wird *ZooKeeper* ebenfalls als eine Art Fundament verwendet, um die internen Vorgänge und Aktionen innerhalb des Clusters zu steuern. Unter anderem verwaltet *ZooKeeper*

hierzu Meta-Daten über Broker und Topics. Darüber hinaus wurden in frühen Versionen die Offsets (siehe Abschnitt 3.2.2) innerhalb von *ZooKeeper* verwaltet. Diese werden in den neueren Versionen jedoch standardmäßig innerhalb von Topics gespeichert und direkt auf den Kafka-Brokern verwaltet.

Konkret handelt es sich bei *ZooKeeper* ebenfalls um ein verteiltes System, welches in einem Verbund von mehreren Servern agiert. Dieser Verbund wird als *Ensemble* bezeichnet [JR16]. Da über dieses Ensemble im Rahmen von Mehrheitsbeschlüssen neue Leader und Controller (siehe Abschnitt 3.2.6) ernannt werden, empfiehlt es sich, dieses mit einer ungeraden Anzahl an ZooKeeper-Knoten auszustatten [NSP17].

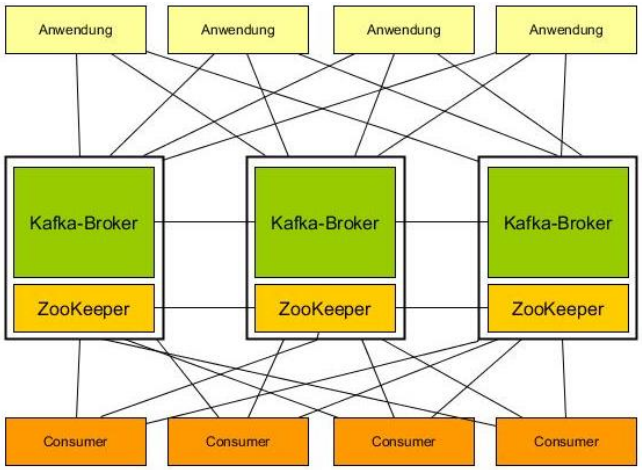


Abbildung 24: Aufbau eines Kafka-Clusters einschließlich ZooKeeper-Ensemble

Im Allgemeinen setzt sich ein Kafka-Cluster – wie in Abbildung 24 dargestellt – aus einem ZooKeeper-Ensemble mit einer ungeraden Anzahl an Teilnehmern sowie einer identischen Anzahl an Kafka-Brokern zusammen. Hierbei kommen üblicherweise 3 *Kafka*- sowie 3 *ZooKeeper*-Instanzen zum Einsatz. Weitere Informationen zu *ZooKeeper* sind [JR16] zu entnehmen.

3.2.2 Messaging Modell

Apache Kafka bietet grundsätzlich im Vergleich zu JMS nur ein Messaging Modell an, bei dem ein *Publish/Subscribe* Ansatz verfolgt wird [NSP17]. Durch eine redundante Speicherung der Daten auf mehreren Brokern und einer Zerteilung eines Topics in mehrere Partitionen bietet *Apache Kafka* jedoch die Möglichkeit, beliebige Konsumierungs-Konstrukte abzubilden [NSP17].

Abbildung 25 zeigt den zugehörigen Aufbau eines Topics. Dieses setzt sich im dargestellten Fall aus 4 Partitionen zusammen. Eine Partition ist hierbei wie ein Array bzw. eine Liste aufgebaut, in der jeder Datensatz durch einen eindeutigen Index – einen sogenannten Offset – repräsentiert wird. Neue Nachrichten werden in diesem Zusammenhang am Ende der jeweiligen Partition angefügt und erhalten einen neuen fortlaufenden Offset [NSP17].

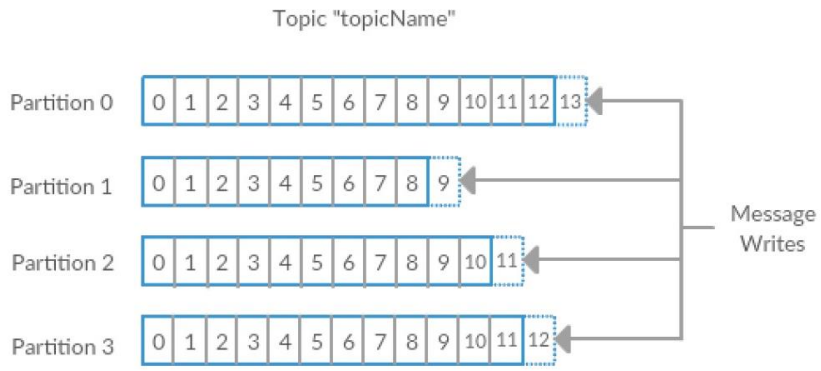


Abbildung 25: [NSP17] Aufbau eines Kafka-Topics

Im konkreten Beispiel in Abbildung 25 wird unter anderem der Partition 0 an den aktuellsten Datensatz mit dem Offset 12 ein neuer Datensatz mit dem Offset 13 angefügt. *Apache Kafka* gewährleistet in diesem Zusammenhang die Reihenfolge nur innerhalb einer Partition und nicht partitionsübergreifend [NSP17].

Der Abruf von Nachrichten durch die Consumer erfolgt zunächst dadurch, dass die Broker die verfügbaren Consumer auf die einzelnen Partitionen verteilen. In diesem Rahmen werden einer Partition niemals zwei Consumer zugewiesen [NSP17]. Sofern beispielsweise 5 Consumer und 4 Partitionen vorhanden sind, werden die 4 Partitionen auf 4 Consumer abgebildet und dem 5. Consumer keine Partition zugewiesen [NSP17].

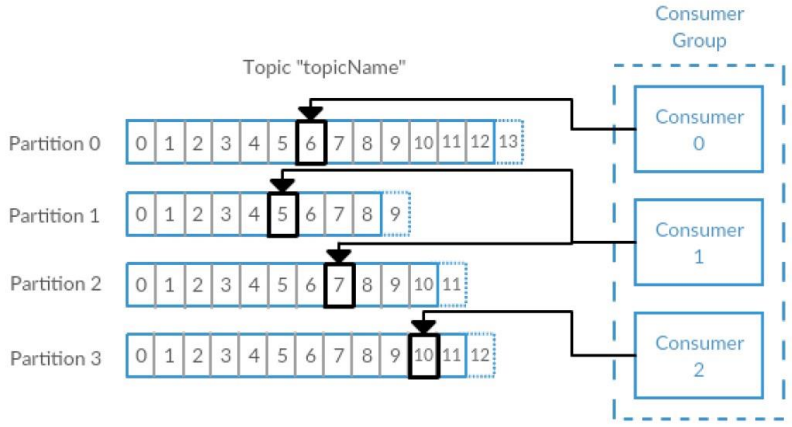


Abbildung 26: [NSP17] Abruf von Nachrichten aus einem Kafka-Topic

Entgegengesetzt ist es jedoch, wie in Abbildung 26 dargestellt, möglich, dass einem Consumer mehrere oder sogar alle Partitionen zugewiesen werden. Diese Betrachtungsweise gilt nur innerhalb einer sogenannten *ConsumerGroup*. Sofern die Nachrichten zusätzlich von einer weiteren Instanz oder System verarbeitet werden sollen, muss sich dieses grundsätzlich in einer anderen *ConsumerGroup* befinden [NSP17].

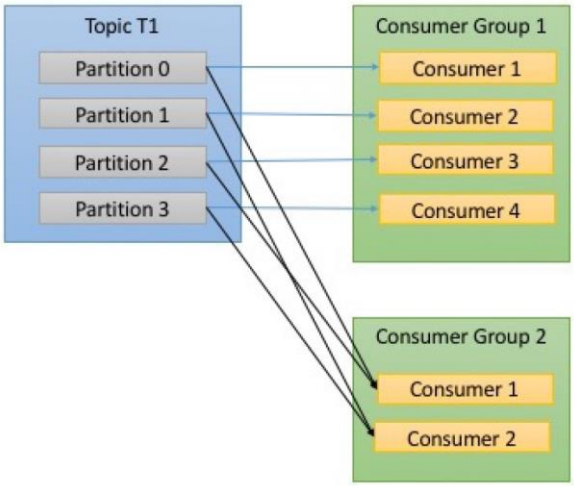


Abbildung 27: [NSP17] Übersicht eines Konsumierungsszenarios

Abbildung 27 zeigt ein mögliches Konsumierungsszenario, in dem die Daten doppelt von zwei verschiedenen *ConsumerGroups* verarbeitet werden. Im Fall der *ConsumerGroup1* werden die enthaltenen 4 Consumer jeweils *eins-zu-eins* auf die vorhandenen Partitionen abgebildet. Im Fall der *ConsumerGroup2* erhält jeder Consumer jeweils *2 der 4* vorhandenen Partitionen.

Im Gegensatz zu JMS verwaltet nicht der Broker, sondern die Consumer den aktuellen Verarbeitungsstand. Darüber hinaus wird bei *Apache Kafka* der Versand nicht eigenständig vom Broker initiiert, sondern erfolgt durch eine aktive Anfrage des Consumers (*polling*) [NSP17].

Ausgehend vom niedrigsten Offset (aktueller Verarbeitungsstand) konsumiert der Consumer im Anschluss an ein *Polling* die erhaltenen Nachrichten und passt in diesem Zuge seinen (Verarbeitungs-)Offset an. Da der Broker jedoch befugt ist, im Rahmen eines sogenannten Rebalance die Partitionen unter den Consumern neu zu verteilen, müssen die Consumer ihren aktuellen Verarbeitungsstand in regelmäßigen Abständen an den Broker übertragen. Sofern nun ein neuer Consumer die Partition eines Vorgängers erhält, weiß dieser über den entsprechenden Verarbeitungsstand Bescheid.

Da im Kafka-Kontext jedoch die Nachrichten für einen definierbaren Zeitpunkt auf der Festplatte gespeichert werden, ist ein Consumer dazu in der Lage, zu älteren oder neueren Offsets zu springen und damit die Nachrichten erneut zu konsumieren oder diese entsprechend zu überspringen.

3.2.3 Clustering-Konzept

Apache Kafka wurde, wie bereits erwähnt, unter Gesichtspunkten des Clusterings konzipiert [NSP17]. Hierzu arbeiten mehrere Broker in einem Verbund zusammen, um die gespeicherten Nachrichten den Consumern bereitzustellen. In diesem Zusammenhang werden die Topics redundant auf einer, im Vorfeld definierbaren, Anzahl an Brokern gespeichert. Diese Anzahl wird

als *ReplicationFactor* bezeichnet [NSP17]. Durch die vorherige Untergliederung eines Topics in einzelne Partitionen kann dieses Topic im Anschluss durch mehrere Broker für den Empfang und die Auslieferung von Nachrichten eingesetzt werden. In diesem Zusammenhang wird für jede Partition einer der Broker zum Leader ernannt, der im Anschluss als Einziger befugt ist, Daten für diese Partition entgegenzunehmen und auszuliefern. Durch dieses Vorgehen wird die eintreffende Last gleichmäßig auf alle beteiligen Broker verteilt [NSP17].

Da die Leader stetig neue Nachrichten von den Producern erhalten, besitzen die anderen Broker zeitnah einen veralteten Datenbestand. Aus diesem Grund synchronisieren sich diese sogenannten Replicas kontinuierlich mit dem Broker indem sie die neuen Datenbestände abrufen und in ihrem eigenen Datenbestand ablegen. Die jeweiligen Replicas werden als insync-Replicas bezeichnet, sofern sie den aktuellen Stand der jeweiligen Partition besitzen. Dieses Vorgehen ermöglicht es, dass eines der insync-Replicas für den Leader einspringen kann, sofern dieser ausfallen sollte. Die Ernennung zum Leader erfolgt über eine entsprechende Wahl (*election*), die im Abschnitt 3.2.6 beschrieben wird.

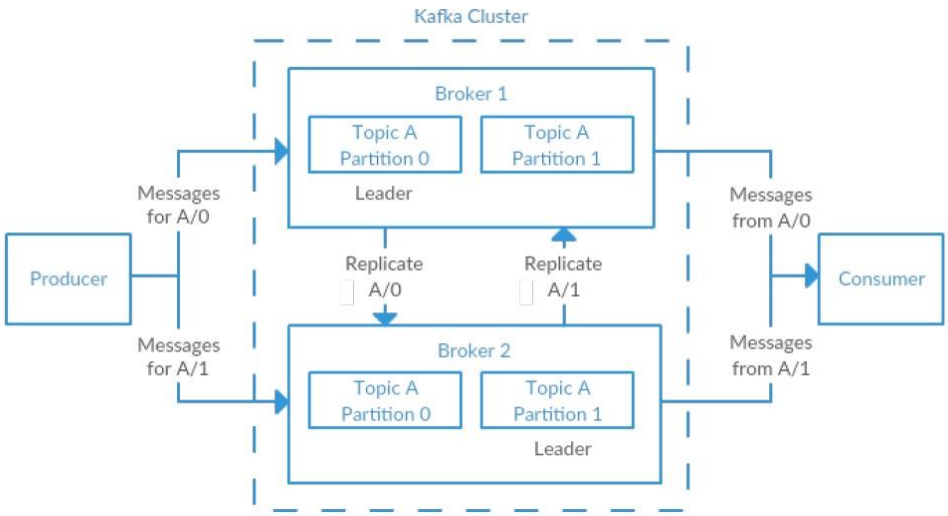


Abbildung 28: [NSP17] Vorgehen innerhalb eines Kafka-Clusters

Abbildung 28 zeigt dieses Vorgehen anhand eines Clusters, welches sich aus zwei Brokern zusammensetzt. Das *Topic A* befindet sich in diesem Zusammenhang vollständig auf beiden Brokern. Im Rahmen einer *Leader Election* wird *Broker 1* als Leader für die Partition 0 und der *Broker 2* als Leader für die *Partition 1* ermittelt. Der Producer richtet seine Nachrichten jeweils alternierend an die beiden Partitionen und kontaktiert damit jeweils abwechselnd die beiden Broker [NSP17]. Die Broker wiederum replizieren gegenseitig die Daten der gegenüberliegenden Leader-Partition. Zuletzt liest im dargestellten Beispiel ein Consumer die Daten aus den Leader-Partitionen beider Broker [NSP17].

3.2.4 Grundlegende Broker-Konfigurationsparameter

Apache Kafka bietet zahlreiche Konfigurationsparameter, mit denen das Verhalten des Brokers beeinflusst werden kann. Dieser Abschnitt bietet einen Überblick über die relevantesten dieser Parameter, die in der *server.properties*-Datei definiert werden.

```
1  broker.id=0
2  log.dirs=C:\dev\kfk\kafka_logs
3  num.partitions=5
4  log.retention.hours=168
5  zookeeper.connect=localhost:2181
6  default.replication.factor=3
7  unclean.leader.election.enable=false
8  min.insync.replicas=2
```

Abbildung 29: Broker-Konfiguration Apache Kafka

Abbildung 29 zeigt einen Ausschnitt aus dieser Datei. Durch den Parameter *broker.id* wird in Zeile 1 dem Broker eine einheitliche ID zugewiesen, über die *ZooKeeper* und die anderen Broker diesen eindeutig identifizieren können. Der Parameter *zookeeper.connect* in Zeile 5 teilt dem Broker mit, mit welcher ZooKeeper-Instanz er sich initial verbinden soll. Auf die anderen relevanten Parameter wird im weiteren Verlauf dieses Kapitel näher eingegangen.

Um dem Broker mitzuteilen, in welchen Verzeichnissen er die Nachrichten abzulegen hat, werden über den Parameter *log.dirs* die zugehörigen Pfade hinterlegt. Darüber hinaus kann im Hinblick auf die Nachrichtenspeicherung über den Parameter *log.retention.** angegeben werden, wie lange die jeweiligen Nachrichten auf dem Datenträger gespeichert werden sollen. Der Platzhalter * kann sowohl durch *bytes* als auch durch eine Zeitangabe der Form *ms*, *minutes* oder *hours* ersetzt werden. Weiterhin ist es möglich, sowohl eine Byte- als auch eine Zeitangabe zu definieren. In diesem Fall werden die Nachrichten gelöscht, sobald eine der beiden Bedingungen erfüllt ist. Da mehrere Nachrichten innerhalb einer log-Datei abgelegt werden, richtet sich der angegebene Zeitraum danach, wann das jeweilige Log-File zuletzt modifiziert wurde. Sollte die Angabe eines Datenvolumens erfolgen, so richtet sich die definierte Größe nach der Gesamtgröße des jeweiligen Topics [@kd].

Zuletzt kann über den Parameter *message.max.bytes* angegeben werden, welche maximale Nachrichtengröße vom Broker akzeptiert wird. Eine vollständige Übersicht aller Konfigurationsparameter ist [@kd] zu entnehmen.

3.2.5 Kafka-API

Apache Kafka bietet für die Anbindung von Clients fünf verschiedene APIs an. Diese lauten *Producer API*, *Consumer API*, *Streams API*, *Connect API* sowie die *Legacy API* [@kd]. Im Kontext dieser Arbeit sind jedoch nur die beiden erstgenannten relevant. Die Producer-API wird in diesem Zusammenhang im Rahmen des Abschnitts 3.2.5.1 genauer beschrieben. Die Betrachtung der Consumer-API ist Bestandteil des Abschnitts 3.2.5.2.

3.2.5.1 Kafka-Producer

Die Producer-API von *Apache Kafka* gibt, ähnlich wie jene von *ActiveMQ*, den Ablauf zum Erstellen eines Consumers bzw. zum Senden einer Nachricht vor. Die notwendigen Schritte sind dabei in der Basisversion jedoch deutlich geringer. Zu Beginn wird ein Objekt des Typs *Properties* erstellt, dem, wie in Abbildung 30 ersichtlich, minimal (Zeile 36) die verfügbaren Kafka-Broker sowie die notwendigen Klassen zum Serialisieren der Nachricht (Zeile 37 und 38) mitgeteilt werden. Diese Properties werden anschließend dem zuvor definierten *KafkaProducer*-Objekt zugewiesen. Die Anzahl der Properties kann durch zusätzliche Parameter aus [@kd] beliebig erweitert werden.

Anschließend kann durch den Aufruf der in Zeile 42 beginnenden *sendMessageToCluster()*-Methode eine Nachricht versendet werden. Hierfür wird ein *ProducerRecord* erstellt, welchem das beabsichtigte Topic sowie der Inhalt der Nachricht zugewiesen werden. Definierte Nachrichtentypen wie bei JMS existieren im Kafka-Umfeld nicht. Es können jedoch ebenfalls durch die Definition eigener Schemata anwendungsspezifische Objekte serialisiert und übertragen werden. In diesem Zusammenhang kann beispielsweise *Apache Avro* [@avro] als Serialisierungs-Framework eingesetzt werden. Weitere Informationen zu dieser Thematik sind [NSP17] zu entnehmen.

Abschließend wird die Nachricht über den Aufruf der *send()*-Methode versandt. In diesem Zusammenhang gibt es grundsätzlich drei verschiedene Versandarten. Da diese im Rahmen dieser Arbeit einen maßgeblichen Einfluss auf die Test-Konzeption haben, werden diese im Rahmen von Kapitel 4 anhand der notwendigen Client-Software beschrieben.

```
35    public Producer(String _topic, int _messagesPerSecond, long _startProcessingAt) {
36        kafkaProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"esbsvr03.ov.otto.de:9092");
37        kafkaProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSerializer.class.getName());
38        kafkaProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,StringSerializer.class.getName());
39        producer = new KafkaProducer<String, String>(kafkaProps);
40    }
41
42    public void sendMessageToCluster(String content){
43        ProducerRecord<String, String> record = new ProducerRecord<String, String>(topic, content);
44        producer.send(record, new ProducerCallback());
45    }
```

Abbildung 30: Erzeugung eines Kafka-Producers

Die darauffolgenden internen Vorgänge des Kafka-Producers im Anschluss an den Aufruf der *send()*-Methode werden in Abbildung 31 dargestellt. Dieser führt im ersten Schritt die Serialisierung der Nachricht durch. Anschließend wird durch den Partitioner und die zugehörige Partitionierungsstrategie die Nachricht einer der verfügbaren Partitionen zugewiesen. Darauffolgend versucht die API, die vorhandenen Nachrichten in mehreren *Batches* an den Broker zu übertragen. Sofern dies scheitert, wird je nach Einstellung automatisch versucht, die Nachrichten erneut zu versenden. Sollte kein Retry-Versuch mehr vorhanden sein, endet der Aufruf der *send()*-Methode mit einer Exception. Im Erfolgsfall liefert die Methode als Rückgabewert ein

RecordMetadata-Objekt, das Informationen über das Topic, die Partition sowie den Offset der Nachricht enthält. Weitere Informationen zu diesem Vorgehen können [NSP17] entnommen werden.

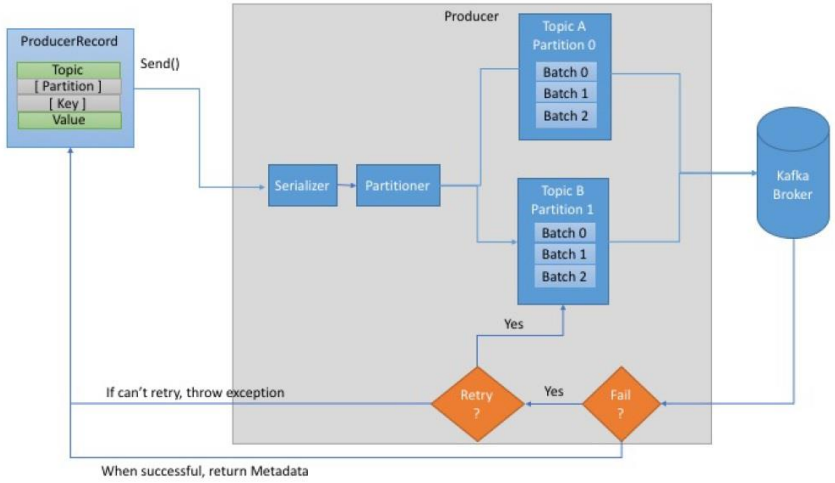


Abbildung 31: [NSP17] Aufbau Kafka-Producer

3.2.5.2 Kafka-Consumer

Die Erzeugung eines Kafka-Consumers erfolgt auf ähnliche Art und Weise wie die Erstellung eines Producers. Dazu werden ebenfalls über die Properties die verfügbaren Kafka-Broker sowie die Klassen für die Deserialisierung angegeben. Zusätzlich wird in Zeile 50 die *ConsumerGroup* definiert, innerhalb welcher der Consumer agieren soll. Darüber hinaus wird in Zeile 52 definiert, für welches Topic der Consumer Nachrichten verarbeiten möchte.

```
46  public Consumer(String _topic, int _messagesPerSecond, long _startProcessingAt) {
47      kafkaProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"esbsvr03.ov.otto.de:9092");
48      kafkaProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,StringDeserializer.class.getName());
49      kafkaProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
50      kafkaProps.put(ConsumerConfig.GROUP_ID_CONFIG,"toNoaConsumer");
51      consumer = new KafkaConsumer<String, String>(kafkaProps);
52      consumer.subscribe(Collections.singletonList(topic), this);
53  }
54
55  public void run(){
56          while(true){
57              ConsumerRecords<String, String> records = consumer.poll(waitMillisecondsIfNotEnoughDataAvailable);
58                  for(ConsumerRecord<String, String> record: records){
59                      System.out.println("Process Message " + record.key());
60                      consumer.commitAsync();
61                  }
62          }
63  }
```

Abbildung 32: Erzeugung eines Kafka-Consumers

Die in Zeile 55 beginnende Verarbeitung der Nachrichten zeigt ein komplett anderes Vorgehen im Vergleich zu *ActiveMQ*. Die Nachrichten werden nicht eigeninitiiert vom Broker an den Client übertragen. Das Konzept von *Kafka* sieht vor, dass der Consumer kontinuierlich beim Broker anfragt, ob neue Nachrichten vorliegen, der diese im Anschluss an den Client ausliefert.

Konkret wird in diesem Zusammenhang in Zeile 56 eine *while()*-Schleife gestartet, die dauerhaft den weiteren Abruf neuer Daten forciert. Im ersten Schritt eines Schleifendurchlaufs werden über die *poll()*-Funktion neue Nachrichten vom Broker angefordert. Der dargestellte Parameter gibt die Länge des Zeitraumes (ms) an, den der Broker warten soll, sofern die angeforderte Nachrichtenmenge aktuell nicht verfügbar ist. Diese Menge kann ebenfalls über die oben dargestellten Properties definiert werden.

Anschließend wird die zurückgelieferte Menge an Nachrichten im Rahmen einer *for()*-Schleife durchlaufen. Nach der Verarbeitung jeder Nachricht wird der neue Offset durch einen *commit* an den Broker übertragen und im Topic *__consumer_offsets* gespeichert. Grundsätzlich wird dabei zwischen automatischen *Commits* und manuellen *Commits* unterschieden [NSP17]. Automatische *Commits* erleichtern die Arbeit des Entwicklers, führen jedoch zu einer erhöhten Menge an doppelt konsumierten Nachrichten, da hier grundsätzlich in größeren Abständen der neue Verarbeitungsstand an den Broker übertragen wird. Beim Einsatz von manuellen Commits ist der Entwickler deutlich flexibler und kann zwischen synchronen und asynchronen *Commits* wählen sowie beide Ansätze kombinieren [NSP17]. Im Rahmen eines synchronen Commits wartet der Consumer mit der weiteren Verarbeitung, bis er eine positive Rückmeldung vom Broker erhalten hat. Bei asynchronen Commits hingegen fährt der Consumer im Anschluss an den Commit direkt mit der weiteren Verarbeitung fort, ohne die Bestätigung des Brokers abzuwarten.

Da die Bestätigung der Nachrichten nicht explizit pro Nachricht, sondern implizit über den aktuellen Verarbeitungs-Offset erfolgt, darf der Consumer im Fall eines gescheiterten asynchronen Commits diesen nicht ohne weiteres erneut senden. Dies liegt daran, dass es bereits einen neueren asynchronen Commit geben könnte, der einen noch neueren Verarbeitungsstand kommuniziert hat. Der erneute Versand des veralteten Commits würde in diesem Fall dazu führen, dass der neuere Verarbeitungsstand zurückgesetzt wird und die verarbeiteten Nachrichten wiederholt verarbeitet werden müssten. Abbildung 33 veranschaulicht dieses Vorgehen. Sofern der gescheiterte 1. Commit (*rot*) erneut im Anschluss an den 2. Commit (*grün*) gesendet wird, so wird damit der Verarbeitungsstand auf den Offset 10 zurückgesetzt.

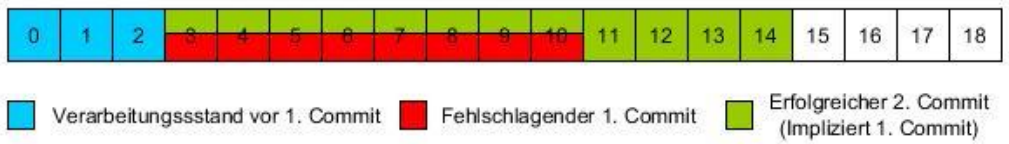


Abbildung 33: Commit-Ablauf

Die Maßnahmen zur Übertragung des aktuellen Datenbestandes sind notwendig, da, wie bereits erwähnt, durch einen sogenannten *Rebalance* den Consumern die aktuelle Partition entzogen werden kann [NSP17]. Sofern die Partition anschließend einem anderen Consumer zugewiesen

wird, muss dieser vom Broker über den aktuellen Verarbeitungsstand in Kenntnis gesetzt werden.

3.2.6 Interne Broker-Abläufe

In diesem Abschnitt werden die grundlegenden internen Vorgänge von *Apache Kafka* beschrieben. In diesem Zusammenhang befasst sich Abschnitt 3.2.6.1 mit der Clusterzugehörigkeit eines Brokers. Im Abschnitt 3.2.6.2 wird Bezug auf die Replizierung der gespeicherten Daten genommen. Zuletzt wird im Abschnitt 3.2.6.3 die Rolle des Controllers beschrieben.

3.2.6.1 Cluster-Zugehörigkeit

Bevor ein Broker aktiv innerhalb eines Clusters agieren kann, muss diesem über die Konfiguration eine eindeutige ID zugewiesen werden. *Apache Kafka* nutzt hierbei *ZooKeeper*, um eine Liste namens */brokers/ids* zu verwalten, welche die aktuellen Mitglieder des Clusters beinhaltet [NSP17]. In diesem Zusammenhang registriert sich ein Broker während des Startvorganges mit seiner eindeutigen ID in *ZooKeeper* indem er einen sogenannten flüchtigen Knoten (*emphemeral node*) erstellt [NSP17]. Sofern der Knoten mit der zugehörigen ID bereits vorhanden ist, wird der Registrierungsversuch durch *ZooKeeper* entsprechend abgelehnt. Darüber hinaus abonnieren die einzelnen Komponenten im Cluster während des Startvorganges die in *ZooKeeper* verwaltete Liste, um darüber informiert zu werden, wenn neue Broker hinzukommen oder das Cluster verlassen.

Sofern ein Broker mit der identischen ID gestartet wird, führt dies zu einem Fehler, da bereits ein Broker mit derselben ID in ZooKeeper vorhanden ist. Darüber hinaus wird der flüchtige Knoten entfernt, sobald der Broker die Verbindung zu *ZooKeeper* verliert und die anderen Broker hierüber durch die Änderung in */brokers/ids* informiert.

In diesem Zusammenhang ist von Interesse, dass trotz der Entfernung des Knotens aus *ZooKeeper* die Broker-ID weiterhin existiert. Das drückt sich darin aus, dass der Broker vor seinem Ausscheiden aus dem Cluster für die Replizierung mehrerer Topics zuständig war und dabei anhand seiner ID referenziert wurde. Da im Rahmen des Ausfalls die insync-Replicas für die verantwortlichen Partitionen des Brokers einspringen, ist das Cluster nicht mehr abhängig von den Daten auf dem ausgefallenen Broker. Aus diesem Grund kann anstelle des ursprünglichen Brokers ein beliebiger Broker mit der gleichen ID gestartet werden, der sich beim Start mit den Leadern der einzelnen Partitionen synchronisiert [NSP17].

3.2.6.2 Replication

Die Replizierung der Daten ist eines der Kernkonzepte von *Apache Kafka* [NSP17]. Bei den verschiedenen Replicas einer Partition wird grundsätzlich zwischen *Leader-Replicas* und *Follower Replicas* unterschieden [NSP17]. Das Leader-Replica ist in diesem Rahmen für die

Entgegennahme und die Auslieferung von Nachrichten der jeweiligen Partition verantwortlich. Die Follower Replicas bearbeiten keine Clientanfragen und besitzen lediglich die Aufgabe, auf dem aktuellen Stand des Leader-Replicas zu bleiben und sich damit kontinuierlich mit diesem zu synchronisieren. Durch dieses Vorgehen ist der Follower bei einem Ausfall des Leaders in der Lage, für diesen einzuspringen [NSP17]. Die Synchronisierung erfolgt durch einen sogenannten *FetchRequest*, welcher der Anfrage eines herkömmlichen Consumers gleicht [NSP17].

Da ein Follower-Replica immer in exakter Reihenfolge das nächste Paket anfordert, das ihm noch fehlt, weiß der Leader durch die Anfrage exakt, wie weit das Replica hinter dem aktuellen Stand liegt und ab wann er dieses als *out-of-sync* Kennzeichen muss. Ein Replica gilt in den Default-Einstellungen von Kafka als *out-of-sync*, wenn es seit 10 Sekunden keine Nachrichten angefordert hat oder zwar Nachrichten anfordert, aber nicht innerhalb der nächsten 10 Sekunden in der Lage ist, den aktuellen Stand zu erreichen [NSP17].

