Hochschule für Telekommunikation Leipzig (FH)

Institut für Telekommunikationsinformatik

Abschlussarbeit zur Erlangung des akademischen Grades Bachelor of Engineering

Thema: Vergleich von Lastverteilungsmechanismen unter Einsatz der Messaging-Protokolle ZeroMQ und AMQP

Vorgelegt von: Martin Stoffers

geboren am: 07.07.1981

in: Borken

Themensteller: Institut für Telekommunikationsinformatik, HfTL
Gustav-Freitag-Str. 43-45
04277 Leipzig

Erstprüfer: Herr Prof. Dr.-Ing. Thomas Meier

Zweitprüfer: Herr Prof. Dr. Mathias Krause

Datum: 27. August 2014

Inhaltsverzeichnis

Abkürzungsverzeichnis

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
GPL	GNU General Public License
HTTP	HyperText Transfer Protocol
IEC	International Electrotechnical Commission
IPC	Inter Process Communication
IP	Internet Protocol
ISO	International Organization for Standardization
JMS	Java Message Service
JTC 1	Joint Technical Committee 1
LGPLv3+	GNU Lesser General Public License Version 3
MOM	Message Oriented Middleware
MSCs	Message Sequence Charts
OASIS	Organization for the Advancement of Structured Information Standards
PGM	Pragmatic General Multicast
QoS	Quality of Service
SASL	Simple Authentication and Security Layer
SCTP	Stream Control Transmission Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
VMs	Virtuellen Maschinen
ZMTP	ZeroMQ Message Transport Protocol

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Message Oriented Middleware (MOM) gilt innerhalb des Business-Umfelds als zuverlässige Lösung zur synchronen sowie asynchronen Verteilung von Daten zwischen verschiedenen Anwendungen in verteilten Systemen. Die Entwicklung von leistungsstarken und autarken Webanwendungen, welche auf einen klassischen Webserver verzichten, rückt Messaging auch in diesem Bereich immer weiter in den Fokus der Entwickler. Die Kombination von MOM und Webanwendung verspricht eine schnelle und effektive Kommunikation über klar definierte Schnittstellen zwischen verschiedenen Webanwendungen. Die im Business-Umfeld genutzte MOM basiert allerdings häufig auf proprietären Protokollen und bietet nur eine API zur Nutzung an. Mit dem Advanced Message Queuing Protocol (AMQP) und ZeroMQ gibt es zwei Lösungen, welche auf eine offene Spezifikation setzen und somit über Implementierungen in vielen Programmiersprachen verfügen. Dies begünstigt den Einsatz im Webumfeld maßgeblich, da gerade dort verschiedenste Softwarelösungen zum Einsatz kommen.

Die vorliegende Arbeit soll die Protokolle AMQP und ZeroMQ in Bezug auf ihre Lastverteilungsmechanismen auf der Anwendungsebene untersuchen. Anhand von zwei Anwendungen aus dem Webumfeld soll das tatsächliche Lastverhalten mit Hilfe von Tests ermittelt und ausgewertet werden. Dabei soll ein besonderes Augenmerk auf das Verhalten bei wachsender Komplexität und Belastung der zu testeten Anwendungen gelegt werden.

Dazu werden in Abschnitt 2, nach einer grundlegenden Einordnung, zunächst die prinzipiellen Funktionsweisen der jeweiligen Protokolle beschrieben. Daran schließt sich eine Betrachtung der daraus resultierenden Lastverteilungsmechanismen an. In Abschnitt 3 werden zunächst Anforderungen und Festlegungen getroffen, welche der einheitlichen Implementierung und dem Aufbau des Testsystems sowie beider Anwendungen dienen. Daran angeschlossen erfolgt die Auswahl der Testumgebung und der zu testenden Anwendungen sowie eine Auswahl der Programmiersprache. Nachfolgend werden in Abschnitt 4 genauere Anforderungen für die Konzeption des Testsystems und des Loggings entwickelt. Danach erfolgen die Konzeption und Implementierung dieser Systemkomponenten. In den Abschnitten 5 und 6 wird zunächst auf die speziellen Anforderungen und die Konzeption der ausgewählten Anwendungen eingegangen. Anschließend erfolgen Implementierung, Testplanung sowie Testdurchführung und eine Auswertung der Ergebnisse. Abschließend wird in Abschnitt 7 ein testübergreifender Überblick gegeben, welcher mögliche Verbesserungen beider Anwendungen sowie des Testsystems aufzeigen soll.

2 Einführung in AMQP und ZeroMQ

2.1 AMQP

2.1.1 Einordnung

Die Entwicklung des Anwendungsprotokolls AMQP wurde 2003 von JPMorgan Chase mit dem Ziel angestoßen eine standardisierte Kommunikation zwischen den Anwendungen des Unternehmens zu schaffen. Im Gegensatz zu den bis dahin verfügbaren MOM-Systemen sollte das Protokoll dabei eine Interoperabilität zwischen den verschiedensten Middleware-Architekturen gewährleisten[1, 2]. Zusätzlich wurde beschlossen, dass wegen der großen Verbreitung der Java Message Service (JMS)-API alle darin enthaltenen Eigenschaften auch im AMQP abgebildet sein sollten, um einen einfach Umstieg zu ermöglichen[3]. In Folge dessen, wurde von Mitte 2004 bis Mitte 2006 die iMatix Corporation mit der initialen Entwicklung und Dokumentation des AMQPs beauftragt[3]. Mit der Beteiligung weiterer Unternehmen, wurden die Ziele und Spezifikationen des Protokolls an die Erfordernisse eines unternehmensübergreifenden Datenaustausches angepasst. Dies erhöhte nach Ansicht von Peter Hintjens die Komplexität des Protokolls erheblich.[4] Nach der Eingliederung in die Organization for the Advancement of Structured Information Standards (OASIS) im August 2011 erfolgte die Gründung des technischen Komitees zur Formung eines offenen ISO/IEC-Standards für AMQP[5, 6]. Seit Ende April 2014 ist AMQP in der Version 1.0 als ISO/IEC-Standard von der JTC 1 freigegeben[7]. Der AMQP Working Group gehören zur Zeit 11 namhafte Unternehmen an. Darunter neben JPMorgan Chase die Bank of America, Deutsche Boerse, Microsoft, Red Hat und VMware[8]. Als eines der ersten kommerziellen Einsatzgebiete des AMQP werden Trading-Anwendungen, der an der Entwicklung beteiligten Börsen und Bankinstitute, genannt[9, 10]. Mittlerweile existieren für das AMQP APIs in allen gängigen Programmiersprachen. Darunter C, C#, Erlang, Java, Python, .NET und Perl.

2.1.2 Funktionsprinzipien

Das AMQP wurde als binäres Anwendungsprotokoll entworfen und setzt auf das darunterliegende Transmission Control Protocol (TCP) auf[2, S. 26]. Die Nutzung des TCP sorgt hierbei für die verlässliche Übertragung der Nachrichten zum Empfänger. Zur Absicherung des Datenverkehrs können Transport Layer Security (TLS) und Simple Authentication and Security Layer (SASL) als Transportverschlüsselungen genutzt werden[11, 12][2, S. 106-112]. Wie auch JMS unterstützt AMQP transaktionale Nachrichten und damit den sicheren und validierbaren Austausch von Nachrichten[2, S. 95]. In diesem Zusammenhang sei auf die sogenannten Message-Acknowledgements verwiesen, welche dem später beschriebenen Broker die Erkennung einer fehlerhaften Nachrichtenverarbeitung ermöglichen[13]. Mit dem bereits erwähnten binären Aufbau gehört das AMQP, in Bezug auf den Durchsatz, zu den effizientesten Protokollen im Messaging. Der maximale Durchsatz wird mit einigen hunderttausend Nachrichten pro Sekunde angegeben[14].

AMQP nutzt wie die meisten anderen Messagingsysteme einen zentralen Server, den sogenannten Broker. Der Broker übernimmt, wie in Abbildung 1 zu sehen, innerhalb des Systems das zentrale Routing der Nachrichten. Auch die persistente Speicherung der Nachrichten zur Unterstützung der gängigen Fehlersemantiken, wie At-Most-Once, erfolgt im Broker. Der bekannteste und erfolgreichste Broker für den Einsatz des AMQP ist der RabbitMQ-Broker von Pivotal Software[15]. Er ist in der Programmiersprache

Erlang geschrieben und steht unter der Mozilla Public License. Die Nachrichten in einem AMQP-System werden durch Producer generiert und an den Broker gesendet. Im Kontext von Messaging spricht man dabei von Publishen. Die so erzeugten Nachrichten werden über einen Virtualhost innerhalb des Broker, mit Hilfe eines Exchange über Queues an die Consumer zugestellt. Producer und Consumer können dabei auch von nur einer Anwendung realisiert sein. Die entstehende Sterntopologie vereinfacht den Aufbau des Systems erheblich, da innerhalb von Producer und Consumer nur die Adresse des Brokers bekannt sein muss. Auch die Komplexität des System und die damit direkt im Zusammenhang stehende Gesamtzahl aller Verbindungen wird so stark reduziert. Jedoch birgt dieses Konzept den Nachteil, dass ein Ausfall des zentralen Servers das ganze System betrifft. Zusätzlich begrenzt die Leistungsfähigkeit des Brokers den maximalen Durchsatz des gesamten Systems. Durch das Clustern von mehreren Brokern kann dieser Nachteil aber ausgeglichen werden. Im Folgenden soll ein kurzer Einblick in die Funktionen von Virtualhost, Queue und Exchange gegeben werden. Anschließend wird anhand eines Beispiels der prinzipielle Verbindungsaufbau von Producer und Consumer zum Broker sowie das sogenannte Binding zwischen Queue und Exchange gezeigt.

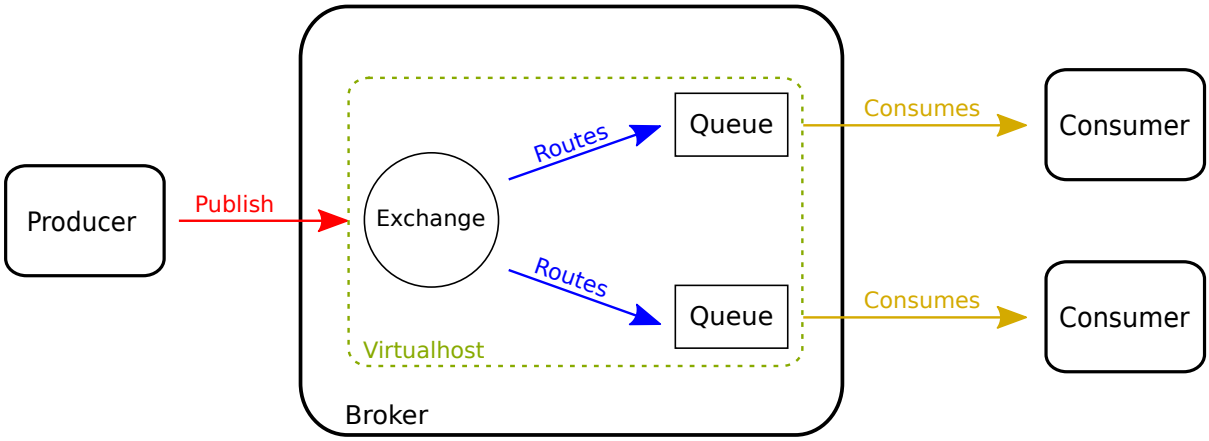

Abbildung 1: AMQP-System mit allen wesentlichen Komponenten (Nach: [16])

Virtualhost Ein Virtualhost, wie er in Abbildung 1 angedeutet ist, übernimmt innerhalb des Brokers vergleichbare Aufgaben wie ein Virtualhost im HyperText Transfer Protocol (HTTP). Er trennt verschiedene Systeme logisch voneinander und schafft so getrennte Systeme[13]. Der Zugang zu einem Virtualhost kann über eine optionale Authentifizierung eingeschränkt werden.

Queue Eine Queue innerhalb des Brokers eines AMQP-Systems unterscheidet sich in der Verwendung kaum von den Queueing-Konzepten anderer Messagingsysteme. Ihre wesentliche Aufgabe ist auch hier das Zwischenspeichern von Nachrichten, welche für einen oder mehrere Consumer bestimmt sind. Sie können sowohl dynamisch durch Consumer oder Producer erzeugt werden, aber auch statisch vom Broker vorgegeben sein. Sind mehrere Consumer mit einer Queue verbunden, so erfolgt die Zustellung der Nachrichten nach dem Round-Robin-Prinzip[13]. Dabei kann eine Queue die in Tabelle 1 aufgelisteten Eigenschaften besitzen.

Eigenschaft	Beschreibung
Name	Name der Queue
Durable	Queue und Nachrichten bleiben nach einem Neustart des Brokers erhalten
Exclusive	Queue für einen Consumer; Wird nach Verbindungsende gelöscht
Auto-delete	Queue für mehrere Consumer; Wird nach dem Ende aller Verbindungen gelöscht
Arguments	Frei verfügbare Argumente für optionale Eigenschaften

Tabelle 1: Eigenschaften einer Queue

Exchange Dem Exchange kommt innerhalb des Brokers in etwa die Rolle eines Routers zu. Er kann, genau wie eine Queue, dynamisch erzeugt werden, oder durch den Broker statisch vorgegeben sein. Ein Exchange kann auch durch mehrere Producer genutzt werden und besitzt, genau wie die Queue, verschiedene Eigenschaften. Diese sind in Tabelle 2 aufgelistet. Die fünf Exchangetypen `Default-`, `Direct-`, `Fanout-`, `Topic-`, und `Headers-`Exchange helfen dabei die unterschiedlichen Pattern eines Messagingsystems innerhalb von AMQP umzusetzen[13]. Die Funktionsweise der wichtigsten vier Exchangetypen wird nachfolgend kurz beschrieben.

Eigenschaft	Beschreibung
Name	Name des Exchange
Typ	Exchange Typ
Durability	Exchange bleibt nach Neustart des Brokers erhalten
Auto-delete	Exchange wird gelöscht, wenn keine Queue mehr verbunden ist
Arguments	Frei verfügbare Argumente für optionale Eigenschaften

Tabelle 2: Eigenschaften eines Exchange

Default-Exchange Der Default-Exchange dient der Umsetzung einfacher Anwendungen in denen das direkte Versenden von Nachrichten ohne Routing ausreicht. Jede mit einem Namen erzeugte Queue, die ohne Angabe eines Exchange erzeugt wird, wird mit diesem verbunden[13]. Dies kann sowohl durch Producer als auch Consumer erfolgen. Das Routing der Nachrichten erfolgt dabei über die Angabe des Queuenames als Routingkey innerhalb der vom Producer versendeten Nachricht. Wie Abbildung 2 zeigt, wird der Exchange niemals direkt durch den Producer angesprochen. Die Kommunikation mit dem Exchange erfolgt also transparent.