Neben diesem Vorgehen wird für jede Partition ein sogenannter *preferred leader* definiert [NSP17]. Hierbei wird jeweils jener Broker als *preferred leader* gekennzeichnet, der bei der initialen Erstellung des Topics Leader der jeweiligen Partition war. Dieses Vorgehen strebt eine gleiche Verteilung der Leader auf die verfügbaren Broker an [NSP17].

3.2.6.3 Controller

Die Wahl eines neuen Leaders für eine Partition erfolgt durch einen dazu ernannten Broker innerhalb des Clusters. Dieser Broker wird als Controller bezeichnet [NSP17].

Der erste Broker, der einem Cluster beitritt, wird automatisch zum Controller, indem er innerhalb von */controller* einen flüchtigen Knoten in *ZooKeeper* erstellt. Die anderen Broker versuchen ebenfalls im Rahmen des Startvorgangs diesen Knoten zu erstellen, erhalten jedoch in diesem Fall eine *node already exists* Exception und erkennen hierdurch, dass bereits ein Controller vorhanden ist. Infolgedessen erstellen diese einen *ZooKeeper Watch* für den vorhandenen Controller-Knoten um bei Veränderungen informiert zu werden. Falls der aktuelle Controller die Verbindung zu *ZooKeeper* verliert, wird der Knoten automatisch aus *ZooKeeper* entfernt. Durch den *ZooKeeper Watch* werden die anderen Broker darüber informiert, dass der ursprüngliche Controller nicht mehr vorhanden ist und versuchen ihrerseits noch einmal einen flüchtigen Knoten in */controller* zu erzeugen. Der erste Broker, dem dies gelingt, ist anschließend der neue Controller des Clusters [NSP17]. Alle anderen Broker erstellen erneut einen *ZooKeeper Watch* auf diesen neuen Knoten. Dieses Vorgehen stellt sicher, dass es zu einem bestimmten Zeitpunkt immer nur einen Controller innerhalb des Clusters gibt.

Jeder neue Controller erhält eine neue und höhere Epochen-Nummer als der alte Controller. Alle anderen Broker kennen diese neue Epochen-Nummer und ignorieren Anweisungen von

einem Controller mit einem kleineren Wert. Dies könnte der Fall sein, wenn sich der alte Controller nach einem kurzzeitigen Verbindungsabbruch immer noch für den Controller hält.

Sofern der Controller erkennt, dass ein Broker das Cluster verlassen hat, sucht er für alle Leader-Replicas neue Leader. Hierfür durchläuft er alle betroffenen Partitionen und legt als neuen Leader den ersten in der Replica-Liste fest. Sofern verfügbar ist dies der *preferred Leader*.

3.2.7 Verlässliche Nachrichten-Übertragung

In diesem Abschnitt wird Bezug auf die zuverlässige Nachrichten-Übertragung genommen. In diesem Zusammenhang ist zunächst zu klären, welche generellen Garantien *Apache Kafka* bei der Nachrichten-Übertragung bietet, und wo die zugehörigen Grenzen dieses Systems liegen. Dieser Themenschwerpunkt ist Inhalt des Abschnitts 3.2.7.1. Anschließend werden im Abschnitt 3.2.7.2 die drei wesentlichen Broker-Parameter erläutert, mit denen sich die maximal mögliche Zustellgarantie erreichen lässt. Darauffolgend wird in den Abschnitten 3.2.7.3 und 3.2.7.4 auf die Eigenschaften der Producer und Consumer in solchen System eingegangen. Zuletzt nimmt Abschnitt 3.2.7.5 Bezug auf eine angestrebte *Exactly-Once* Zustellgarantie.

3.2.7.1 Kafka-Garantien

Apache Kafka bietet grundsätzlich vier verschiedene Garantien. Hierzu zählt im ersten Schritt, dass *Kafka* immer die Reihenfolge der Nachrichten innerhalb einer Partition sicherstellt [NSP17]. Darüber hinaus garantiert *Kafka*, dass Nachrichten erst als *committed* betrachtet werden, wenn diese auf allen *insync*-Replicas vorhanden sind. Producer können in diesem Zusammenhang wählen, ob sie eine direkte Bestätigung des Brokers erwarten oder diese erst gesendet werden soll, wenn die Nachricht als *committed* betrachtet wird [NSP17].

Die dritte Garantie besagt, dass Nachrichten, die als *committed* betrachtet werden, nicht verloren gehen, solange mindestens ein Replica verfügbar ist. Zuletzt versichert *Kafka*, dass Consumer lediglich Nachrichten lesen können, die sich im Zustand *committed* befinden [NSP17].

3.2.7.2 Broker-Konfiguration

Grundsätzlich sind drei verschiedene Broker-Konfigurationsparameter vorhanden, die das Verhalten des Brokers in Bezug auf die verlässliche Nachrichtenübertragung beeinflussen.

Der erste Parameter, der Auswirkungen auf die verlässliche Nachrichtenübertragung hat, ist der *ReplicationFactor*. Dieser gibt an, auf wie viele Broker das Topic bzw. die enthaltenen Partitionen repliziert werden sollen. Daraus resultierend ergibt sich, dass bei einem *ReplicationsFactor* von N insgesamt der Ausfall von N-1 Brokern kompensiert werden kann. Die Wahl eines geeigneten *ReplicationFactor* richtet sich nach den eigenen Anforderungen hinsichtlich der Ausfallsicherheit. Sofern kein Ausfall zu tolerieren ist, empfiehlt sich ein Wert von mindestens 3 Replicas [NSP17].

Der zweite relevante Parameter *Unclean Leader Election* befasst sich mit der Problemstellung, wie damit umgegangen werden soll, wenn der Leader ausfällt und zu diesem Zeitpunkt keine *insync*-Replicas vorhanden sind. Grundsätzlich muss in dieser Konstellation der ausgefallene Leader wieder betriebsbereit gemacht werden, da es sonst zu einem Datenverlust kommen würde [NSP17]. Um jedoch das Topic wieder schnell produktiv einsetzen zu können, kann durch die Aktivierung der *Unclean Leader Election* dem Controller gestattet werden, auch ein Replica das *out-of-sync* ist, zum neuen Leader zu ernennen. Da dieses Vorgehen jedoch einen Datenverlust zur Folge hat, können durch die Aktivierung dieser Einstellung die Zustellgarantien *At-Least-Once* und *Exactly-Once* nicht mehr erreicht werden.

Basierend auf der Problematik, dass trotz eines *ReplicationFactors* von beispielsweise 5 Replicas im schlechtesten Fall nur ein insync-Replica vorhanden ist, befasst sich der Parameter *Minimum Insync Replicas* damit, wie viele synchronisierte Replicas minimal vorhanden sein müssen, damit der Leader neue Nachrichten entgegen nimmt. Dieser Parameter stellt folglich sicher, dass Nachrichten nur entgegen genommen werden, sofern das angestrebte Replizierungsniveau aktuell gegeben ist [NSP17]. Im Hinblick auf eine hohe Zuverlässigkeit sollte hierbei mindestens ein Wert von 2 verwendet werden.

3.2.7.3 Producer in verlässlichem System

Trotz einer bestmöglichen Konfiguration des Brokers hinsichtlich der Verlässlichkeit bei der Datenübertragung kann es trotzdem innerhalb des Gesamtsystems zu einem Datenverlust kommen, sofern die zugehörigen Producer nicht richtig konfiguriert wurden [NSP17].

Bei der Producer-Konfiguration ist in diesem Zusammenhang darauf zu achten, dass dieser die passende Bestätigungsart vom Broker fordert. Diese Bestätigungsart kann über den Parameter *acks* festgelegt werden. Ein Wert von 0 besagt, dass die Nachricht als erfolgreich übertragen gilt, wenn der Client in der Lage war, die Nachricht abzuschicken. Der Wert 1 gibt an, dass der Broker direkt nach Erhalt der Nachricht den Empfang quittiert, ohne dass diese bereits auf den insync-Replicas vorhanden ist. Zuletzt gibt der Wert *all* an, dass der Broker den Erhalt erst bestätigt, wenn die Nachricht auf allen *insync*-Replicas vorliegt [NSP17].

Weiterhin muss auf Seiten des Producers für ein entsprechendes Fehler-Handling gesorgt werden, in welchem Rahmen versucht wird, Nachrichten erneut zu versenden bzw. das Scheitern entsprechend protokolliert wird [NSP17].

3.2.7.4 Consumer in verlässlichen Systemen

Bei der Verarbeitung der Nachrichten durch die Consumer ist darauf zu achten, durch den Parameter *auto.offset.reset* ein passendes Verhalten vorzugeben, wie damit verfahren werden soll, wenn der zuletzt gespeicherte Offset nicht mehr verfügbar ist. Hierbei kann entweder durch den Wert *earliest* beim frühestmöglichen Offset begonnen oder durch latest zum Ende der Partition

gesprungen werden. Ersteres sorgt ggf. dafür, dass Nachrichten doppelt konsumiert werden. Letzteres führt dazu, dass Nachrichten nicht verarbeitet werden. Da dies einem Datenverlust gleicht, empfiehlt es sich, sofern zumindest die Zustellgarantie *At-Least-Once* benötigt wird, die Einstellung *earliest* zu verwenden.

Darüber hinaus sollte der aktuelle Verarbeitungstand entsprechend sorgfältig gepflegt und der zugehörige Offset nach der Verarbeitung jeder Nachricht explizit *committed* werden. Da ein Broker im Rahmen eines Rebalance die Partitionen neu zuweisen kann, sollten durch einen Rebalance-Listener [NSP17] entsprechende Maßnahmen ergriffen werden, um vorher den aktuellen Stand zu sichern.

Darüber hinaus führt der Broker einen Rebalance durch, sofern ein Consumer aus der *ConsumerGroup* ausscheidet. In diesem Zusammenhang senden die Consumer durch den Aufruf der *poll()*-Funktion einen *Heartbeat* an den Broker. Im Rahmen einer länger andauernden Verarbeitung der Nachricht ist hierdurch dafür Sorge zu tragen, dass weiterhin kontinuierlich ein *Heartbeat* gesendet wird, damit der Broker nicht fälschlicherweise einen Rebalance durchführt.

3.2.7.5 Exactly-Once Zustellgarantie

Abschließend betrachtet ermöglicht *Apache Kafka* bei passender Konfiguration eine *At-Least-Once* Zustellgarantie, in der jede Nachricht mindestens einmal an die beabsichtigten Consumer übertragen wird [NSP17].

Jedoch ist *Kafka* mit den in diesem Abschnitt dargestellten Basiskonzepten nicht in der Lage, zu gewährleisten, dass jede Nachricht exakt einmal die beabsichtigten Consumer erreicht [NSP17]. Das ist insbesondere in Systemen, wie beispielsweise Bestellsystemen, kritisch, in denen eine Nachricht wirklich nur einmal ihr Ziel erreichen darf, da andernfalls eine Bestellung doppelt getätigt würde.

Um diese *Exactly-Once* Eigenschaft zu gewährleisten, muss das Gesamtsystem dahingehend optimiert werden, dass die Consumer beispielsweise durch die Verwendung von Sequenznummern Duplikate erkennen und diese verwerfen.

4 Aufbau der Testumgebung

In diesem Kapitel wird der Aufbau der Testumgebung einschließlich der clientseitigen Software zum Senden und Empfangen der Nachrichten beschrieben. Hierzu werden im Abschnitt 4.1 die verschiedenen Testarten erläutert. Anschließend erfolgt im Abschnitt 4.2 eine Beschreibung der verwendeten Einstellungen und Parameter anhand von Quellcodeausschnitten der Clientsoftware. Zuletzt wird im Abschnitt 4.3 die verwendete Test-Infrastruktur einschließlich der einzelnen Systeme erläutert.

Die Ziele der im Folgenden beschriebenen Testszenarien liegen darin, das Brokerverhalten bei steigender Last zu ermitteln und zu visualisieren. In diesem Zusammenhang ist außerdem von Interesse, ob und ab welcher Belastungshöhe die Systeme nicht mehr in der Lage sind, weitere Daten aufzunehmen.

4.1 Testszenarien

Neben einer grundsätzlichen Trennung zwischen Producer- und Consumer-Tests wurden in beiden Messaging-Systemen jeweils 3 verschiedene Testszenarien durchgeführt. Diese werden im weiteren Verlauf als *Nachrichten/Sekunden-basiert*, *Consumer/Producer-basiert* und *Datenmengen-basiert* bezeichnet und in den Abschnitten 5.1, 5.2 und 5.3 detaillierter erläutert. Die Tabelle 2 zeigt eine Übersicht über die durchgeführten Test-Konstellationen.

Die Trennung vom Nachrichten-Empfang (Producer-Tests) und dem Weiterversand (Consumer-Tests) von Nachrichten wurde vollzogen, um das Verhalten des Brokers in beiden Situationen gezielter betrachten zu können. Bei einem *Ende-zu-Ende* Test ist nicht mehr nachvollziehbar, welchen Ausschlag die Consumer oder Producer am Ressourcen-Verbrauch haben. Aus den einzelnen Betrachtungsweisen kann später abgeleitet werden, welche Consumer- und Producer-Anzahl sinnvollerweise in einem *Ende-zu-Ende* Test gewählt werden sollte. Ein solcher Test wird im Kapitel 8 anhand des konkreten OTTO-Anwendungsfalles beschrieben.

Apache ActiveMQ		**Apache Kafka**	
Producer	**Consumer**	**Producer**	**Consumer**
T.1a N/S-basiert	**T.4a** N/S-basiert	**T.1b** N/S-basiert	**T.4b** N/S-basiert
T.2a Producer-basiert	**T.5a** Consumer-basiert	**T.2b** Producer-basiert	**T.5b** Consumer-basiert
T.3a Daten-basiert		**T.3b** Daten-basiert	

Tabelle 2: Test-Übersicht

Das allgemeine Vorgehen bei der Testdurchführung sieht einen Belastungszeitraum von 30 Sekunden vor, in welchem sich der jeweilige Broker mit dem Empfang bzw. der Auslieferung einer bestimmten Nachrichten- bzw. Datenmenge konfrontiert sieht. In diesem Zeitraum wird auf dem System, auf dem der Broker läuft, parallel der CMD-Befehl *vmstat* ausgeführt.

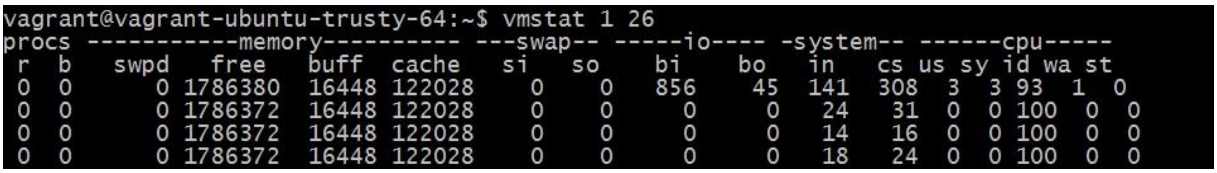


Abbildung 34: Ausgabe vmstat

Der Befehl *vmstat* gibt – wie in Abbildung 34 dargestellt – hierbei kontinuierlich einmal pro Sekunde auf Betriebssystemebene unter anderem Werte für den Arbeitsspeicher *(free, buff, cache)*, die IO-Operationen *(bi, bo)* sowie für die Auslastung des Prozessors *(us, sy, id, wa)* aus. Die wesentlichen Werte die innerhalb der hier beschriebenen Tests Berücksichtigung gefunden haben werden in Tabelle 3 aufgeführt und beschrieben.

Wert	Bedeutung
bi	Anzahl Datenblöcke, die pro Sekunde von der Festplatte gelesen werden.
bo	Anzahl Datenblöcke, die pro Sekunde auf die Festplatte geschrieben werden.
us	Prozentualer Anteil, in dem die CPU von Benutzerprozessen verwendet wird.
sy	Prozentualer Anteil, in dem die CPU von Betriebssystemprozessen verwendet wird.
wa	Prozentualer Anteil, in dem die CPU aufgrund von Speicherzugriffen warten muss.
id	Prozentualer Anteil, in dem die CPU nicht in Anspruch genommen wird.

Tabelle 3: Beschreibung der vmstat-Ausgabewerte

Im Anschluss an einen Testdurchlauf wurden die Ausgabewerte des Testzeitraums in eine Excel-Tabelle übertragen, die Mittelwerte gebildet und in ein Diagramm überführt.

Hierbei bilden die Durchschnittswerte von *us*, *sy*, und *wa* in Summe die prozentuale Auslastung der CPU. Die Werte *bo* und *bi* geben an, wie viele Schreib- und Leseoperationen der Broker pro Sekunde durchführt. Die Bedeutungen der einzelnen Werte sind der Tabelle 3 zu entnehmen.

4.1.1 Nachrichten/Sekunden-basiertes Testszenario

Beim Nachrichten/Sekunden-basierten Vorgehen wird der Broker über einen geplanten Zeitraum von 30 Sekunden mit einer konstanten Nachrichtenmenge pro Sekunde konfrontiert. Hierbei wird iterativ die Nachrichtenmenge systematisch um jeweils 500 Nachrichten pro Sekunde gesteigert.

Die Nachrichtenmenge pro Sekunde wird durch eine Verzögerung innerhalb der Producer bzw. Consumer erreicht. Ausgehend von mehreren Nachrichten pro Sekunde wird im Anschluss an den Versand einer Nachricht für einen Zeitraum von *1000 ms,* geteilt durch die Menge an beabsichtigten Nachrichten, gewartet. Dies entspricht beispielsweise bei fünf Nachrichten pro Sekunde einer Verzögerung von *200 ms*.

Ausgehend von der geplanten (optimalen) Testdauer von 30 Sekunden ergibt sich ein zusätzlicher Zeitbedarf, der aus den Übertragungszeiten der Nachrichten sowie aus weiteren Vorgängen innerhalb der Clients und des Brokers resultiert. Sofern sich dieser zusätzliche Zeitbedarf unverhältnismäßig bei zunehmender Belastung erhöht, ist dies ein Indiz dafür, dass der Broker an eine Belastungsgrenze stößt und in irgendeiner Form nicht mehr ausreichend Ressourcen zur Verfügung stehen.

Neben dem Zeitbedarf werden zusätzlich die IO-Operationen sowie die CPU-Auslastung im Verhältnis zur Menge an Nachrichten pro Sekunde untersucht. Diese Faktoren geben bei steigender Belastung ebenfalls Aufschluss über die noch zur Verfügung stehenden Kapazitäten und Ressourcen.

Im Rahmen einer parallelen Ausführung wurden die Nachrichten auf mehrere Producer aufgeteilt, die jeweils in eigenen Threads und zum Teil auch auf getrennten Maschinen laufen.

4.1.2 Consumer/Producer-Basiertes Testszenario

Beim Consumer/Producer-basierten Vorgehen wird, beginnend mit einem Producer bzw. Consumer, systematisch die Anzahl an parallelen Consumern/Producern erhöht, um zu ermitteln, welche Nachrichtenkapazität der Broker innerhalb von 30 Sekunden maximal aufnehmen bzw. ausliefern kann.

Im Gegensatz zum Nachrichten/Sekunde-basierten Vorgehen werden an dieser Stelle die Nachrichten nicht verzögert. Die Producer/Consumer versuchen im jeweiligen Zeitraum die größtmögliche Nachrichtenmenge an den Broker zu übertragen. Die Testlaufzeit liegt damit konstant bei 30 Sekunden, unabhängig davon, wie viele Nachrichten letztendlich übertragen werden.

Innerhalb dieses Testszenarios werden neben der Nachrichtenmenge, im Verhältnis zur steigenden Producer/Consumer-Anzahl, ebenfalls die Werte der IO-Operationen und der CPU-Auslastung untersucht.

Das Resultat dieses Tests liefert einen Überblick über die maximale Nachrichtenmenge, die der Broker bewältigen kann. Des Weiteren gibt das Resultat Aufschluss darüber, mit welcher Menge an Consumern/Producern dieses Maximum erzielt werden kann und wie die Ressourcenauslastung bei den jeweiligen Nachrichtenmengen ist.

4.1.3 Datenmengenbasiertes Testszenario

Das Datenmengen-basierte Vorgehen ist weitestgehend analog zum Nachrichten/Sekunden-basierten Vorgehen aus Abschnitt 4.1.1. Allerdings wird bei dieser Herangehensweise die Nachrichtengröße von *14 KB* auf *1,7 MB* erhöht, um zusammen mit der passenden Anzahl an Nachrichten ein bestimmtes Datenaufkommen zu erreichen.

Ziel dieses Vorgehens ist zum einen die Steigerung der IO-Operationen um Thesen aus anderen Tests zu belegen, dass die jeweiligen Systeme mit zusätzlicher Hardware bis zu einem gewissen Grad weiterer Belastung gerecht werden würden. Zum anderen abstrahiert dieses Testverfahren von der letztendlichen Nachrichtengröße. In den hier beschriebenen Tests wird ansonsten von einer konstanten Nachrichtengröße von *14 KB* ausgegangen. Da sich das Broker-Verhalten unter Umständen bei anderen Nachrichtengrößen verändert, bietet dieses Testszenario einen anderen (allgemeineren) Blickwinkel auf das Datenverarbeitungsverhalten des Brokers.

4.2 Testeinstellungen und Rahmenparameter

In diesem Abschnitt werden die Test-Parameter, Übertragungsmodi und allgemeine Einstellungen thematisiert, die innerhalb der Testdurchführung verwendet wurden und in direktem Zusammenhang zu den relevanten nicht-funktionalen Anforderungen stehen.

Dazu werden exemplarisch nur Quellcodebeispiele gezeigt, die im Hinblick auf diese Anforderungen relevant sind oder grundsätzliche Auswirkungen auf die Performance und den Durchsatz haben. Die Grundlagen zum Senden und Empfangen von Nachrichten können Kapitel 3 entnommen werden.

4.2.1 Producer-Tests ActiveMQ

Dieser Abschnitt befasst sich mit den Producer-Besonderheiten im JMS-Kontext. Hierzu zeigt Abbildung 35 die zugehörigen Quellcode-Ausschnitte. Die in Zeile 70 beginnende *run()*-Methode ist Bestandteil der Klasse *Thread*, von der die hier gezeigte Klasse *Producer* erbt. Diese Methode startet den jeweiligen Thread und damit die parallele Ausführung. In Zeile 71 wird für einen vorher definierten Zeitraum gewartet, damit alle Threads in der Lage sind, gleichzeitig mit dem Nachrichtenversand zu beginnen. Dieses Vorgehen soll dafür sorgen, dass, aufgrund des Zeitbedarfs zur Verbindungserstellung, nicht die ersten Threads schon mit der Ausführung beginnen und damit die vollen Broker-Ressourcen ausschließlich für sich beanspruchen können. Das könnte im Extremfall dazu führen, dass der erste Thread seine Arbeit bereits beendet

hat, wenn der letzte Thread mit der Ausführung beginnt. Dieses Verzögerungs-Vorgehen wird analog in allen anderen Test-Anwendungen weitergeführt.

```
55    private void sendMessage(String text) throws JMSException{
56        messageCounter++;
57        TextMessage message = queueSession.createTextMessage(text);
58        queueSender.send(queue, message, DeliveryMode.PERSISTENT, 4, 0);
59        if(messageCounter % 10 == 0){
60            queueSession.commit();
61        }
62    }
70    public void run(){
71        while(System.currentTimeMillis() < this.startProcessingAt){
72            //Wait for start...
73        }
74
75        try {
76            System.out.println("Start producing " + amountOfMessages + " Messages");
77            trackStartTime();
78
79            while(amountOfMessages > 0){ //System.currentTimeMillis() < this.durationTime
80                sendMessage("Nachricht " + amountOfMessages + "Payload: " + generateRandomContent() );
81                amountOfMessages--;
82                delay();
83            }
84
85            calculateAndPrintDurationTime();
86            this.queueConnection.close();
87        } catch (JMSException e) {
88            e.printStackTrace();
89        }
90    }
```

Abbildung 35: Producer-Test ActiveMQ

In Zeile 77 wird die Startzeit gespeichert, um diese später mit der Endzeit in Relation zu setzen. Anschließend wird eine *while()*-Schleife gestartet, die – je nach Testszenario – entweder so lange durchlaufen wird, bis die komplette Anzahl an Nachrichten versandt wurde oder das beabsichtigte Zeitintervall beendet ist. In Zeile 80 wird eine Nachricht versendet, für die explizit ein zufälliger Nachrichten-Inhalt (14 KB) generiert wird. Anschließend wird in Zeile 82 (je nach Testszenario) der Versand der Folgenachricht um eine bestimmte Dauer verzögert. Zuletzt wird, nachdem die *while()*-Schleife verlassen wurde, in Zeile 85 der benötigte Zeitbedarf und die Anzahl der gesendeten Nachrichten berechnet und auf der Kommandozeile ausgegeben.

In Zeile 55 startet die Methode *sendMessage()*, die den eigentlichen Nachrichtenversand vollzieht. In diesem Zuge wird in Zeile 57 eine Textnachricht erstellt, welche in der darauffolgenden Zeile versendet wird. Hierzu ist anzumerken, dass der Übertragungsmodus PERSISTENT verwendet wird, der dafür sorgt, dass die Nachrichten auf der Festplatte des Brokers gespeichert werden. (siehe Kapitel 3). Dieser Modus ist notwendig, um der nicht-funktionalen Anforderung *Zuverlässigkeit* gerecht zu werden und um *ActiveMQ* bzw. JMS in eine möglichst gleiche Ausgangslage wie *Apache Kafka* zu versetzen.

Zuletzt wird in den Zeilen 59 – 61 bei jeder 10. Nachricht die Transaktion *committed*. Hierdurch sind die 10, in dieser Transaktion gesendeten, Nachrichten für die Consumer abrufbar. Dies

bewirkt für den Broker eine Entlastung, da dieser ausschließlich bei jeder 10. Nachricht im Rahmen des *Commits* synchron mit dem Client kommunizieren muss [SBD11].

4.2.2 Consumer-Tests ActiveMQ

Das in Abschnitt 4.2.1 beschriebene Vorgehen beim Versand von Nachrichten im JMS-Umfeld deckt sich im Wesentlichen mit dem generellen Ablauf beim Empfang der Nachrichten. Es werden gleichermaßen Start- und Endzeiten gespeichert und ausgewertet. Abgesehen von dem anderen Basis-Aufbau aufgrund des Nachrichtenempfanges sind keine maßgeblichen Änderungen zu dem beschriebenen Grundaufbau aus Kapitel 3 zu nennen.

Es wird auf gleiche Art und Weise wie bei den Producer-Tests nach jeder 10. Nachricht die Transaktion *committed,* und die Nachrichten in diesem Zuge vom Server gelöscht. Dies schafft ebenfalls eine Entlastung für den Consumer, da dieser nur bei jeder 10. Nachricht synchron den Abschluss der Transaktion an den Broker melden muss.

Grundsätzlich problematisch bei der Durchführung der *ActiveMQ*-Consumer-Tests war, dass der Consumer lediglich einen Listener implementiert, aber den Broker nicht aktiv auffordern kann, eine höhere Nachrichtenmenge zu übertragen.

4.2.3 Producer-Tests Kafka

In diesem Abschnitt werden die Besonderheiten beschrieben, die innerhalb der Kafka-Producer-Tests Relevanz haben. Da im Allgemeinen die Clients (Producer, Consumer) im Kafka-Kontext mehr Verantwortung besitzen als im JMS-Kontext, gibt es an dieser Stelle ebenfalls deutlich mehr zu betrachtende Einstellungsoptionen.

Abbildung 36 zeigt die wesentlichen Quellcode-Ausschnitte der Kafka-Producer-Test-Software. Im Rahmen der Erstellung des Producer-Objektes im Konstruktor (Zeile 35 – 47) werden zahlreiche dieser Einstellungen definiert. Neben den Basiseinstelllungen aus Kapitel 3 wird in Zeile 43 die ACKS_CONFIG definiert. Diese gibt – wie bereits erwähnt – an, auf wie viele weitere Broker im Cluster der aktuelle Broker die Nachricht replizieren soll, bevor er eine Bestätigung an den Client zurücksendet. Durch die hier gewählte Einstellung *all* wird vom Broker gefordert, dass er mit der Bestätigung an den Client wartet, bis er selbst eine Bestätigung von allen replizierenden Servern erhalten hat. Dieses Vorgehen führt zwar zu Performanceeinbußen, stellt jedoch sicher, dass bestimmte *Quality of Service* Eigenschaften geboten werden. Durch die hier gewählte Einstellung ist ein Verlust von Nachrichten ausgeschlossen. Folglich wird im Rahmen der nicht-funktionalen Anforderung *Zuverlässigkeit* das Level *At-Least-Once* erreicht. Die Einstellung schließt jedoch nicht aus, dass Nachrichten doppelt konsumiert werden. Diese Problematik muss durch andere Maßnahmen (siehe Kapitel 3 und 8) verhindert werden.

Die Einstellung *RETRIES_CONFIG* gibt an, wie oft die Kafka-API automatisch versucht, die Nachricht erneut zu senden, bevor sie eine Exception wirft, die dann vom Programm entsprechend gehandhabt werden kann. An dieser Stelle wurde der Wert 2 gewählt, da es zwar zu fehlerhaften Sendeversuchen kommen darf, jedoch auch nicht zu lange vergeblich versucht soll die Nachricht zu versenden.

```
35    public Producer(String _topic, int _messagesPerSecond, long _startProcessingAt) {
36        messagesPerSecond = _messagesPerSecond;
37        startProcessingAt = _startProcessingAt;
38        amountOfMessages = messagesPerSecond * DURATION_IN_SECONDS;
39        topic = _topic;
40        kafkaProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"esbsvr03.ov.otto.de:9092");
41        kafkaProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,StringSerializer.class.getName());
42        kafkaProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,StringSerializer.class.getName());
43        kafkaProps.put(ProducerConfig.ACKS_CONFIG, "all");
44        kafkaProps.put(ProducerConfig.RETRIES_CONFIG, 2);
45        kafkaProps.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, 1869538);
46        producer = new KafkaProducer<String, String>(kafkaProps);
47    }
48
49    public void sendMessageToCluster(String content){
50        ProducerRecord<String, String> record = new ProducerRecord<String, String>(topic, content);
51        producer.send(record, new ProducerCallback());
52    }
53
54    public void run(){
55        while(System.currentTimeMillis() < this.startProcessingAt){ /* Wait for start... */ }
56        System.out.println("Start running ...");
57        trackStartTime();
58
59        while( amountOfMessages > 0 ){ /*System.currentTimeMillis() < this.durationTime */
60            sendMessageToCluster("Nachricht " + amountOfMessages + "Payload: " + generateRandomContent() );
61            amountOfMessages--;
62            messageCounter++;
63            delay();
64        }
65
66        calculateAndPrintDurationTime();
67        producer.close();
68        System.out.println("Connection closed");
69    }
```

Abbildung 36: Producer-Test Kafka

Die in Zeile 54 beginnende *run()*-Methode verhält sich identisch zu jener Methode des *ActiveMQ* Producers-Tests. Lediglich die darin verwendete Methode *sendMessageToCluster()* bietet potenziell drei verschiedene Arten der Nachrichtenübertragung an. Bei diesen Übertragungsarten handelt es sich um eine synchrone Übertragung, eine asynchrone Übertragung sowie eine asynchrone Übertragung mit Callback.

In dem hier erläuterten Kafka-Producer-Test wurde die Übertragungsart *asynchron mit Callback* gewählt. Der Client kann somit auf schnellere Art und Weise die Nachrichten an den Broker übertragen, ohne die Antwort einer einzelnen Nachricht abwarten zu müssen. Im Gegensatz zur reinen asynchronen Übertragung wird der Producer jedoch durch den Callback über das Scheitern des Nachrichtenversandes informiert und kann entsprechende Maßnahmen, wie beispielsweise das erneute Versenden der Nachricht, ergreifen.

Die Erzeugung der eigentlichen Nachricht erfolgt in Zeile 50 analog zur Beschreibung aus Kapitel 3. An dieser Stelle ist lediglich anzumerken, dass in Zeile 51 im Rahmen der send()-Methode die verantwortliche Callback-Klasse als Parameter übergeben wird.

4.2.4 Consumer-Tests Kafka

In diesem Abschnitt werden die Besonderheiten beschrieben, die im Zusammenhang mit den Kafka-Consumer-Tests zu beachten sind. Diese Besonderheiten sind von gesonderter Bedeutung, da an dieser Stelle aufgrund der Kafka-Konzepte fundiertes Fachwissen auf Entwicklerseite vorhanden sein muss. Bei einer falschen Konfiguration kann es zu einem sehr häufig durchgeführten Rebalancing kommen, das ein Pausieren der Nachrichtenverarbeitung zur Folge hätte. Dieses Verhalten würde zu einer Verfälschung der Testergebnisse führen.

Aufgrund dieser Problematik wurde bei der Testdurchführung darauf geachtet, jegliches Rebalancing während des Testzeitraums zu unterbinden. Aus diesem Bestreben und einer generell höheren Client-Komplexität ergibt sich ein deutlich umfangreicherer Consumer-Aufbau, der in Abbildung 37 dargestellt wird. Eine detaillierte Beschreibung zur Arbeitsweise eines Kafka-Consumers sowie den zugehörigen generellen Konzepten zur Nachrichtenübertragung im Kafka-Kontext können Kapitel 3 entnommen werden.

In Zeile 53 des dargestellten Konstruktors wird dem Consumer eine *ConsumerGroup* zugeordnet. Diese sorgt dafür, dass die Nachrichten gerecht auf alle Consumer innerhalb der *ConsumerGroup* verteilt werden. Detaillierte Informationen zu diesen Strukturierungsmöglichkeiten sowie den damit verbundenen Messaging-Modellen können Kapitel 3 entnommen werden.

Die Einstellung *PATITION_ASSIGNMENT_STRATEGY_CONFIG* in Zeile 54 gibt an, auf welche Weise die Nachrichten auf die Teilnehmer der ConsumerGroup verteilt werden sollen.

Die Einstellung *AUTO_OFFSET_RESET_COMMIT* gibt an, an welchem Startpunkt der Consumer mit der Verarbeitung beginnen soll, sofern der zuletzt gespeicherte Verarbeitungsstand nicht mehr verfügbar ist. Bezugnehmend auf die Beschreibung aus Kapitel 3 wird an dieser Stelle die Einstellung *earliest* verwendet, um einen Datenverlust auszuschließen.

Die Einstellung *ENABLE_AUTO_COMMIT_CONFIG* in Zeile 57 befähigt bzw. untersagt der Consumer-API, eigenständig den aktuellen Offset (Verarbeitungsstand) im Rahmen des Abrufs neuer Daten an den Broker zu übermitteln. Um im Rahmen der Testdurchführung flexibler mit der Übertragung des Verarbeitungstands umgehen zu können, wurde diese Option entsprechend deaktiviert.

```
49  public Consumer(String _topic, int _messagesPerSecond, long _startProcessingAt) {
50      kafkaProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"esbsvr03.ov.otto.de:9092");
51      kafkaProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,StringDeserializer.class.getName());
52      kafkaProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
53      kafkaProps.put(ConsumerConfig.GROUP_ID_CONFIG,"toNoaConsumer");
54      kafkaProps.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
55              "org.apache.kafka.clients.consumer.RoundRobinAssignor");
56      kafkaProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,"earliest");
57      kafkaProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
58      kafkaProps.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 5);
59      consumer = new KafkaConsumer<String, String>(kafkaProps);
60      consumer.subscribe(Collections.singletonList(topic), this);
61  }
62
63  public void run(){
64      while(System.currentTimeMillis() < this.startProcessingAt){ /* Waiting to start... */ }
65      System.out.println("Start consuming ... " + amountOfMessages );
66      trackStartTime();
67
68      try{
69          while(1 == 1){
70              ConsumerRecords<String, String> records = consumer.poll(0);
71              if(paused == false){
72                  for(ConsumerRecord<String, String> record: records){
73                      consumer.commitAsync(new OffsetCommitCallback() {
74                          public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {
75                              if (exception != null){
76                                  failedCommits++;
77                              }
78                          }
79                      });
80
81                      messageCounter++;
82                      amountOfMessages--;
83                      if(System.currentTimeMillis() > this.durationTime){  // amountOfMessages <= 0
84                          Collection<TopicPartition> partion = records.partitions();
85                          consumer.pause(partion);
86                          paused = true;
87                          this.terminationTime = System.currentTimeMillis() + 15000;
88                          calculateAndPrintDurationTime();
89                          break;
90                      }
91                  }
92              }else{
93                  delay();
94                  if(this.terminationTime < System.currentTimeMillis()){
95                      System.out.println("Quit Consumer!");
96                      break;
97                  }
98              }
99          }
100     }finally {
101         consumer.close();
102         System.out.println("Connection closed");
103     }
104 }
```

Abbildung 37: Consumer-Test Kafka

Als letzte Einstellung legt *MAX_POLL_RECORDS_CONFIG* in Zeile 58 fest, wie viele Nachrichten maximal innerhalb eines *poll*-Vorganges an den Client übertragen werden. Die hier definierte Nachrichtenmenge hat maßgeblichen Einfluss auf die Übertragungsgeschwindigkeit. An dieser Stelle wurde innerhalb des Tests ein verhältnismäßig kleiner Wert von 5 gewählt, da durch einen höheren Wert die Wahrscheinlichkeit eines Rebalance erhöht wird.

Nach der finalen Definition der zuvor beschriebenen Parameter beginnt die konkrete Nachrichtenverarbeitung innerhalb der *run()*-Methode ab Zeile 63. Um einen gleichzeitigen Start zu forcieren, wird in Zeile 64 ein einheitlicher Startzeitpunkt an die Threads kommuniziert.

Anschließend erfolgt in Zeile 69, konform zur *Kafka*-Dokumentation, die Ausführung einer *while()*-Schleife, die in Zeile 70 in jedem Schleifendurchlauf über die poll()-Funktionen einen

neuen Satz an Nachrichten vom Broker anfordert. Dieser wird in der darauf folgenden *for()*-Schleife über einen *Iterator* durchlaufen und die Nachrichten entsprechend ausgewertet.

Da, wie bereits erläutert, die automatische Übertragung des Verarbeitungsstandes deaktiviert wurde, erfolgt in Zeile 73 die explizite Übertragung des Offsets. Orientierend an den in Kapitel 3 beschriebenen möglichen Vorgehensweisen wird der Verarbeitungsstand asynchron an den Broker übertragen. Das ist dahingehend unkritisch, da bei einer gescheiterten Übertragung die anschließende Übertragung die vorherige impliziert (siehe Kapitel 3).

Zuletzt erkennt die in Zeile 83 beginnende *IF*-Bedingung, dass der Testzeitraum beendet ist und setzt das *Polling* auf *pause*. Dies führt dazu, dass der Consumer weiter über die *poll()*-Funktion einen *Heartbeat* an den Broker sendet, jedoch dabei keine neuen Nachrichten mehr zur Verarbeitung erhält. Anschließend verbleibt der Consumer noch weitere 15 Sekunden in diesem Zustand bevor er terminiert. Dieses Vorgehen ist notwendig, da der Stopp eines Consumers ansonsten einen Rebalance zur Folge hätte. Hierdurch werden die Thread die ihre Verbreitung noch nicht beendet haben, bei ihrer Fertigstellung nicht behindert. Ein Rebalance würde dafür sorgen, dass die Partitionen neu unter den noch verfügbaren Consumern aufgeteilt werden. Da dies jedoch einen kurzzeitigen Stopp der Nachrichtenverarbeitung zur Folge hat, würde dies zu einer Verfälschung der Testergebnisse führen. Weitere Informationen zu dieser Thematik können Kapitel 3 entnommen werden.

4.3 Testinfrastruktur

In diesem Abschnitt wird die verwendete Testinfrastruktur einschließlich der zugehörigen Server und derer Performance beschrieben.

Für einen repräsentativen Vergleich beider Messaging-Ansätze wurden die beteiligten Komponenten – Broker, Producer und Consumer – stets auf getrennten Systemen ausgeführt.

In ersten Testversuchen wurden die jeweiligen Tests zwar auf einem lokalen Rechner durchgeführt. Jedoch stellte sich hierbei schnell heraus, dass sich die einzelnen Komponenten (Consumer, Producer, Broker) stark beeinträchtigten und sich gegenseitig die Ressourcen entziehen. Dieses Problem wird zusätzlich dadurch verstärkt, dass Kafka-Clients wesentlich mehr CPU-Leistung fordern als jene von *ActiveMQ*. Ein Vergleich beider Systeme wäre damit nicht fundiert, da eines der Systeme letztendlich mit weniger Ressourcen auskommen müsste.

Abbildung 38 zeigt jene Test-Infrastruktur, die innerhalb der *OTTO*-Rechenzentren verwendet wurde. Sie setzt sich aus insgesamt 4 Linux-Servern zusammen, die über identische Hardware-Kapazitäten verfügen. Die Server unterscheiden sich lediglich dahingehend, dass es sich bei *esbadmpho01* und *esbadmpho02* um Systeme handelt, auf denen die Client-Software innerhalb

von Docker-Containern und nicht direkt auf dem Basisbetriebssystem ausgeführt wird. Die Server *esbsvr03* und *esbsvr04* werden im Gegensatz dazu direkt als eigenständige Linux-Server eingesetzt.

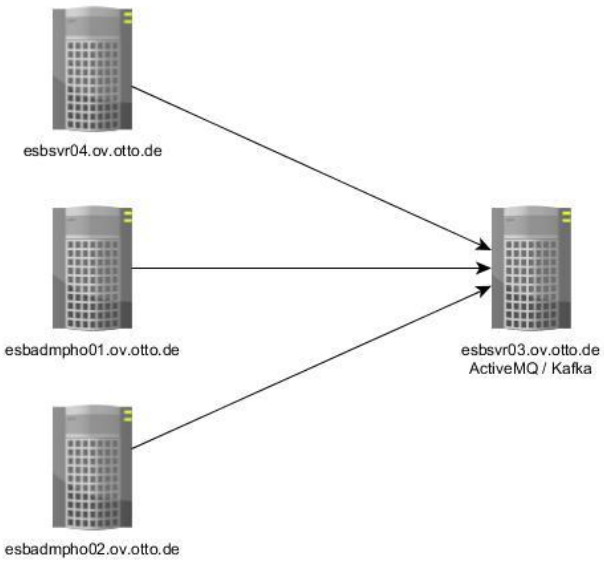


Abbildung 38: Test-Infrastruktur

Innerhalb der Testinfrastruktur wurde der Server *esbsvr03* je nach Anwendungsfall entweder als Kafka- oder ActiveMQ-Broker betrieben. Die anderen Server fungierten als Producer bzw. Consumer, welche die Nachrichten entsprechend an den *esbsvr03*-Server übertragen. Hierzu wurde die Client-Software als JAR-Datei an die Server übertragen und, entweder direkt oder im Rahmen eines Docker-Containers, ausgeführt.

Bei den einzelnen Servern handelt es sich um virtuelle Maschinen, die innerhalb der OTTO-Rechenzentren verwaltet werden. Die Maschinen verfügen jeweils über zwei *Intel® Xeon® CPU-E5-4657L v2 @ 2.40GHz* Prozessoren, 16 GB Arbeitsspeicher sowie eine 50 GB SSD-Festplatte.

5 Testdurchführung & Evaluation

In diesem Kapitel werden die Testergebnisse und die damit verbundenen Erkenntnisse, die sich aus den Testszenarien des letzten Kapitels ergeben, beschrieben. In diesem Zusammenhang werden die ermittelten Werte von *Apache Kafka* und *ActiveMQ* innerhalb der Diagramme unmittelbar gegenübergestellt.

Im Rahmen des Abschnitts 5.1 werden die Producer-seitigen Testergebnisse veranschaulicht. Anschließend erfolgt die Beschreibung der Consumer-seitigen Testergebnisse in Abschnitt 5.2. Zuletzt befasst sich Abschnitt 5.3 mit dem Einfluss des Betriebssystem-Cache auf die Performance der Broker.

5.1 Ergebnisse Producer-Tests

In diesem Abschnitt werden die Ergebnisse der Producer-Tests auf Basis der, in Kapitel 4 definierten, Testszenarien beschrieben. Hierzu befasst sich Abschnitt 5.1.1 mit der Gegenüberstellung der Producer-seitigen Nachrichten/Sekunde-basierten Tests. Im Abschnitt 5.1.2 werden die Producer-basierten Tests und in dem darauffolgenden Abschnitt 5.1.3 die Datenmengen-basierten Tests behandelt.

5.1.1 Nachrichtenbasierte Producer Testergebnisse (T.1a und T.1b)

In den Nachrichten/Sekunden-basierten Tests auf Producer-Seite wurde systematisch die Nachrichtenmenge von 500 Nachrichten pro Sekunde auf insgesamt 5500 Nachrichten pro Sekunde gesteigert. Sofern sich die dargestellten Kurven nicht über das komplette Intervall erstrecken, liegt das daran, dass die Test-Last in diesem Spezialfall nicht weiter gesteigert werden konnte.

5.1.1.1 Zeitbedarf

Die Gegenüberstellung des Zeitbedarfs von *Apache Kafka* (rot) und *Apache ActiveMQ* (blau) zeigt innerhalb der Abbildung 39, dass der zusätzlich benötigte Zeitbedarf bei *Apache Kafka* über das volle Testintervall konstant bleibt.

Im Gegensatz dazu steigt bei *ActiveMQ* der zusätzliche Zeitbedarf bis zu einem Nachrichtenaufkommen von 3000 Nachrichten pro Sekunde konstant an. Ab dieser Schwelle nimmt der Zeitbedarf für die Verarbeitung weiterer Nachrichten überproportional zu. Konkret bedeutet dies, dass sich der Zeitbedarf bei einer Steigerung der Nachrichtenmenge von 3000 auf 4000 Nachrichten bereits mehr als verdoppelt hat.

Das Verhalten der beiden Systeme hinsichtlich des zusätzlichen Zeitbedarfs legt die Vermutung nahe, dass *ActiveMQ* spätestens ab einer Belastung von 3000 Nachrichten pro Sekunde an eine Belastungsgrenze stößt, die sich darin ausdrückt, dass eine bestimmte Ressource innerhalb des Brokers erschöpft ist.

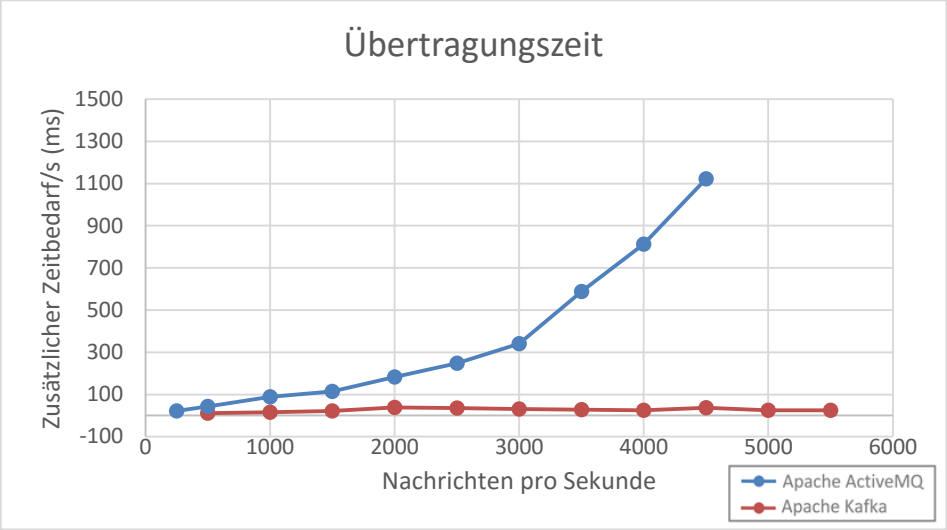


Abbildung 39: Zusätzlicher Zeitbedarf im Verhältnis zu Nachrichten

Im Gegensatz dazu bleibt *Apache Kafka* über das komplette Testintervall im Hinblick auf den zusätzlichen Zeitbedarf konstant. Dies lässt den Schluss zu, dass das System noch ausreichend Ressourcen zur Verfügung hat.

5.1.1.2 IO-Operationen

Die Gegenüberstellung der Schreiboperationen, die von beiden Systemen pro Sekunde verursacht werden, unterstreicht die These, dass ab einer Menge von 3000 Nachrichten/s ein Ressourcenengpass im Hinblick auf *ActiveMQ* vorliegt.

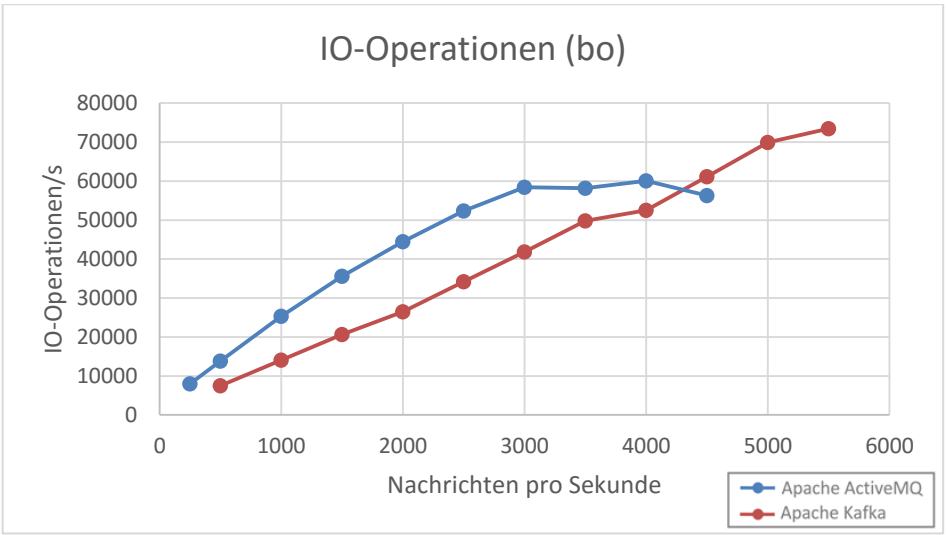


Abbildung 40: IO-Operationen im Verhältnis zu Nachrichten pro Sekunde

Im konkreten Fall drückt sich dies in Abbildung 40 dadurch aus, dass auch hier die IO-Operationen bis zu einer Menge von 3000 Nachrichten pro Sekunde linear ansteigen. Anschließend stagnieren die IO-Operationen, obwohl die Nachrichtenmenge weiterhin erhöht wird.

Daraus lässt sich schließen, dass der erhöhte Zeitbedarf (siehe Abbildung 39) aus der fehlenden Möglichkeit, die IO-Operationen weiter zu steigern, resultiert. Der Grund dafür ist, dass der Broker mit der Weiterverarbeitung warten muss, bis wieder IO-Operationen getätigt werden können. Der zuletzt erkennbare Abfall der Kurve ist vermutlich dadurch zu erklären, dass durch die ineffiziente Durchführung der IO-Operationen die CPU noch deutlicher beeinträchtigt wird.

Im Gegensatz dazu zeigt das Verhalten von *Apache Kafka* (rot) in der Abbildung 40 eine nahezu konstante lineare Steigerung der IO-Operationen, im Verhältnis zur Anzahl an gesendeten Nachrichten, über das gesamte Testintervall hinaus. Das spricht dafür, dass im Fall von *Apache Kafka* bei einer Menge von 5500 Nachrichten und insgesamt fast 75000 IO-Operationen pro Sekunde das Belastungs-Limit noch nicht erreicht ist.

5.1.1.3 CPU-Auslastung

Unter Einbeziehung der CPU-Auslastung beider Systeme zeigt sich bei *ActiveMQ* ebenfalls, dass die CPU ab 3000 Nachrichten deutlich erkennbar bei einer Auslastung von knapp 75% stagniert. Folglich wäre auf dem Broker noch genug Rechenleistung verfügbar, um weitere Nachrichten zu verarbeiten, jedoch hemmt die Limitierung von weiteren IO-Operationen (siehe Abschnitt 5.1.1.2) eine Nutzung der restlichen CPU-Kapazität.

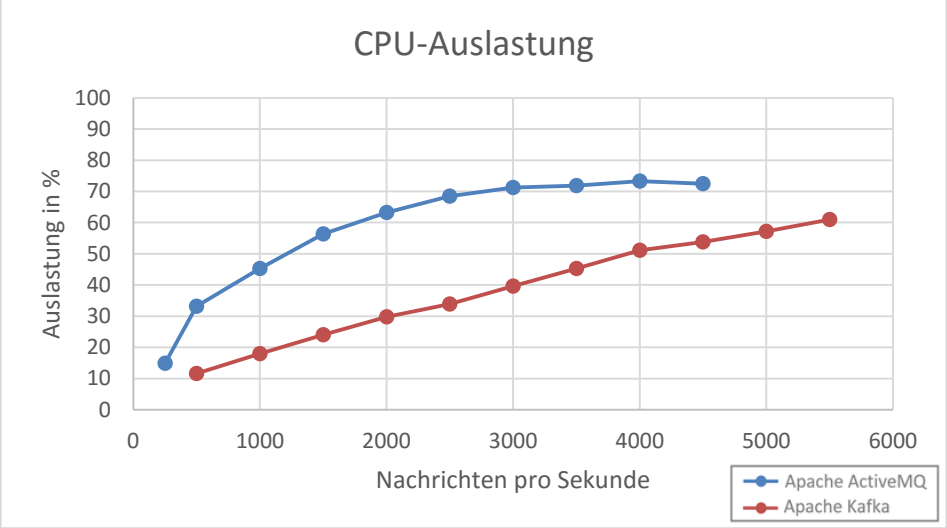


Abbildung 41: CPU-Auslastung im Verhältnis zu Nachrichten pro Sekunde

Im Fall von *Apache Kafka* ist erneut ein weitestgehend lineares Verhalten über das komplette Testintervall zu erkennen. Dieses deckt sich mit den zuvor dargestellten IO-Operationen (siehe Abschnitt 5.1.1.2) und dem konstanten Zeitbedarf (siehe Abschnitt 5.1.1.1).

5.1.1.4 Zusammenfassende Test-Erkenntnisse

Zusammenfassend zeigt der Nachrichten/Sekunden-basierte Test, dass bei *Apache ActiveMQ* der limitierende Faktor die nicht mehr steigerbaren IO-Operationen sind. Im Hinblick auf eine

vertikale Skalierung des Systems wäre es damit nicht zielführend, weitere Rechenleistung oder weiteren Arbeitsspeicher bereitzustellen. Lediglich eine Steigerung der Festplatten-Geschwindigkeit würde eine Optimierung des Systems hervorrufen. Das Defizit an vorhandenen Ressourcen drückt sich in einem erhöhten Zeitbedarf aus.

Im Fall von *Apache Kafka* zeigt sich beim zusätzlichen Zeitbedarf ein konstantes Verhalten über das komplette Testintervall. Darüber hinaus steigen die IO-Operationen und die CPU-Auslastung im Verhältnis zur Anzahl an Nachrichten pro Sekunde linear an. Das konstante Verhalten von *Apache Kafka* führt zu dem Schluss, dass innerhalb dieses Systems nicht die IO-Operationen, sondern die CPU der limitierende Faktor sein wird.

Dadurch, dass bei *Apache Kafka* die IO-Operationen wesentlich effizienter durchgeführt werden, muss die CPU nur in sehr geringem Maße auf die Fertigstellung von IO-Operationen warten. Daher kann die CPU wesentlich effizienter bis zu einer Auslastung von voraussichtlich 100% genutzt werden. Das Verhalten hat ebenfalls maßgeblichen Einfluss auf die Ressourcenauslastung der Client-Systeme, da diese nicht transitiv auf die Ressourcen des Brokers warten müssen. Bei *ActiveMQ* hingegen befindet sich innerhalb der CPU-Auslastung des Brokers ein sehr hoher prozentualer Anteil an Wartezeit, der für die Fertigstellung bzw. Durchführung von IO-Operationen anfällt. Dies führt innerhalb der sendenden Clients dazu, dass transitiv ebenfalls auf die CPU des Brokers gewartet werden muss. Diese Charakteristika der beiden Systeme haben damit zusätzlich maßgeblichen Einfluss auf das Verhalten der jeweiligen Clientanwendungen.

Da bei *Apache Kafka* die Producer-Systeme damit wesentlich schneller an ihre Grenzen stoßen, konnte innerhalb der hier verwendeten Testinfrastruktur, der *Kafka*-Broker nicht vollständig ausgelastet werden. Das zuvor erwähnte Verhalten hinsichtlich des limitierenden Faktors CPU, deutet darauf hin, dass sich die Menge an Nachrichten pro Sekunde durch eine weitere Beanspruchung der CPU im Testsystem noch auf knapp 9000 Nachrichten pro Sekunde erweitern ließe. Ein Ziel, der in den nächsten Abschnitten beschriebenen Tests, besteht darin, eine Steigerung der IO-Operationen zu forcieren, um diese Thesen zu belegen.

Abschließend lässt sich als Ergebnis des Nachrichten/Sekunden-basierten Testlaufs bereits anmerken, dass bei *ActiveMQ* die Festplattengeschwindigkeit bzw. die IO-Operationen der limitierende Faktor ist. Bei *Apache Kafka* hingegen ist dies die CPU-Auslastung. Da es im Allgemeinen deutlich einfacher ist, die Systeme mit mehr Rechenleistung auszustatten als die Festplattengeschwindigkeit zu erhöhen, lässt sich *Kafka* damit deutlich besser vertikal skalieren.

5.1.2 Producer-basierte Testergebnisse (T.2a und T.2b)

In diesem Abschnitt werden die Test-Ergebnisse der Producer-basierten Testdurchläufe von *Apache ActiveMQ* und *Apache Kafka* gegenübergestellt. Hierfür wurde in beiden Systemen systematisch die Anzahl an Producern von dem Wert 1 auf den Wert 10 erhöht.

5.1.2.1 Anzahl Nachrichten

Bei der Gegenüberstellung der Anzahl der übertragenen Nachrichten pro Sekunde zeigt sich in Abbildung 42 im Allgemeinen eine deutlich höhere Empfangsrate bei *Apache Kafka* (rot) als bei *ActiveMQ* (blau). Die maximale Empfangsrate von *ActiveMQ* liegt bei knapp 4300 Nachrichten pro Sekunde. Dieses Maximum wird bereits bei 6 Producern erreicht und lässt sich im weiteren Verlauf nicht mehr weiter steigern.

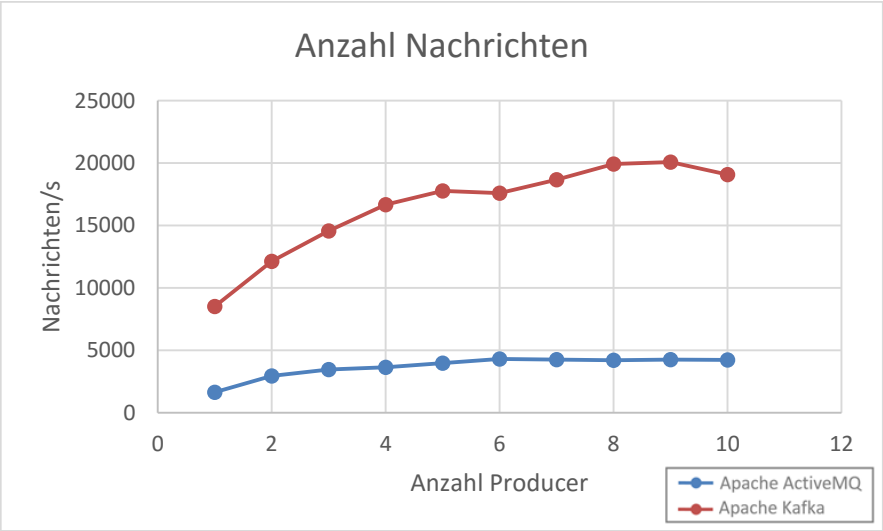


Abbildung 42: Anzahl Nachrichten im Verhältnis zur Anzahl an Producern

Bei *Apache Kafka* hingegen steigt die Anzahl der Nachrichten deutlich länger mit der Menge an eingesetzten Producern an. Hierbei liegt die maximale Nachrichtenmenge bei knapp 20.000 Nachrichten pro Sekunde. Die maximale Auslastungsgrenze ist damit bei *Apache Kafka* im verwendeten Testsystem mit 20.000 Nachrichten pro Sekunde um den Faktor 4 höher als bei *ActiveMQ*.

Abschließend lässt der Kurvenverlauf der Nachrichtenmenge von *ActiveMQ* – ähnlich wie im Nachrichten/Sekunden-basierten Test aus Abschnitt 5.1.1 – erneut den Rückschluss zu, dass das System schon beim Einsatz von 6 Consumern aufgrund eines Ressourcenmangels komplett ausgelastet ist.

5.1.2.2 IO-Operationen

In Bezug auf die IO-Operationen zeigt sich bei den Producer-basierten Tests ebenfalls, dass diese sich bei *ActiveMQ* bereits bei 6 Producern nicht mehr signifikant steigern lassen und damit die maximale Menge an Operationen pro Sekunde erreicht ist. Diese liegt, wie anhand des Kurvenverlaufs in Abbildung 43 ersichtlich, bei etwa 90.000 Schreiboperationen pro Sekunde.

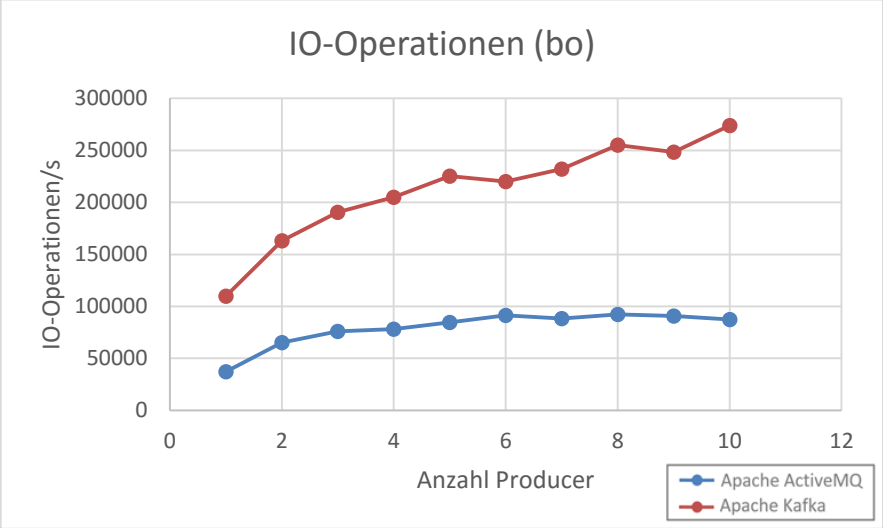


Abbildung 43: IO-Operationen im Verhältnis zur Anzahl an Producern

Bei *Apache Kafka* ist zum einen ersichtlich, dass die Anzahl an IO-Operationen, wie auch zuvor die Menge an Nachrichten, deutlich über jener von *ActiveMQ* liegt. Zum anderen ist ersichtlich, dass die IO-Operationen mit der steigenden Anzahl an Producern über das komplette Intervall mitwachsen. Darüber hinaus stehen die erzeugten IO-Operationen im Verhältnis zur Nachrichtenmenge (siehe Abschnitt 5.1.2.1). Dies spricht dafür, dass das System kontinuierlich weiter ausgelastet wird und keine Ressourcenengpässe vorliegen.