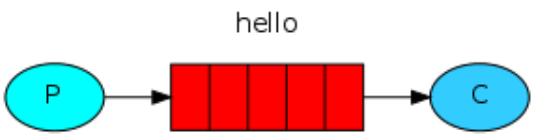

Abbildung 2: Default-Exchange (Entnommen von: [17])

Direct-Exchange Ein Direct-Exchange stellt die eintreffenden Nachrichten anhand eines Routingkeys an die verbundenen Queues zu. Dazu wird bei der Erzeugung einer Queue, die mit diesem Exchange verbunden werden soll, ein sogenannter Bindingkey angegeben[13]. Sind der Bindingkey der Queue und der Routingkey der Nachricht im Exchange identisch, wird die Nachricht in die Queue gelegt. Wie in Abbildung 3 gezeigt, kann eine Queue auch Nachrichten mit unterschiedlichen Routingkeys über mehrere Bindings auswählen[13].

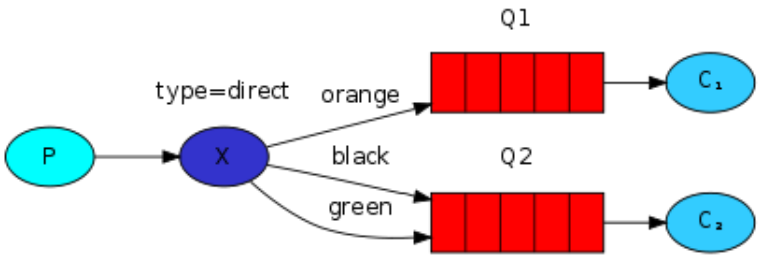

Abbildung 3: Direct-Exchange (Entnommen von: [18])

Topic-Exchange Die Funktionsweise eines Topic-Exchange, gleicht der eines Direct-Exchange. Auch er routet die Nachrichten anhand eines Routingkeys. Allerdings ist der Routingkey komplexer aufgebaut und besteht aus einer verketteten Wortliste, welche durch Punkte getrennt wird[13]. Eine mit diesem Exchange verbundene Queue empfängt verschiedene Gruppen aus dieser Wortliste über einen speziell angepassten Bindingkey. Wie in Abbildung 4 zu sehen ist, werden hierbei die Ausdrücke * und # verwendet[13]. Der Stern kennzeichnet ein beliebiges Wort an der angegebenen Stelle innerhalb der Wortliste. Das Doppelkreuz kann eingesetzt werden um mehrere aufeinander folgende Wortgruppen gemeinsam auszuwählen und so alle Nachrichten unterhalb der angegeben Struktur auszuwählen. Das Routing mittels Topic-Exchange eignet sich gerade deshalb für komplexere Anwendungsfälle.

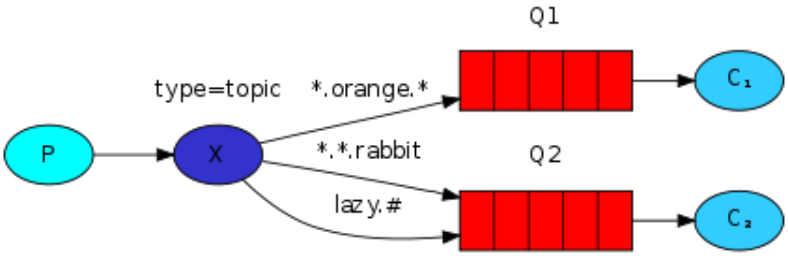

Abbildung 4: Topic-Exchange (Entnommen von: [19])

Fanout-Exchange Der Fanout-Exchange entspricht dem Publish/Subscribe-Pattern anderer Messagingsysteme. Alle eintreffenden Nachrichten am Exchange werden an alle mit ihm verbundenen Queues verteilt. Ein möglicherweise angegebener Routingkey in der Nachricht wird hierbei ignoriert[13]. Dieser kann dennoch im Consumer für die Zuordnung der Nachricht zu einem Producer genutzt werden. Der prinzipielle Aufbau dieses Pattern ist in Abbildung 1 auf Seite 3 dargestellt.

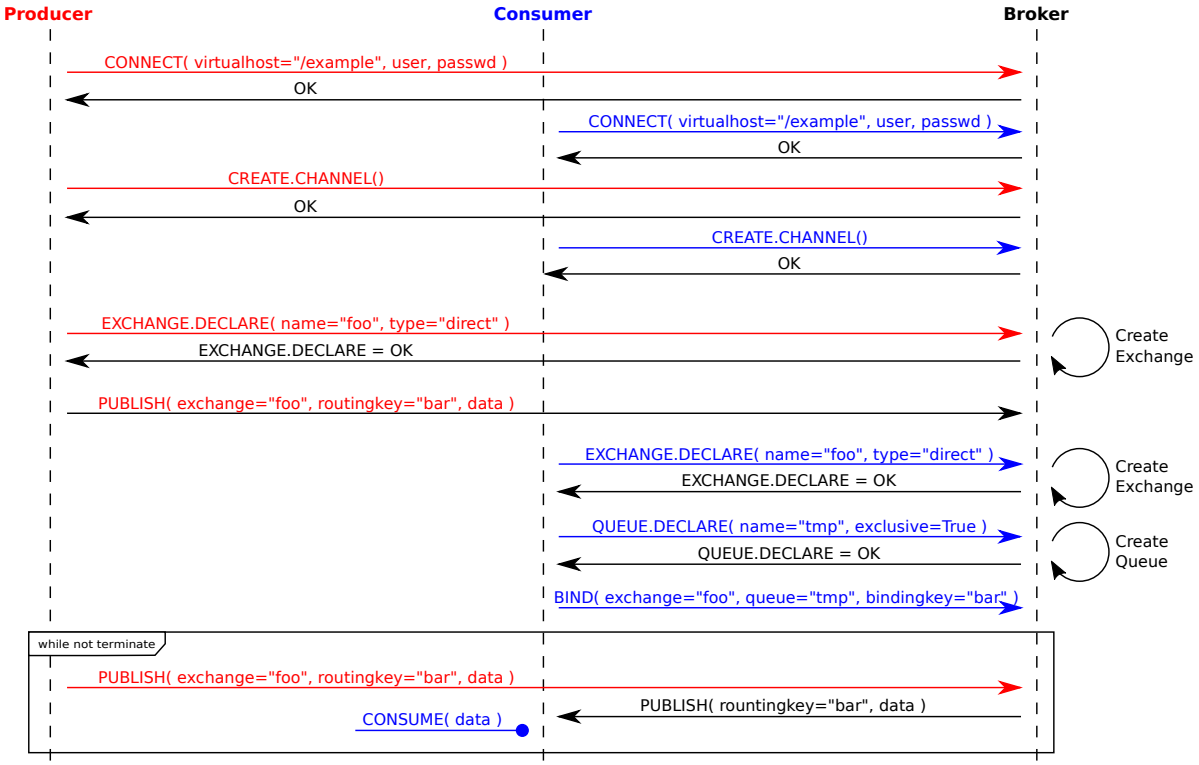

Abbildung 5: Verbindungsaufbau und Binding am Beispiel eines Direct-Exchange

Systembeispiel Die Verbindung von Producer und Consumer mit einem Broker erfolgt grundsätzlich nach dem gleichen Schema. Je nachdem welches Pattern das Messagingsystem umsetzen soll, unterscheiden sich aber die von Producer und Consumer ausgewählten Eigenschaften von Queue und Exchange. In diesem Beispiel soll exemplarisch die Verbindung von Consumer und Producer über einen Broker mit Direct-Exchange erläutert werden. Wie in Abbildung 5 erkennbar ist, erfolgt im ersten Schritt zunächst der Verbindungsaufbau des Producers und Consumers zum AMQP-Broker. Dazu geben beide Komponenten den zu nutzenden Virtualhost des Brokers an. Erfordert die Verbindung zum Virtualhost eine Authentifizierung, so geschieht dies im selben Schritt. Nachdem die Verbindung aufgebaut wurde, öffnen sowohl Consumer als auch Producer einen Channel innerhalb der bestehenden Verbindung. Um Ressourcen zu schonen, können über eine Verbindung mehrere Channel gleichzeitig betrieben werden. Die Kommunikation und Datenübertragung findet im Channel als eigene Session statt.

Nach dem Aufbau des Channels erfolgt durch den Producer die Erzeugung eines neuen Exchange vom Typ *Direct*, unter Angabe eines Namens, im Broker. Dabei ist es unerheblich, ob der Exchange bereits existiert oder zuvor durch den Broker erzeugt werden muss. Nach der Bestätigung durch den Broker kann der Producer beginnen Nachrichten an diesen Exchange zu schicken. Dazu setzt er in jeder Nachricht einen Routingkey.

Damit nun der Consumer die Nachrichten empfangen kann, schickt auch er an den Broker die Aufforderung zur Erzeugung des Exchange. Anschließend erfolgt die Anforderung einer Queue durch Angabe ihres Namens und des Exchange mit dem diese Queue verbunden werden soll. Dabei ist es möglich eine sogenannte *Nameless*-Queue vom Broker anzufordern. Der Name der Queue wird dann zufällig im Broker erzeugt und dem Consumer anschließend mitgeteilt. Lediglich das Anfordern einer bereits existierenden Queue mit der Eigenschaft *Exclusive* ist nicht möglich, da diese nur durch einen Consumer genutzt werden darf. Ist dies erfolgt, teilt der Consumer dem Broker durch das Binding mit, welche Nachrichten er

empfangen möchte. Dabei gibt der Consumer einen Bindingkey an, der dem Broker die Auswahl und Zuordnung der Nachrichten über den Routingkey des Exchange ermöglicht. Der letzte Schritt kann durch den Consumer beliebig oft mit unterschiedlichen Bindingkeys wiederholt werden.

2.1.3 Lastverteilungsmechanismen

In einem Messagingsystem mit Broker erfolgt die Lastverteilung immer über den zentralen Broker. Die Effizienz und Skalierbarkeit des gesamten Systems hängt also stark von der Leistungsfähigkeit dieser Komponente ab. Möchte man beispielsweise den Durchsatz des Systems erhöhen, ohne die Leistungsfähigkeit in Consumer oder Producer zu maximieren, so kommt nur eine Veränderung am Broker in Betracht. In diesem Fall ist es möglich, den Broker durch bessere Hardware leistungsfähiger zu machen. Üblicherweise wird aber das Zusammenschalten von Brokern zu Clustern diesem Ansatz vorgezogen. Wie in Abbildung 6 zu sehen ist, hat dieses Konzept den Vorteil, dass der vorhandene Broker direkt entlastet wird, da er nun weniger Verbindungen verwalten muss. Gleichzeitig wird die Ausfallsicherheit des Systems durch den zusätzlichen Broker erhöht.

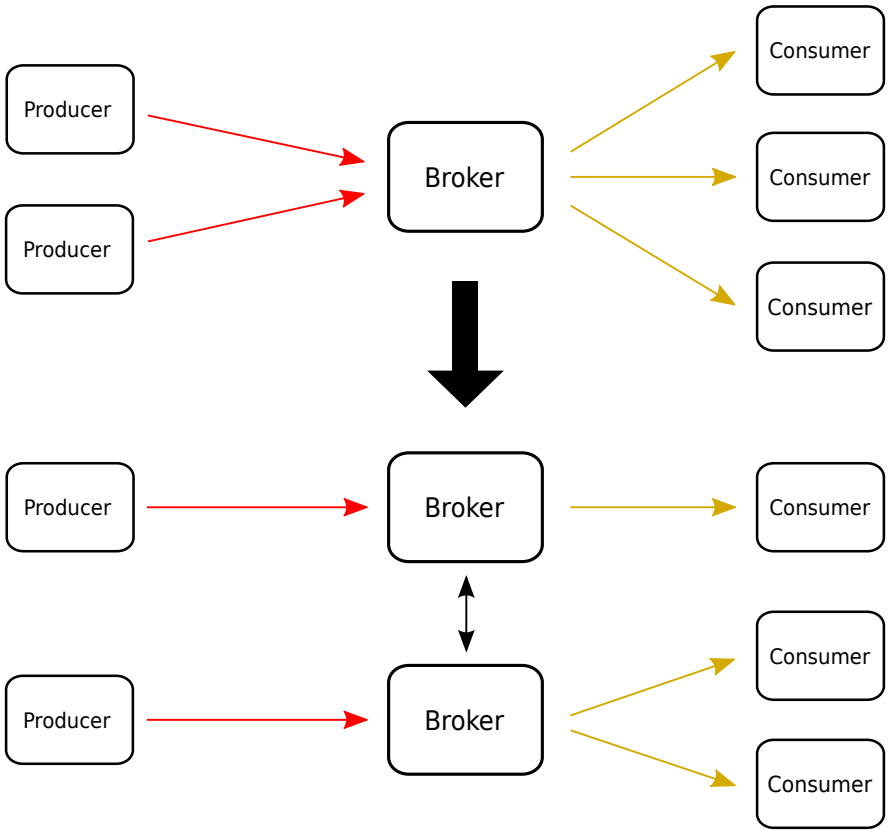

Abbildung 6: Skalierung eines AMQP-Systems durch Clustern von Brokern

Wie die Abarbeitung von Nachrichten, und damit die Verteilung der Last innerhalb des Brokers zu implementieren ist, wird im AMQP nicht explizit vorgeschrieben und unterscheidet sich deshalb in jeder Broker-Implementierung. Da in den Testaufbauten auf einen RabbitMQ-Broker zurückgegriffen wird, soll dessen allgemeines Verhalten hier dennoch kurz erläutert werden. Eine definierte Verteilung der Nachrichten im RabbitMQ-Broker mittels Round-Robin-Verfahren erfolgt nur über die Consumer an einer Queue[13]. Die Verteilung von Nachrichten an die Queues selbst, scheint dagegen vom Eintreffzeitpunkt der Nachrichten am Exchange abhängig zu sein. Wird also eine einfach Task-Queue für mehrere Consumer erzeugt, so werden die Nachrichten nacheinander an alle Consumer zugestellt und die Last damit fair verteilt. Werden die Nachrichten dagegen auf mehrere Queues mit je einem Empfänger über einen Fanout-Exchange verteilt, so erfolgt keine direkte Lastverteilung. Die Queues werden verschieden stark belastet, wenn die Consumer die Nachrichten unterschiedlich schnell verarbeiten. Neben den im Broker implementierten Lastverteilungsmechanismen bietet das AMQP weitere Möglichkeiten zur direkten Steuerung des Nachrichtenflusses und des Consumerverhaltens.