In der direkten Gegenüberstellung der IO-Operationen beider Systeme ist im verwendeten Testsystem zu erkennen, dass *Apache Kafka* im Optimalfall in der Lage ist, eine dreimal höhere Menge an IO-Operationen als *ActiveMQ* durchzuführen.

5.1.2.3 CPU-Auslastung

Die Gegenüberstellung der CPU-Auslastung der beiden Systeme in Abbildung 44 zeigt, dass *ActiveMQ,* wie bereits erwartet, frühzeitig mit der CPU-Auslastung stagniert und sich nur noch mäßig steigern lässt. Ferner ist anzumerken, dass sich innerhalb der CPU-Auslastung von *ActiveMQ* ein sehr hoher Anteil befindet (durchschnittlich 35%), in dem die CPU auf IO-Operationen wartet und somit nicht effektiv genutzt werden kann. Dies führt dazu, dass die eigentliche Netto-CPU-Auslastung nur bei durchschnittlich 50% liegt.

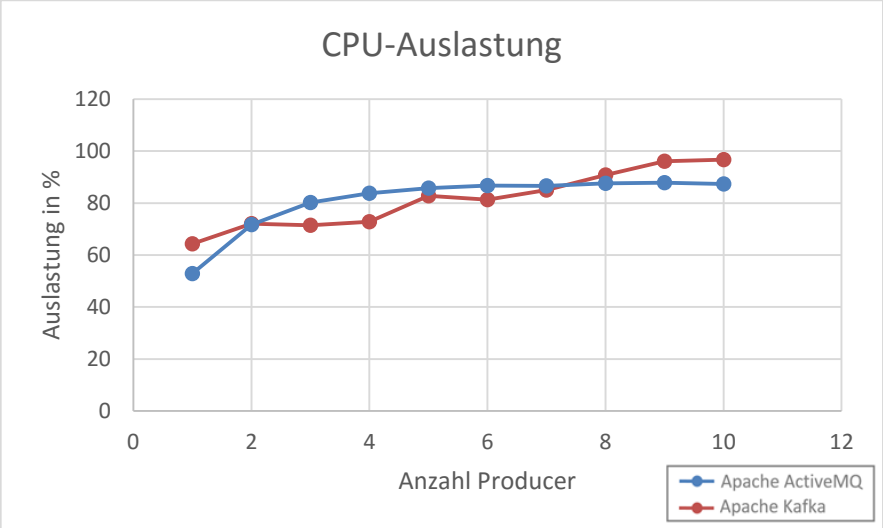


Abbildung 44: CPU-Auslastun im Verhältnis zur Anzahl an Consumern

Apache Kafka hingegen läuft mit der Steigerung der Producer kontinuierlich auf eine Komplettauslastung der CPU hinaus und erreicht diese bei einer Anzahl 9 – 10 Producern. Der Vollständigkeit halber ist anzumerken, dass hier der Faktor bei dem die CPU auf IO-Operationen wartet, äußerst gering ist und deutlich unter 10% liegt.

5.1.2.4 Zusammenfassende Testergebnisse

Zusammengefasst zeigt auch der Producer-basierte Test, dass bei *ActiveMQ* der limitierende Faktor die ineffiziente Vorgehensweise im Umgang mit den Schreiboperationen ist. Bei *Apache Kafka* ist dieser Faktor hingegen die CPU-Auslastung. Diese Erkenntnis und die zusätzliche Leistungssteigerung in Bezug auf den Test aus Abschnitt 5.1.1 belegt außerdem die These, dass sich die Anzahl an verarbeitbaren Nachrichten proportional mit der noch verfügbaren CPU-Leistung hochskalieren lässt. Der in diesem Abschnitt beschriebene Test zeigt in diesem Zusammenhang, dass die prognostizierte Menge an IO-Operationen zu erreichen ist und dass die CPU in vollem Umfang ausgenutzt werden kann.

Des Weiteren ist eine deutlich höhere Leistungssteigerung als zuvor erwartet erkennbar. Dies ist unter anderem darin begründet, dass durch einen konstanten Sendevorgang entsprechende Optimierungskonzepte in den Clients, wie beispielsweise eine Nachrichtenübertragung in Batches, effizienter funktionieren und der Broker somit entlastet wird. Folglich reduziert sich, durch geringere Quittierungsoperationen, der Ressourcenverbrauch im Broker.

Innerhalb des hier beschriebenen Tests zeigt sich in der verwendeten Infrastruktur, dass *Apache Kafka* in der Lage ist, eine um den Faktor 4,5 höhere Nachrichtenmenge im Vergleich zu *ActiveMQ* aufzunehmen.

5.1.3 Datenbasierte Producer Testergebnisse (T.3a und T3.b)

Als Randbetrachtung zu den beiden Testszenarien aus den Abschnitten 5.2.1 und 5.2.2 wurde zusätzlich ein datenbasierter Test durchgeführt, welcher von der verwendeten Nachrichtengröße abstrahiert. Darüber hinaus wurde in diesem Test versucht, die IO-Operationen gezielt zu maximieren, um potenzielle Grenzen aufzudecken.

5.1.3.1 Zeitbedarf

Im Hinblick auf das zeitliche Verhalten ist in Abbildung 45 erkennbar, dass dieses bei *Apache Kafka* bis zu einer Datenmenge von 300 MB konstant bleibt und anschließend leicht ansteigt. Zwischen der Erhöhung der Datenmenge von 300 auf 400 MB liegt damit die maximale Belastungsschwelle des Systems. An dieser Stelle kann erstmalig für dieses System vermutet werden, dass eine Ressource innerhalb des Brokers erschöpft ist.

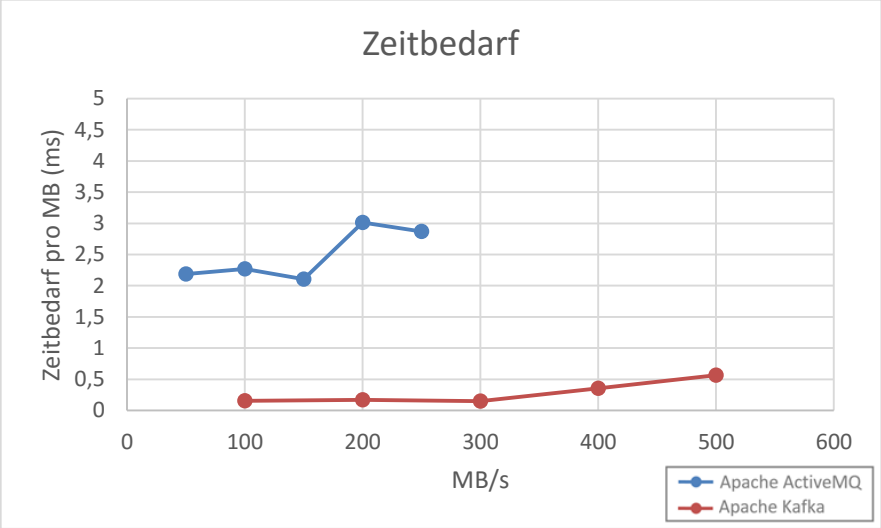


Abbildung 45: Zusätzlicher Zeitbedarf im Verhältnis zu MB pro Sekunde

Im Fall von *ActiveMQ* zeigt sich von Beginn an ein deutlich höherer Zeitbedarf als bei *Apache Kafka*. Dieser Zeitbedarf steigt zwischen 150 und 200 MB auf das Doppelte an. Dies lässt auch hier die Annahme zu, dass ab dieser Menge eine Belastungsgrenze erreicht ist.

5.1.3.2 IO-Operationen

Im Hinblick auf die IO-Operationen ist in Abbildung 46 bei *ActiveMQ* erkennbar, dass sich diese ab der vermuteten Belastungsgrenze von 150 MB nicht mehr signifikant steigern lassen und die Steigung der Kurve abflacht. Dies lässt auf Basis der Erkenntnisse aus den vorherigen Tests den Rückschluss zu, dass die Menge an maximalen IO-Operationen erreicht ist und somit ein Ressourcenengpass vorliegt.

Im Fall von *Apache Kafka* ist ersichtlich, dass die Kurve von IO-Operationen ab 400 MB deutlich abflacht und sich anschließend nur noch geringfügig erhöht. Da bereits ermittelt wurde, dass die CPU der limitierende Faktor bei *Apache Kafka* ist, lässt sich vermuten, dass die IO-Operationen transitiv durch einen Mangel an CPU-Leistung zur Neige gehen. Diese These wird im anschließenden Abschnitt 5.1.3.3 näher betrachtet.

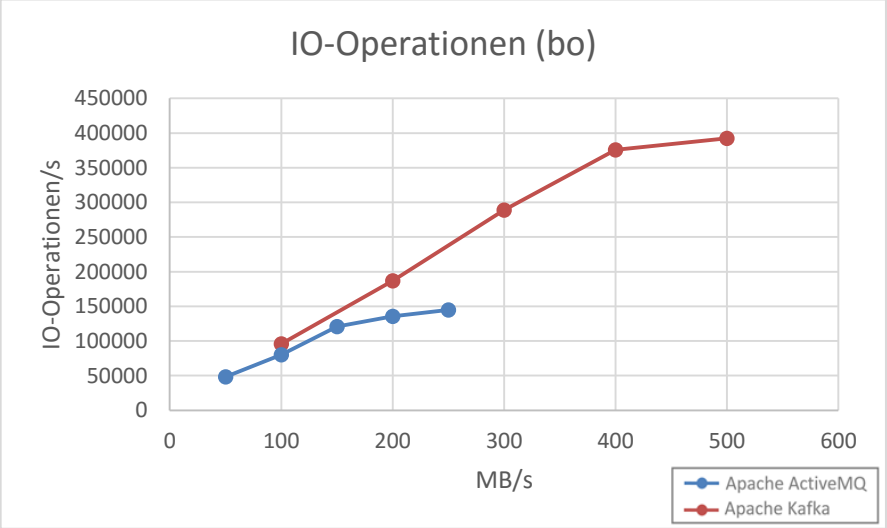


Abbildung 46: IO-Operationen im Verhältnis zu MB pro Sekunde

Allgemein lässt sich jedoch erkennen, dass bei beiden Systemen die IO-Operationen durch ein hohes Datenaufkommen noch weiter gesteigert werden konnten - bei *ActiveMQ* auf knapp 150.000 und bei *Apache Kafka* sogar auf knapp 400.000 IO-Operationen pro Sekunde.

5.1.3.3 CPU-Auslastung

Anhand der CPU-Auslastung zeigt sich in Abbildung 47, dass diese im Fall von *ActiveMQ* bei 200 MB ihr Maximum erreicht hat und bei 85% stagniert.

Erneut sei an dieser Stelle angemerkt, dass in diesem Wert ein signifikanter Anteil enthalten ist, in welchem die CPU ineffizient auf IO-Operationen warten muss. Weiterhin bestätigt sich erneut die These, dass noch CPU-Leistung verfügbar wäre, diese aber nicht ausgenutzt werden kann, da der Broker nicht in der Lage ist weitere Schreiboperationen zu produzieren.

Zudem bestätigt sich außerdem, die These, dass die IO-Operationen im Fall von *Apache Kafka*, aufgrund von mangelnder CPU-Auslastung, zur Neige gehen (Abschnitt 5.1.3.2). Die CPU erreicht bei einem Datenaufkommen von 400 MB pro Sekunde ihr Maximum. Somit wird ihre Leistung komplett ausgeschöpft.

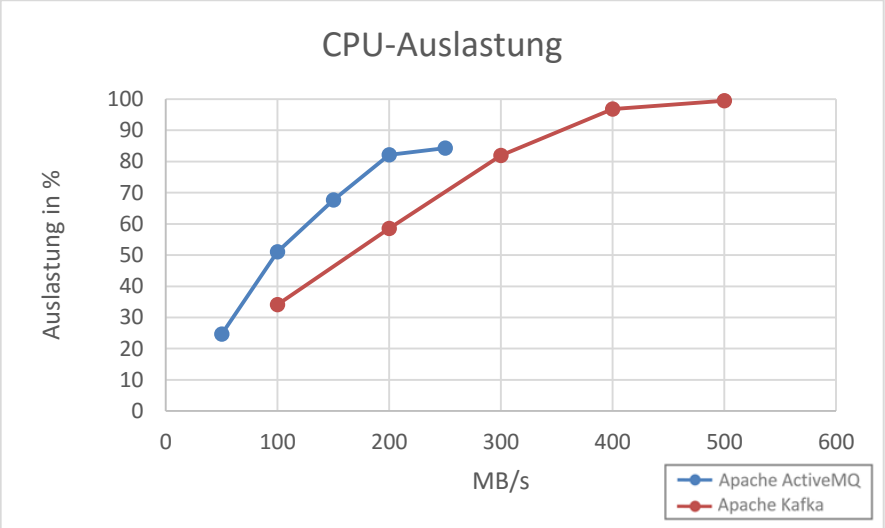


Abbildung 47: CPU-Auslastung im Verhältnis zu MB pro Sekunde

5.1.3.4 Zusammenfassende Ergebnisse

Abschließend zeigt sich erneut, durch die in diesem Kapitel beschriebenen Tests, dass die Leistung von *ActiveMQ* durch die IO-Operationen und bei *Apache Kafka* durch die CPU limitiert ist.

Ferner zeigen die Tests, dass *ActiveMQ*, in der hier verwendeten Testinfrastruktur und unter den verwendeten Rahmenparametern, mit 150 – 200 MB knapp halb so viele Daten pro Sekunde aufnehmen kann wie Apache Kafka. Bei Apache Kafka liegt die maximale Datenmenge bei 300 – 400 MB.

5.2 Ergebnisse Consumer-Tests

In diesem Abschnitt werden die Ergebnisse der Consumer-Tests beschrieben. Hierzu befasst sich Abschnitt 5.2.1 mit den durchgeführten Nachrichten/Sekunden-basierten Tests. Darauffolgend wird im Abschnitt 5.2.2 auf die Consumer-basierten Tests, welche systematisch die Consumer-Anzahl steigern, eingegangen.

Grundsätzlich ist im Zusammenhang mit den Consumer-Tests anzumerken, wie bereits in Kapitel 3 beschrieben, dass die beiden Test-Systeme bei der Nachrichtenauslieferung unterschiedlich vorgehen. *ActiveMQ*-Broker werden hierbei eigenständig aktiv und senden neue Nachrichten von sich aus an die Clients. *Kafka*-Broker hingegen lassen die Clients kontinuierlich anfragen, ob neue Nachrichten vorliegen und liefern diese im Gegenzug aus. Durch diese unterschiedlichen Herangehensweisen ergeben sich bei der Test-Durchführung an einigen Stellen Herausforderungen, die eine Steigerung der Test-Last erschweren bzw. verhindern. Diese Hindernisse werden an den entsprechenden Stellen beschrieben.

5.2.1 Nachrichtenbasierte Consumer Testergebnisse (T.4a und T.5b)

Im Nachrichten/Sekunden-basierten Testszenario wurde analog, wie im Fall der Producer-Tests, die Nachrichtenmenge systematisch um jeweils 500 Nachrichten pro Sekunde erhöht.

Zu Beginn dieses Abschnitts fällt auf, dass hier, anders als bei den Producer-Tests, *ActiveMQ* unter deutlich größerer Last getestet werden konnte als *Apache Kafka* (siehe Abbildungen). Dieser Sachverhalt ist nicht dadurch zu erklären, dass *Apache Kafka* an seine Leistungsgrenze stößt, sondern dass der Vorgang bei der Auslieferung von Nachrichten deutlich komplexer ist.

Mit steigender Last und einer höheren Anzahl an Consumern steigt die Wahrscheinlichkeit eines Rebalancings (siehe Kapitel 3). Da innerhalb dieses Zeitraums keine Nachrichten konsumiert werden können, würde dies zu einer Verfälschung der Testergebnisse führen. Durch geeignete Gegenmaßnahmen, wie dem Abruf kleinerer Batches, wurde versucht dieser Problemstellung entgegenzuwirken.

In einem späteren produktiven Einsatz sind diese Effekte zwar mit angemessener Sorgfalt zu betrachten, können jedoch bei passender Konfiguration weitestgehend vernachlässigt werden. Das liegt daran, dass ein Rebalancing einer besseren Nachrichtenverteilung auf die Consumer dient und damit die Gesamt-Produktivität langfristig verbessert.

5.2.1.1 Zeitbedarf

Im Hinblick auf den Zeitbedarf zeigt sich bei *Apache Kafka*, ein ähnlich lineares Verhalten, wie auch bei den Producer-Tests (siehe Abbildung 48). Jedoch liegt der zusätzliche Zeitbedarf zu Beginn deutlich über dem anfänglich konstanten Zeitbedarf von *ActiveMQ*.

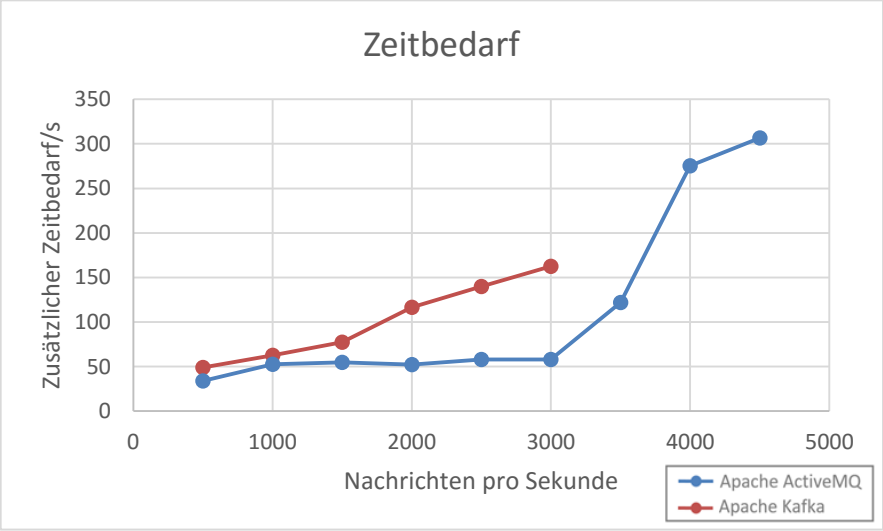


Abbildung 48: Zeitbedarf im Verhältnis zu Nachrichten pro Sekunde

Dieser zu Beginn konstante Zeitbedarf bei *ActiveMQ* nimmt jedoch ab einer Belastung von 3000 Nachrichten pro Sekunde drastisch zu. Dieses Verhalten legt, ähnlich wie bei den Producer-Tests, die Vermutung nahe, dass ab diesem Zeitpunkt die maximale Belastungsgrenze erreicht ist und somit eine bestimmte Ressource innerhalb des Brokers erschöpft ist.

5.2.1.2 IO-Operationen

Anhand von Abbildung 49 bestätigt sich die These, dass im Fall von *ActiveMQ* die IO-Operationen ab einer Auslieferungsmenge von 3500 Nachrichten pro Sekunde nicht mehr signifikant gesteigert werden können und damit auch hier die Schreib- und Leseoperationen der limitierende Faktor sind.

Zusätzlich ist an dieser Stelle anzumerken, dass sich bei den Consumer-Tests die Menge an IO-Operationen überwiegend aus Leseoperationen (bi) zusammensetzt. Es entsteht jedoch zusätzlich ein kleiner Anteil an Schreiboperationen, die auf Löschvorgänge im Broker zurückzuführen sind.

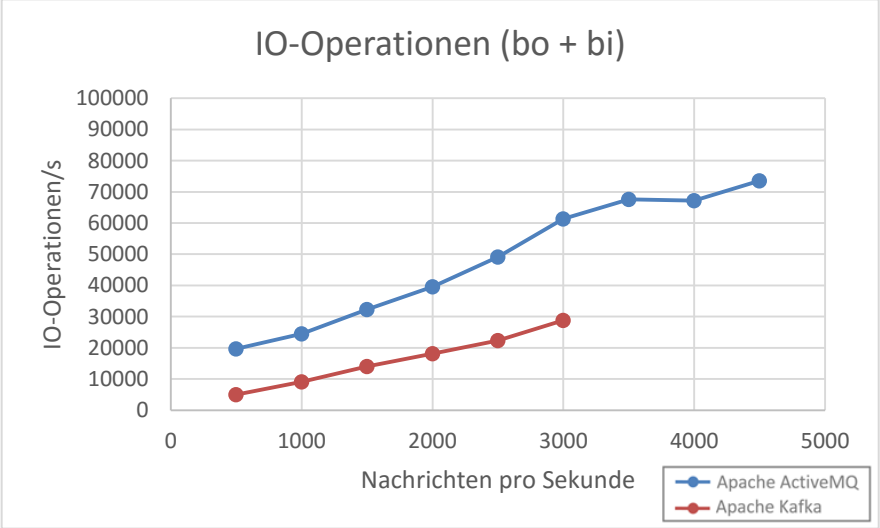


Abbildung 49: IO-Operationen im Verhältnis zu Nachrichten pro Sekunde

Im Fall von *Apache Kafka* zeigt sich zum einen, dass die IO-Operationen deutlich unter jenen von *ActiveMQ* liegen. Dies liegt unter anderem daran, dass innerhalb dieses Systems die Nachrichten im Anschluss an die Konsumierung nicht gelöscht und daher keine Schreiboperationen produziert werden.

Zum anderen ist ersichtlich, dass die IO-Operationen konstant im Verhältnis zur steigenden Nachrichtenmenge pro Sekunde mitwachsen und somit kein Ressourcenengpass zu erkennen ist.

5.2.1.3 CPU-Auslastung

Bei der Gegenüberstellung der zugehörigen CPU-Auslastung in Abbildung 50 zeigt sich im Fall von *ActiveMQ* ebenfalls, dass die Kurve ab einem Wert von 3500 Nachrichten pro Sekunde nicht mehr gesteigert werden kann. Da zu diesem Zeitpunkt jedoch lediglich 65% der CPU verwendet wird, bestätigt sich an dieser Stelle erneut, dass der Broker grundsätzlich noch mehr Nachrichten verarbeiten könnte, jedoch nicht mehr in der Lage ist, eine höhere Anzahl an IO-Operationen zu produzieren.

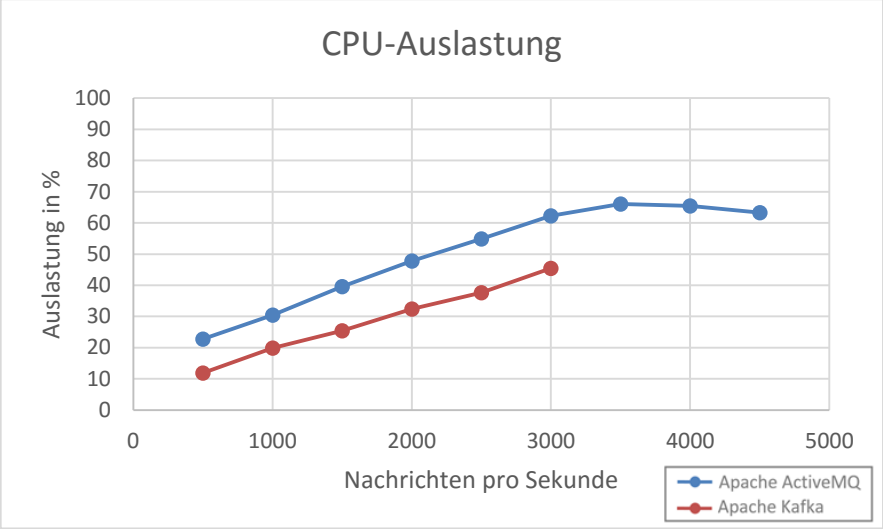


Abbildung 50: CPU-Auslastung im Verhältnis zu Nachrichten pro Sekunde

Bei *Apache Kafka* verläuft die Kurve erneut weitestgehend linear im Verhältnis zu der steigenden Anzahl an Nachrichten pro Sekunde. Da auch hier keine Engpässe zu erkennen sind und die CPU als limitierender Faktor betrachtet wird, lässt sich, ähnlich wie bei den Producer-Tests, die Belastung auf das Maximum der CPU-Auslastung prognostizieren. Dies würde innerhalb der hier verwendeten Testinfrastruktur in etwa 6500 Nachrichten pro Sekunde entsprechen. Ziel ist es im Rahmen der anschließenden Consumer-basierten Tests, diese These durch eine erhöhte Auslastung und gesteigerte IO-Operationen zu belegen.

5.2.2 Consumer-basierte Testergebnisse (T.5a und T.5b)

Im Consumer-basierten Testszenario wurde, ähnlich wie bei den Producer-basierten Tests, systematisch die Menge an Consumern erhöht und dementsprechend das Verhalten des Brokers analysiert.

5.2.2.1 Anzahl Nachrichten

Im Hinblick auf die Menge an auslieferbaren Nachrichten zeigt sich, dass bei *ActiveMQ* die Nachrichtenmenge bis zu einer Anzahl von 4 Consumern konstant ansteigt und anschließend

bei fast 4000 Nachrichten ihr Maximum erreicht. Anschließend lässt sich die Menge an auslieferbaren Nachrichten pro Sekunde nicht mehr steigern und es kann damit an dieser Stelle erneut angenommen werden, dass ein Ressourcenengpass vorliegt.

Im Fall von *Apache Kafka* zeigt sich ebenfalls bis zu 4 Consumern ein konstanter Anstieg, bei dem die Kurve jedoch im Anschluss leicht abflacht. In dem in Abbildung 51 dargestellten Diagramm beträgt die maximale Nachrichtenmenge 10.000 pro Sekunde, die mit 6 Consumern erreicht wird. Anschließend konnte die Anzahl an Consumern aufgrund der zu Beginn angesprochenen Handhabbarkeit nicht weiter gesteigert werden. Unter Einbeziehung der CPU-Auslastung ist zu erwarten, dass sich dieser Wert noch geringfügig auf maximal 12.000 Nachrichten pro Sekunde steigern ließe.

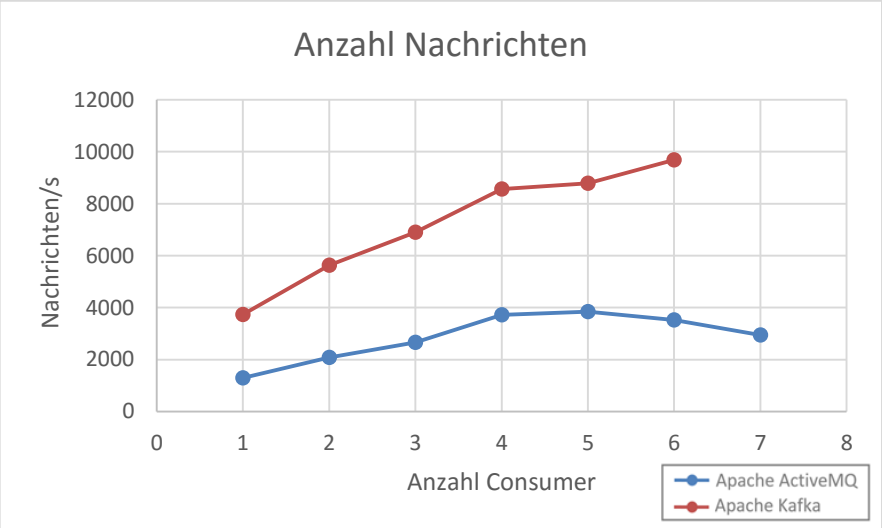


Abbildung 51: Anzahl Nachrichten im Verhältnis zur Anzahl an Consumern

5.2.2.2 IO-Operationen

Die Darstellung der zugehörigen IO-Operationen in Abbildung 52 zeigt in Bezug auf *ActiveMQ* ebenfalls, dass diese ab einer Menge von 4 Consumern und knapp 4.000 Nachrichten/s stagnieren, obwohl eine erhöhte Auslieferungslast durch die Consumer forciert wird. Das Resultat dieses Maximums drückt sich grundsätzlich in einem erhöhten zusätzlichen Zeitbedarf aus, wie dieser im Testszenario in Abschnitt 5.1.1 zu erkennen war.

Die Kurve von *Apache Kafka* beschreibt einen etwas unbeständigen Anstieg auf insgesamt 120.000 IO-Operationen pro Sekunde, die sich im Anschluss durch die Hinzunahme eines 6. Consumers nicht weiter steigern lassen.

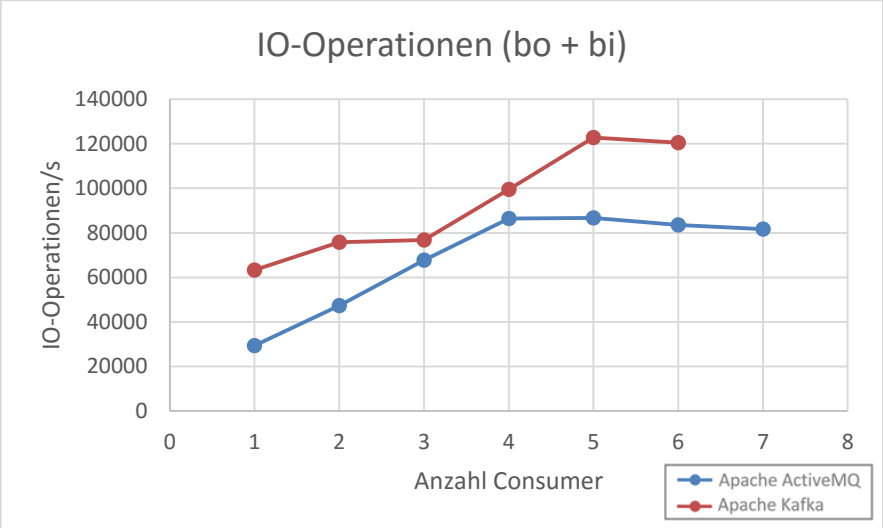


Abbildung 52: IO-Operationen im Verhältnis zur Anzahl an Consumern

5.2.2.3 CPU-Auslastung

Zuletzt ist in der Darstellung der CPU-Auslastung in Abbildung 53 ersichtlich, dass diese im Fall von *ActiveMQ* erwartungsgemäß ab dem 4. Consumer stagniert. Der erreichte Wert lässt sich im Anschluss nicht über eine CPU-Auslastung von 75% steigern. Dieses Verhalten macht erneut deutlich, dass noch weitere Rechenleistung verfügbar wäre, diese jedoch nicht ausgenutzt werden kann, da der Broker die dazu notwendigen IO-Operationen nicht ausführen kann.

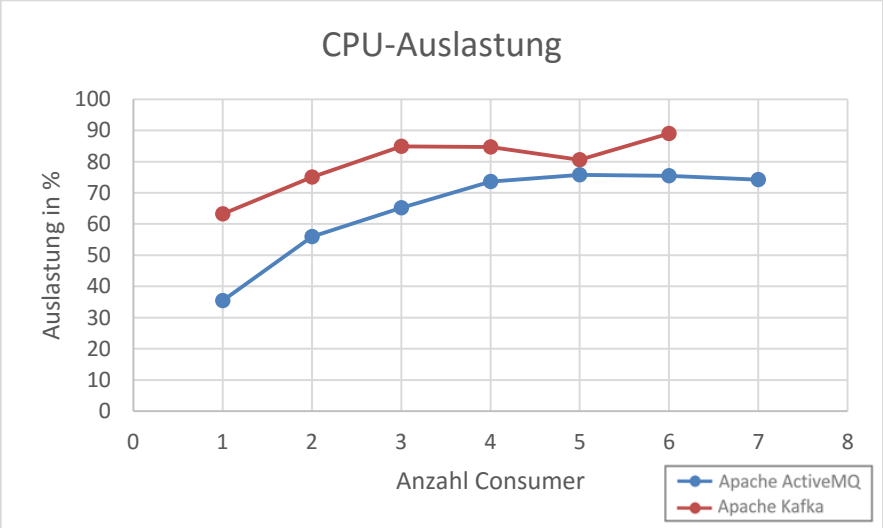


Abbildung 53: CPU-Auslastung im Verhältnis zur Anzahl an Consumern

Die Kurve von *Apache Kafka* zeigt einen Anstieg auf knapp 85% der CPU-Kapazität. Anschließend stagniert diese und lässt sich nur noch geringfügig erhöhen. Dennoch ist zu erwarten, dass auch hier durch eine entsprechende Steigerung und weitere Optimierungen die CPU komplett ausgelastet werden kann.

5.2.2.4 Datenbasierte Testergebnisse

Aufgrund der zu Beginn dieses Kapitels angesprochenen unzureichenden Handhabbarkeit der *Kafka*-Consumer konnten keine repräsentativen Tests hinsichtlich der maximalen Datenmenge durchgeführt werden.

Die vorherigen Testergebnisse zeigen, dass ein *Kafka*-Broker lediglich halb so viele Nachrichten ausliefern kann, wie er in der Lage ist entgegenzunehmen. Aus diesem Sachverhalt lässt sich schließen, dass erwartungsgemäß auch nur die halbe Datenmenge im Verhältnis zu den Daten-basierten Producer-Tests aus Abschnitt 5.1.3 erreichbar ist. Dies entspräche somit in etwa einer Datenmenge von 150 – 200 MB pro Sekunde.

Im Fall von *ActiveMQ* liegt die maximal aufnehmbare Datenmenge bei 150-200 MB innerhalb der Daten-basierten Producer-Tests. Da die Anzahl an auslieferbaren Nachrichten mit 4000 um 20% geringer ist als die empfangbare Rate von 5000, lässt sich auch hier prognostizieren, dass ein *ActiveMQ*-Broker eine um etwa 20% geringere Datenmenge ausliefern kann. Dies entspräche einem Wert von 120 – 160 MB.

5.2.3 Zusammenfassende Testergebnisse

Zusammengefasst zeigt sich innerhalb der Consumer-seitigen Tests, wie auch bei den Producer-Tests, dass im Fall von *Apache Kafka* eine deutlich höhere Nachrichtenmenge ausgeliefert werden kann. Diese ist mit knapp 10.000 Nachrichten/s mehr als doppelt so groß wie die von *ActiveMQ* (circa 4.000 Nachrichten/s).

Des Weiteren fällt jedoch bei *Apache Kafka* auf, dass nur etwa halb so viele Nachrichten konsumiert werden können, wie das System in der Lage ist aufzunehmen. Im Hinblick auf eine *Ende-zu-Ende* Betrachtung vom Nachrichtenversand beim Sender und dem Eingang beim Empfänger orientiert sich der Gesamtdurchsatz dabei an dem kleineren Durchsatz bei der Nachrichtenauslieferung. Damit bildet die Auslieferungskapazität im Rahmen des Gesamtdurchsatzes mit 10.000 Nachrichten ein Bottleneck.

Hier ist erneut anzumerken, dass die Kafka-Consumer deutlich komplexer sind und daher mit höherer Sorgfalt konzipiert werden müssen. Sofern dies nicht erfolgt, treten erkennbare Seiteneffekte auf, die häufige Rebalances hervorrufen und damit den Durchsatz des Systems einschränken.

5.3 Einfluss des Betriebssystem-Cache

In diesem Abschnitt wird auf den Betriebssystem-Cache, der maßgeblichen Einfluss auf die Performance des Brokers hat, Bezug genommen. Insbesondere bei den Consumer-Tests führt der Broker eine signifikante Menge an Lese-Operationen auf die Festplatte aus. Sofern das Betriebssystem den IO-Cache mit einbezieht, wird ein Großteil der Daten als Kopie im flüchtigen

Arbeitsspeicher abgelegt. Dies führt dazu, dass die Nachrichten nicht erst von der Festplatte geladen werden müssen, sondern direkt zur Auslieferung bereitstehen. Der Broker produziert folglich nicht die sonst benötigten Leseoperationen.

Abbildung 54 stellt das IO-Verhalten eines *Kafka*-Brokers sowohl mit als auch ohne Einbeziehung des Betriebssystem-Cache dar. Es zeigt sich, dass die IO-Operationen ohne die Verwendung des Caches (rot) konstant mit der Menge an Nachrichten/s ansteigen. Im Gegensatz dazu beschreibt die blaue Kurve das Brokerverhalten unter Einbeziehung des Betriebssystem-Cache. Hier führt der Broker keine bzw. nur geringfügige IO-Operationen durch.

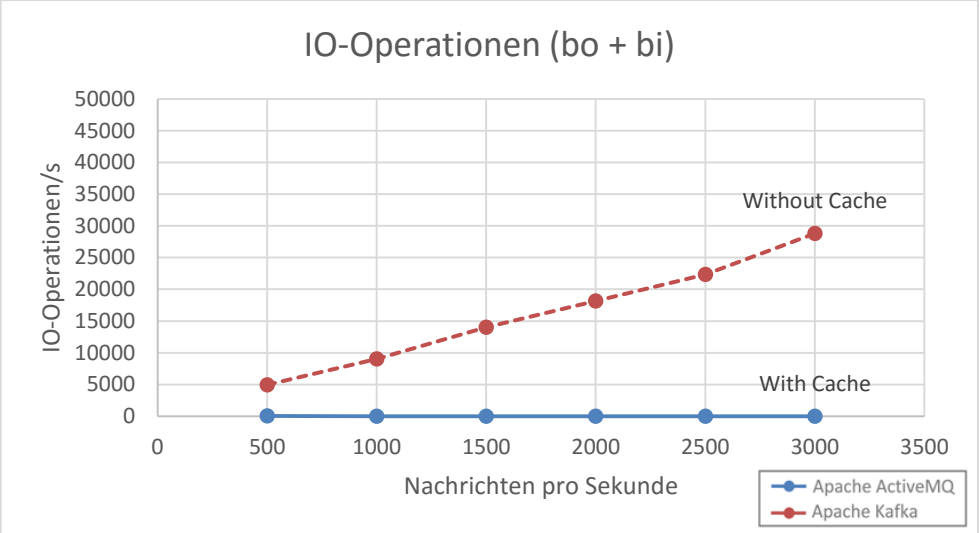


Abbildung 54: Broker-Verhalten mit und ohne Betriebssystem Cache (IO-Operationen)

Als unmittelbare Folge zeigt sich in Abbildung 55, dass der CPU-Bedarf ohne die Nutzung des Caches deutlich höher ist als unter dessen Einbeziehung. An dieser Stelle ist der CPU-Bedarf mit 45% um knapp 30% höher als unter Verwendung des Caches.

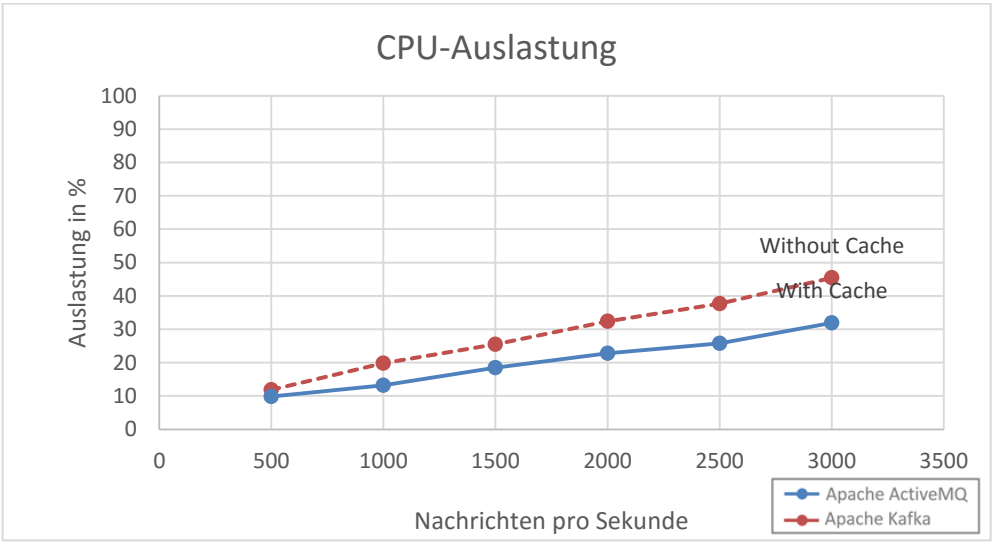


Abbildung 55: Broker-Verhalten mit und ohne Betriebssystem-Cache (CPU-Auslastung)

Diese Erkenntnis verdeutlicht, dass der Betriebssystem-Cache eine positive Auswirkung auf den Durchsatz und die Performance des Brokers hat. Eine entsprechende Hochrechnung der CPU-Werte auf 100% prognostiziert, dass der Broker in der Lage ist, unter Einbeziehung des Caches, knapp 10.000 Nachrichten pro Sekunde auszuliefern. Sofern der Cache jedoch nicht verwendet wird, liegt maximale Nachrichtenmenge lediglich bei etwa 6500 Nachrichten/s.

Auch wenn die Einbeziehung des Caches grundsätzlich einen positiven Effekt hervorruft, wurde in den hier beschriebenen Tests darauf Wert gelegt, von möglichst realitätsnahen Bedingungen auszugehen, welche nicht von den Eigenschaften des Betriebssystems abhängig sind. Das damit ermittelte Performance-Minimum kann in einem späteren produktiven Einsatz gegebenenfalls durch Betriebssystem-Optimierungen begünstigt werden, es verschlechtert sich jedoch nicht.

In dem in Abbildung 54 und Abbildung 55 dargestellten Fall führt dies zu der Erkenntnis, dass bei *Apache Kafka* der Consumer-Durchsatz durch die Verwendung des IO-Caches erwartungsgemäß um 30% gesteigert werden könnte.

Bei *ActiveMQ* könnte sich dieser Effekt noch deutlicher bemerkbar machen, da die IO-Operationen in diesem System als limitierender Faktor betrachtet werden.

6 Herausforderungen (Bezug auf Anforderungen)

Dieses Kapitel befasst sich mit Herausforderungen bzw. individuellen Gegebenheiten, mit denen sich verschiedene Unternehmen oder Projekte im Rahmen des Messagings konfrontiert sehen. Im Konkreten handelt es sich bei diesen Herausforderungen um nicht-funktionale Anforderungen.

Im Rahmen dieses Kapitels werden die einzelnen relevanten nicht-funktionalen Anforderungen in Relation mit *Apache Kafka* sowie *ActiveMQ* (als Repräsentant von JMS-konformen Systemen) gestellt. Hierzu werden im Abschnitt 6.1 zwei Beispiel-Cluster beschrieben, die im Folgenden als Veranschaulichung der Betrachtungsweisen dienen sollen. Darauffolgend werden in den weiteren Abschnitten die einzelnen nicht-funktionalen Anforderungen, beginnend mit der Skalierbarkeit, betrachtet.

6.1 Clusteraufbau

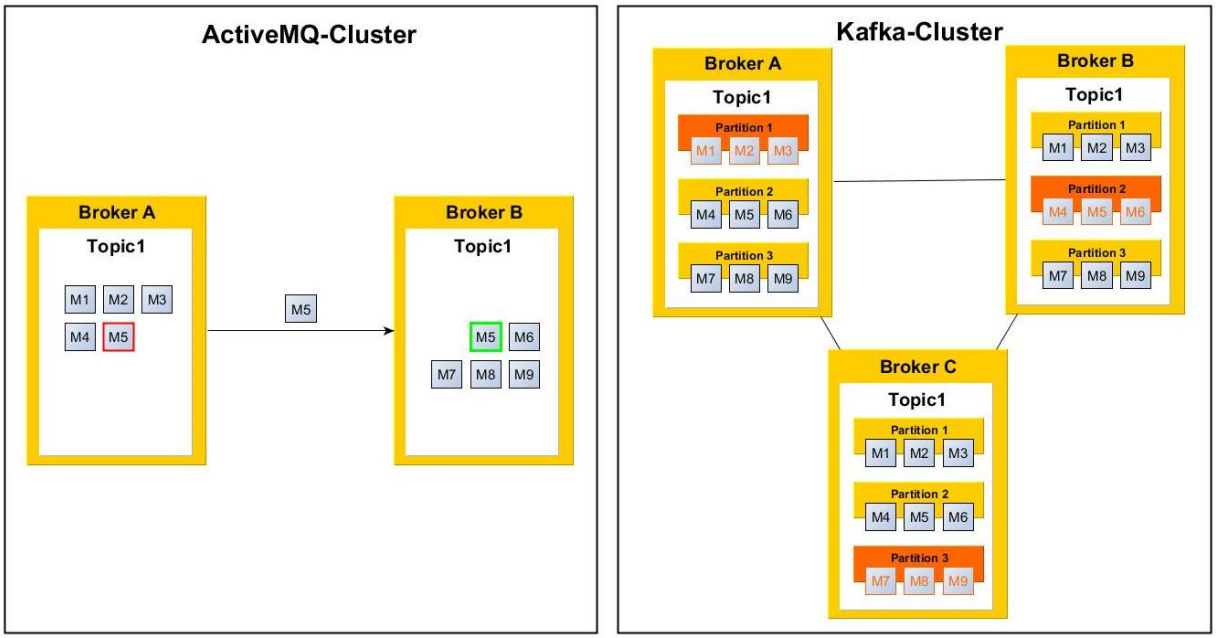


Abbildung 56: Clusterstrukturen von Apache ActiveMQ und Apache Kafka

Abbildung 56 zeigt einen beispielhaften Aufbau und die damit verbundenen systemspezifischen Eigenschaften der beiden in dieser Arbeit betrachteten Systeme. Das linke Netzwerk innerhalb der Abbildung zeigt ein *ActiveMQ*-Cluster (*Network-of-Brokers*), in dem zwei Broker aktiv sind, die über eine sogenannte *Demand Forwarding Bridge* kommunizieren.

Essenziell für die Gegenüberstellung beider Systeme ist an dieser Stelle, dass in einem *ActiveMQ*-Cluster – wie bereits erwähnt – immer nur eine Instanz der jeweiligen Nachricht vorliegt. Diese eine Kopie wird je nach Bedarf zwischen den Brokern ausgetauscht. Hierzu zeigen die Broker im linken Netzwerk den Austausch der Nachricht *M5*. Vor der Übertragung enthält

der *Broker A* die Nachrichten *M1 – M5* und der *Broker B* die Nachrichten *M6 – M9*. Im Rahmen der Übertragung sichert *Broker B* die neue Nachricht in seinem *Speicher (grün)*, was dazu führt, dass *Broker A* im Anschluss die Nachricht *M5* aus seinem Speicher (rot) entfernt.

Im rechten Cluster von Abbildung 56 liegen die Nachrichten zu jedem Topic vollständig, in Partitionen unterteilt, auf jedem Broker vor. Diese Partitionen ermöglichen eine Auslieferung der Daten über mehrere Broker hinweg. Die rot markierten Partitionen geben an, welcher Broker als Leader für diese Partition verantwortlich ist und damit als einziger Daten für diese entgegennehmen und ausliefern darf. Im konkreten Fall würde dem ersten Consumer die Partition 1 auf *Broker A* zugewiesen werden, dem 2. Consumer die Partition 2 auf *Broker B* und Consumer 3 die Partition 3 auf *Broker C*.

6.2 Skalierbarkeit

Die nicht-funktionale Anforderung *Skalierbarkeit* beschreibt, wie in einem System die Leistung durch das Hinzufügen von neuen Ressourcen weiter gesteigert und damit das System einer höheren Last gerecht werden kann.

Bei *ActiveMQ* wird, wie bereits erläutert, zwischen den Ansätzen *Network-of-Broker* und (*Shared Storage) Master/Slave* unterschieden. Da bei dem *Master/Slave*-Ansatz eine Ausfallsicherheit im Vordergrund steht, eignet sich dieser nicht für eine Skalierung. Dieser Aufbau kann lediglich durch eine vertikale Skalierung erweitert werden. Die Testdurchläufe in Kapitel 5 haben gezeigt, dass die Leistung von *ActiveMQ* durch die IO-Operationen limitiert wird. Dies hat zur Folge, dass eine vertikale Skalierung im Allgemeinen schwerer zu realisieren ist als in anderen System. Eine vertikale Skalierung beschreibt das Hinzufügen neuer Hardware, um einer höheren Last gerecht zu werden. Eine horizontale Skalierung hingegen verfolgt den Ansatz, mehrere Instanzen parallel zu betreiben, um die Last gleichmäßig auf diese verteilen zu können.

Im Gegensatz zum *Master/Slave*-Ansatz lässt das *Network-of-Broker* eine horizontale Skalierung zu. Hierdurch sind mehrere aktive Broker in der Lage, Nachrichten von Clients entgegenzunehmen und auszuliefern. Im produktiven Einsatz ist die verwendete Anzahl an Brokern üblicherweise nicht größer als zwei. Das liegt daran, dass bereits ab einer verhältnismäßig geringen Clustergröße die Nachrichten sehr ineffizient zwischen den Brokern hin und her gereicht werden, bevor letztendlich die Auslieferung an den Client erfolgt.

Apache Kafka eignet sich sowohl für eine vertikale als auch für eine horizontale Skalierung. Im Fall der vertikalen Skalierung hat *Apache Kafka* eine deutlich bessere Ausgangslage, da dieses System anhand der CPU skaliert und damit wesentlich leichter um diese Ressource erweitert werden kann.

Da *Apache Kafka* von Beginn an unter Aspekten des Clusterings konzipiert wurde, liegen die Stärken innerhalb der horizontalen Skalierung ebenfalls bei diesem System. Da das komplette

Topic innerhalb eines Clusters auf alle Broker verteilt wird, die jeweils nur einen Teil davon (bestimmte Partitionen) bereitstellen, bildet sich eine sehr ausgeglichene Lastverteilung auf alle beteiligten Broker.

Darüber hinaus lässt sich ein Kafka-Cluster mit sehr geringem Aufwand um einen weiteren Broker erweitern. Dieser neue Broker muss lediglich mit einer eindeutigen ID versehen und anschließend gestartet werden.

Zusammenfassend ist im Hinblick auf die Skalierbarkeit das Fazit zu ziehen, dass in beiden Systemen die Skalierbarkeit gegeben ist, *Apache Kafka* durch sein direktes Clustering-Konzept jedoch deutliche Vorteile in diesem Bereich aufweist.

6.3 Verfügbarkeit

Bei der nicht-funktionalen Anforderung *Verfügbarkeit* handelt es sich um einen Maßstab dafür, ob ein System zu einem bestimmten Zeitpunkt oder über einen bestimmten Zeitraum hinweg verfügbar ist [Kle13]. Im Rahmen einer Hochverfügbarkeit wird angestrebt, dass ein System möglichst immer verfügbar ist. Hierbei ist eine Verfügbarkeit von 99% in kritischen Systemen häufig bereits nicht mehr tragbar, da diese auf ein Jahr gerechnet schon mehrere Tage Ausfall zur Folge hätte.

An dieser Stelle muss die Unterscheidung in die Verfügbarkeit des Systems und die Verfügbarkeit der darin enthaltenen Daten getroffen werden. Die *Systemverfügbarkeit* hat dabei grundsätzlich eine höhere Bedeutung für die Producer, da diese Daten in das System senden. Die *Datenverfügbarkeit* hingegen ist eher für die Consumer relevant, da diese zu jedem Zeitpunkt auf die Daten im System zugreifen möchten.

Im Fall von *ActiveMQ* bietet der *(Shared Storage) Master/Slave*-Ansatz sowohl eine hohe Systemverfügbarkeit als auch eine hohe Datenverfügbarkeit. Das liegt zum einen daran, dass *n-1* Broker ausfallen können. Zum anderen greifen alle Broker auf den gleichen gemeinsam genutzten Datenbestand zurück und können damit zu jedem Zeitpunkt vollständig auf diesen zugreifen. Das setzt jedoch voraus, dass auch dieser gemeinsam genutzte Speicher durch entsprechende Maßnahmen hochverfügbar gehalten wird.

Diese Situation gestaltet sich im Hinblick auf den *Network-of-Brokers*-Ansatz anders. In diesem Zusammenhang ist ebenfalls eine Systemverfügbarkeit bis zu einem Ausfall von *n-1* Brokern gewährleistet. Da jedoch von jeder Nachricht– wie in dem linken Cluster in Abbildung 56 ersichtlich – jeweils nur ein Exemplar im System vorhanden ist, sind bei einem Serverausfall die darauf enthaltenen Daten innerhalb der Ausfallzeit nicht verfügbar. Durch eine entsprechende Kombination mit dem *Master/Slave*-Ansatz kann dieses Verfügbarkeitsdefizit ausgeglichen werden.

Im Gegensatz dazu ist bei *Apache Kafka* aufgrund der redundanten Speicherung aller Daten, sowohl die Systemverfügbarkeit als auch die Datenverfügbarkeit, in vollem Maße bis zu einem Ausfall von *n-1* Brokern gegeben. Die Menge *n-1* richtet sich dabei nicht nach der Gesamtanzahl an Brokern im Cluster, sondern nach dem *ReplicationFactor* des jeweiligen Topics. Sofern durch einen *ReplicationFactor* von 3 die Daten auf jeweils 3 Brokern vorliegen, toleriert das System einen Ausfall von 2 Brokern, bevor es zu einer Einschränkung der Datenverfügbarkeit kommt.

Im Zusammenhang mit der nicht-funktionalen Anforderung *Verfügbarkeit* ist in beiden Systemen grundsätzlich eine Hochverfügbarkeit zu erreichen. Allerdings ist im Fall von *ActiveMQ* ein sehr hoher Aufwand für das gleiche Maß an *Verfügbarkeit* notwendig. Bei *Apache Kafka* hingegen ist die System-und Datenverfügbarkeit direkt enthalten und muss lediglich über einen Einstellungsparameter definiert werden.

6.4 Zuverlässigkeit

Die nicht-funktionale Anforderung *Zuverlässigkeit* orientiert sich im Fall des Messagings an den *Quality-of-Service*-Klassen *At-most-Once*, *At-Least-Once* sowie *Exactly-Once*. Diese Klassen geben an, wie zuverlässig bzw. genau das System bei der Übertragung von Nachrichten vorgeht. Die Definition dieser *QoS*-Klassen kann Kapitel 2 entnommen werden.

Im Rahmen der *Zuverlässigkeit* zeigt sich bei *ActiveMQ* die große Stärke des JMS-Standards. Dieser legt verbindlich fest, wie die Nachrichten übertragen werden müssen, um einer bestimmten *QoS*-Klasse gerecht zu werden. Durch den Einsatz von *persistent*-Messaging und anderen Konfigurationsparametern ist damit die höchste der hier betrachteten Zuverlässigkeitsklassen ohne größeren Aufwand zu erreichen.

Im Fall von *Apache Kafka* kann ohne größeren Aufwand die *QoS*-Klasse *At-Least-Once* erzielt werden. Dieses Verhalten, welches keinen Datenverlust toleriert, jedoch den doppelten Nachrichtenempfang akzeptiert, wird im Wesentlichen durch den *ReplicationFactor* sowie die Einstellung *ACKS_CONFIG = all* realisiert. Detaillierte Informationen hierzu können Kapitel 3 entnommen werden.

Diese Einstellung bietet in der aktuellen Kafka Version *0.10.1.0* die höchste Form der Zuverlässigkeit im Zusammenhang mit der Nachrichtenübertragung. Durch diese Konfiguration wird zwar sichergestellt, dass Nachrichten mindestens einmal ihr Ziel erreichen, es kann jedoch nicht ausgeschlossen werden, dass sie gegebenenfalls dupliziert werden. Die Erzeugung von Duplikaten kann durch verschiedene Ereignisse hervorgerufen werden. Das wahrscheinlichste Ereignis besteht darin, dass einem Consumer die aktuelle Partition entzogen wird, er jedoch seinen derzeitigen Verarbeitungsstand (*Offset*) nicht mehr sichern kann.

Abschließend lässt sich festhalten, dass im Bereich der Zuverlässigkeit, aufgrund des Standards und der einheitlichen und bewährten Verfahren, die Stärken bei den klassischen JMS-konformen Systemen liegen. Diese bieten mit den passenden Einstellungen, ohne größeren Aufwand, eine verlässliche *Exactly-Once* Übertragung. Das reine Produkt *Apache Kafka* hingegen bietet dieses Niveau an Zuverlässigkeit jedoch nicht. Sofern diese Eigenschaft im *Kafka*-Kontext gefordert wird, muss diese entsprechend clientseitig, beispielsweise durch die Verwendung von Sequenznummern, nachgebildet werden.

6.5 Robustheit

Die nicht-funktionale Anforderung *Robustheit* befasst sich mit der Fähigkeit des Systems, mit außerplanmäßigen Situationen umgehen zu können und demzufolge eine nachvollziehbare Maßnahme durchzuführen [Kle13]. Ein Programm, bei dem eine entsprechende Robustheit vorliegt, ist auch unter suboptimalen Gegebenheiten in der Lage, korrekt zu arbeiten und seine Produktivität weiter aufrechtzuerhalten.

Beide hier betrachteten Systeme sind in ausreichendem Maße robust bzw. lassen sich durch entsprechende Einstellungen in diesen Zustand bringen.

Insbesondere bei *Apache Kafka* haben die Konfigurationsparameter einen maßgeblichen Einfluss auf die Robustheit des Systems. Ein Beispiel hierfür ist die *Unclean Leader Election*.

In Abbildung 56 ist der *Broker A* Leader für die Partition 1. Sofern dieser nun ausfällt, kann im Zuge einer *Election* (Wahl) eines der *insync*-Replicas zum Leader gewählt werden. Sollte zu diesem Zeitpunkt jedoch, beispielsweise aufgrund von Netzwerkschwierigkeiten, kein *insync*-Replica vorhanden sein, steht das System vor einem Problem. Um die Robustheit zu erhöhen, kann in diesem Fall die *Unclean Leader Election* aktiviert werden. Da diese jedoch einen Datenverlust zur Folge haben kann, hat diese Einstellung maßgeblichen Einfluss auf die im vorherigen Abschnitt betrachtete *Zuverlässigkeit*.

Ein weiterer Aspekt der *Robustheit* von *Apache Kafka* ist, wie die Consumer damit umgehen, wenn der zuletzt gespeicherte Verarbeitungsstand (*Offset*) nicht mehr vorhanden ist. An dieser Stelle kann durch den Konfigurationsparameter *auto.offset.reset* zwischen den Optionen *latest*, *earliest* und *none* gewählt werden. Im Hinblick auf ein robustes System sollte hierbei auf die Einstellung *none* verzichtet werden. Da jedoch *latest* zu einem Datenverlust führen kann, ist *earliest* die geeignetere Wahl im Sinne der *Zuverlässigkeit*. Weitere Informationen zum Verhalten dieses Konfigurationsparameters können Kapitel 3 entnommen werden.

Im Allgemeinen ist *ActiveMQ* mit den Basiseinstellungen deutlich robuster als *Apache Kafka.* Jedoch können auch in diesem System Einstellungen vorgenommen werden, welche die Robustheit steigern. Dazu gehört beispielsweise der Umgang mit einer vollen *Queue*. An dieser

Stelle kann entweder eine *Exception* geworfen oder der Producer so lange verzögert werden, bis wieder genügend Platz verfügbar ist.

Grundsätzlich kann in beiden Systemen eine angemessene Robustheit erreicht werden, allerdings haben bei *Apache Kafka* viele Parameter direkte und massive Auswirkungen auf andere nicht-funktionale Anforderungen wie zum Beispiel die *Zuverlässigkeit*.

6.6 Wartbarkeit und Änderbarkeit

Die nicht-funktionale Anforderung *Wartbarkeit* beschreibt die Eigenschaft eines Systems, bestimmte Aktionen durchführen zu können, ohne dass dieses komplett ausgeschaltet werden muss. Beispiele für derartige Aktionen sind das Einspielen eines Server-Patches oder der Austausch einer Komponente innerhalb des Produktiv-Betriebs. Hierbei wird angestrebt, diese Aktionen auszuführen, ohne dass das Gesamtsystem davon beeinträchtigt wird. Die nicht-funktionale Anforderung *Änderbarkeit* gibt an, wie flexibel ein System im Hinblick auf Änderungen der Struktur und Anpassung der Konfiguration ist.

Im Fall von *ActiveMQ* ist die Wartbarkeit im Allgemeinen kritischer zu betrachten als bei *Apache Kafka*. Dies liegt daran, dass bei *ActiveMQ* nicht ohne weiteres ein Broker im Cluster ausgeschaltet werden kann. In diesem Fall ist während der *Downtime* die Datenverfügbarkeit grundsätzlich nicht mehr gewährleistet. Darüber hinaus müssen für einen dauerhaften Austausch die enthaltenen Daten auf den neuen Broker übertragen werden.

Im Fall von *Apache Kafka* kommt es nicht zu diesen Problemen, da die Daten redundant auf mehreren Brokern vorliegen und somit *n-1* Broker abgeschaltet werden können, ohne dass es zu Beeinträchtigungen kommt. Darüber hinaus kann das Cluster im Hinblick auf die Änderbarkeit ohne großen Aufwand um einen neuen Broker erweitert werden.

6.7 Portierbarkeit

Die nicht-funktionale Anforderung *Portierbarkeit* beschreibt, wie gut sich das System bzw. ein Teilstück daraus (beispielsweise ein Broker aus dem Cluster) auf einer anderen Plattform ausführen lässt und welche Maßnahmen hierfür erforderlich sind [Kle13].

In diesem Zusammenhang bieten beide Systeme, durch den Einsatz von Java bzw. der JVM, gute Portierungseigenschaften. Darüber hinaus stellen beide Plattformen *Scripte* für mehrere Betriebssysteme zur Verfügung.

6.8 Performance

Die *Performance* beider Systeme wurde im Kapitel 5 eingehend betrachtet. Hier hat sich herausgestellt, dass beide Systeme unter ähnlichen *Quality-of-Service* Eigenschaften deutliche Performance-Unterschiede aufweisen.

Apache Kafka ist mit 20.000 Nachrichten pro Sekunde in der Lage eine viermal höhere Nachrichtenmenge aufzunehmen als *ActiveMQ*. Bei der Datenauslieferung erzielt *Apache Kafka* mit knapp 10.000 Nachrichten in etwa doppelt so viele wie *ActiveMQ*.

In einer entsprechenden *Ende-zu-Ende* Betrachtung können damit innerhalb derselben Zeitspanne in etwa doppelt so viele Nachrichten von den Producern zu den Consumern transportiert werden. Sofern der Fokus auf einer schnellen Speicherung der Daten und einer sporadischen Konsumierung liegt, kann *Apache Kafka* in der gleichen Zeit sogar die vierfache Menge an Daten aufnehmen.