Durch die Nutzung des *Consumer-Prefetch* kann der Nachrichtenaustausch zwischen Consumer und Broker beeinflusst werden[20]. Dieses Quality of Service (QoS) Merkmal dient dazu die Anzahl, der noch nicht vom Consumer bestätigten Nachrichten zu begrenzen und steht im direkten Zusammenhang mit Message-Acknowledgements und Transaktionen. Es hat damit aber auch einen Einfluss auf die Lastverteilung zwischen Consumer und Broker, da die Anzahl der Nachrichten im Consumer, durch den sogenannten *Prefetch-Count*, begrenzt wird. Für den Consumer bedeutet dies eine Entlastung. Wird dieses Merkmal nicht aktiviert, so ruft der Consumer jede verfügbare Nachricht direkt aus der Queue ab. Dies kann bei vielen kleinen Nachrichten zu einer Überlastung des Consumers führen, da dieser die Nachrichten zwischenspeichern muss. Die Nutzung dieses Merkmals setzt voraus, dass im Consumer die Bestätigung von Nachrichten explizit eingeschaltet wird. In Bezug auf die Last des Brokers ist eine Einschätzung des Verhaltens nicht so leicht abzuschätzen. Mit der Aktivierung des Consumer-Prefetch wird die Abarbeitung der Nachrichten innerhalb der Queue verlangsamt und der Broker zusätzlich belastet. Da aber weniger unbestätigte Nachrichten im Broker zu verwalten sind, kann auch von einer Entlastung des Brokers ausgegangen werden. Hierbei wird vermutet, dass sich der Consumer-Prefetch nicht negativ auf den Broker auswirkt, da die Queue zu den effizientesten Komponenten im Broker gehören.

Eine weitere Eigenschaft des AMQP ist, dass für jeden Consumer an einer Queue eine Priorität gesetzt werden kann[21]. Damit wird eine Verteilung der Last an die Consumer in Abhängigkeit ihrer Leistungsfähigkeit möglich. Die Priorität wird dabei von jedem Consumer selbst festgelegt. Der Broker verteilt folglich die Nachrichten nicht mehr nach dem Round-Robin-Verfahren an die Consumer.

Beide zuvor genannten Eigenschaften des AMQP können deutliche Auswirken auf das System haben, wenn sie gemeinsam genutzt werden. So bewirkt ein zu niedriger *Prefetch count* eines Consumers mit hoher Priorität, dass alle anderen Consumer mit niedriger Priorität wesentlich früher belastet werden. Geht man davon aus, dass die zuletzt genannten Consumer weniger leistungsfähig sind, kann dies eine erhebliche negative Auswirkung haben. Die Veränderung dieser Werte sollte also unter Beachtung der Parameter, wie dem maximalen Nachrichtenaufkommen, im System vorgenommen werden.

2.2 ZeroMQ

2.2.1 Einordnung

Die Entwicklung von ZeroMQ durch die iMatix Corporation startete 2007 mit dem Ziel ein leichtgewichtiges und gut zu skalierendes verteiltes Messagingsystem zu entwickeln[22]. Mitte August 2008 folgte die erste Veröffentlichung der libzmq in Version 0.3, welche den Kern des Systems bildet. Als Inspiration diente der Aufbau des Internet Protocol (IP) sowie die, laut Peter Hintjens, gemachten Fehler beim Design des AMQP[23, 4]. Im Juli 2009 erschien Version 1.0 des ZeroMQ und wurde ein Jahr später durch Version 2.0 abgelöst[23] Im Gegensatz zum AMQP stehen sowohl Quellcode als auch alle Spezifikationen zu ZeroMQ vollständig unter der GNU General Public License (GPL) beziehungsweise der GNU Lesser General Public License Version 3 (LGPLv3+)[24]. Auf Grund der offenen Lizenzen, einer starken Community und der ausführlichen Dokumentation, existieren für alle populären Programmiersprachen entsprechende Module oder Bibliotheken. Neben C sind dies zum Beispiel Java, C#, C++, Erlang, Haskell, Python, und Perl[25]. Als prominentestes Einsatzgebiet des ZeroMQ wird das CERN genannt. Nach einer Analyse verschiedener Messagingsysteme im Jahr 2011 wurde beschlossen die Middleware zur Steuerung der Collider auf ZeroMQ ab dem Jahr 2012 umzustellen[26, 27]. Seit März 2014 ist ZeroMQ in der Version 4.04 verfügbar[28].

2.2.2 Funktionsprinzipien

Wie bereits im vorherigen Abschnitt angedeutet, unterscheidet sich der Aufbau eines ZeroMQ-Systems erheblich von denen anderer Messagingsysteme. ZeroMQ nutzt, wie in Abbildung 7 angedeutet, ausschließlich die sogenannten ZeroMQ-Sockets, zur direkten Verbindung der einzelnen Anwendungen untereinander. Dabei kann, wie bei App C zu sehen, ein Socket auch mehrere Verbindungen und unterschiedliche Messaging-Pattern gleichzeitig verwalten. Auf einen fest definierten Broker als zentrale Komponente des Systems wird verzichtet.

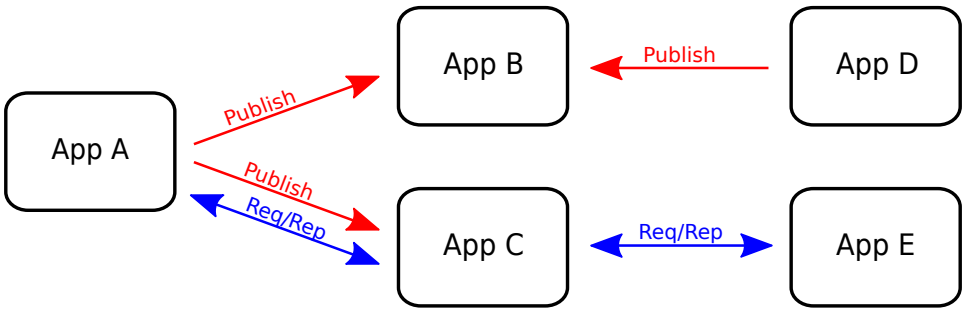

Abbildung 7: Exemplarischer Aufbau eins ZeroMQ-Systems

Das Application Programming Interface (API) der ZeroMQ-Sockets ist bewusst an den Aufbau von Unix-Sockets angelehnt und soll den Entwicklern die Arbeit mit dem System erleichtern. Im Unterschied zu Unix-Sockets kann über das API eines ZeroMQ-Socket aber bereits eine Verbindung zu einer noch nicht existierenden Gegenstelle erzeugt werden. Dies steigert die Flexibilität des System maßgeblich und sorgt gleichzeitig für eine größere Fehlertoleranz. Das eigentliche Handling der Verbindungen sowie Auf- und Abbau erfolgen dabei innerhalb des ZeroMQ-Sockets und werden damit vor der Anwendung verborgen.

Auch die Verwaltung und Zwischenspeicherung von noch nicht verarbeiteten Daten erfolgt im Socket selbst. Auf Grund dieser Tatsachen wird ZeroMQ auch als „Socket based messaging“ oder „Brokerless messaging“bezeichnet[29]. Die Kommunikation zwischen den Sockets erfolgt mittels des ZeroMQ Message Transport Protocol (ZMTP), welches meistens direkt auf das TCP aufsetzt[30]. Neben den üblichen Transportprotokollen werden zusätzlich das Stream Control Transmission Protocol (SCTP) Pragmatic General Multicast (PGM), inproc, Inter Process Communication (IPC) und SOCKS5 unterstützt[31]. Mit inproc und IPC kann ZeroMQ somit auch zur Kommunikation zwischen verschiedenen Threads einer Anwendung oder den Prozessen eines Computers eingesetzt werden.

ZeroMQ unterstützt vier grundlegende Messaging-Pattern. Dazu gehören Publish/Subscribe, Request/Replay, Pipeline und Exclusive Pair[29]. Die Umsetzung erfolgt mittels der nachfolgend genannten Sockettypen.

- Publish- und Subscribe-Socket
- XPUB- und XSUB-Socket
- Request- und Replay-Socket
- Push- und Pull-Socket
- Router- und Dealer-Socket
- Pair-Socket

Seit dem Erscheinen des ZMTP in Version 3.0 unterstützt ZeroMQ die Verschlüsselung der Transportwege mit SASL[30]. Zusätzlich wird über eine weitere Absicherung der Kommunikation mit einem Zertifikatssystem im ZMTP nachgedacht[32]. Der maximale Durchsatz eines ZeroMQ-Systems wird mit einigen Millionen Nachrichten pro Sekunde bei hunderten von mit einander verbunden Sockets angegeben[14]. Trotz all dieser Fähigkeiten wird Wert auf eine sehr geringe Komplexität des ZMTP gelegt. Typische Eigenschaften anderer brokerbasierter Messagingsysteme, wie Transaktionen und Hochverfügbarkeit, sind im ZeroMQ nicht explizit vorgesehen. Die vollständige Zustellung von Nachrichten wird viel mehr dem gewählten Transportprotokoll überlassen. Gerade die Nutzung des PGM als reliable Multicastprotokoll kann somit bei der Umsetzung dieser Eigenschaften helfen. Zusätzlich existieren bereits Verweise auf bestehende Frameworks und Konzeptbeispiele im ZeroMQ-Guide[29]. Auf Grund des Umfanges soll in dieser Arbeit aber nicht weiter auf diese Problematik eingegangen werden.

Der Aufbau eines ZeroMQ-Systems hat jedoch, neben der fehlenden Unterstützung von Transaktionen, zwei weitere offensichtliche Nachteile gegenüber den Broker basierten Systemen. Ein Problem ist, dass die Komplexität eines ZeroMQ-Systems schnell sehr groß werden kann. Insbesondere dann, wenn das System aus vielen verschiedenen Akteuren mit unterschiedlicher Funktionalität besteht. Ein weiteres Problem ist, dass jeder Anwendung im System immer alle Anwendungen bekannt sein müssen, mit denen sie sich verbinden sollen. Dieses führt zu einer signifikanten Erhöhung des Aufwandes bei der Verwaltung eines ZeroMQ-Systems. Denn bei einer Erweiterung des Systems müssen gegebenenfalls mehrere Anwendungen gleichzeitig angepasst werden.

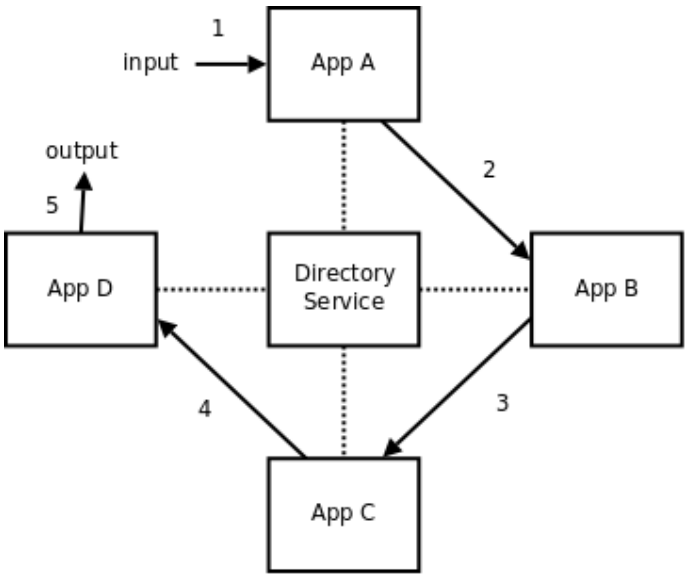

Abbildung 8: ZeroMQ-System mit Directory-Service (Entnommen von: [33])

Zur Lösung dieses Problems wird beispielsweise vorgeschlagen einen zentralen Directory-Service zu nutzen (s. Abbildung 8). Dieser teilt den Anwendungen dann die Adressen der anderen Anwendungen mit[33]. So ist sichergestellt, dass bei einer Änderung der Adressvergabe dennoch alle Anwendungen miteinander kommunizieren können. Im Folgenden soll ein kurzer Einblick in das Funktionsprinzip des ZeroMQ-Sockets gegeben werden. Dazu wird zunächst ein Blick auf das Nachrichtenformat von ZeroMQ geworfen.

Nachrichtenformat Nachrichten in einem ZeroMQ-System setzen sich aus einem oder mehreren Frames zusammen. Eine Nachricht, welche aus mehreren Frames besteht wird als Multipart-Nachricht bezeichnet. Wie in Abbildung 9 zu sehen ist, kann eine solche Nachricht um beliebige Frames erweitert werden. Damit kann der Aufbau der Nachrichten an die jeweiligen Erfordernisse eines Systems angepasst werden. Das Konzept der Multipart-Nachrichten setzt aber voraus, dass allen Anwendungen der Aufbau der Nachrichten bekannt ist. Ein Veränderung im Nachrichtenaufbau bedingt also schlimmsten Falls die Anpassung eines großen Teils des Messagingsystems

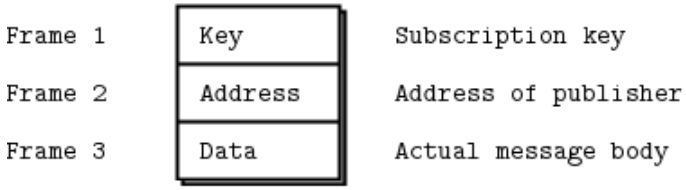

Abbildung 9: Möglicher Aufbau Multipart-Nachricht (Entnommen von: [34])

Systembeispiel Das in Abbildung 10 abgebildete System soll, anhand des Messaging-Pattern Publish/Subscribe, die Arbeitsweise der ZeroMQ-Sockets verdeutlichen. Anwendung B und C sollen, als Subscriber, Nachrichten in Abhängigkeit eines Keys empfangen. Diese Nachrichten werden durch Anwendung A generiert und an alle verbundenen Anwendungen verteilt. Sie agiert in diesem System als Publisher.

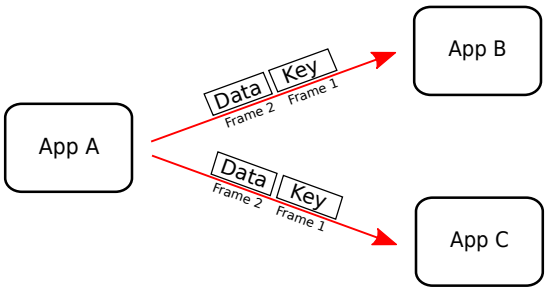

Abbildung 10: Publish/Subcribe-System

Damit zwischen den dargestellten Anwendungen eine Verbindung aufgebaut werden kann, erstellt zunächst jede Anwendung einen neuen ZeroMQ-Context (s. Abbildung 11). Dieser Context beinhaltet dabei die Verwaltung aller Sockets und Verbindungen innerhalb der jeweiligen Anwendung. Anschließend erfolgt die Generierung eines neuen ZeroMQ-Sockets. Dabei erstellt Anwendung A einen sogenannten Pub-Socket. Anwendung B und C erzeugen einen Sub-Socket. Nach dem beide Subscriber den Socket generiert haben, weisen sie den Socket an, sich mit einem Publisher zu verbinden. Dieser Aufruf ist im Gegensatz zu gewöhnlichen Unix-Sockets nicht blockierend. Auch kann dieser Schritt mehrfach wiederholt werden, wenn der Socket sich zu verschiedenen Publishern verbinden soll. Ist der Publisher zum Zeitpunkt des Connects nicht erreichbar, so erfolgt ein regelmäßiges Pollen durch den Socket, bis der Publisher erreichbar ist. Anschließend erfolgt das Setzen der Socket-Option *ZMQ.SUBSCRIBE* mit einem Key. Der Aufruf dieser Methode kann ebenfalls mehrfach erfolgen und erlaubt das Subscriben auf mehrere Keys. Das Setzen dieser Option bewirkt, dass der ZeroMQ-Socket die Daten innerhalb des ersten Frames jeder Nachricht mit den angegebenen Keys vergleicht. Sind die Daten mit einem der Keys identisch, wird die Nachricht an die Anwendung weiter gereicht. Der Abruf der empfangen Nachrichten erfolgt mit dem Aufruf einer Receive-Methode. Das Beispiel zeigt die blockierende Variante.

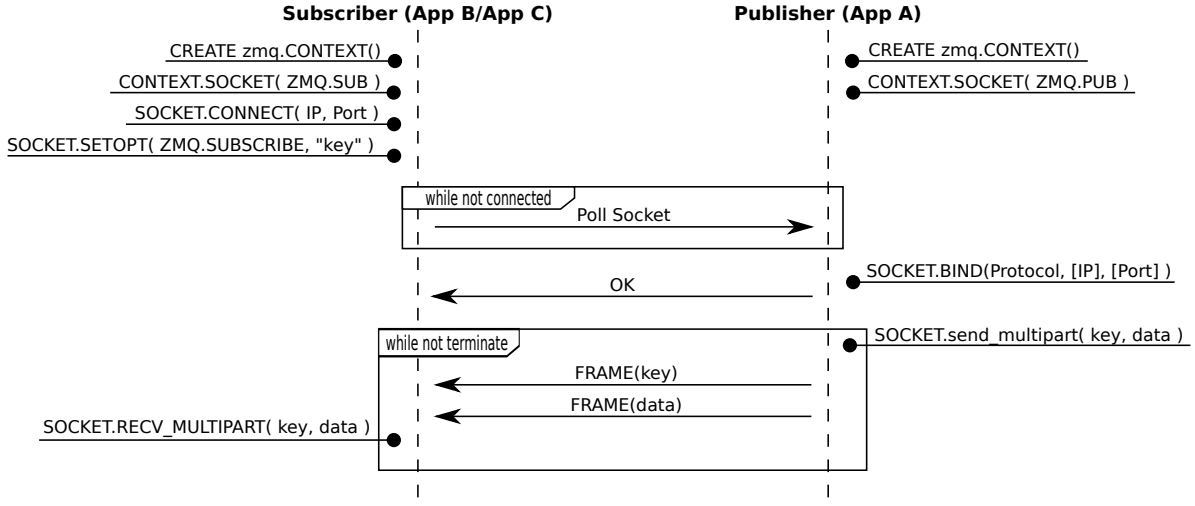

Abbildung 11: Verbindungsaufbau am Beispiel von Publish/Subscribe

Ist die Erstellung des ZeroMQ-Socket auf Seiten des Publishers erfolgreich, so erfolgt ein Bind des Sockets auf eines der unterstützten Protokolle. Ebenso wie das Connect im Subscriber, kann das Binding mehrfach ausgeführt werden. Damit ermöglicht der ZeroMQ-Socket das gleichzeitige Versenden einer Nachricht über verschiedene Protokolle, Interfaces oder Ports. Ist das Binding erfolgreich, so kann der Publisher beginnen Daten in Form von Multipart-Nachrichten zu versenden.

Routing Das Konzept der Multipart-Nachrichten erlaubt auch ein flexibles Routing von Nachrichten zwischen verschiedenen Anwendungen. Dies ist insbesondere deshalb wichtig, weil ZeroMQ auf einen zentralen Broker verzichtet. Das Routing erfolgt anhand von zusätzlichen Frames innerhalb einer Nachricht und ist in Abbildung 12 vereinfacht dargestellt.

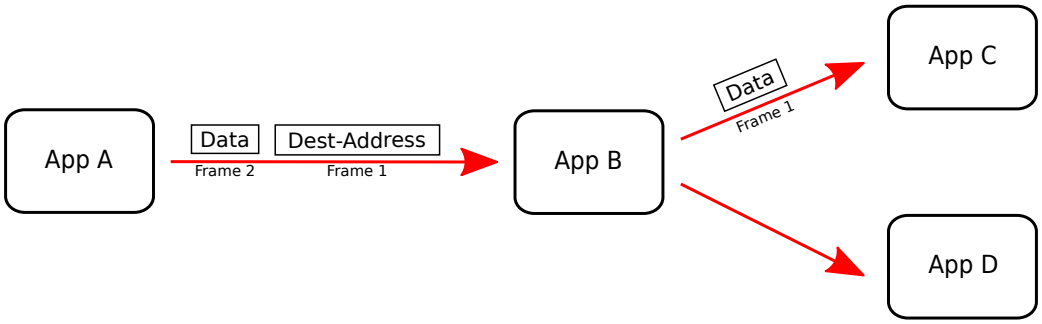

Abbildung 12: Routing mit Multipart-Nachrichten

Eine entscheidende Rolle beim Routing spielen die Sockettypen *Router* und *Dealer*. Mit ihnen kann ein zentraler Router, wie zum Beispiel in Abbildung 13 gezeigt, umgesetzt werden. Dadurch wird es möglich, Requests der Client-Anwendungen an beliebig viele Services zu verteilen. Dieser Ansatz ermöglicht zudem eine einfache Skalierung des Systems, da die Anzahl der angeschlossen Anwendungen beiderseits durch das Starten weiterer Anwendungen jeder Zeit angepasst werden kann. Aufgrund dessen, dass die Implementierung des Codes, im gezeigten Beispiel, zwischen Router- und Dealer-Socket vollständig in der Hand des Entwicklers liegt, ist es möglich das Verhalten und die Fähigkeiten des Routers beliebig anzupassen. Dies umfasst sowohl die Art der Lastverteilung auf die Services in Abhängigkeit verschiedener Parameter als auch das Zwischenspeichern von Nachrichten im Router bis eine erfolgreich Abarbeitung der Daten durch einen Worker zurückgemeldet wurde. Im weiteren kann durch die Anbindung einer Datenbank auch ein vollwertiger Broker mit persistenter Datenspeicherung in ZeroMQ implementiert werden.

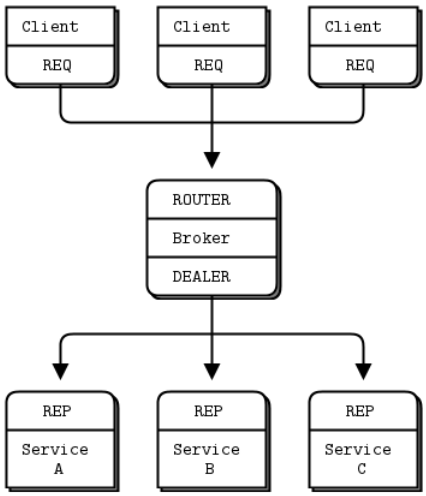

Abbildung 13: Routing mit Request und Replay-Sockets (Entnommen von: [35])

Ein ähnliches Konzept kann auch genutzt werden um Nachrichten mittels Proxy/Bridge in ein anderes Netzsegment zu routen. Wie in Abbildung 14 zu sehen ist, werden für den Proxy die Sockettypen XPub und XSub verwendet. Dieser Sockettyp funktioniert wie die schon

bekannten Pub- und Sub-Sockets, ermöglichen aber die Weiterleitung der Subscriptions der Subscriber an den Publisher. Abseits der eigentlichen Proxyfunktion, kann dieser Aufbau auch zur einfachen Skalierung eines Systems verwendet werden. Außerdem ist auch hier die Möglichkeit vorhanden, innerhalb des Proxys weiteren Einfluss auf die Verteilung der Nachrichten zu nehmen.

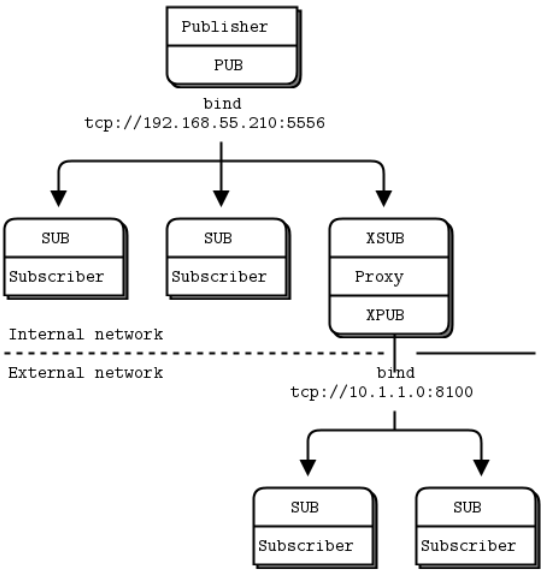

Abbildung 14: Routing mit Publish/Subscribe-Sockets (Entnommen von: [36])

Push- und Pull-Socket Neben den bereits beschriebenen Sockettypen existieren mit den Pull- und Push-Socket zwei weitere Sockettypen. Wie in Abbildung 15 zu erkennen ist, erfolgt die Verteilung der Nachrichten durch den Pull-Socket. Dabei werden die Nachrichten nach dem Round-Robin-Prinzip an alle verbunden PULL-Sockets weitergereicht. Auf eine detaillierte Beschreibung des Systems soll an dieser Stelle aber verzichtet werden, da es in der vorliegenden Arbeit keine weitere Verwendung findet.

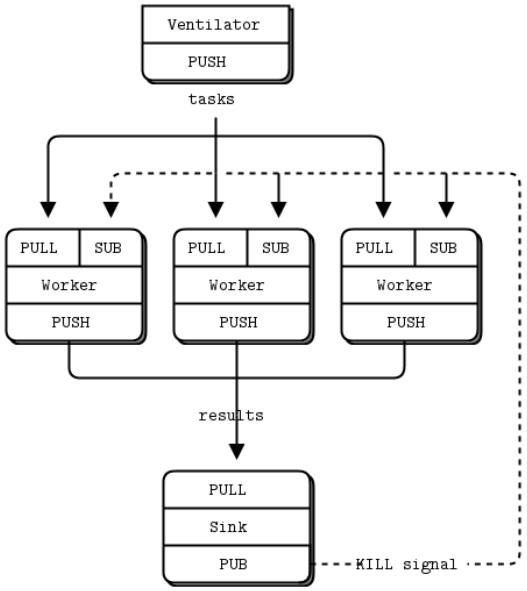

Abbildung 15: System Push und Pull-Socket (Entnommen von: [37])

2.2.3 Lastverteilungsmechanismen

Wie bereits erwähnt, erreicht ZeroMQ mit dem flexiblen Aufbau von Nachrichten und den damit einhergehenden Routingfähigkeiten eine sehr gute Skalierung des entstehenden Messagingsystems. Durch das Einfügen von Routern und Proxys sowie der Verwendung der unterschiedlichen Sockettypen, kann in einem ZeroMQ-System an nahezu jeder Stelle eine Skalierung erfolgen und die Lastverteilung direkt beeinflusst werden. Das so entstehende Messagingsystem hat damit aber auch eine wesentlich höhere Komplexität und einen höheren Entwicklungsaufwand gegenüber brokerbasierten Systemen.

In Bezug auf die Nachrichtenverarbeitung weisen ZeroMQ-Sockets eine interessante Eigenschaft auf, welche direkten Einfluss auf die Verteilung der Last im System hat. Ist ein Subscriber mit mehreren Publishern verbunden und empfängt über diese Nachrichten, so erfolgt die Verarbeitung der Nachrichten nicht in der Reihenfolge des Eintreffens. Viel mehr sorgt der ZeroMQ-Socket, wie in Abbildung 16 zu sehen, für eine Abarbeitung aller Nachrichten aus den verbundenen Publishern nach dem Round-Robin-Prinzip[29]. Es kann also davon ausgegangen werden, dass die Verarbeitung der Nachrichten aus den Publishern immer fair erfolgt.

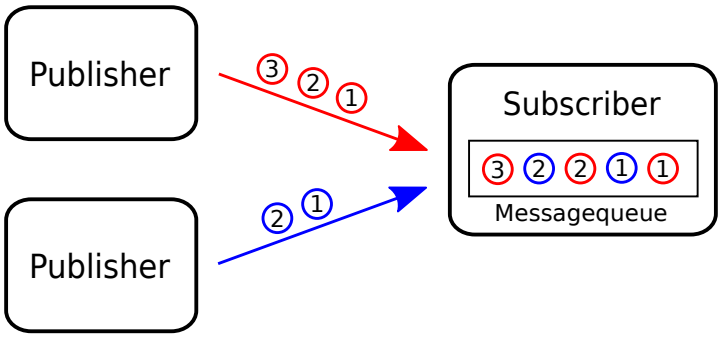

Abbildung 16: Fair-Queueing durch einen ZeroMQ-Socket

Ebenfalls ist in Abbildung 16 ersichtlich, dass jeder ZeroMQ-Socket einen eigenen Empfangspuffer für eingehende Nachrichten verwaltet. Das Halten der Messagequeue im Socket kann bei einem hohen Nachrichtenaufkommen aber zu einer stärkeren Belastung auf Seiten des Empfängers führen. Ist die Anwendung in der Folge nicht in der Lage die Nachrichten schnell genug zu verarbeiten, kann dies im schlimmsten Fall zu einer vollständigen Auslastung des Arbeitsspeichers führen. Kommt es zusätzlich zu einer Auslagerung der Daten vom RAM auf einen persistenten Datenträger, so wird die Abarbeitung der Nachrichten weiter verlangsamt. Dieses Problematik kann dazu führen, dass es zu einer unregelmäßigen Verteilung der Last im System kommt. Im Extremfall ist es möglich, dass das System in Teilbereichen ausfällt. Da in einem ZeroMQ-System jede Anwendung, im Gegensatz zu einem Broker basierten System, einen potentiellen Single-Point of Failure darstellt, ist eine Betrachtung der möglichen Lastverhältnisse bereits vor Implementierung des Systems erforderlich.