Sofern grundsätzlich auf den Einsatz von *persistent*-Messaging verzichtet und damit ein Nachrichtenverlust toleriert werden kann, steigert sich die Performance von *ActiveMQ* deutlich.

6.9 Interoperabilität

Die nicht-funktionale Anforderung *Interoperabilität* beschreibt, wie gut ein Systemen mit anderen Systemen zusammenarbeiten kann bzw. was es für Software, Protokolle und Features anbietet, um von einer breiten Masse an Betriebssystemen, Software und Protokollen genutzt zu werden.

ActiveMQ bietet für eine breitflächige Nutzung APIs für Java, C++, C# sowie für das Messaging im Web eine Rest-Api und eine Ajax an [SBD11]. *Apache Kafka* bietet durch beispielsweise Java, C++, C#, Ruby, Python, GO, NodeJS, Scala ein ähnliches Portfolio.

Im Hinblick auf die Speicherung der Daten bietet *ActiveMQ* eine Vielzahl an verschiedenen *Stores*. An dieser Stelle kann unter anderem zwischen dem *AMQ Message Store*, dem *KahaDB Message Store* sowie dem *JDBC Message Store* gewählt werden. Im Gegensatz dazu besteht bei *Apache Kafka* derzeit nicht die Möglichkeit, eine andere Store-Variante zu wählen.

ActiveMQ bietet für die Kommunikation zwischen den Clients und dem Broker, sowie zwischen den Brokern untereinander, verschiedene Protokolle an. Hierzu zählen TCP, STOMP, NIO, UDP, SSL und HTTP/HTTPS. *Apache Kafka* bietet in diesem Zusammenhang keine direkte Auswahl des zu verwendenden Netzwerkprotokolls und verwendet hierbei grundsätzlich TCP für die Kommunikation.

Eine Interoperabilität zwischen den beiden hier betrachteten Systemen untereinander ist zum jetzigen Zeitpunkt nicht direkt gegeben. Es können lediglich eigenständig Verknüpfungspunkte entwickelt werden, welche die Nachrichten als Consumer aus einem der Systeme entgegennehmen und diese direkt in das andere System speisen.

Aufgrund dessen, dass es sich bei *ActiveMQ* um ein etabliertes System handelt, dass großflächig zum Einsatz kommt, gewährt es im Allgemeinen eine deutlich bessere Interoperabilität. Da es

sich bei jedoch *Apache Kafka* um ein sehr junges Projekt handelt, sind in Zukunft weitere Optimierungen diesbezüglich zu erwarten. Darüber hinaus haben die Initiatoren Jay Kreps, Neha Narkhede und Jun Rao das Unternehmen *Confluent* [@cfl] gegründet, innerhalb dessen sie durch die *Confluent Platform* kommerzielle Erweiterungen anbieten, die weitere Möglichkeiten hinsichtlich der Interoperabilität eröffnen.

6.10 Gesamtbetrachtung

In diesem Abschnitt werden die vorherigen Einzelbetrachtungen im Hinblick darauf, wie gut sich die zuvor thematisierten nicht-funktionalen Anforderungen durch die beiden Systeme abbilden lassen, in einen Gesamtkontext gesetzt.

		Apache ActiveMQ (JMS)		Apache Kafka
		Network-of-Broker	Shared Storage Master/Slave	
Skalierbarkeit				
Verfügbarkeit	System			
	Daten			
Zuverlässigkeit (QoS)				
Robustheit				
Wartbarkeit & Änderbarkeit				
Portierbarkeit				
Performance				
Interoperabilität				

Zufriedenstellend Eingeschränkt Unzureichend

Tabelle 4: Eignung hinsichtlich Nichtfunktionaler Anforderungen

Tabelle 4 stellt die Eignung beider Systeme hinsichtlich der nicht-funktionalen Anforderungen gegenüber. Neben der generellen Unterscheidung beider Systeme wird bei *ActiveMQ* zwischen den beiden Clustervarianten *Network-of-Broker* und *Master/Slave* unterschieden. Die mit grün markierten Felder geben an, dass innerhalb des Systems gute Voraussetzungen für die jeweiligen Anforderungen vorliegen. Orange markierte Felder weisen darauf hin, dass in irgendeiner Form Defizite vorliegen, manuell nachgebessert werden muss oder dass im anderen System eine bessere Ausgangslage vorliegt. Der Vollständigkeit halber geben rot markierte Felder an, dass sich eine bestimmte Anforderung nicht abbilden lässt. Diese Situation liegt jedoch bei keiner der betrachteten Anforderungen vor.

Hinsichtlich der Skalierbarkeit weisen beide Systeme ausreichende Fähigkeiten auf. Dennoch liegen die Vorteile aufgrund der in Abschnitt 6.2 beschriebenen Faktoren bei *Apache Kafka*.

Die Verfügbarkeit kann grundsätzlich in beiden Systemen gewährleistet werden. Bei *ActiveMQ* besteht jedoch die Problematik, dass zwar das System als solches verfügbar ist, es jedoch be-

stimmte Daten innerhalb des Systems vorrübergehend nicht sind. Um dieses Problem zu umgehen, müssen dementsprechend die produktiven Broker mit dem *Master/Slave*-Ansatz redundant ausgelegt werden. Im Allgemeinen bildet *Apache Kafka* diese nicht-funktionale Anforderung jedoch deutlich besser ab, da alle Broker am Produktivbetrieb teilnehmen, dabei jedoch auch über den kompletten redundanten Datenbestand verfügen. Dies macht das System weniger anfällig gegenüber Ausfällen.

Für die Anforderung *Zuverlässigkeit* liegen bei *ActiveMQ* grundsätzlich bessere Gegebenheiten vor, da sich alle Zuverlässigkeitsgarantien erzielen lassen. Sofern bei *Apache Kafka* die *Exactly-Once* Zustellgarantie erforderlich ist, lässt sich dies nur über zusätzliche Maßnahmen in den Clients erreichen.

Im Fall der *Robustheit* ist dies ähnlich, da *ActiveMQ* hier grundsätzlich bessere Eigenschaften als *Apache Kafka* bietet. Darüber hinaus resultieren aus diesen Maßnahmen, welche bei *Apache Kafka* ergriffen werden können, massive Nebeneffekte auf andere nicht-funktionale Anforderungen, wie die *Zuverlässigkeit*. Dies kann ohne entsprechendes Wissen zu einem Verlust der angestrebten Zuverlässigkeitsgarantie führen.

Abschließend bestehen weitere Defizite von *ActiveMQ* in der *Wart- und Änderbarkeit* sowie in der *Performance*. Hier kann, sofern *persistent*-Messaging zum Einsatz kommt, nur ein Bruchteil des Durchsatzes von *Apache Kafka* erreicht werden. *Apache Kafkas* abschließendes Defizit liegt in der schlechteren *Interoperabilität* im Vergleich zu *ActiveMQ*.

Die Frage, welches System sich für einen produktiven Einsatz besser eignet, lässt sich nicht pauschal beantworten. Hierbei steht zum einen die Priorisierung der nicht-funktionalen Anforderungen im Vordergrund. Darüber hinaus sollten weitere Aspekte betrachtet werden. Hierzu zählt beispielsweise, dass eine höhere Fachkompetenz auf Entwicklerseite vorliegen muss, da *Apache Kafka* deutlich komplexe Abläufe besitzt und damit nicht so gut handhabbar ist wie JMS. Ferner ist relevant, wie die im System enthaltenen Daten konsumiert werden sollen und für welche Anwendungsfälle die Verarbeitung gewünscht ist. Um eine Entscheidungsgrundlage für das passende System zu bieten bzw. um einen Umstieg auf *Apache Kafka* abzuwägen, wird im folgenden Kapitel ein Leitfaden vorgestellt, der als entsprechende Entscheidungsgrundlage dienen soll.

7 Resultierender Leitfaden für Unternehmen

In diesem Kapitel wird ein Leitfaden definiert, der die Erkenntnisse der vorherigen Kapitel strukturiert und zusätzlich jene Faktoren betrachtet, die nicht direkt auf nicht-funktionale Anforderungen projiziert werden können.

Da sich grundsätzlich alle nicht-funktionalen Anforderungen in beiden Systemen abbilden lassen, gibt es keine Argumente, die pauschal gegen eines der beiden Produkte sprechen oder es ausschließen. Lediglich im Zusammenhang mit der Interoperabilität könnte es gegebenenfalls Systeme geben, die sich zum jetzigen Zeitpunkt noch nicht für die Zusammenarbeit mit *Kafka* eignen.

Abgesehen davon können erwartungsgemäß eher äußere Faktoren, wie fehlendes Entwicklerknowhow oder ein mangelndes Budget, direkte Ausschlusskriterien sein.

Zu Beginn dieses Kapitels wird die visuelle Darstellung des Leitfadens aufgeführt und beschrieben (siehe Abschnitt 7.1). Im Anschluss wird der Aufbau des Leitfadens im Rahmen der darauffolgenden Abschnitte näher erläutert. Hierbei bilden die einzelnen Unterkapitel die jeweiligen Schritte, die während der Anforderungsanalyse betrachtet werden sollten.

7.1 Leitfaden

Der in Abbildung 57 dargestellte Leitfaden beinhaltet die, innerhalb einer Anforderungsanalyse, zu betrachtenden Schritte. Die rechteckigen Kästen beinhalten jeweils eine Frage zu einem der zu betrachtenden Aspekte. In Abhängigkeit der Beantwortung durch *Ja* oder *Nein* wird der aktuelle Kasten durch die Nutzung der zugehörigen Kante verlassen.

Durch entsprechendes Abfragen wird zwischen der linken *ActiveMQ*-Seite und der rechten *Kafka*-Seite gewechselt. Abschließend mündet der Leitfaden entweder in einer Empfehlung für Apache Kafka oder *ActiveMQ*.

Im Rahmen der folgenden Abschnitte wird der visuelle Leitfaden anhand der enthaltenen Schritte gezielt durchlaufen. Hier wird zuerst in Abschnitt 7.2 der Frage nachgegangen, ob eine Mehrfach- und/oder eine Offlineverarbeitung der Daten angestrebt wird. Anschließend geht Abschnitt 7.3 auf Performance-Aspekte ein. Der darauf folgende Abschnitt hinterfragt, ob ein besonderer Fokus auf den nicht-funktionalen Anforderungen Skalierbarkeit und Verfügbarkeit liegt. Sofern im Anschluss die aktuelle Tendenz gegen *Apache Kafka* spricht, ist zu klären, ob der von *Kafka* gebotene Mehrwert perspektivisch doch benötigt werden könnte.

Zuletzt wird, sofern die Entscheidung in Richtung *Apache Kaf*ka tendiert, in den Abschnitten 7.6, 7.7 und 7.8 hinterfragt, ob der Aufwand im Verhältnis zum Mehrwert steht, die Interoperabilität gewährleistet und die Umsetzung letztendlich möglich ist.

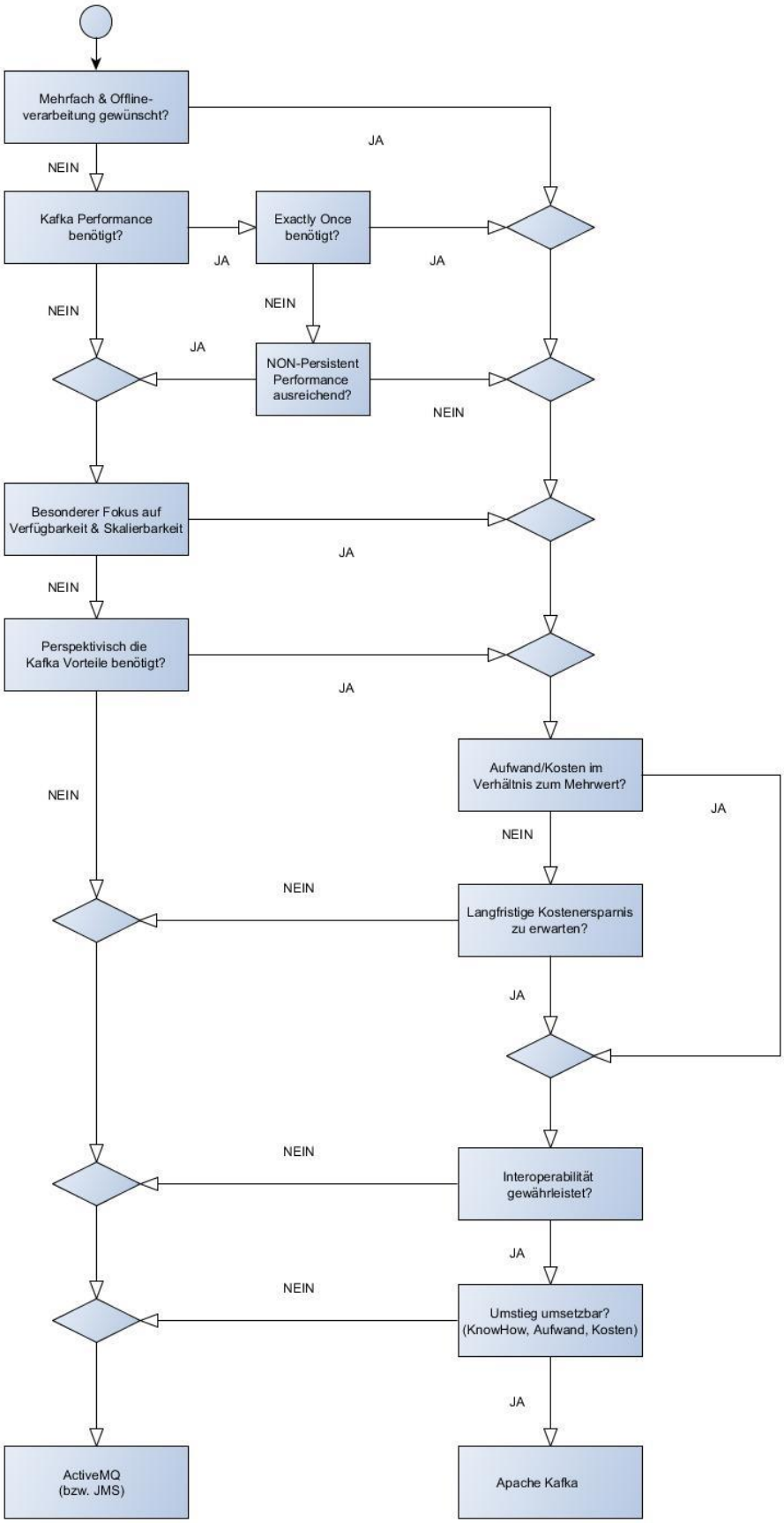


Abbildung 57: Leitfaden zum Einsatz von Apache Kafka

7.2 Mehrfach- und Offlineverarbeitung

Eines der wesentlichen Features von *Apache Kafka* besteht, wie bereits erwähnt, darin, dass Nachrichten für einen definierbaren Zeitraum dauerhaft auf der Festplatte gespeichert werden. Da die Consumer außerdem die Verwaltung des Verarbeitungsstands übernehmen, besteht die Möglichkeit im Datenbestand zu älteren oder auch zu neueren Stellen zu springen. Hierdurch können die Daten entsprechend mehrfach und zu beliebigen Zeitpunkten verarbeitet werden.

Im Fall von *ActiveMQ* erfolgt eine vernichtende Konsumierung der Daten. Dies bedeutet, dass die einzige Kopie der Nachricht an den verarbeitenden Consumer ausgeliefert und anschließend vom Broker aus dem Datenbestand entfernt wird. Dieses Vorgehen stellt im *ActiveMQ*-Kontext die Zustellgarantie *Exactly-Once* sicher.

Zusammenfassend bietet *Apache Kafka* damit eine deutlich bessere Ausgangslage, um Daten mehreren Anwendungen bzw. Anwendungsteilen zur Verfügung zu stellen. Durch diese Eigenschaften können zum Beispiel nachgelagerte Analysen durchgeführt werden, die keinen direkten Einfluss auf den Produktivbetrieb haben. Weitere Informationen zu diesem Mehrwert können Kapitel 3 entnommen werden.

Zu Beginn der Anforderungsanalyse sollte folglich überlegt werden, welche Datenmenge innerhalb der eigenen Anwendung produziert wird und auf welche Art und Weise diese Daten verarbeitet werden sollen.

Sofern die eigene Anwendung ein hohes Datenaufkommen produziert, für das eine Mehrfachverarbeitung angestrebt wird, spricht dies im ersten Schritt eher für den Einsatz von *Apache Kafka* als für die Verwendung von *ActiveMQ*. In diesem Fall wird in Abbildung 57 der erste Kasten, der die Mehrfach und Offline-Verarbeitung repräsentiert, über die *Ja*-Kante verlassen. Anschließend wird der Fragestellung, ob Mehrwerte wie Performance benötigt werden, nicht weiter nachgegangen, da durch den hier beschriebenen Punkt bereits ein wesentlicher Mehrwert identifiziert wurde.

Sofern der Bedarf an einer Mehrfachverarbeitung nicht gegeben ist, wird der Kasten entsprechend über die *Nein*-Kante verlassen. Im Anschluss an diese Entscheidung wird jedoch im Folgenden die Performance hinterfragt, da bisher kein Mehrwert für *Kafka* ermittelt werden konnte.

7.3 Performance

Falls im Rahmen von Abschnitt 7.2 kein Mehrwert für die Verwendung von *Apache Kafka* ermittelt werden konnte, gilt es im Anschluss zu klären, ob der benötigte Performance-Bedarf einen Einsatz von *Apache Kafka* rechtfertigt.

Nun muss zunächst festgestellt werden, welcher Datenmenge das System gerecht werden muss. Die durchgeführten Tests in Kapitel 5 haben gezeigt, dass *ActiveMQ* unter ähnlichen Bedingungen, hinsichtlich der Zustellgarantie mit 10.000 Nachrichten, in etwa halb so viele Nachrichten/s aufnehmen kann wie *Apache Kafka*. In Betrachtung der MB ist ein *ActiveMQ*-Broker mit 150 – 200 MB in der Lage, fast halb so viele Daten aufzunehmen wie ein *Kafka*-Broker.

Sofern die Frage, ob Performance-Werte in der Größenordnung von *Apache Kafka* benötigt werden im ersten Schritt mit *Ja* beantwortet wird, sollte das Augenmerk auf die angestrebte Zustellgarantie gelegt werden.

An dieser Stelle zeigt sich deutlich, wie sich die Eigenschaften hinsichtlich der Zustellgarantie und der Performance beider Systeme gegenseitig bedingen. Sofern bei *ActiveMQ* eine Zustellgarantie der Form *Exactly-Once* gefordert wird, schränkt dies massiv die Performance-Eigenschaften ein, da jede Nachricht auf die Festplatte geschrieben werden muss. Daraus lässt sich schlussfolgern, dass, sofern eine *Exactly-Once* Übertragung erforderlich ist, aber Performance-Werte oberhalb der Möglichkeiten von *ActiveMQ* benötigt werden, ein Mehrwert für einen Einsatz von *Apache Kafka* gegeben ist.

Sofern keine *Exactly-Once* Zustellgarantie benötigt wird, stellt sich die Situation anders dar. Das liegt daran, dass *ActiveMQ* in diesem Fall einen Nachrichtenverlust toleriert und damit die Nachrichten nicht mehr auf der Festplatte gespeichert werden müssen. Hieraus ergeben sich komplett andere Performance-Aspekte, wodurch *ActiveMQ* eine deutlich größere Nachrichtenmenge verarbeiten kann.

Sollte der hierdurch erzielte Performancegewinn von *ActiveMQ* für den beabsichtigten Einsatzzweck immer noch nicht ausreichend sein, ist an dieser Stelle ein Mehrwert für die Verwendung von *Apache Kafka* gegeben. In diesem Fall wird der Kasten *Non-Persistent Performance ausreichend?* über die *Nein*-Kante verlassen und der Leitfaden wechselt auf die Seite von *Apache Kafka*. Andernfalls verbleibt der Leitfaden auf der linken *ActiveMQ*-Seite, da kein Mehrwert identifiziert werden konnte.

Zusammengefasst ist ein Mehrwert hier nur geboten, wenn der Einsatz von *persistent* Messaging unabdingbar und in diesem Zusammenhang die gebotene Performance von *ActiveMQ* nicht ausreichend ist oder der Einsatz von *non-persistent* Messaging immer noch unter den benötigten Performance-Werten liegt. In diesem Fall wird dieser Abschnitt des Leitfadens aus Abbildung 57 in Richtung *Apache Kafka* verlassen und die Entscheidung tendiert aktuell in Richtung dieses Systems. Der zu leistende Aufwand, um innerhalb von *Kafka* die *Exactly-Once* Zustellgarantie zu realisieren, fließt im Rahmen des Abschnitts 7.6 mit in die Betrachtung des Gesamtaufwands ein.

7.4 Besonderer Fokus auf Verfügbarkeit und Skalierbarkeit

Sofern in den beiden vorherigen Abschnitten bzw. Schritten im zugehörigen visuellen Leitfaden kein Mehrwert erkannt wurde, ist nach aktuellem Stand ein Umstieg auf *Apache Kafka* nicht gerechtfertigt.

An dieser Stelle sollte jedoch nochmals ein Blick auf die einzelnen nicht-funktionalen Anforderungen aus Kapitel 6 geworfen werden. Aufgrund der allgemein besseren Skalierbarkeits- und Verfügbarkeitseigenschaften von *Apache Kafka* kann sich dementsprechend ein Mehrwert ergeben.

Bei der Betrachtung der Anforderungen sollte jedoch ebenfalls die Robustheit mit berücksichtigt werden, da diese bei *Apache Kafka* von vielen Faktoren abhängig ist und dabei insbesondere maßgeblichen Einfluss auf die Zuverlässigkeit und die Verfügbarkeit hat.

Sofern ein Mehrwert erkannt wurde, gilt es im Anschluss zu klären, ob dieser im Verhältnis zum Aufwand bzw. zu den Kosten für die Umsetzung steht (siehe Abschnitt 7.6). Andernfalls wird im Folgenden hinterfragt, ob die Eigenschaften von *Apache Kafka* perspektivisch benötigt werden könnten.

7.5 Perspektivische Betrachtung

Auch wenn sich aktuell sowohl durch die Mehrfachverarbeitung, den Performance-Gewinn als auch durch die nicht-funktionalen Anforderungen kein Mehrwert für den Einsatz von *Apache Kafka* ergibt, sollte zusätzlich geprüft werden, ob diese Eigenschaften in Zukunft von Relevanz sein könnten.

Obwohl dem Umstieg auf *Apache Kafka* grundsätzlich immer ein konkreter Mehrwert zugrunde liegen sollte, eröffnet dieses System langfristig komplett neue Möglichkeiten für die eigene Anwendung.

Daher sollte, trotz eines aktuellen fehlenden Bedarfs, geprüft werden, ob in Zukunft Anpassungen oder Erweiterungen bevorstehen, die einen Umstieg bzw. einen Einsatz rechtfertigen.

Sofern diese Frage ebenfalls mit *Nein* beantwortet werden kann, endet an dieser Stelle der Leitfaden in dem finalen *ActiveMQ*-Kasten, der repräsentativ zu einem Einsatz dieses Systems rät bzw. zu diesem tendiert. Sollte der Benutzer jedoch spätestens zu diesem Zeitpunkt innerhalb des Leitfadens auf die Seite von *Apache Kafka* wechseln, werden im Anschluss Faktoren betrachtet, die final zum Scheitern eines Umstiegs führen könnten.

7.6 Aufwand im Verhältnis zum Mehrwert

Sofern der aktuelle Ausführungsstand des Leitfadens zu einem Einsatz von *Apache Kafka* tendiert, existiert ein Mehrwert für den Einsatz dieses Produktes. In dieser Ausgangslage müssen

jedoch weiterführende Aspekte betrachtet werden, die zum Scheitern eines Umstiegs auf *Apache Kafka* führen könnten.

In diesem Zusammenhang sollte im ersten Schritt hinterfragt werden, ob der zuvor extrahierte Mehrwert im Verhältnis zum notwendigen Aufwand steht. Angenommen, für den Umstieg auf *Apache Kafka* muss zusätzliches Fachpersonal eingestellt werden, so rentiert sich ein Umstieg nur geringfügig, wenn dadurch beispielsweise die Performance nur minimal verbessert wird bzw. eine Mehrfachverarbeitung nur in einem Randbereich des Systems als Sonderfall zum Einsatz kommt.

Da der notwendige Aufwand individuell dadurch variiert, unter welchen Qualitätsanforderungen das System eingesetzt werden soll, empfiehlt es sich, diesen individuell im Rahmen einer Aufwandsschätzung einzuordnen. Beispielsweise steigt der Aufwand durch die Maßnahmen, die notwendig sind, um eine *Exactly-Once* Zustellgarantie zu gewährleisten.

Ob der Aufwand im Verhältnis zum Mehrwert steht, ist eine sehr individuelle und subjektive Beurteilung, die vom jeweiligen Projekt und den äußeren Faktoren abhängt. In diesem Stadium empfiehlt es sich, dieser Fragestellung projektintern im Rahmen einer Kosten-Nutzen-Analyse nachzugehen. Ferner sollte der ermittelte Aufwand, sofern dies möglich ist, mit dem Aufwand verglichen werden, der im jetzigen JMS-System noch zu leisten ist.

Falls der notwendige Aufwand grundsätzlich nicht im Verhältnis zum Mehrwert steht, sollte zusätzlich noch hinterfragt werden, ob langfristig eine Kostenersparnis zu erwarten ist und ab welchem Zeitpunkt sich eine Umstellung entsprechend amortisieren würde.

Sofern sich ein Umstieg nicht lohnt, führt der Leitfaden zurück auf die Seite von *ActiveMQ* und endet mit einer finalen Empfehlung zu diesem System. Im positiven Fall wird im Folgenden geprüft, ob eine entsprechende Interoperabilität gewährleistet ist.

7.7 Interoperabilität gewährleistet

Im nächsten Schritt sollten alle Systeme, die mit *Apache Kafka* interagieren, identifiziert werden. Hierzu gehören insbesondere Programme, die Nachrichten in das System übertragen und Nachrichten aus diesem konsumieren. An dieser Stelle müssen entsprechende APIs für die jeweilige Programmiersprache vorhanden sein. In diesem Zusammenhang werden die relevantesten Programmier- und Script-Sprachen bereits abgedeckt. Dennoch gilt es, dies im Einzelfall zu überprüfen.

Darüber hinaus sollte für alle anderen Systeme und Programme, die in irgendeiner Form Berührungspunkte mit *Apache Kafka* haben, sichergestellt werden, dass die Interaktion zwischen diesen Systemen wie geplant funktioniert. Zu diesen Systemen gehören beispielsweise Monitoring-Systeme und Administrierungs-Konsolen.

Des Weiteren sollte hinterfragt werden, ob und wie sich verwendete Protokolle sowie geplante Sicherheitsmaßnahmen mit *Kafka* in Einklang bringen lassen.

Sofern die Interoperabilität zwischen *Apache Kafka* und einem bestimmten System nicht direkt besteht, ist zunächst zu prüfen, welche möglichen Kompensierungsmaßnahmen vorhanden sind. Sollte für dieses Problem keine Lösung zu finden sein, führt dies insbesondere bei elementaren Systemen dazu, dass auf einen Einsatz von *Apache Kafka* verzichtet werden muss. In diesem Fall wechselt auch hier der zugehörige Leitfaden in Abbildung 57 auf die Seite von *ActiveMQ*. Andernfalls wird abschließend überprüft, ob sich die Umsetzung unter Firmengesichtspunkten und äußeren Faktoren überhaupt durchführen lässt.

7.8 Umsetzung durchführbar

Im letzten Schritt sollte, sofern alle vorherigen Bedingungen für den Einsatz von *Apache Kafka* sprechen, geprüft werden, ob sich der Umstieg auf dieses System innerhalb des Unternehmens oder Projektes realisieren lässt.

In diesem Zusammenhang ist zu entscheiden, wie mit dem benötigten *Kafka*-Fachwissen umgegangen wird. Dieses kann durch neue Entwickler mit entsprechenden Fachwissen oder durch Weiterbildungsmaßnahmen des vorhandenen Personals erfolgen. Außerdem kann auch in Erwägung gezogen werden, externe Beratungsunternehmen mit einzubeziehen. Des Weiteren ist zu prüfen, ob grundsätzlich genug Mitarbeiter vorhanden sind, um dem bevorstehenden Aufwand gerecht zu werden.

Fortführend sollten alle Kostenpunkte identifiziert und anschließend hinterfragt werden, ob aus finanziellen Gesichtspunkten ein Umstieg rentabel ist. Die Kosten hängen unter anderem von den vorhandenen Anforderungen, dem daraus resultierenden Aufwand sowie davon ab, ob zusätzlich in weitere Maßnahmen investiert werden muss.

Sofern sich ein Umstieg wirtschaftlich realisieren lässt, endet der in Abbildung 57 dargestellte Leitfaden in der finalen *Kafka-Box*. Diese repräsentiert eine Empfehlung bzw. eine Tendenz für den Einsatz von *Apache Kafka*.

7.9 Abschließende Anmerkung

Abschließend ist erneut anzumerken, dass es sich bei der Entscheidung für oder gegen eines der in dieser Arbeit betrachteten Messaging-Systeme um eine individuelle Beurteilung handelt, die sich von Fall zu Fall unterscheidet. Trotz einer durch den Leitfaden ermittelten Tendenz bestehen eventuell noch weitere Rahmenbedingungen, die für ein anderes Produkt sprechen. Zuletzt basieren viele der notwendigen Entscheidungen auf subjektiven Fragestellungen, die entsprechend unterschiedlich eingeschätzt werden können.

Allgemein kann jedoch festgehalten werden, dass insbesondere dann ein Einsatz von *Apache Kafka* in Betracht kommt, wenn eine Mehrfachverarbeitung angestrebt oder eine Performance benötigt wird, die über die Möglichkeiten von klassischen Systemen hinausgeht.

Abschließend sollte trotz aller Euphorie bezüglich der Möglichkeiten, die ein Einsatz von *Apache Kafka* eröffnet, eine angemessene Skepsis vorhanden sein und nur dann ein Umstieg erfolgen, wenn ein erkennbarer Mehrwert vorliegt.

Da es sich bei *Apache Kafka* um ein verhältnismäßig junges Projekt handelt, ist außerdem damit zu rechnen, dass es in künftigen Versionen zu weiteren Konzeptänderungen kommen kann. Diese könnten dazu führen, dass sich die ermittelte Tendenz im hier betrachteten Leitfaden entsprechend verändert.

8 Anwendungsfall OTTO

Dieses Kapitel befasst sich mit dem konkreten Anwendungsfall der *OTTO (GmbH & Co KG)*. In diesem Zusammenhang werden die Gegebenheiten des zugehörigen Systems sowie dessen nicht-funktionale Anforderungen auf den in Kapitel 7 definierte Leitfaden abgebildet.

In Abschnitt 8.1 wird hierzu ein Teilbereich der Systemarchitektur sowie die zugehörige Art und Weise der Nachrichtenverarbeitung beschrieben. Anschließend nimmt der Abschnitt 8.2 Bezug auf die zugehörigen nicht-funktionalen Anforderungen. Des Weiteren befasst sich Abschnitt 8.3 mit einer *Exactly-Once* Übertragung innerhalb der Infrastruktur. In diesem Zusammenhang werden Plausibilitätstests in beiden Systemen durchgeführt. Abschnitt 8.4 beschreibt eine prototypische Clienterweiterung, die sicherstellt, dass sich bei *Apache Kafka* eine *Exactly-Once* Zustellgarantie tatsächlich erzielen lässt. Darauf folgend wird in Abschnitt 8.5 ein *Ende-zu-Ende* Test beschrieben, der sich an den produktiven Gegebenheiten innerhalb der Infrastruktur orientiert. Zuletzt werden die konkreten Anforderungen des Unternehmens in Abschnitt 8.6 auf den Leitfaden aus Kapitel 7 projiziert.