3 Allgemeine Anforderungen und Festlegungen

3.1 Anforderungen

Aus der Zielstellung ergeben sich mehrere Anforderungen, die für die Realisierung erfüllt sein müssen. Mit Hilfe dieser Anforderungen sollen erste übergreifende Festlegungen für den späteren Entwurf und die Implementierung der zu testenden Systeme sowie des umgebenden Testsystems getroffen werden.

1. Einsatz der Messaging-Systeme AMQP und ZeroMQ
2. Festlegung geeigneter Testparameter und Messpunkte mit Fokus auf die Lasterfassung
3. Auswahl geeigneter webbasierter Systeme für die Tests
4. Auswahl einer Testumgebung unter Beachtung der verfügbaren Hard- und Software
5. Auswahl einer geeigneten Programmiersprache
6. Entwurf und Implementierung eines Testsystems und der zu testenden Systeme
7. Entwurf und Implementierung eines Systems zur Erfassung der Testdaten

3.2 Festlegungen

3.2.1 Auswahl der Testparameter und Messpunkte

Die sorgfältige Auswahl der Testparameter und Messpunkte ist für die Beurteilung der Messagingsysteme von entscheidender Bedeutung. Durch die Variation dieser Parameter soll im Verlauf der Arbeit eine Bewertung des Lastverhalten unter verschiedenen Bedingungen ermöglicht werden. Die damit festzulegenden Messpunkte zur Erzeugung der Messwerte wurden nach diesen Testparametern ausgewählt. Unter Beachtung der Zielstellung wurden deshalb die folgenden Parameter und Messpunkte ausgewählt und festgelegt.

Testparameter

1. Direkte Beeinflussung der Systemlast durch:
   - Nachrichtenaufkommen pro Sekunde
   - Nachrichtengröße
2. Beeinflussung der Systemkomplexität durch:
   - Anzahl der Verbindungen innerhalb der Messagingsysteme

Messpunkte

1. Messung des Nachrichtenflusses durch:
   - Nachrichtenaufkommen pro Sekunde (Input und Output)
   - Verarbeitete Anzahl von Nachrichten während des Testlaufs
   - Fehlerhafte Verarbeitung von Nachrichten (Output)
2. Messung der Systemlast durch:
   - Messung der CPU-Belastung
   - Messung des RAM-Verbrauchs

3.2.2 Auswahl der zu testenden Systeme

Die Erzeugung aussagekräftiger Testdaten erfordert einen kontinuierlichen Nachrichtenfluss innerhalb der verwendeten Messagingsysteme. Unter Beachtung der zuvor ausgewählten Testparameter wird damit die Auswahl der möglichen Systeme für die Tests stark eingeschränkt. Nach einer Analyse verschiedener Webanwendungen wurden die folgenden zwei Kandidaten, nach diesen Kriterien, ausgewählt.

Chat-System Ein Chat-System zeichnet sich, wie beispielhaft in Abbildung 17 zu sehen, durch eine flexible Anzahl an miteinander verbundenen Chat-Clients aus. Die bidirektionale Kommunikation dieser Clients sorgt für einen kontinuierlichen Nachrichtenfluss im gesamten System. Jede Nachricht eines Chat-Clients wird dabei an alle anderen Chat-Clients über einen Broadcast verteilt.

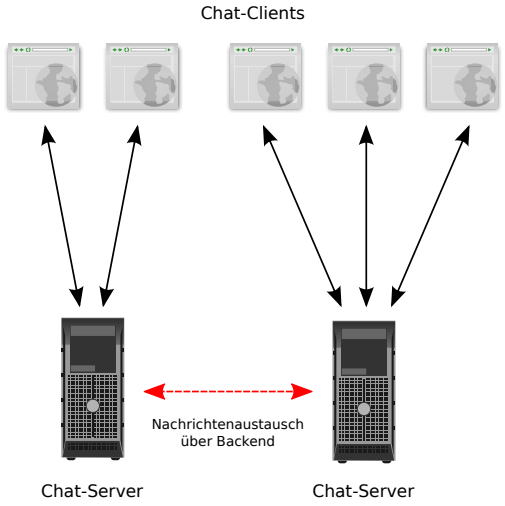

Abbildung 17: Chat-System mit angedeutetem Messagingsystem (rot) im Backend

Die Messagingsysteme AMQP und ZeroMQ können in diesem Szenario zum Austausch der Nachrichten zwischen den Chat-Servern eingesetzt werden. Unter Verwendung mehrerer Chat-Server können folglich das Nachrichtenaufkommen und die Anzahl der Verbindungen in den Messagingsystemen beeinflusst werden. Das zu übertragende Datenvolumen ist zudem stark von den Fähigkeiten des verwendeten Chat-Systems abhängig und kann deshalb frei variiert werden. Die Anbindung der Chat-Clients an den Chat-Server kann über das bidirektionale Websocket-Protokoll erfolgen. Diese Kombination von Messagingsystem und Websocket-Protokoll erlaubt eine vollständig bidirektionale und nachrichtenbasierte Kommunikation zwischen allen Clients. Damit kann eine Veränderung der geforderten Testparameter durch die in Tabelle 3 gezeigten Parameter erreicht werden.

Parameter	**Beeinflussung durch**
Nachrichtenaufkommen	Anzahl der Chat-Clients Generierte Nachrichten pro Sekunde im Chat-Client
Nachrichtengröße	Chat-Clients
Anzahl der Verbindungen	Anzahl der Server

Tabelle 3: Geforderte Testparameter im Chat-System

Nachrichtenticker Ebenso wie das Chat-System zeichnet sich auch ein Nachrichtenticker durch eine flexible Anzahl von Teilnehmern aus. Jedoch beschränkt sich der Nachrichtenfluss hier, wie in Abbildung 18 beispielhaft gezeigt, nur auf den Empfang von zuvor ausgewählten Nachrichtentypen. Als Nachrichtenquellen dienen eine variable Anzahl an Anwendungen, welche jede für sich Nachrichten eines bestimmten Typs anbietet. Die Empfänger der Nachrichten wählen aus dieser Menge an Nachrichtenquellen eine beliebige Anzahl aus. Damit entspricht das System einem typischen Publish/Subscribe-Szenario, welches sowohl durch ZeroMQ als auch AMQP umgesetzt werden kann. Als Publisher sind hier die Nachrichtenquellen zusehen. Die Empfänger agieren als Subscriber.

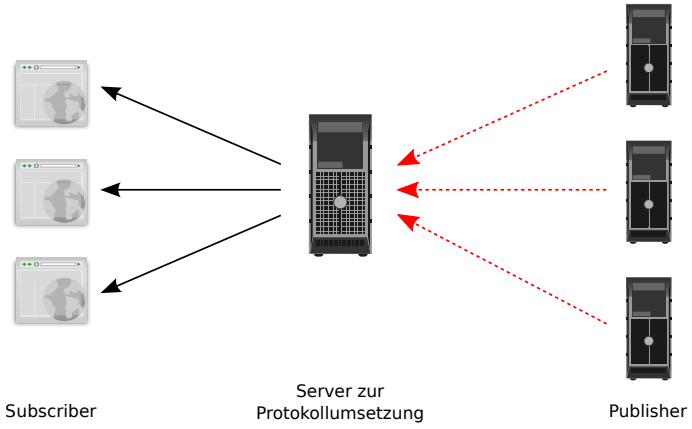

Abbildung 18: Nachrichtenticker mit Messagingsystem (rot) im Backend zu den Publishern

Um eine Anbindung der Subscriber an das Messagingsystem zu ermöglichen, kann auch hier unter Verwendung eines Webservers das Websocket-Protokoll verwendet werden. Sowohl durch die Veränderung der Anzahl der Publisher als auch der Server kann die Summe der Verbindungen innerhalb der Messagingsysteme direkt beeinflusst werden. Die Größe und Menge der Nachrichten wird dabei alleine durch die Publisher bestimmt. Damit kann eine Veränderung der geforderten Testparameter durch die in Tabelle 4 gezeigten Parameter erreicht werden.

Parameter	**Beeinflussung durch**
Nachrichtenaufkommen	Anzahl der Subscriber Generierte Nachrichten pro Sekunde im Subscriber
Nachrichtengröße	Chat-Teilnehmer
Anzahl der Verbindungen	Anzahl der Server Anzahl der Publisher

Tabelle 4: Geforderte Testparameter im Nachrichtenticker-System

3.2.3 Auswahl der Testumgebung

Eine weitere Anforderung aus der Zielstellung ist die Auswahl und die anschließende Festlegung auf eine geeignete Testumgebung. Innerhalb der Hochschule für Telekommunikation Leipzig boten sich zwei mögliche Testumgebungen zur Nutzung an. Um eine Auswahl zu ermöglichen, wurden die in den möglichen Testumgebungen verfügbaren Ressourcen sowie der nötige Aufwand zur Inbetriebnahme der Hardware eingehend betrachtet. Zusätzlich

erfolgte, soweit sinnvoll, eine Betrachtung der verfügbaren Software und ihrer Versionsstände. Im Folgenden sollen die beiden Testumgebungen mit ihren Vor- und Nachteilen beschrieben und anhand dessen die Gründe für die Auswahl dargelegt werden.

Virtuelle Maschinen Die Hochschule bietet mit dem zur Verfügung stellen von Virtuellen Maschinen (VMs) die Möglichkeit ein autarkes Testsystem zu entwerfen. Diese Lösung hat den Vorteil, dass sowohl die Wahl von Software und Betriebssystem als auch die Festlegung der Hardwareressourcen sehr flexibel ist. Damit würden die Entwürfe und Implementierungen von Anwendungen und Testsystem nur geringen Einschränkungen unterliegen. Jedoch wurde diese Testumgebung als zeitkritisch eingestuft, da es neben der genauen Festlegung der Ressourcen auch zu einer möglichen Verzögerung bei der grundlegenden Einrichtung durch das Rechenzentrum hätte kommen können. Im Weiteren wurde die Anpassung der VMs, trotz möglicher Duplizierung von identischen VMs, als ebenso kritisch gesehen, da sie eine teilweise Einarbeitung in das Themengebiet der Virtualisierung erfordert hätte. Zudem erschien es dem Autor unmöglich das Verhalten und den Einfluss der VMs untereinander und auf die Testergebnisse abzuschätzen. Speziell aus diesem Grund entschied sich der Autor gegen diese Lösung als Testumgebung.

Computerpools Die Verwendung der drei, frei zur Verfügung stehenden, Computerpools bietet den Vorteil einer bereits existierenden Testumgebung. Jedoch sind die Hardwareressourcen auf 60 Workstations mit unterschiedlicher Leistungsfähigkeit beschränkt. Ebenso ist ein exklusiver Zugriff auf die Workstations nicht garantiert werden. Die verfügbare Hardware der Computerpools setzt sich dabei wie in Tabelle 5 aufgelistet zusammen.

Pool	Bezeichnung	Prozessor	Kerne	RAM
A1.26	Optiplex 990	Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz	8	8 GByte
A1.27	Precicsion 380	Intel(R) Pentium(R) D CPU @ 2.80GHz	2	3 GByte
A1.28	Optiplex 980	Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz	8	8 GByte

Tabelle 5: Hardware der Computerpools

Durch das, auf allen Workstations, vorinstallierte Debian 6.0 besteht eine Festlegung auf bestimmte Softwareversionen. Dies betrifft sowohl die Version der Programmiersprachen als auch die Verfügbarkeit der Softwarebibliotheken der zu verwendenden Messagingsysteme. Die bestehende Infrastruktur erlaubt aber auch die zentrale Installation von zusätzlichen Softwarepaketen. Damit werden etwaige Anpassungen an den Systemen maßgeblich erleichtert. Bezüglich der Tests und der daraus resultierenden Testergebnisse kann durch die real existierende Hardware eine genaue Abschätzung der Einflüsse durch die Testumgebung abgegeben werden. Auf Grund dieses Vorteils, dem einfachen Zugriff auf die Systeme und unter Beachtung der schlussendlich ausgewählten Programmiersprache, wurde trotz der genannten Nachteile diese Testumgebung ausgewählt.

3.2.4 Auswahl der Programmiersprache

Die Auswahl der Programmiersprache für die Implementierung der Anwendungen und des Testsystems fand zunächst auf Grund der Bibliotheken und Module für die zu testenden Messagingsysteme statt. Dabei wurden alle Sprachen ausgeschlossen, welche nicht für beide Messagingsysteme verfügbar waren und über keine aktuelle Bibliothek zur Nutzung des Websocket-Protokolls verfügten. Ebenso wurden alle Programmiersprachen ausgeschlossen, in denen der Autor keine Erfahrungen vorweisen kann. Damit kamen die folgenden Programmiersprachen C, Python, Java und Perl für eine Implementierung in Betracht. Abschließend erfolgte eine Analyse dieser Sprachen hinsichtlich der Leistungsfähigkeit, des Programmieraufwands sowie der verfügbaren Dokumentationen.

Der Autor entschied sich auf Grund der umfassenden Dokumentation für beide Messagingsysteme schlussendlich für die Festlegung auf die Programmiersprache Python zur Implementierung des Testsystems und der Anwendungen. Gründe für diese Entscheidung waren die Verfügbarkeit mehrerer Module für die Unterstützung von Websockets und das virtualenv-Modul[38, 39, 40]. Das zuletzt genannte Modul ermöglicht dabei eine Art virtuelle Pythonumgebung. Dies ermöglicht die Festlegung des Pythoninterpreters und der verwendeten Module auf einen in der Testumgebung verfügbaren Versionsstand. Damit wird eine Entwicklung der Anwendungen außerhalb der gewählten Testumgebung ermöglicht und die spätere Lauffähigkeit sichergestellt. Ein weiterer Grund für die Wahl von Python ist die gute Modularisierung des Programmcodes und die C nahe Implementierung des Pythoninterpreters. Zudem wird Python häufig im Webumfeld eingesetzt und verfügt deshalb über alle nötigen Module zur Unterstützung von Webtechniken und Protokollen wie dem HTTP.

Der Ausschluß der Programmiersprachen C und Perl erfolgte auf Grund des zu erwartenden Aufwandes bei der Entwicklung in Bezug auf Multithreadingtechniken. In Perl hätte zudem die geringe Verfügbarkeit von Dokumentationen im Bereich der Messaging-Bibliotheken zu einem hohen Implementierungsaufwand geführt. Die Entscheidung gegen Java erfolgte, trotz ähnlicher Vorteile wie bei Python, wegen der geringen Erfahrung des Autors mit der Programmiersprache und einer ausführlicheren Dokumentation der Messagingbibliotheken in Python.

4 Testsystem und Logging

4.1 Anforderung

Aus den allgemeinen Anforderungen und den damit entstandenen Festlegungen der Testumgebung, Testparameter und den zu testenden Systemen ergeben sich weitere spezielle Anforderungen für das zu erstellende Testsystem. Basierend auf den daraus entstehenden Festlegungen soll das Konzept entstehen. Die ermittelten Anforderungen sind:

1. Sicherstellung der Wiederverwendbarkeit des Testsystem und der Teststeuerung
2. Sicherstellung einer flexiblen Konfiguration der Testparameter
3. Flexible Verteilung der zu testenden Anwendungs-Software ermöglichen (Deployment)
4. Entwurf einheitlicher Logging-Verfahren für die Tests

4.2 Testsystem

4.2.1 Konzeption

Um eine hohe Wiederverwendbarkeit des Testsystems sicherzustellen, soll die Steuerung der Tests über einen zentralen Testcontroller realisiert werden. Dieses Vorgehen ermöglicht das zentrale Starten der Tests und vermeidet ein manuelles Starten der einzelnen Systemkomponenten auf verschiedenen Workstations. Das einheitliche Erfassen der anfallenden Messwerte soll nach Abschluss der Tests über einen zentralen Logging-Server geschehen. Dies stellt die Bereitstellung einer Log-Datei mit allen Messwerten auf einer Workstation sicher und ermöglicht zudem eine schnelle Prüfung des abgeschlossenen Tests auf Verwendbarkeit. Die Bereitstellung der Anwendungssoftware und der Systemkonfigurationen soll durch ein Git-Versionsverwaltungsystem realisiert werden. Durch diese Entscheidung erfolgt immer die vollständige Auslieferung aller Softwarebestandteile an alle Workstations innerhalb der Testumgebung. Zusätzlich stellt die Versionsverwaltung sicher, dass alle Anwendungen innerhalb der Testumgebung mit der gleichen Softwareversion und Konfiguration laufen. Damit teilt sich das Testsystem, wie in Abbildung 19 zu sehen, in drei verschiedene Komponenten auf. Deren Konzeption soll im Folgenden näher beschrieben werden.

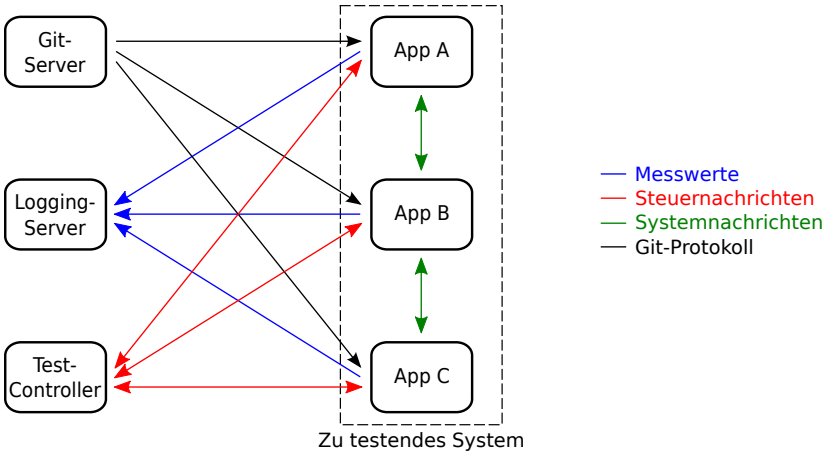

Abbildung 19: Prinzipieller Aufbau des Testsystems

Testcontroller und Teststeuerung Der in Abbildung 19 zu sehende Testcontroller sorgt für die zentrale und zeitliche Ansteuerung der einzelnen Anwendungen innerhalb des jeweils zu testenden Systems. Durch das Versenden von einfachen Kommandos soll er den Anwendungen sowohl Startzeitpunkt und Endzeitpunkt eines Testlaufs signalisieren sowie den Versand der erfassten Messwerte an den Logging-Server initialisieren. Zusätzlich soll die Möglichkeit bestehen das laufende System nach dem Testlauf zu beenden. Um den reibungslosen Ablauf der Tests sicherzustellen, sollen die Anwendungen zu bestimmten Zeitpunkten Rückmeldungen an den Testcontroller geben. Dazu gehören die Bestätigungen der erfolgten Ermittlung der Messwerte und des erfolgreichen Versendens dieser Messwerte an den Logging-Server. Ebenfalls soll eine Nachricht vor Beendigung der Anwendung an den Testcontroller gesendet werden. Dies stellt sicher, dass zu keiner Zeit laufende Prozesse aus vorangegangen Testläufen existieren, welche die Testergebnisse eines nachfolgenden Tests beeinflussen können. Die Ermittlung der insgesamt anzusteuernden Anwendungen soll über die globale Konfigurationsdatei erfolgen. Für die Verbindung der einzelnen Anwendungen mit dem Testcontroller wurde das TCP gewählt. Die erforderlichen Kommandos sind in Tabelle 6 aufgelistet. Die Anwendungen werden dabei je nach Typ mittels eines Präfixes vom Controller angesprochen. Damit wird eine ortsunabhängige Unterscheidung der einzelnen Komponenten des zu testenden Systems möglich. Eine eventuelle Bestätigung durch die Anwendung erfolgt mit dem selben Typ. Basierend auf den in Abschnitt 3.2.2 ausgewählten Anwendungen wurden die in Tabelle 7 aufgelisteten Anwendungstypen festgelegt.

Kommando	**Zweck**	**Bestätigung auf**
[Typ] hello	Anmeldung am Controller	-
[Typ] start	Startsignal durch den Controller	-
[Typ] log	Aufforderung zum Loggen der aktuellen Messwerte	-
[Typ] logdone	Bestätigung der erfolgten Messwerterfassung	[Typ] log
[Typ] getlog	Aufforderung zum versenden der Messwerte	-
[Typ] logsend	Bestätigung, dass die Messwerte versandt wurden	[Typ] getlog
[Typ] exit	Aufforderung zum Terminieren / Bestätigung der Terminierung	[Typ] exit
error	Allgemeiner Fehler im Testablauf	-
[IP]	Übertragung der IP eines angeschlossenen Systems (Sonderfall)	-

Tabelle 6: Kommandos zur Teststeuerung

Typ	**Verwendet für**	**System**
s	Server	Chat & Nachrichtenticker
w	Publisher	Nachrichtenticker
c	Chat-Client/Subscriber	Chat & Nachrichtenticker
l	Lastermittlung	Chat & Nachrichtenticker

Tabelle 7: Anwendungstypen

Testablauf Um einen einheitlichen Test sicherzustellen, wurde vor Beginn der Implementierung der grobe Testablauf für alle Anwendungen unter Beachtung des Gesamtsystems festgelegt. Dabei wurden die Startzeitpunkte der einzelnen Anwendungen festgelegt und die benötigten Kommandos ausgewählt. Die Kommunikation zwischen Testcontroller und Anwendung sollte dabei wie in Abbildung 20 für ein minimales Chat-System und in Abbildung 21 auf Seite 24 für ein minimales Nachrichtenticker-System gezeigt verlaufen.

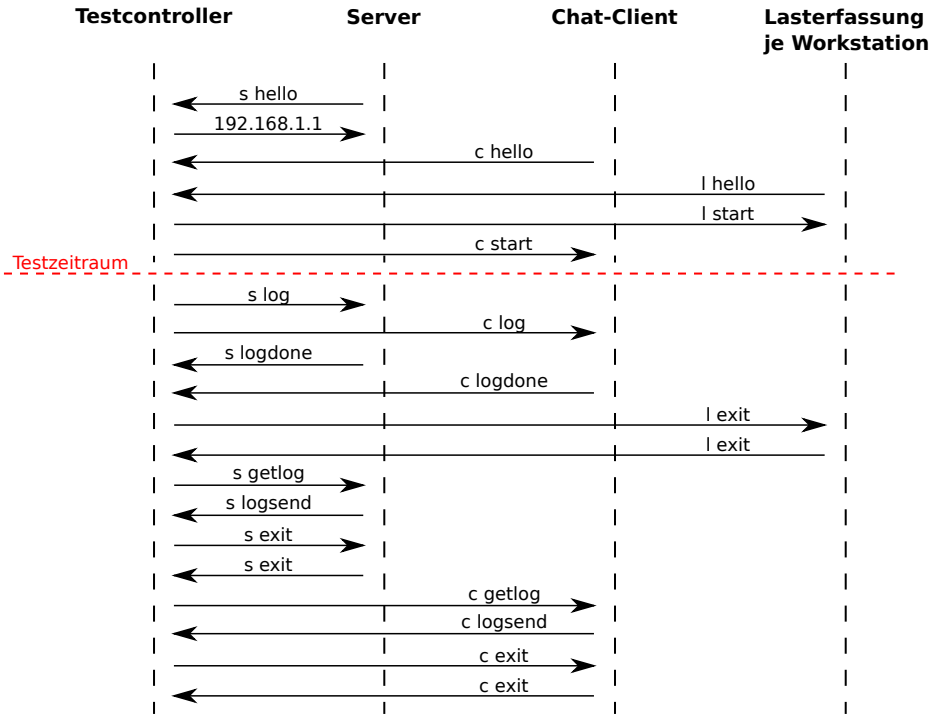

Abbildung 20: Kommunikation im Testablauf Chat-System

Wie in beiden Abbildungen zu sehen, soll zunächst die Anmeldung aller Anwendungen am Testcontroller erfolgen. Der Start der Anwendungen soll zuvor durch Bash-Skripte mit Hilfe der zentralen Konfigurationsdatei erfolgen. Das versenden der IP-Adresse an den Server im Chat-System sowie an die Publisher im Nachrichtenticker erfolgt aus Implementierungsgründen und wird später genauer erläutert werden. Die versandte IP-Adresse entspricht dabei der IP-Adresse der sich anmeldenden Anwendung. Haben sich alle für den Testlauf erforderlichen Komponenten am Testcontroller angemeldet, so soll der Test gestartet werden. Dazu werden alle Anwendungen, aufgefordert sich miteinander zu verbinden und die Kommunikation zu beginnen. Da der Server zum Startzeitpunkt des Testlaufs bereits stabil laufen soll, entfällt hier das Start-Kommando. Unter der Annahme, dass die Verbindung von Chat-Client beziehungsweise Subscriber innerhalb kürzester Zeit nach dem Erhalt des Start-Kommandos erfolgt, soll dieser Zeitpunkt als Startzeitpunkt im Server verwendet werden. Nach dem Ablauf des Testintervalls sollen die Anwendungen des Systems aufgefordert werden die erfassten Messwerte abzurufen. Dabei sollen bereits erste Berechnungen anhand dieser Werte vorgenommen werden. Die erfolgreiche Bearbeitung wird anschließend an den Testcontroller gemeldet. Haben alle Anwendungen dies getan, so werden die Lasterfassungs-Anwendungen auf allen Workstations aufgefordert die erfassten Messwerte an den Logging-Server zu senden und sich danach zu beenden. Abschließend erfolgt das Versenden der Messwerte durch die Server, die Subscriber beziehungsweise die Chat-Clients und die Publishern. Nach dem erfolgreichen Versenden der Messwerte wird die jeweils angesprochene Anwendung aufgefordert sich zu beenden. Ist dies für alle Anwendungen im zu testenden Systems geschehen, so gilt der Testlauf als beendet.

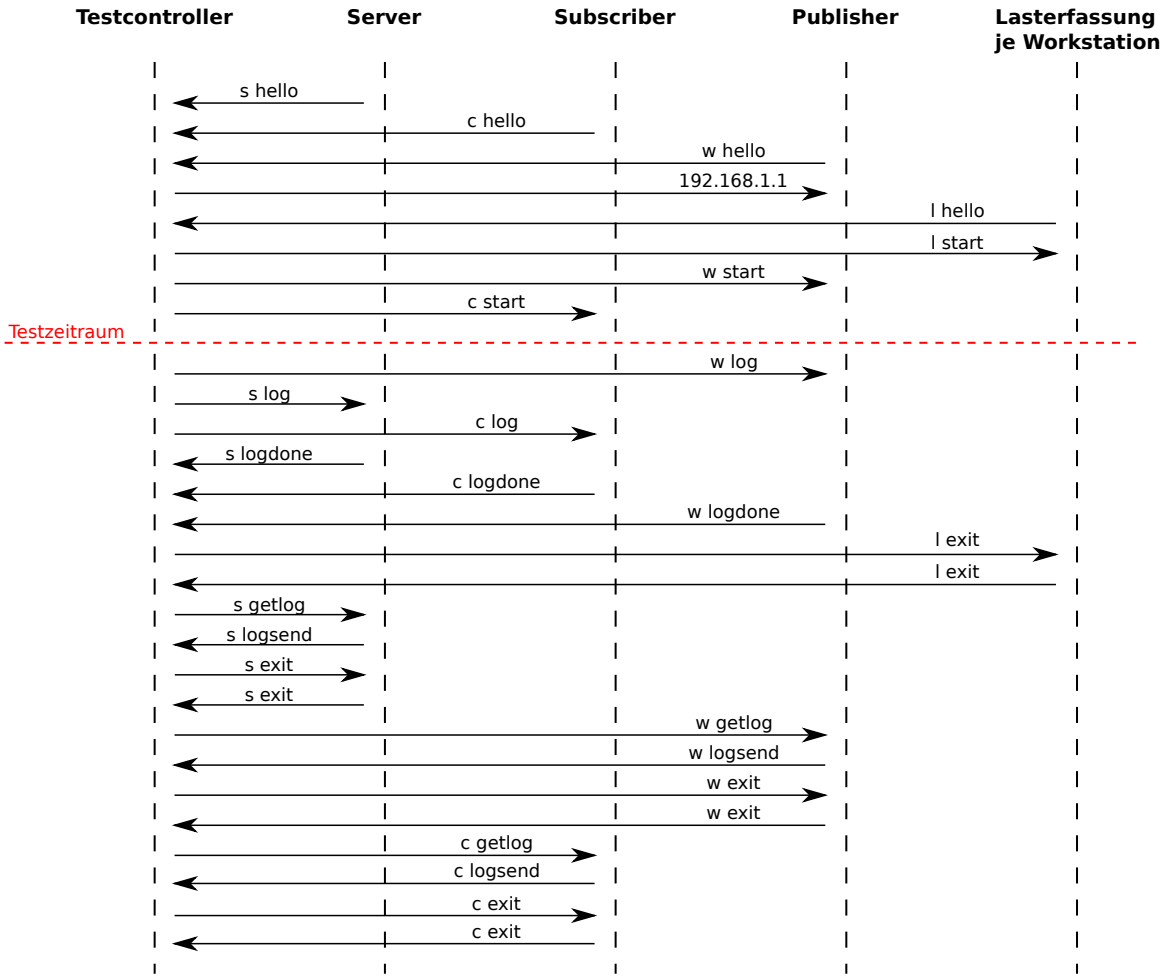

Abbildung 21: Kommunikation im Testablauf Nachrichtenticker

Deployment Wie bereits in der groben Konzeption des Testsystems erwähnt, soll das Ausliefern der Software über einen zentralen Git-Server erfolgen. Dabei sollen die für einen Testlauf benötigten Workstations über einfache SSH-Verbindungen vor dem Test aufgefordert werden, ein Update der Software durchzuführen. Durch dieses Update erfolgt auch die Auslieferung der Testkonfiguration. Zusätzlich ermöglicht dieser Ansatz das zuverlässige Einspielen von Patches im Falle einer Softwareanpassung. Die Auswahl der anzusprechenden Workstations, soll dabei ebenfalls über die zentrale Konfigurationsdatei für die Testläufe erfolgen. Um den Entwicklungsaufwand des Deployments möglichst gering zu halten, soll es mittels einfacher Bash-Skripte umgesetzt werden.