8.1 Beschreibung Systemarchitektur

Der im Folgenden beschriebene Teilbereich der Systemarchitektur dient dazu, Lagerbestände von externen Handelspartnern entgegenzunehmen, um diese Artikel anschließend auf der *OTTO*-Plattform anbieten zu können. Abbildung 58 zeigt den entsprechenden Aufbau der zugehörigen Systemarchitektur.

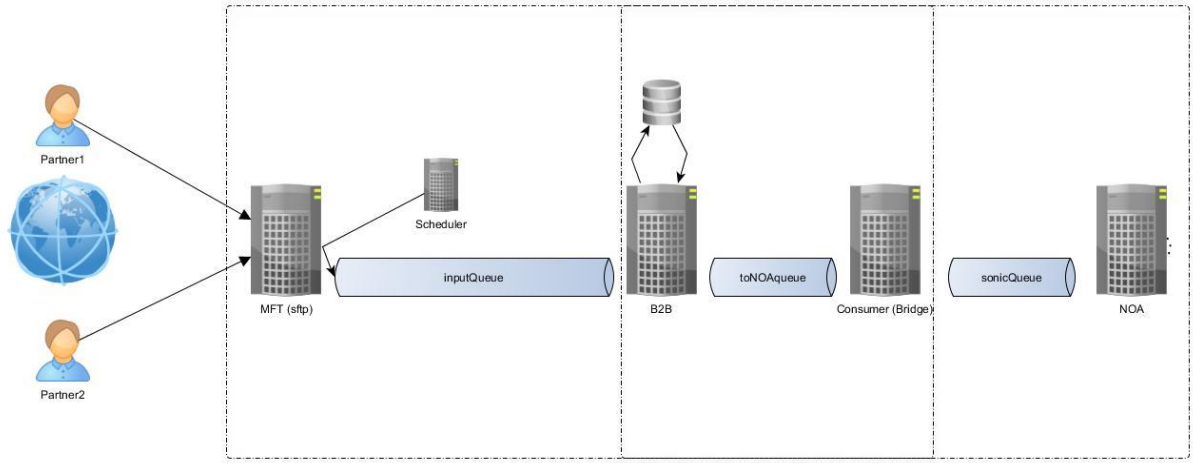


Abbildung 58: Teilbereich der Systemarchitektur der OTTO-(GmbH & Co KG)

In diesem Zusammenhang kontaktieren die jeweiligen Handelspartner in Abbildung 58 beginnend von links den MFT-Server (*Managed-File-Transfer*), der die Daten synchron entgegennimmt. Die auf dem MFT-Server gespeicherten Datensätze werden anschließend über einen Scheduler in eine *Input*-Queue eines JMS-Systems übertragen. Der Scheduler dient dazu, die

eintreffende Last konstant über die Zeit zu verteilen. Anschließend werden diese Daten von einem *JavaEE*-Server mit dem Namen *B2B* über *Message Driven Beans* entgegengenommen und in einer Datenbank abgelegt.

Da der weitere Verarbeitungsprozess täglich im Rahmen eines Events gestartet wird, müssen die Partner bis zu einer bestimmten Uhrzeit ihre Produkte in die Infrastruktur übertragen haben. Andernfalls werden diese Produkte am Folgetag nicht im Artikelbestand auf der *OTTO*-Plattform berücksichtigt.

Dieser Vorgang führt dazu, dass sich täglich der eintreffende Datenbestand auf dem *B2B*-Server sammelt und vollständig an das Folgesystem übertragen wird. Dieser Übertragungsabschnitt, der in Abbildung 58 durch *toNOAQueue* gekennzeichnet ist, wird aktuell durch ein *ActiveMQ* System realisiert und stellt einen der wesentlichen potenziellen Einsatzzwecke für *Apache Kafka* dar. Die Menge an zu übertragenden Nachrichten liegt etwa bei 300.000 Nachrichten täglich. Hierbei wird eine maximale Übertragungsdauer von 1,5 Minuten angestrebt.

Zuletzt werden in Abbildung 58 die Nachrichten von einer Sonic-Bridge aus der *toNOAQueue* geladen und in das Sonic-Messaging-System gespeist. Diese Daten werden nun final vom NOA-System entgegengenommen und in den *OTTO*-Artikelbestand eingepflegt. Die Bezeichnung NOA steht hierbei für *Neue OTTO Abwicklung*.

8.2 Vorhandene nicht-funktionale Anforderungen

Aus dem in Abschnitt 8.1 dargestellten Umgang hinsichtlich der Verarbeitung der Datenbestände von Handelspartnern sowie den grundsätzlichen Gegebenheiten, ergeben sich entsprechende nicht-funktionale Anforderungen, die maßgebliche Auswirkungen auf einen potenziellen Umstieg auf *Apache Kafka* haben.

Zu diesen Anforderungen zählt unter anderem eine exakte und verbindliche Zustellgarantie der einzelnen Nachrichten. Es wird damit im Rahmen der nicht-funktionalen Anforderung *Zuverlässigkeit* die höchste Zustellgarantie *Exactly-Once* benötigt. Diese erfordert im Fall von *ActiveMQ* eine Verwendung des *persistent*-Messagings. Da die Betrachtung der nicht-funktionalen Anforderungen in Kapitel 6 ergeben hat, dass diese Zustellgarantie mit den Basismechanismen von *Apache Kafka* nicht realisierbar ist, muss an dieser Stelle die Machbarkeit mithilfe entsprechender Gegenmaßnahmen sichergestellt werden. In diesem Zusammenhang wird in Abschnitt 8.3 ein Plausibilitätstest durchgeführt, der überprüft, ob die gesendete Nachrichtenmenge der empfangenen Menge entspricht. Erwartungsgemäß trifft das bei *ActiveMQ* zu. Bei *Apache Kafka* hingegen ist jedoch davon auszugehen, dass dies nicht der Fall ist. Anschließend wird in Abschnitt 8.4 die zuvor angesprochene Machbarkeit sichergestellt.

Eine weitere nicht-funktionale Anforderung ergibt sich aus der angestrebten Übertragungsdauer von 1,5 Minuten bei einer Menge von 300.000 Nachrichten. In diesem Zusammenhang wird in

Abschnitt 8.5 die Ausgangslage entsprechend realitätsnah nachgestellt und ein *Ende-zu-Ende* Test durchgeführt, der die absolut benötigte Zeit in beiden Systemen ermittelt.

Ausgehend von dem hier beschriebenen Anwendungsfall wird keine direkte Mehrfachverarbeitung benötigt, da die Daten nur vom NOA-Server verarbeitet werden. Jedoch gibt es im Gesamt-System weitere Bereiche, in denen ein entsprechender Bedarf diesbezüglich gegeben ist.

Zuletzt wird allgemein ein besonderer Fokus auf die nicht-funktionale Anforderung *Verfügbarkeit* gelegt.

8.3 Gewährleistung Exactly-Once

Da die in Abschnitt 8.2 dargelegten Anforderungen des Unternehmens eine *Exactly-Once* Zustellgarantie fordern, muss in diesem Zusammenhang sichergestellt werden, dass sich diese mit *Apache Kafka* wie beabsichtigt realisieren lässt. Hierzu wird in diesem Abschnitt ein Plausibilitätstest durchgeführt, der insgesamt 300.000 Nachrichten in die jeweiligen Systeme überträgt und zeitgleich ausliest. Ziel hiervon ist es, das tatsächliche Verhalten beider Systeme aufzuzeigen, um auf dieser Grundlage anschließend den Erfolg einer optimierten Variante bemessen zu können.

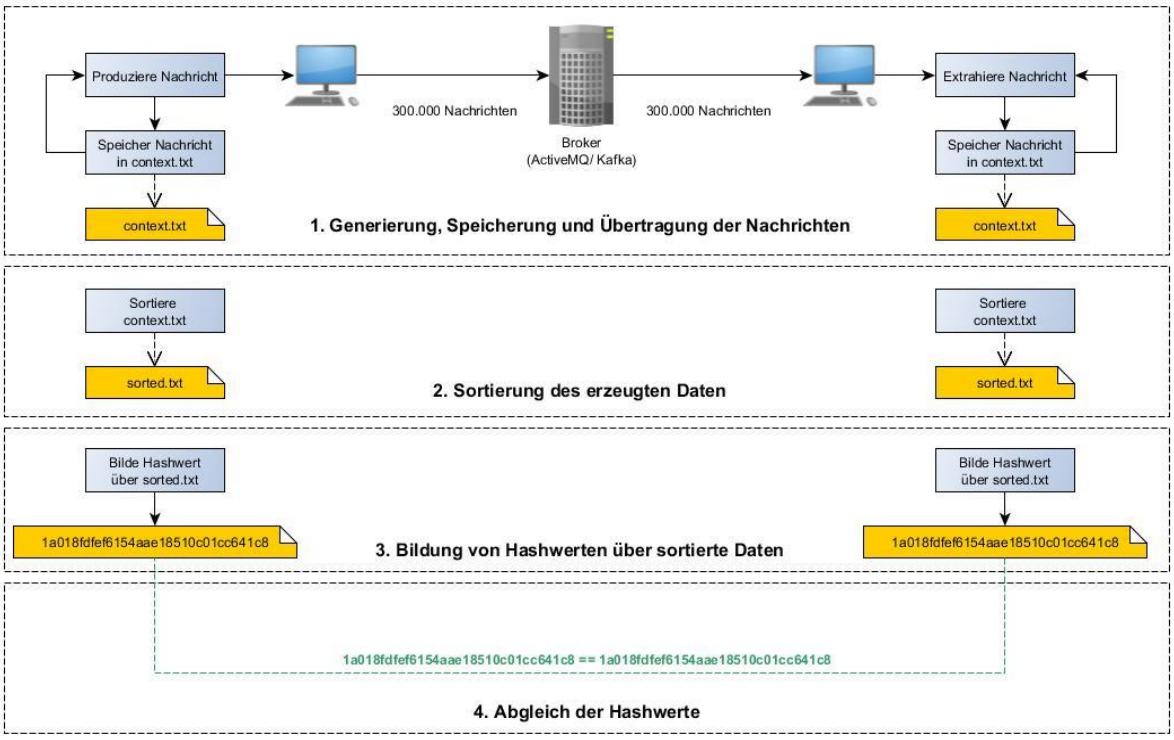


Abbildung 59: Aufbau und Vorgehen des Plausibilitätstests

Sowohl beim Senden als auch beim Empfangen werden die jeweiligen Nachrichten in einer Textdatei gespeichert (siehe Abbildung 59). Im Anschluss an den Testdurchlauf werden diese

Dateien in *Schritt 2* jeweils auf Sende- und Empfangsseite sortiert. Darauffolgend werden in *Schritt 3* Hashwerte über die einzelnen Dateien gebildet. Im letzten *Schritt 4* erfolgt ein Vergleich der erzeugten Hashwerte. Sofern diese auf Producer- und auf Consumer-seite übereinstimmen, ist sichergestellt, dass die empfangene Menge an Nachrichten der gesendeten Menge entspricht. Erwartungsgemäß ist dies bei *ActiveMQ* der Fall. Bei *Apache Kafka* hingegen ist, sofern lediglich das Basis-Produkt verwendet wird, ein anderes Verhalten zu erwarten. Dies ist darin begründet, dass das Auftreten von Duplikaten nicht verhindert werden kann und sich damit die Inhalte der Textdateien entsprechend unterscheiden.

8.3.1 Plausibilitätstest ActiveMQ

Die Abbildung 60 zeigt den durchgeführten Plausibilitätstest für *ActiveMQ*. Hierbei werden zu Beginn über einen Docker-Container zwei Consumer gestartet, die insgesamt 300.000 Nachrichten an den Broker übertragen. Anschließend wird die erzeugte *content.txt*-Datei, wie zuvor beschrieben, sortiert und ein Hashwert über diese gebildet.

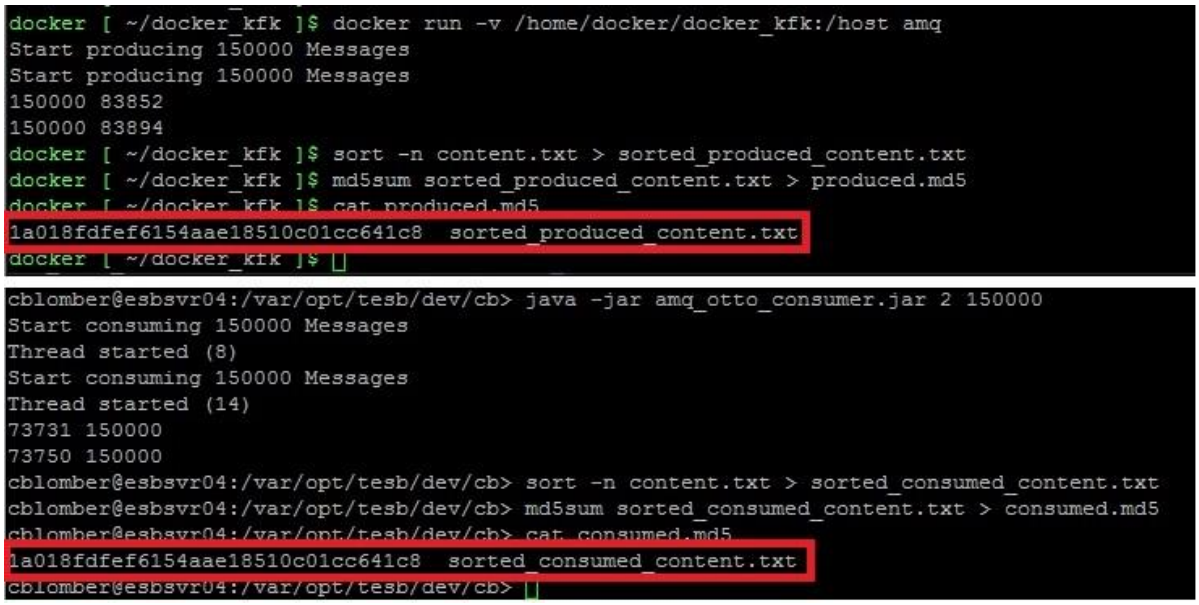


Abbildung 60: Plausibilitätstest ActiveMQ

Der untere Teil der Abbildung 60 zeigt den Consumer-seitigen Teil des durchgeführten Tests. An dieser Stelle werden die Consumer über eine JAR-Datei gestartet. Anschließend erfolgt die Weiterverarbeitung analog zum zuvor beschriebenen Vorgehen auf Consumer-Seite. Die rot umrandeten Ausgaben aus Abbildung 60 zeigen, dass die Hashwerte auf Sender- und Empfänger-Seite übereinstimmen. Dies bestätigt die Annahme, dass bei *ActiveMQ* die gesendete Menge der empfangenen Menge entspricht und somit ist die Zustellgarantie *Exactly-Once* in diesem System gewährleistet.

8.3.2 Plausibilitätstest Apache Kafka

Die Abbildung 61 zeigt den durchgeführten Plausibilitätstest für *Apache Kafka* auf Producer- sowie auf Consumer-Seite. An dieser Stelle wird ebenfalls ein Docker-Container mit zwei Producern gestartet, die erzeugte Textdatei sortiert und abschließend ein Hashwert über diese gebildet.

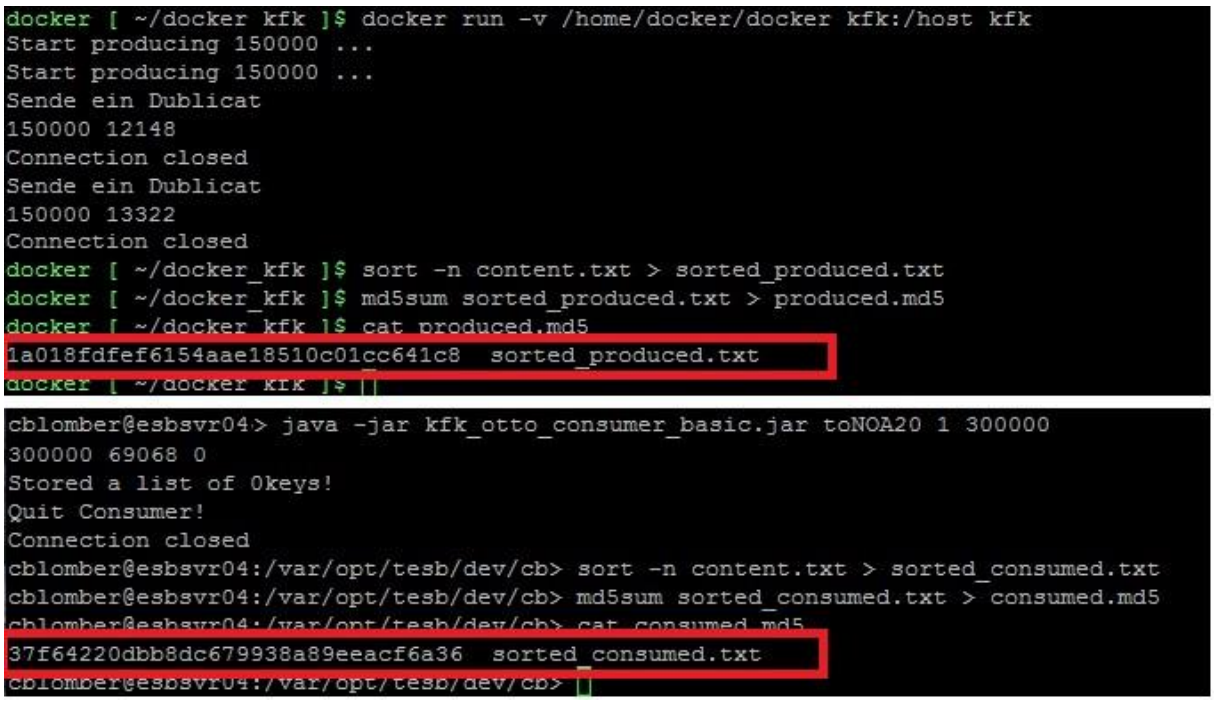


Abbildung 61: Plausibilitätstest Apache Kafka

Die Ausgabe der Hashwerte belegt die zu Beginn dieses Abschnitts aufgestellte These, dass sich die gesendete und die empfangene Nachrichtenmenge aufgrund von Duplikaten entsprechend unterscheiden. In der oberen Producer-Ausgabe ist zu erkennen, dass jeder Producer jeweils ein Duplikat produziert. Dieses Verhalten wurde künstlich durch die Producer hervorgerufen, indem eine Nachricht doppelt gesendet und das Duplikat jeweils nicht in der Textdatei gespeichert wurde. Dieses Vorgehen dient lediglich der Veranschaulichung. Identische Tests ohne dieses erzwungene Vorgehen zeigten ein identisches Verhalten von *Apache Kafka*.

8.4 Prototypische Entwicklung der Gegenmaßnahmen

In diesem Abschnitt wird ein prototypischer Lösungsansatz erklärt, der den fehlgeschlagenen Plausibilitätstest aus Abschnitt 8.3 optimiert. Diese Verbesserung soll dazu führen, dass die Hashwerte im Anschluss an eine erneute Durchführung identisch sind und damit ein *Exactly-Once*-Verhalten gewährleistet ist.

Wie bereits erwähnt, ist eine mögliche Gegenmaßnahme, die gesendeten Nachrichten mit einer eindeutigen Sequenznummer zu versehen. Anschließend wird diese Sequenznummer auf Consumer-Seite extrahiert und geprüft, ob diese bereits zu einem vorherigen Zeitpunkt verarbeitet wurde. In diesem Zusammenhang können die Nachrichten bzw. Sequenznummern beispielsweise in einer Datenbank oder in einer *Map* gespeichert werden.

8.4.1 Producer-seitige Anpassungen

In der hier beschriebenen prototypischen Umsetzung, die lediglich dazu dient, die Machbarkeit sicherzustellen, wurde der *Map*-Ansatz verfolgt. Die im Folgenden dargestellten Codeausschnitte dienen ausschließlich der Veranschaulichung und eignen sich nicht für einen produktiven Einsatz.

```
49    public void sendMessageToCluster(String content){
50        MessageContainer messageContainer = new MessageContainer();
51        messageContainer.setContent(content);
52        String containerSchema = loadAvroSchema();
53
54        GenericRecord mappedMessage = useSchemaToMapObjectToMessage(containerSchema, messageContainer);
55        ProducerRecord<String, GenericRecord> data = new ProducerRecord<String, GenericRecord>("messageContainer", "message", mappedMessage);
56
57        producer.send(data);
58    }
```

Abbildung 62: Producer-Optimierung

Abbildung 62 zeigt die *sendMessageToCluster()*-Methode des Producer-Threads. Diese erzeugt zu Beginn ein eigenes Java-Objekt des Typs *MessageContainer*. Dem Container wird anschließend in Zeile 51 über die Exemplar-Variable *content* der ursprüngliche Nachrichteninhalt zugewiesen. Bei der Erzeugung des *MessageContainer*-Objekts wird über den Konstruktor automatisch eine eindeutige Sequenznummer des Typs *UUID* erzeugt und diese der Exemplar-Variable *identifier* zugewiesen.

Anschließend wird in Zeile 52 das zugehörige Schema geladen und in der darauf folgenden Zeile das Java-Objekt *MessageContainer* auf dieses Schema *gemappt*. Das dadurch erzeugte *GenericObject* wird in Zeile 55 einem *ProducerRecord* zugewiesen und in Zeile 57 an den Broker gesendet.

8.4.2 Consumer-seitige Anpassungen

Auf Seiten der Consumer müssen die von den Producern hinzugefügten Sequenznummern entsprechend extrahiert und ausgewertet werden. Abbildung 63 zeigt die *run()*-Methode des Consumer-Threads. Die Methode ruft in Zeile 70 eine neue Menge an Nachrichten vom Server ab. Anschließend wird diese Menge in Form einer *for()*-Schleife abgearbeitet. In diesem Rahmen wird in Zeile 72 mithilfe des Templates der Inhalt der Nachricht wieder in das ursprüngliche *MessageContainer*-Objekt überführt.

Anschließend prüft Zeile 73 anhand der enthaltenen Sequenznummer, ob die jeweilige Nachricht bereits verarbeitet wurde. Sofern das der Fall ist, wird die Nachricht als Duplikat betrachtet

und dementsprechend verworfen. Falls die Sequenznummer noch nicht bekannt ist, wird in der Zeile 74 die Nachrichtenverarbeitung durchgeführt und anschließend in Zeile 75 die Sequenznummer gespeichert, um Duplikaten vorzubeugen.

```
67    public void run(){
68        try{
69            while(true){
70                ConsumerRecords<String, String> records = consumer.poll(0);
71                    for(ConsumerRecord<String, String> record: records){
72                        MessageContainer messageContainer =  convertMessageWithAvroTemplateToObject(record);
73                        if ( checkInMapOrDatabaseForDublicates(messageContainer.getIdentifier()) ){
74                            processMessage(messageContainer.getContent());
75                            storeIdentifierToAvoidTwiceProcessing(messageContainer.getIdentifier(), messageContainer.getContent());
76                        }else{
77                            System.out.println("Dublikat erkannt");
78                        }
79                        consumer.commitAsync(new OffsetCommitCallback() {
88                    }
89            }
90        }finally {
91            consumer.close();
92            System.out.println("Connection closed");
93        }
94    }
```

Abbildung 63: Consumer-Optimierung

Abschließend ist anzumerken, dass sich diese einfache Form der Vermeidung von Duplikaten lediglich für die Verwendung von einem Consumer eignet. Dies liegt daran, dass jeder Consumer aktuell eigenständig für sich festhält, welche Nachrichten er bereits verarbeitet hat. Sofern es zu einem Rebalance kommt und den Consumern in diesem Rahmen andere Partitionen zugewiesen werden, fehlt dem neuen Producer der aktuelle Verarbeitungsstand. Dieser ist fälschlicherweise mit dem anderen Consumer-Thread in eine andere Partition gewechselt. Das hier beschriebene Vorgehen müsste um einen gemeinsamen Verarbeitungsstand erweitert werden, auf den alle Consumer Zugriff haben.

8.4.3 Plausibilitätsprüfung mit optimierten Clients

Um zu überprüfen, ob eine *Exactly-Once*-Zustellgarantie mit den optimierten Clientanwendungen erreicht werden kann, wurde mit dieser Version ein erneuter Plausibilitätstest durchgeführt. Abbildung 64 zeigt die entsprechenden Kommandozeilen-Ausgaben der Testdurchführung.

An dieser Stelle wird analog zu den in Abschnitt 8.3 beschriebenen Plausibilitätstests vorgegangen. Die Producer werden über einen Docker-Container gestartet und produzieren insgesamt 300.000 Nachrichten sowie 2 Duplikate.

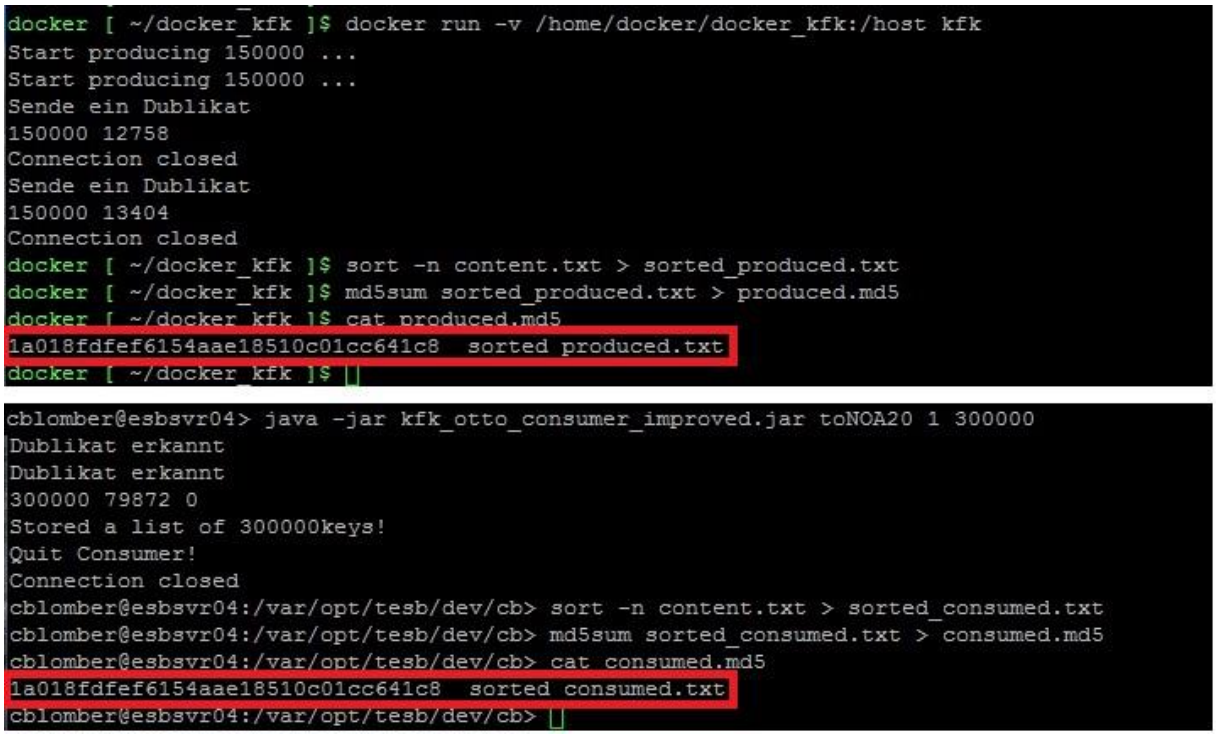


Abbildung 64: Plausibilitätstest mit optimierten Clientanwendungen

Im Gegensatz zu dem vorherigen Plausibilitätstest aus Abschnitt 8.3.2 erkennen die Consumer im unteren Teil der Abbildung 64 nun die beiden Duplikate und verwerfen diese. Im Anschluss zeigt sich nun durch einen Abgleich der Hashwerte, dass die gesendete Menge an Nachrichten der empfangenen Menge entspricht. Mit diesem Ergebnis ist sichergestellt, dass sich die *Exactly-Once* Zustellgarantie realisieren lässt.

8.5 Ende-zu-Ende Test

Der *Ende-zu-Ende*-Test dient dazu, den Durchsatz vom Versand der ersten Nachricht bis zur Auslieferung der letzten Nachricht zu ermitteln. Konkret wird untersucht, wie hoch der zeitliche Bedarf für die Übertragung der 300.000 Nachrichten ist. Aus Unternehmenssicht ist dieser ermittelte Bedarf relevant, um innerhalb des zuvor definierten Leitfadens (siehe Kapitel 7) einordnen zu können, ob die Performance der einzelnen Systeme ausreichend bzw. erforderlich ist.

Abbildung 65 stellt die ermittelten Testergebnisse gegenüber. Es zeigt sich, dass *ActiveMQ* für die Übertragung der 300.000 Nachrichten circa 2,5 Minuten benötigt. Dies entspricht einem Durchsatz von 2000 Nachrichten pro Sekunde.

Im Gegensatz dazu weist *Apache Kafka* mit 1 Minute und 10 Sekunden eine Zeitersparnis von mehr als der Hälfte auf. Der Nachrichtendurchsatz liegt mit circa 4.350 Nachrichten pro Sekunde um den Faktor 2,15 höher als bei *ActiveMQ*.

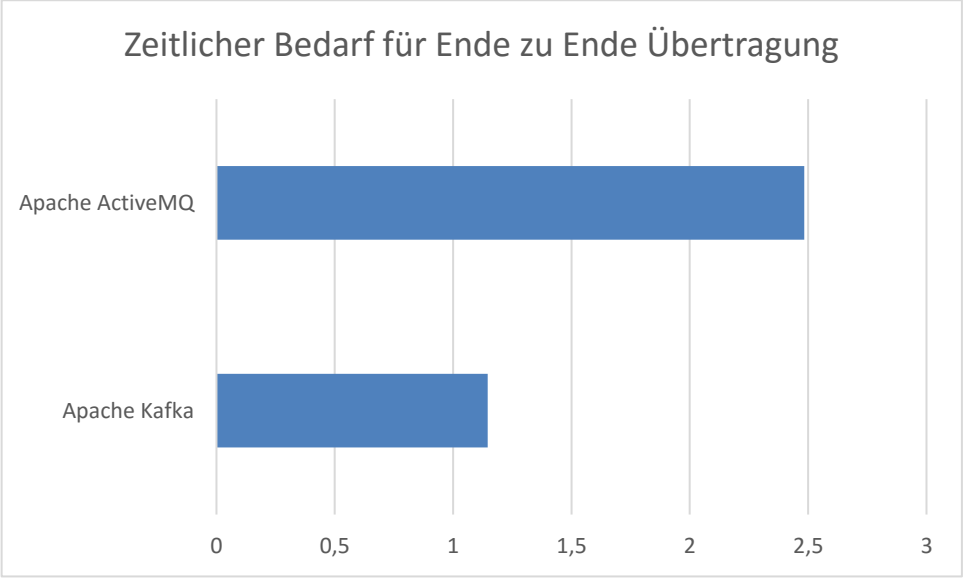


Abbildung 65: Zeitlicher Bedarf für Ende-zu-Ende Übertragung

Da das Unternehmen in Bezug auf die nicht-funktionale Anforderung *Performance* eine angestrebte Zeitspanne von maximal 1,5 Minuten definiert, wirkt sich die Zeitdifferenz zwischen beiden Systemen innerhalb des Leitfadens positiv für *Apache Kafka* aus. Aus Sicht der Performance ist damit ein Mehrwert für den Einsatz von *Apache Kafka* gegeben.