4.2.2 Implementierung

Testkonfiguration Wie bereits in den vorherigen Abschnitten angesprochen, erfolgt die Konfiguration aller Anwendungen und des Testsystem über eine zentrale Konfigurationsdatei. Die Datei *system.cfg* beinhaltet deshalb die Einstellungen für alle geforderten Testparameter, sowie die Zuteilung aller benötigten Workstations für einen Testlauf. Wie Listing 1 auf Seite 25 zeigt, wurde die Datei dazu in Abschnitte aufgeteilt, welche durch Einfassung in eckigen Klammern gekennzeichnet sind. Die Abschnitte Controller, Logger, Server, Broker, Client und Worker dienen der Zuweisung einer Aufgabe zu einer speziellen Workstation. Diese erfolgt durch die Angabe eines Schlüssels und der IP-Adresse der Workstation innerhalb der Testumgebung. Der Abschnitt Test definiert einen Teil der

Testparameter, darunter die Laufzeit des Tests, die Frequenz der Nachrichtengenerierung und die Größe der zu erzeugenden Nachrichten. Der Schlüssel *load_checks* dient dabei nur den Anwendungen zur Lasterfassung auf den einzelnen Workstations. Er bestimmt wie oft ein Auslesen der aktuellen Lastwerte aus dem System erfolgt. Die Schlüssel *max_clients* und *max_workers* sorgen für den korrekten Aufbau des zu testenden Systems durch die Anwendungen auf den einzelnen Workstations.

```
1  [Controller]
2  cnc_ip = 192.168.1.40
3
4  [Logger]
5  logger_ip = 192.168.1.40
6
7  [Server]
8  srv_ip_1 = 192.168.5.17
9
10 [Broker]
11 broker_ip = 192.168.1.40
12
13 [Client]
14 client_ip_1 = 10.1.127.5
15 client_ip_2 = 10.1.127.6
16 client_ip_3 = 10.1.127.7
17 client_ip_4 = 10.1.127.8
18
19 [Worker]
20 worker_ip_1 = 192.168.1.1
21
22 [Test]
23 # test time
24 runtime = 300
25
26 # Load check every x seconds
27 load_checks = 10
28
29 # maximum clients in the system
30 max_clients = 100
31
32 # maximum worker in the system
33 max_workers = 25
34
35 # time each worker sleep between two messages
36 sleep_seconds = 1
37
38 # data_size in byte per message
39 data_size_byte = 512
```

Listing 1: Konfigurationsdatei für den Nachrichtentickers

Wie das prinzipielle Auslesen der Konfigurationsdatei durch die Python-Anwendungen erfolgt, soll im Folgenden anhand eines Beispiels erklärt werden. Durch das, in Listing 2 auf Seite 26 zu sehende, Modul `ConfigParser` kann eine Datei mit dem zuvor beschriebenen Aufbau über ein einfaches Interfaces ausgelesen werden. Dazu wir zunächst ein neues ConfigParser-Objekt erzeugt. Anschließend wird die Konfigurationsdatei angegeben und ausgelesen (Z. 36-37). Danach wird, unter Angabe des Abschnittes und des Schlüssels, über die Methoden `get`, `getint` und `getfloat` der Wert eingelesen. Durch die Auswahl der verwendeten Methode erfolgt dabei eine sofortige Prüfung des Wertes auf den korrekten Datentyp. Damit ist sichergestellt, dass keine falsch konfigurierte Datei für einen Testlauf verwendet werden kann. Die Methode *items* gibt, unter Angabe des Abschnitts, alle verfügbaren Einträge als Liste zurück (Z. 40, 42). Eine Prüfung des Datentyps findet hierbei nicht statt.

```
35 # Get configs
36 config = ConfigParser.RawConfigParser()
37 config.read('system.cfg')
38 SERVERSTATIONS = config.items('Server')
39 CLIENTS = config.getint('Test', 'max_clients')
40 CLIENTSTATIONS = config.items('Client')
41 WORKERS = config.getint('Test', 'max_workers')
42 WORKERSTATIONS = config.items('Worker')
43 MAXCONNECTION = 2*len(CLIENTSTATIONS) + 2*len(SERVERSTATIONS) + ↩
       2*len(WORKERSTATIONS)
44 LOGGER_IP = config.get('Logger', 'logger_ip')
45 LOGGER = TCPLogger(LOGGER_IP, 'CNC')
46 RUNTIME = config.getint('Test', 'runtime')
47 TIMER = Timer(RUNTIME, __endtest, [SERVER_LIST, CLIENT_LIST, WORKER_LIST],{})
```

Listing 2: Auslesen der Konfigurationsdatei im Testcontroller (`controller.py`)

Teststeuerung im Controller Die Umsetzung des Testcontrollers für Teststeuerung erfolgte für die zu testenden Systeme nach dem gleichen Prinzip. Lediglich im Testcontroller, für das zu testende Nachrichtenticker-System, wurde die Steuerung um die dort vorhandenen Publisher-Komponenten erweitert. Die Implementierung beider Controller richtete sich dabei nach den Message Sequence Charts (MSCs) aus der Konzeption. Die Umsetzung der Teststeuerung innerhalb der einzelnen Anwendungen soll an dieser Stelle nicht weiter betrachtet werden, da auf diese in der Implementierung der Anwendungen eingegangen wird.

Durch die Anpassung der Konfigurationsdatei vor jedem Testlauf mit neuen Parametern, muss der Testcontroller zu Beginn jedes Tests neu gestartet werden. Dies bringt zum einem einen größen Aufwand während der Testphase mit sich, erlaubt es aber vor Beginn des Tests eine automatische Überprüfung der Konfiguration vorzunehmen. Wie im Listing 3 des Testcontrollers für den Nachrichtenticker zu erkennen ist, erfolgt diese Prüfung auf den Zeilen 82 bis 86. Überprüft wird, ob die Anzahl der zu startenden Subscriber ganzzahlig durch die Anzahl der zugewiesen Workstations teilbar ist und ob überhaupt ausreichend Subscriber gestartet werden (Z. 82). Diese Prüfung wird nachfolgend auch für die Publisher durchgeführt und entfällt damit folglich im Chat-System. Damit ist sichergestellt, dass kein System mit einer ungültigen Zuteilung der Ressourcen getestet wird. In Bezug auf die anderen Testparameter bleibt aber eine Unsicherheit bestehen.

```
79     #register exit method
80     atexit.register(end)
81
82     if (not (CLIENTS % len(CLIENTSTATIONS)) == 0) or len(CLIENTSTATIONS) > ↩
           CLIENTS or CLIENTS < 1:
83         print 'Configuration failure in client section of system.cfg'
84         os._exit(1)
85
86     if (not(WORKERS % len(WORKERSTATIONS)) == 0) or len(WORKERSTATIONS) > ↩
           WORKERS:
87         print 'Configuration failure in worker section of system.cfg'
88         os._exit(1)
89
90     SERVER_SOCKET.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
91     SERVER_SOCKET.bind(('0.0.0.0', PORT))
92     SERVER_SOCKET.listen(MAXCONNECTION)
93     CONNECTION_LIST.append(SERVER_SOCKET)
94     print 'Controller started'
```

Listing 3: Prüfung der Systemkonfiguration im Testcontroller (`controller.py`)

Ist die Überprüfung der Systemkonfiguration erfolgreich verlaufen, wird der TCP-Server gestart. Die Annahme der Verbindungen und die Verarbeitung der Nachrichten wurde dabei, wie in Listing 4 zu sehen, mit der nicht blockierenden Select-Methode umgesetzt (Z. 100). Sie ermöglicht die effektive sequenzielle Abarbeitung der eintreffenden Nachrichten von mehreren verbundenen Gegenstellen. Die Berechnung der maximal anzunehmenden Verbindungen erfolgt dabei, wie in Listing 2 auf Seite 26 auf Zeile 43 zu erkennen, dynamisch über das Auslesen der entsprechenden Werte aus der Systemkonfiguration.

Anschließend wartet der Server auf neue Verbindungen und die folgende Anmeldung der Anwendungen durch das definierte Kommando. Dabei wird, wie Listing 4 ab Zeile 114 zeigt, jede Anwendung anhand des Typs im Kommando identifiziert und einer entsprechenden Liste zugeordnet.

```
97      while 1:
98          try:
99              # Get the list sockets which are ready to be read through select
100             read_sockets, write_sockets, error_sockets = ↩
                    select.select(CONNECTION_LIST, [], [])
101
102             for sock in read_sockets:
103                 #New connection
104                 if sock == SERVER_SOCKET:
105                     # Handle the case in which there is a new connection ↩
                            recieved through SERVER_SOCKET
106                     sockfd, addr = SERVER_SOCKET.accept()
107                     CONNECTION_LIST.append(sockfd)
108
109                 #Some incoming message from a client
110                 else:
111                     # Data recieved from client, process it
112                     try:
113                         data = sock.recv(RECV_BUFFER)
114                         if data == 'c hello':
115                             print 'Clients - (%s, %s) online' % ↩
                                    sock.getpeername()
116                             CLIENT_LIST.append(sock)
117                         elif data == 'l hello':
118                             print 'Logger - (%s, %s) online' % ↩
                                    sock.getpeername()
119                             LOGGER_LIST.append(sock)
120                         elif data == 's hello':
121                             print 'Server - (%s, %s) online' % ↩
                                    sock.getpeername()
122                             SERVER_LIST.append(sock)
123                         elif data == 'w hello':
124                             # send ip to worker
125                             ip = str(sock.getpeername()[0])
126                             sock.send(ip)
127                             time.sleep(0.1)
128                             print 'Worker - (%s, %s) online' % ↩
                                    sock.getpeername()
129                             WORKER_LIST.append(sock)
130                         elif data == 'c exit':
131                             print 'Clients - (%s, %s) offline - Aborting test' ↩
                                    % sock.getpeername()
132                             raise TestErrorException
133                         elif data == 'l exit':
```

Listing 4: Start des TCP-Servers und Anmeldung im Testcontroller (`controller.py`)

Nach dem der Testcontroller anhand der If-Abfrage auf Zeile 146 im Listing 5 auf Seite 28 festgestellt hat, dass alle Anwendungen online sind, werden diese gestartet. Dazu erfolgt ein Broadcast mit den entsprechenden Kommandos, indem über die zuvor generierten Listen iteriert wird. Gleichzeitig wird, wie auf den Zeilen 156 bis 158 zu sehen ist, der Startzeitpunkt gespeichert und ein Timer gestartet.

```
146             if (len(CLIENT_LIST) + len(SERVER_LIST) + len(WORKER_LIST) + ↩
                    len(LOGGER_LIST)) == MAXCONNECTION and len(CONNECTION_LIST)-1 ↩
                    == MAXCONNECTION:
147                 print 'Stabilze system - Waiting 5sec'
148                 time.sleep(5)
149                 print 'Start up Logger'
150                 for l in LOGGER_LIST:
151                     try:
152                         l.send('l start')
153                     except:
154                         print 'Lost Logger - Aborting test '
155                         sys.exit()
156                 test_start_time = time.time()
157                 TIMER.start()
158                 LOGGER.add_logmessage('test-starttime:' + str(test_start_time))
159                 print 'Start up worker'
```

Listing 5: Start des Tests im Testcontroller des Nachrichtentickers `controller.py`

Dieser Timer sorgt über den auf Zeile 47 definierten Callback *___endtest*, in Listing 2, für das Einleiten der Messwerterfassung in den Anwendungen nach Ablauf der Testzeit. Die Funktion ist in Listing 6 abgebildet.

```
13 def __endtest(*args, **kwargs):
14         """
15         Timer callback
16         @brief Let server, clients and worker log there state
17         """
18         for s in args[0]:
19             s.send('s log')
20         for c in args[1]:
21             c.send('c log')
22         for w in args[2]:
23             w.send('w log')
```