Die ermittelten Testergebnisse deckten sich im weitesten Sinne mit den Performance-Tests aus Kapitel 5. Die Reduzierung der übertragbaren Nachrichtenmenge pro Sekunde ist dadurch zu erklären, dass nun gleichzeitig sowohl Producer als auch Consumer aktiv sind.

8.6 Abbildung des Leitfadens auf den konkreten Anwendungsfall

In diesem Abschnitt wird der in Kapitel 7 definierte Leitfaden auf den konkreten Anwendungsfall des Unternehmens abgebildet. Während dieses Vorgangs nehmen zum einen die in Abschnitt 8.2 definierten nicht-funktionalen Anforderungen Einfluss. Zum anderen werden die Erkenntnisse aus den Abschnitten 8.3 - 8.5 hinsichtlich der Zustellgarantie und der Performance mit einbezogen. Im Zusammenhang damit wird der Leitfaden in zwei Bereiche unterteilt. Der 1. Teil befasst sich in Abschnitt 8.6.1 mit der Identifizierung eines Mehrwertes für den Einsatz von *Apache Kafka*. Der 2. Teil in Abschnitt 8.6.2 dient dazu potenzielle Risiken auszuschließen.

8.6.1 Anwendung des Leitfadens – Identifizierung von Mehrwert (Teil 1)

Zu Beginn des Leitfadens wird im ersten Schritt hinterfragt, ob eine Mehrfachverarbeitung erforderlich ist. Diese wird im Anwendungsfall des Unternehmens nur eingeschränkt gefordert und hängt vom jeweiligen Projekt ab. Da grundsätzlich ein Bedarf besteht, kann diese Fragestellung im konkreten Fall mit *Ja* zu beantworten werden. Da hierdurch bereits ein grundsätz-

licher Mehrwert identifiziert wurde, müssten normalerweise die anderen Aspekte wie Performance nicht explizit betrachtet werden. Zur Veranschaulichung werden die Aspekte jedoch im Folgenden, unabhängig vom vorherigen Schritt, durchlaufen.

In Schritt 2 wird geprüft, ob die Kafka-Performance benötigt wird und somit die Performance von *ActiveMQ* nicht ausreichend ist. Da durch die *Ende-zu-Ende* Tests erkannt wurde, dass dies zutreffend ist, kann diese Fragestellung im ersten Teilschritt mit *Ja* beantwortet werden.

Anschließend hinterfragt der Leitfaden, ob die Zustellgarantie *Exactly-Once* erforderlich ist, welche im Fall von ActiveMQ dazu führen würde, dass zwingend *persistent*-Messaging zum Einsatz kommen muss. Da dies, basierend auf den nicht-funktionalen Anforderungen, der Fall ist, reicht die *ActiveMQ*-Performance im Anwendungsfall des Unternehmens definitiv nicht aus. Bei einem Verzicht auf *Exactly-Once* müsste überprüft werden, ob der damit verbundene Performance-Gewinn von *ActiveMQ* ausreichend wäre.

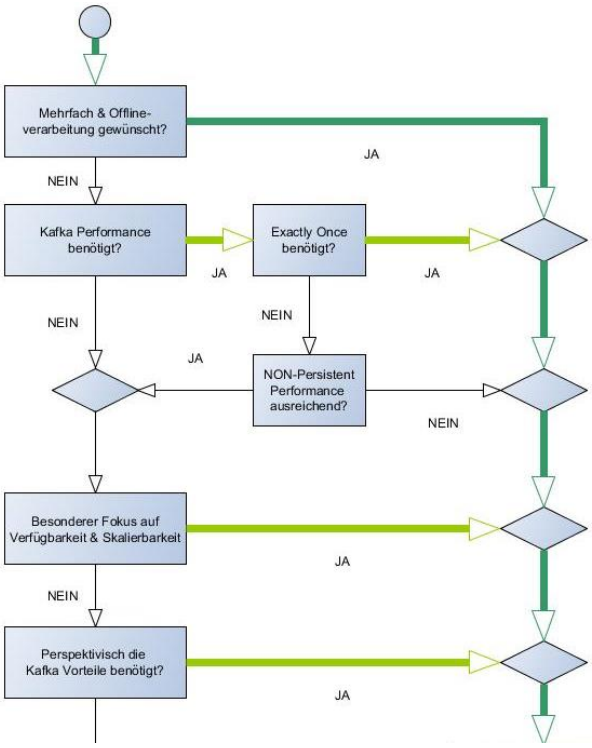


Abbildung 66: Anwendung des Leitfadens - Identifizierung von Mehrwerten

In Schritt 3 wird geprüft, ob ein besonderer Fokus auf die nicht-funktionalen Anforderungen *Skalierbarkeit* und *Verfügbarkeit* gelegt wird. Dies ist ebenfalls für das Unternehmen zutreffend und führt dazu, dass auch dieser Schritt über die *Ja*-Kante verlassen wird.

Im letzten Schritt, der sich mit der Identifizierung eines Mehrwertes befasst, wird begutachtet, ob langfristig durch Projekterweiterungen oder neue Projekte die zuvor betrachteten Aspekte benötigt werden. Da bisher ohnehin alle Aspekte für einen Einsatz von *Apache Kafka* gesprochen haben, ist diese Frage ebenfalls mit *Ja* zu beantworten.

Abschließend veranschaulicht die Abbildung 66 den Verlauf des Leitfadens. Da bereits nach dem ersten Schritt ein Mehrwert identifiziert wurde, zeigt der dunkelgrüne Verlauf den offiziellen Pfad durch den Leitfaden. Die hellgrünen Pfade, welche von Schritt 2, 3 und 4 ausgehen, veranschaulichen die alternativen Wege durch den Leitfaden, sofern bestimmte Aspekte nicht gegeben wären. Darüber hinaus verdeutlichen die zusätzlichen Pfade, dass im Fall des Unternehmens der Bedarf für den Einsatz von *Apache Kafka* auf allen Ebenen gegeben ist.

8.6.2 Anwendung des Leitfaden – Betrachtung der Risikofaktoren (Teil 2)

Nachdem in Abschnitt 8.6.1 ein deutlicher Mehrwert für den Einsatz von *Apache Kafka* identifiziert wurde, sind in diesem zweiten Teil des Leitfadens jene Risikofaktoren zu betrachten, die im Nachgang dazu führen könnten, dass ein Umstieg auf *Apache Kafka* scheitert.

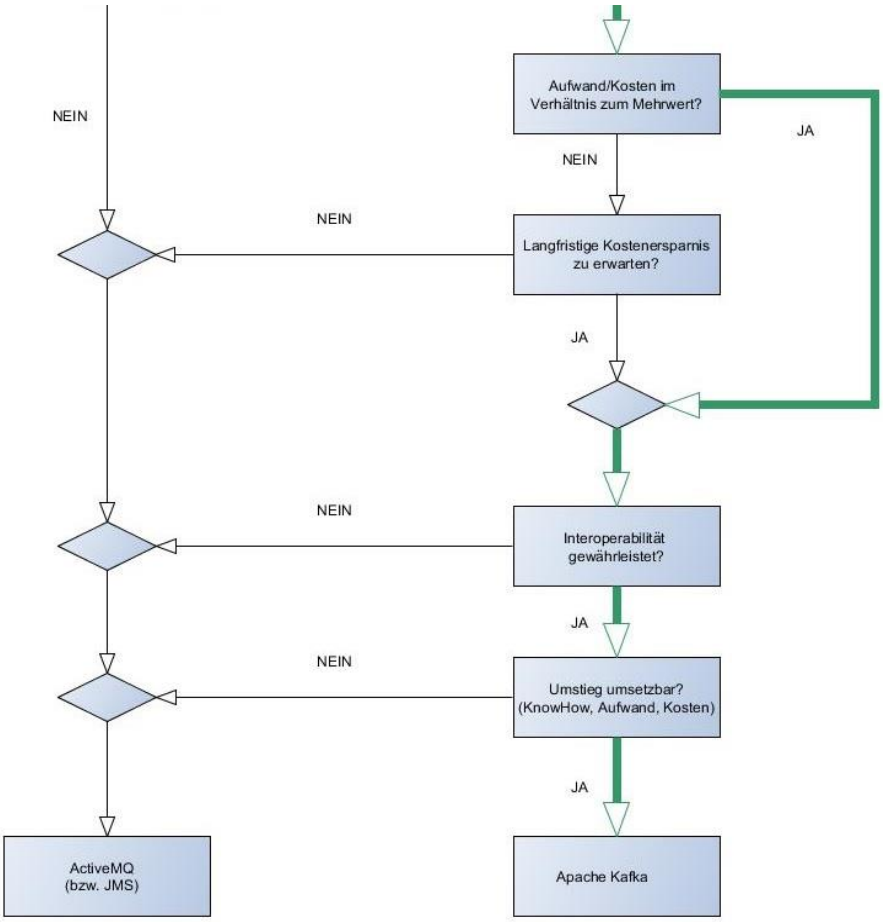


Abbildung 67: Anwendung des Leitfadens – Betrachtung der Risiko-Faktoren

In diesem Zusammenhang wird zuerst geprüft, ob der erkannte Mehrwert im Verhältnis zum Aufwand steht. Dieser Aufwand spiegelt sich im Wesentlichen in der Erzeugung von entsprechender Fachkompetenz, den Umsetzungs- bzw. Migrationsmaßnahmen sowie zusätzlichem Wartungsaufwand wider. Da im Vorfeld bei allen Aspekten ein Mehrwert geboten war, steht dieser in vollem Maße im Verhältnis zum notwendigen Aufwand.

Sofern diese Frage nicht in vollem Umfang mit *Ja* beantwortet werden könnte, würde anschließend geprüft, ob langfristig eine Kostenersparnis zu erwarten ist. Ausgehend von einer simplen Aufwandsschätzung (siehe Tabelle 5) entstehen einmalige Kosten in einer Größenordnung von 135.000 Euro für die Schulung von 4 Mitarbeitern und der Migration der 3 Haupt-Backendsysteme. Da im Fall von *Apache Kafka* aufgrund der Leistungssteigerung weniger Server-Instanzen benötigt werden, ist langfristig mit einer Kostenersparnis zu rechnen. Angesichts des stetig steigenden E-Commerce-Wachstums strebt das Unternehmen jedoch keine unmittelbare Kostenersparnis, sondern einen Leistungszuwachs auf identischer Hardware an. In diesem Fall wird sich die Kostenersparnis finanziell erst beim Einkauf zusätzlicher Hardware bemerkbar machen. Grundsätzlich wird bei der reinen Betrachtung der Nachrichtenmenge die Hälfte an Serverinstanzen benötigt. Da ein zugehöriges *Kafka*-Cluster jedoch mindestens 3 anstatt bisher 2 Broker benötigt, liegt die Ersparnis beim Neueinkauf von Servern damit bei 25%.

Kostenpunkt		**Kostenaufschlüsselung**	**Kosten-Gesamt**
Schulungsmaßnahmen	Interne Kosten	5 PT * 4 Mitarbeiter	14.000 Euro
	Schulungskosten	4000 Euro * 4 Mitarbeiter	16.000 Euro
Umsetzung & Migrationsmaßnahmen (3 Backend-Systeme)	Analyse	3 Systeme * 10 PT	21.000 Euro
	Umsetzung & Testing	3 Systeme * 40 PT	84.000 Euro
			135.000 Euro

Tabelle 5: Aufwand/Kosten für den Umstieg auf Apache Kafka

Anschließend stellt sich im nächsten Schritt die Frage, ob die Interoperabilität zu allen Systemen gegeben ist, die Berührungspunkte mit *Apache Kafka* besitzen. Da es sich bei diesen Systemen im Wesentlichen um Java-basierte Anwendungen handelt, ist die Interoperabilität gewährleistet.

Zuletzt hinterfragt der Leitfaden, ob in der aktuellen Situation ein Umstieg auf *Apache Kafka* unter Gesichtspunkten wie Fachkompetenz, Aufwand und Kosten realisierbar ist. Aufgrund der Unternehmensgröße trifft auch dies grundsätzlich zu und kann damit mit *Ja* beantwortet werden.

Mit der Beantwortung dieser letzten Frage spricht sich der Leitfaden grundsätzlich für den Einsatz von *Apache Kafka* aus. Abbildung 67 zeigt hierzu den durchlaufenen Pfad innerhalb des Leitfadens. Es handelt sich hierbei jedoch lediglich um eine Empfehlung, die im konkreten Fall noch detaillierter hinterfragt werden sollte.

9 Schlussbetrachtung

Dieses Kapitel stellt die gewonnen Ergebnisse und Erkenntnisse dieser wissenschaftlichen Arbeit dar. Im Fokus steht die wissenschaftliche Frage- bzw. Problemstellung, ob sich *Apache Kafka* als Ersatz für klassische Message Queuing Systeme eignet. In diesem Zusammenhang bietet der Abschnitt 9.1 auf Basis der Ergebnisse ein abschließendes Fazit über die Verwendung von *Apache Kafka*. Der darauf folgende Abschnitt 9.2 betrachtet die Ergebnisse im Verhältnis zu zukünftigen Anpassungen dieses Systems und formuliert potenzielle Forschungsfragen, denen im Rahmen von Folgeprojekten nachgegangen werden kann.

9.1 Fazit

Die Kernaufgabe dieser Arbeit bestand darin, Klarheit darüber zu gewinnen, inwieweit sich *Apache Kafka* als Ersatz für klassische Message Queuing Systeme eignet. In diesem Zusammenhang wurde analysiert, welchen Mehrwert der Einsatz von diesem System bietet. Bei einem dieser Aspekte handelt es sich um eine Mehrfach- und Offlineverarbeitung der Daten. Der Umgang mit den gespeicherten Nachrichten ermöglicht es, diesen Datenbestand mehreren Teilbereichen im System für verschiedene Verarbeitungsschritte zur Verfügung zu stellen. Dieser Mehrwert wurde im Rahmen dieser Arbeit nicht explizit untersucht, da er sich aus der vorhandenen Literatur bzw. den zugrunde liegenden *Kafka*-Konzepten ergibt.

Wesentlich relevanter war in Bezug auf die Kernfrage, wie sich das neue System unter steigender Last im Gegensatz zu bestehenden JMS-Systemen wie *ActiveMQ* verhält. In diesem Zusammenhang wurde verstärktes Augenmerk auf eine möglichst exakte Konfiguration der Test-Systeme gelegt. Diese sollte zwar eine bestmögliche Performance bieten, aber im weitesten Sinne identische Qualitätsanforderungen an die Übertragung der Daten stellen. Dieses Vorgehen war von signifikanter Bedeutung, um möglichst aussagekräftige Testergebnisse zu erzielen. Im Fall von *ActiveMQ* führt dies jedoch dazu, dass eine permanente Speicherung der Daten unvermeidbar ist.

Basierend auf einer identischen Ausgangslage wurden in beiden Systemen getrennt voneinander sowohl Tests durchgeführt, in denen Nachrichten in das System übertragen werden, als auch Tests, in denen Nachrichten vom Broker abgerufen werden. Hierbei wurden jeweils drei verschiedene Testszenarien durchgeführt. Die Testergebnisse zeigen, dass bei *ActiveMQ*, aufgrund von ineffizienten Speichervorgängen, der limitierende Faktor die IO-Operationen sind. Diese haben unter anderem zur Folge, dass sich in der CPU-Auslastung ein sehr hoher Anteil wiederfindet, in welchem die CPU auf die Fertigstellung von IO-Operationen warten muss. Dieses Phänomen sorgt außerdem dafür, dass das System ab einem gewissen Punkt die Verarbeitungsmenge nicht weiter steigern kann, obwohl noch CPU-Kapazität zur Verfügung steht.

Bei *Apache Kafka* zeigt sich aufgrund eines sehr effizienten Umgangs mit den Speichervorgängen ein komplett anderes Verhalten. Hier lassen sich die IO-Operationen um ein vielfaches steigern. Darüber hinaus wächst die Menge an IO-Operationen sowie die CPU-Auslastung linear mit der Anzahl an Nachrichten. Hierbei kann die CPU bis zu einem Wert von 100% ausgelastet werden. Der limitierende Faktor in diesem System ist damit die CPU.

Aus dieser Erkenntnis ergibt sich, dass sich *ActiveMQ* grundsätzlich schlechter vertikal skalieren lässt als *Apache Kafka*. Dies liegt daran, dass es deutlich aufwändiger ist (beispielsweise durch RAID) die Lese- bzw. Schreibgeschwindigkeit auf der Festplatte zu erhöhen, als mehr CPU-Leistung zur Verfügung zu stellen.

Die konkreten Performance-Werte zeigen einen Performance-Zuwachs um den Faktor 2 bei *Apache Kafka*. Das System schafft es, etwa 20.000 Nachrichten pro Sekunde entgegenzunehmen sowie 10.000 Nachrichten pro Sekunde auszuliefern. Bei *ActiveMQ* liegt diese Menge in beiden Fällen konstant bei etwa 5.000 Nachrichten pro Sekunde. *Kafka* ist zwar in der Lage, die vierfache Menge an Nachrichten entgegenzunehmen, schafft es jedoch nur die doppelte Menge im Vergleich zu *ActiveMQ* auszuliefern.

Eine wesentliche Problematik bei der Testdurchführung bestand darin, dass beide Systeme eine komplett andere Herangehensweise bei der Auslieferung der Nachrichten verfolgen. *ActiveMQ* sendet die Nachrichten aktiv an den Consumer. Im Fall von *Apache Kafka* hingegen müssen die Consumer die Nachrichten eigenständig anfordern.

Rückblickend betrachtet empfiehlt es sich, von Beginn an eine konsequente Trennung zwischen den beteiligten Komponenten Broker, Producer und Consumer vorzusehen. In anfänglichen Tests auf einem lokalen Computer kam es zu Verfälschungen der Testergebnisse, da sich die beteiligten Systeme gegenseitig die notwendigen Ressourcen entzogen. Dieses Verhalten war dahingehend kritisch, da die Producer und Consumer von *Apache Kafka* eine deutlich höhere CPU-Auslastung aufweisen als jene von *ActiveMQ*.

Im Anschluss an die durchgeführten Performance-Tests wurden die wesentlichen nicht-funktionalen Anforderungen, die im Rahmen von *Message Queuing Systemen* relevant sind, betrachtet. In diesem Zusammenhang hat sich gezeigt, dass *Apache Kafka* deutliche bessere Eigenschaften in Bezug auf die Skalierbarkeit und besonders auf die Verfügbarkeit besitzt.

Trotz sehr guter Performance-, Skalierbarkeits- und Verfügbarkeitseigenschaften zeigen sich bei *Apache Kafka* jedoch leichte Defizite in der Zuverlässigkeit sowie der Robustheit. Das System schafft es zwar einen Nachrichtenverlust zu vermeiden, kann jedoch nicht sicherstellen, dass Nachrichten exakt einmal beim Empfänger ankommen. Durch diese Problemstellung leitet sich eine weiterführende Fragestellung aus der Kernfrage ab. Diese befasst sich damit, ob sich mit *Kafka* überhaupt eine *Exactly-Once*-Zustellgarantie erreichen lässt. Diese Frage wurde im

Rahmen des 8. Kapitels zufriedenstellend beantwortet. In Verbindung damit wurde eine prototypische Optimierung für die clientseitige Software geschrieben, welche die Verarbeitung von Duplikaten unterbindet.

Im Anschluss an die Betrachtung der einzelnen nicht-funktionalen Anforderungen wurde ein Leitfaden definiert, der auf Basis dieser, eine Empfehlung bzw. eine Tendenz für einen der beiden Messaging-Ansätze ausspricht. Im darauf folgenden Kapitel 8 wurde dieser Leitfaden auf den konkreten Anwendungsfall der *OTTO (GmbH und Co KG)* abgebildet. Hierbei stellte sich heraus, dass für das Unternehmen ein Umstieg auf *Apache Kafka* erstrebenswert ist.

Abschließend ist anzumerken, dass es sich – nach Meinung des Autors – bei *Apache Kafka* um ein sehr vielversprechendes Projekt handelt, das komplett neue Möglichkeiten innerhalb der Verarbeitung und Übertragung von Nachrichten eröffnet. Es verfolgt dabei ein sehr aussichtsreiches Konzept, in dem zahlreiche Defizite aus klassischen Messaging-Systemen optimiert wurden. Darüber hinaus bietet der direkte Clustering-Ansatz eine sehr gute Grundlage für nicht-funktionale Anforderungen wie Verfügbarkeit und Skalierbarkeit. Dennoch sollte ein potenzieller Einsatz jedoch konsequent hinterfragt und ein Umstieg nur vollzogen werden, wenn ein entsprechender Mehrwert gegeben ist.

9.2 Ausblick

Im Rahmen dieser Arbeit lag der Fokus primär darin, beide Systeme unter möglichst gleichen Bedingungen zu betrachten. Aufgrund dieser Tatsache und den Anordnungen des betreuenden Unternehmens lag das Augenmerk dabei schwerpunktmäßig auf der Betrachtung einer *Exactly-Once* Zustellgarantie. Zwar wurden als Randbetrachtung auch Aspekte hinterfragt, die sich mit *non-persistent* Messaging befassen, jedoch könnten diese Untersuchungen im Rahmen einer weiterführenden Arbeit entsprechend intensiviert werden.

Darüber hinaus könnten verschiedene Persistence-Stores wie der *AMQ Message Store*, *KahaDB* und *LevelDB* genauer untersucht werden. An dieser Stelle wäre zu prüfen, ob sich klassische Systeme um ähnliche Mechanismen hinsichtlich der Verbreitung und Bereitstellung der Daten erweitern lassen. Ein Beispiel für die Erweiterung von *ActiveMQ* durch einen solchen Ansatz ist *Replicated LevelDB* [@rld]. Dieses Projekt verwendete ebenfalls *ZooKeeper*, um die Daten auf mehreren Systemen bereitzustellen. Dieses wurde jedoch nicht weiterentwickelt.

Abschließend ist im Rahmen des expliziten Ausblicks auf *Apache Kafka* damit zu rechnen, dass in den kommenden Releases und insbesondere im ersten Mayor-Release noch weitere Features und Optimierungen hinzukommen werden. Sofern für den eigenen Einsatzzweck noch essentielle Features fehlen, empfiehlt es sich, regelmäßig den *Future Release Plan* [@frp] von *Apache Kafka* zu überprüfen.

10 Zusammenfassung

Diese wissenschaftliche Arbeit wurde im Rahmen des Studiengangs *Verteilte und mobile Anwendungen* der Hochschule Osnabrück in Kooperation mit der *OTTO (GmbH & Co KG)* verfasst. Die Arbeit befasst sich mit der Fragestellung, ob sich die neuartigen und nicht JMS-konformen Messaging-Ansätze von *Apache Kafka* dazu eignen, klassische Messaging Systeme durch dieses Produkt zu ersetzen.

In diesem Zusammenhang wird im Rahmen des ersten Kapitels die grundsätzliche Problematik sowie der allgemeine Mehrwert einer *Message Oriented Middleware* (MOM) thematisiert.

Das zweite Kapitel beschreibt die relevanten Grundlagen, die für das weitere Verständnis der Arbeit förderlich sind. Darauffolgend werden im 3. Kapitel die beiden Systeme, *ActiveMQ* und *Apache Kafka*, sowie deren Konzepte detailliert beschrieben.

Anschließend wird im 4. Kapitel der Aufbau einer Test-Infrastruktur einschließlich drei verschiedener Testszenarien erläutert. Dieser Versuchsaufbau dient im anschließenden 5. Kapitel durch eine entsprechende Testdurchführung dazu, die maximalen Belastungsgrenzen beider Systeme unter möglichst gleichen Bedingungen zu ermitteln.

Im Rahmen des 6. Kapitels werden einzelne nicht-funktionale Anforderungen betrachtet, die im Zusammenhang von *Message Queuing Systemen* von Bedeutung sind. Zu diesen Anforderungen gehören unter anderem *Skalierbarkeit*, *Verfügbarkeit* und *Zuverlässigkeit*. Die Anforderungen werden anhand der Konzepte beider Systeme gegenüber gestellt, um eine Einschätzung darüber treffen zu können, wie gut sich diese jeweils realisieren lassen.

Im darauffolgenden Kapitel 7 wird ein Leitfaden definiert, der Unternehmen als Entscheidungsgrundlage dienen soll, inwieweit sich *Apache Kafka* für die eigenen Einsatzzwecke eignet. In Verbindung hiermit wird im ersten Schritt versucht, einen Mehrwert für den Einsatz von *Apache Kafka* zu identifizieren. Nachdem dieser erkannt wurde, werden entsprechende Risiken ausgeschlossen, die ein Scheitern des Umstiegs zur Folge hätten.

Zuletzt wird der konkrete Anwendungsfall der *OTTO (GmbH & Co KG)* auf den Leitfaden in Kapitel 8 projiziert und entsprechende Kompensierungsmaßnahmen geprüft.

Die zentralen Ergebnisse zeigen, dass *Apache Kafka* unter ähnlichen Zuverlässigkeitsbedingungen in der verwendeten Testinfrastruktur um den Faktor 2,15 effizienter im Vergleich zu *ActiveMQ* ist. Darüber hinaus hat sich gezeigt, dass sich die neuartigen Konzepte von *Apache Kafka* für einen Einsatz innerhalb der *OTTO*-Infrastruktur eignen.

11 Literatur

[DH03] Dunkel, Jürgen; Holitschke, Andreas; *Softwarearchitektur für die Praxis,* Berlin u.a. Springer, 2003.

[Gar15] Garg, Nishant; *Learning apache kafka,* Birmingham. Packt Publishing Limited, 2015.

[Ion15] Ionescu, Valeriu Manuel; *The analysis of the performance of RabbitMQ and ActiveMQ.* IEEE, 2015.

[JR16] Junqueira, Flavio; Reed, Benjamin; *ZooKeeper,* Cambridge, O'Reilly, 2016.

[Kle13] Kleuker, Stephan; *Grundkurs Software-Engineering mit UML,* 3. Auflage, Wiesbaden. Springer Fachmedien Wiesbaden, 2013.

[KNR11] Kreps, Jay; Narkhede, Neha; Rao, Jun; *Kafka - a Distributed Messaging System for Log Processing.* ACM, 2011.

[NSP17] Narkhede, Neha; Shapira, Gwen; Palino, Todd; *Kafka: The Definitive Guide,* Cambridge, Mass. O'Reilly, 2017.

[RMC09] Richards, Mark; Monson-Haefel, Richard; Chappell, David A.; *Java message service,* 2. Auflage, Cambridge, Mass. O'Reilly, 2009.

[Rup09] Rupp, Chris; *Requirements-Engineering und -Management,* 5. Auflage, München, Wien. Hanser, 2009.

[SS12] Schill, Alexander; Springer, Thomas; *Verteilte Systeme,* 2. Auflage, Berlin Heidelberg. Springer Berlin Heidelberg, 2012.

[SBD11] Snyder, Bruce; Bosnanac, Dejan; Davies, Rob; *ActiveMQ in action,* Greenwich, CT. Manning, 2011.

[Sta15] Starke, Gernot; *Effektive Softwarearchitekturen,* 7. Auflage, München. Hanser, 2015.

[TG02] Tran, Phong; Greenfield, Paul; *Behavior and Performance of Message-Oriented Middleware Systems.* IEEE, 2002.

[@apa] *Apache Software Foundation,* https://www.apache.org/. Abgerufen am 28.02.2017.

[@avro] *Apache Avro,* https://avro.apache.org/. Abgerufen am 28.02.2017.

[@cfl] *Confluent,* https://www.confluent.io/. Abgerufen am 28.02.2017.

[@cip] *Compound Patterns,* http://soapatterns.org/compound_patterns/overview. Abgerufen am 28.02.2017.

[@com] *ActiveMQ, Qpid, HornetQ und RabbitMQ im Vergleich,* https://www.predic8.de/activemq-hornetq-rabbitmq-apollo-qpid-vergleich.htm. Abgerufen am 28.02.2017.

[@frp] *Future Release Plan Apache Kafka,* https://cwiki.apache.org/confluence/display/KAFKA/Future+release+plan. Abgerufen am 28.02.2017.

[@jsr] *JSR 343: JavaTM Message Service 2.0,* https://jcp.org/en/jsr/detail?id=343. Abgerufen am 28.02.2017.

[@kd] *Apache Kafka Documentation,* https://kafka.apache.org/documentation/. Abgerufen am 28.02.2017.

[@lkd] *LinkedIn,* https://www.linkedin.com/. Abgerufen am 28.02.2017.

[@mq] *ActiveMQ,* http://activemq.apache.org/. Abgerufen am 28.02.2017.

[@omq] *About OpenMQ,* https://mq.java.net/about.html. Abgerufen am 28.02.2017.

[@omq2] *OpenMQ,* https://mq.java.net/. Abgerufen am 28.02.2017.

[@qos] *QoS-Level,* https://docs.oracle.com/cd/E24329_01/web.1211/e24375/design.htm#BRDGE135. Abgerufen am 28.02.2017.

[@rab] *RabbitMQ,* https://www.rabbitmq.com/. Abgerufen am 28.02.2017.

[@rjc] *RabbitMQ JMS Client,* https://www.rabbitmq.com/jms-client.html. Abgerufen am 28.02.2017.

[@rld] *Replicated LevelDB Store*, http://activemq.apache.org/replicated-leveldb-store.html. Abgerufen am 28.02.2017.

[@snc] *About SonicMQ,* https://documentation.progress.com/output/ua/OpenEdge_latest/index.html#page/dvesb/sonicmq.html. Abgerufen am 28.02.2017.

[@snc2] *Getting Started with SonicMQ,* https://docs.oracle.com/cd/E18867_01/SRE/GettingStarted.pdf. Abgerufen am 28.02.2017.

[@wsm] *WebSphere MQ,* http://www-03.ibm.com/software/products/de/ibm-mq. Abgerufen am 28.02.2017.

[@zoo] *Apache Zookeeper,* https://zookeeper.apache.org/. Abgerufen am 28.02.2017.

[Deg05] Degenring, Arne; *Enterprise Service Bus,* 2005, http://www.degenring.com/esb/degenring_JS_04_05.pdf. Abgerufen am 28.02.2017.

[GL13] Ghadir, Phillip; Landwehr, Arne; *Log-Daten effektiv verarbeiten mit Apache Kafka,* 2013, https://www.innoq.com/de/articles/2013/08/log-daten-verarbeiten-mit-kafka/ Abgerufen am 28.02.2017.

Anhang A Inhalt der CD

In der beigefügten CD sind folgende Ordner und Dateien enthalten.

Ordnerverzeichnis	Dateien	Beschreibung
\Masterarbeit	Masterarbeit_Blomberg.pdf	Die Masterarbeit im Portable Document Format (PDF)
	Masterarbeit_Blomberg.docx	Die Masterarbeit im Microsoft Word Format
\Abbildungen	Abbildung-**.jpg	Die in der Arbeit enthaltenen Abbildungen als JPG-Datei
\Internet-Quellen	*.jpg	Die in der Arbeit verwendeten Internetquellen als Screenshots.
\Software	Producer-Testsoftware/../*.*	*Eclipse*-Projekte mit der Producer-Testsoftware
	Consumer-Testsoftware/../*.*	*Eclipse*-Projekte mit der Consumer-Testsoftware
	Broker-Software/../*.*	Enthält die verwendete Broker-Software

Tabelle A.1: Inhalt der CD

Erklärung

Hiermit versichere ich, dass ich meine Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: …….....

(Unterschrift)