Listing 6: Callback-Funktion ___endtest im Testcontroller `controller.py`

Nachdem alle Anwendungen bestätigt haben, dass die Messwerte verarbeitet wurden erfolgt das Beenden der Anwendungen für die Lasterfassung. Haben sich diese Anwendungen beendet, so werden nacheinander alle anderen Systemkomponenten aufgefordert ihre Messergebnisse zu versenden und sich dann zu beenden. Dabei wird jeder Anwendungstyp separat behandelt. Das diese Implementierung immer dem gleichen Schema folgt und in beiden Systemen identisch umgesetzt ist, soll das Vorgehen nur einmal anhand der Publisher im Nachrichtenticker-System erläutert werden. Im Listing 7 auf Seite 29 sieht man auf Zeile 358, wie zunächst ein Publisher aufgefordert wird seine Messdaten an den Logging-Server zuschicken. Wird auf Zeile 372 die Bestätigung registriert, wird der Publisher auf Zeile 387 aufgefordert sich zu beenden. Erfolgt auch hier eine Bestätigung, so wird der Publisher aus den Listen entfernt und mit dem nächsten Publisher fortgefahren. Eine explizite Trennung der Verbindung durch Testcontroller erfolgt dabei nicht, da diese direkt nach dem Versenden der Exit-Nachricht durch den Publisher getrennt wird. Für das gezeigte Vorgehen wurde sich aufgrund früherer Versuche entschieden. In diesen wurde festgestellt, dass eine große Anzahl an Verbindungen in Kombination mit vielen Messwerten den Logging-Server überlasten und einzelne Logeinträge verloren gehen. Die schlussendlich gewählte Implementation führt zwar zu einer langsameren Abarbeitung eines Testlaufs, gewährleistet aber das stabile Erfassen aller Messwerte.

```
354     wlist = WORKER_LIST[:]
355     for w in wlist:
356         try:
357             print 'Worker - (%s, %s) - Get logs' %w.getpeername()
358             w.send('w getlog')
359         except:
360             print 'Lost worker - Aborting test'
361             sys.exit()
362 
363         ready = 0
364         while 1:
365             try:
366                 # Get the list sockets which are ready to be read through select
367                 read_sockets, write_sockets, error_sockets = ↩
                        select.select(CONNECTION_LIST, [], [])
368 
369                 for sock in read_sockets:
370                     try:
371                         data = sock.recv(RECV_BUFFER)
372                         if data == 'w logsend':
373                             print 'Worker - (%s, %s) did send the logs' % ↩
                                sock.getpeername()
374                             ready = ready + 1
375                         elif data == 'w exit':
376                             print 'Clients - (%s, %s) offline - Aborting test' ↩
                                % sock.getpeername()
377                             raise TestErrorException
378                         elif data == 'c exit':
379                             print 'Clients - (%s, %s) offline - Aborting test' ↩
                                % sock.getpeername()
380                             raise TestErrorException
381                     except TestErrorException:
382                         sys.exit()
383 
384                 if ready == 1:
385                     try:
386                         print 'Worker - (%s, %s) - send exit' %w.getpeername()
387                         w.send('w exit')
388                     except:
389                         print 'Lost worker - Aborting test'
390                         sys.exit()
391                     break
392             except:
393                 sys.exit()
394 
395         # Wait until all workers exited the system
396         ready = 0
397         while 1:
398             try:
399                 # Get the list sockets which are ready to be read through select
400                 read_sockets, write_sockets, error_sockets = ↩
                        select.select(CONNECTION_LIST, [], [])
401 
402                 for sock in read_sockets:
403                     try:
404                         data = sock.recv(RECV_BUFFER)
405                         if data == 'w exit':
406                             print 'Worker - (%s, %s) offline' % ↩
                                sock.getpeername()
407                             ready = ready + 1
408                             WORKER_LIST.remove(sock)
409                             CONNECTION_LIST.remove(sock)
410                     except TestErrorException:
411                         sys.exit()
412 
413                 if ready == 1:
414                     break
415             except:
416                 sys.exit()
```

Listing 7: Verarbeitung der Publisher im Testcontroller (`controller.py`)

Deployment und Start der Anwendungen Wie bereits angesprochen, erfolgte die Umsetzung des Deployments und der Start der Anwendungen mit Hilfe von Bash-Skripten. Das Einrichten der Testumgebung vor einem Start geschieht mit dem Skript `update.sh`. Auf Zeile 4 in Listing 8 ist zu erkennen, dass das Skript zunächst alle IP-Adressen aus der globalen Konfigurationsdatei mit Hilfe von regulären Ausdrücken parst. Dabei unterscheidet sich die Version, der hier gezeigten update.sh des Chat-Systems, nur durch den regulären Ausdruck zur update.sh des Nachrichtentickers, da in dieser mit den Publishern noch eine weitere Komponente existiert. Anschließend wird in einer For-Schleife die Liste der IP-Adressen abgearbeitet. Damit wird zu jeder Workstation des aktuellen Testlaufs eine SSH-Verbindung aufgebaut und mit dem Befehl `git pull origin master` die Software über den Git-Server aktualisiert (Z. 9). Durch die Verwendung des Befehls `sshpass` werden dabei Benutzername und Passwort an das SSH-Programm übergeben. Dies unterbindet die sonst übliche Abfrage in jedem Schleifendurchlauf. Nach Ablauf des Skripts ist das zu testende System für den Start der Anwendungen bereit.

```
4  ips=$(grep -r -E '^srv_ip|^client_ip' ../system.cfg | sed -e 's/^.* = ↩
       \(.*\)$/\1/' | sort | uniq)
5
6  for ip in $ips
7  do
8      echo "###### Deploy on $ip"
9      sshpass -p $pass ssh $user@${ip} "cd ba-system3; git pull origin master"
10 done
```

Listing 8: *update.sh* des Chat-Systems

Zum Starten der Anwendungen existieren mit `runzmq.sh` und `runamqp.sh` pro System je zwei Skripte. Im Listing 9 erkennt man, dass auf den Zeilen 4 bis 6 ebenfalls ein Parsen der IP-Adressen aus der Konfigurationsdatei mit regulären Ausdrücken erfolgt. Anders als bei dem zuvor gezeigten Skript geschieht dies hier aber für jeden Abschnitt der Konfigurationsdatei getrennt. Die Start-Skripte für das Nachrichtenticker-System sind dabei wieder um die zusätzlichen Publisher erweitert worden. Damit entsteht für jeden Anwendungstyp eine eigene Liste, über die im Anschluss mit getrennten For-Schleifen iteriert wird.

```
4  servers=$(grep -r -E '^srv_ip' ../system.cfg | sed -e 's/^.* = \(.*\)$/\1/' | ↩
       sort | uniq)
5
6  clients=$(grep -r -E '^client_ip' ../system.cfg | sed -e 's/^.* = \(.*\)$/\1/' ↩
       | sort | uniq)
```

Listing 9: Startskript runzmq.sh des Chat-Systems 1/2

Innerhalb der For-Schleifen erfolgt, wie in Listing 10 auf Seite 31 zu sehen, das Ausführen der benötigten Anwendungen über eine SSH-Verbindung. Dabei ist auf Zeile 12 und 19 zu erkennen, dass neben dem Start der eigentlichen Anwendung, auf den Zeilen 11 und 18, auch der Start der Lasterfassung erfolgt. Da alle laufenden Programminstanzen innerhalb einer SSH-Verbindung bei deren Terminierung ebenfalls beendet werden, wurde auf das Programm `at` zurück gegriffen um dieses Verhalten zu umgehen. Das Programm ähnelt dabei im Verhalten einem CRON-Daemon und ist damit in der Lage, Programme als eigene Instanz im Hintergrund auszuführen. Dazu wird `at` der auszuführende Befehl als String übergeben. Der Parameter *now* sorgt für die sofortige Ausführung des Programmes.

```
8  for server in $servers
9  do
10     echo "###### Start up server on $server"
11     sshpass -p $pass ssh -t $user@$server 'bash -cl "cd ~/ba-system3; echo ↩
           \"./zmq-server.py\" | at now"'
12     sshpass -p $pass ssh -t $user@$server 'bash -cl "cd ~/ba-system3; echo ↩
           \"./load.py\" | at now"'
13 done
14
15 for client in $clients
16 do
17     echo "###### Start up client auf $client"
18     sshpass -p $pass ssh -t $user@$client 'bash -cl "cd ~/ba-system3; echo ↩
           \"./clientstarter.py\" | at now"'
19     sshpass -p $pass ssh -t $user@$client 'bash -cl "cd ~/ba-system3; echo ↩
           \"./load.py\" | at now"'
20 done
```

Listing 10: Startskript runzmq.sh des Chat-Systems 2/2

4.3 Logging

4.3.1 Konzeption

Wie bereits mehrfach angedeutet, wird zur Umsetzung eines vollständigen Testsystems ein zentraler Logging-Server und eine Anwendung zur Erfassung der Last der Workstations benötigt. Außerdem wird ein Modul für die Verbindung zum Logging-Server zur Verwendung in den Anwendungen benötigt. Um den Aufwand der Implementierung gering zu halten, soll dabei auf das in Python integrierte Modul *logging* zurückgegriffen werden[41]. Dieses Modul bietet bereits alle nötigen Klassen zur Umsetzung eines Logging-Systems auf Basis des TCP.

Für den Logging-Server soll eine Implementation aus dem Python *logging cookbook* dienen[42]. Da dieser Server bereits alle benötigten Funktionen enthält und als ausgereift betrachtet werden kann, sind nur minimale Änderungen an der Konfiguration erforderlich. Auf Grund dieser Entscheidung entfällt die eigene Implementierung des Servers und ermöglicht eine schnellere Bereitstellung dieser Systemkomponente. In der folgenden Beschreibung der Implementierung wird deshalb nicht weiter auf diese Server-Komponente eingegangen.

Das zu implementierende globale Modul für die Anwendungen soll eine hohe Wiederverwendbarkeit ermöglichen und redundanten Code vermeiden. Dazu soll im Modul eine Klasse mit einheitlichen Methoden implementiert werden. Über die Methoden soll den Anwendungen das Erfassen der Messwerte im RAM ermöglicht werden. Im Weiteren soll die Klasse das Übertragen der Messwerte an den Logging-Server implementieren. Die Ansteuerung erfolgt dabei über eine extern verfügbare Methode. Aus Gründen der Redundanz soll diese Klasse auch eine persistente Speicherung aller Messwerte umsetzen. Dies stellt sicher, dass selbst bei einer fehlerhaften Übertragung der Messwerte ein Backup existiert. Neben der allgemeinen Logging-Funktion soll das Modul auch die Klasse zur Erfassung der CPU- und RAM-Auslastung auf der Workstation enthalten. Das Auslesen soll dabei, wie bereits in den Anforderungen beschrieben, in festen Intervallen erfolgen. Dies soll die zusätzliche Belastung der Workstation durch die damit verbunden Datei-Zugriffe des Betriebssystem möglichst gering halten, um die Testergebnisse nicht zu beeinflussen. Die Übertragung der erfassten Messwerte soll dabei mit Hilfe der zuvor konzipierten Klasse erfolgen.

Die Anwendung zur Erfassung der Last auf den Workstations soll auf dem zuvor beschriebenen Modul basieren. Dabei soll sie die Kommunikation mit dem Testcontroller übernehmen und mit den erhaltenden Kommandos für die korrekte Abarbeitung der Lasterfassung sorgen. Ihr Aufbau richtet sich nach dem Testablauf, wie er im Abschnitt Teststeuerung definiert wurde.

Während der bereits laufenden Testphase wurde festgestellt, dass zur Einschätzung des Lastverhaltens der Systeme unter Verwendung des AMQP auch das Lastverhalten des Brokers zu berücksichtigen ist. Da zu diesem Zeitpunkt das System bereits vollständig implementiert war und die Testreihen für das Chat-System bereits abgeschlossen waren, kam ein Einsatz des zuvor konzipierten Testsystems für den Broker nicht mehr in Frage. Dies begründet sich auch durch die Tatsache, dass der Broker eines AMQP-System eine dauerhaft laufende Komponente ist und somit nicht gestartet werden muss. Es wurde entschieden den Speicherverbrauch des Brokers und die CPU-Last über ein Bash-Skript zu erfassen, welches zu Beginn jedes Testlaufs gestartet werden sollte. Dabei soll der Speicherverbrauch des Brokers über das mitgelieferte Monitoring-Tool des RabbitMQ-Brokers erfolgen. Die CPU-Last soll direkt im Betriebssystem erfasst werden.

4.3.2 Implementierung

In den folgenden Abschnitten soll nun auf die Implementierung der zuvor konzipierten Klassen eingegangen werden. Dazu wird im ersten Abschnitt der Aufbau der Klasse *TCPLogger* erläutert. Diese implementiert die allgemeinen Logging-Funktionen und die Übertragung der Messwerte an den Logging-Server. Danach wird die Implementation der Klasse *LoadLoggerThread*, welche die Lasterfassung auf Betriebssystemebene realisiert, besprochen. Abschließend soll auf die Implementierung der Lasterfassung des Brokers eingegangen werden.

TCP-Logger Wie Listing 11 auf Seite 33 auf Zeile 112 zeigt, wurde die Klasse TCPLogger von der Python internen Threadklasse abgeleitet. Die Umsetzung über Threading wurde gewählt, um eine Blockierung der Anwendung während des Loggens zu vermeiden. Über die, als Konstruktor wirkende Init-Methode auf Zeile 116 werden der Klasse sowohl die IP-Adresse des Logging-Servers als auch eine Id als Parameter übergeben. Die Id entspricht dabei dem Typ der Anwendung und dient der späteren Identifizierung der Logeinträge in der Log-Datei. Zum gleichen Zweck ermittelt die Klasse den Hostnamen der Workstation. Diese Art der Implementierung unterstützt aber auch die flexible Nutzung der Klasse in anderen Bestandteilen des System. Die auf Zeile 125 erzeugte Liste `self.__messages` enthält später die einzelnen Strings mit den erfassten Messwerten. Über die Methode `getLogger` mit dem Parameter self.___ID, auf Zeile 127, wird ein Logger mit der zugewiesenen ID erzeugt. Es wird davon ausgegangen, dass diese ID eindeutig ist. Um Ressourcen zu schonen erfolgt aber noch keine Zuweisung eines definierten Log-Handlers. Mit dem Aufruf von *Thread.___init___(self)* wird die Elternklasse initialisiert. Dies entspricht dem Methodenaufruf `super` in Java.

```
112 class TCPLogger(Thread):
113     """
114     TCP-Logger Objekt
115     """
116     def __init__(self, loggerip, id):
117         """
118         Init
119         @param loggerip log server IP
120         @param id Identification for logger
121         """
122         self.__ID = id
123         self.__HOSTNAME = socket.gethostname()
124         self.__LOGGER_IP = loggerip
125         self.__messages = []
126 
127         logger = logging.getLogger(self.__ID)
128         logger.setLevel(logging.DEBUG)
129         logger.propagate = False
130         self.__LOGGER = logger
131         Thread.__init__(self)
```

Listing 11: Init-Methode der Klasse TCPLogger (`Logger.py`)

Anhand des Codes in Listing 11 ist erkennbar, dass in der Init-Methode kein Aufruf der Run-Methode aus Listing 12 erfolgt. Dieses Vorgehen wurde gewählt, da zum Zeitpunkt eines laufenden Tests noch keine Verbindung zum Logging-Server aufgebaut werden soll, da dies zu einer möglichen Verfälschung der Testergebnisse führen könnte. Erst wenn die Run-Methode direkt aufgerufen wird, erfolgt auf Zeile 140 das Erzeugen eines Socket-Handlers inklusive dem Aufbau der Verbindung für die Kommunikation mit dem Logging-Server.

```
133     def run(self):
134         """
135         Run-Method
136         @brief Send all available log messages to log server. In addition ↩
                backup on harddisk
137         """
138         out = open(self.__ID + ".txt","w")
139         try:
140             socketHandler = logging.handlers.SocketHandler(self.__LOGGER_IP, ↩
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
141             frm = logging.Formatter("%(message)s")
142             socketHandler.setFormatter(frm)
143             self.__LOGGER.addHandler(socketHandler)
144 
145             self.__LOGGER.debug("\"" + self.__HOSTNAME + "\",\"start\"")
146             for msg in self.__messages:
147                 self.__LOGGER.debug("\"" + self.__HOSTNAME + "\",\"" + msg[0] ↩
                        + "\"")
148                 out.write("\"" + self.__HOSTNAME + "\",\"" + msg[0] + "\"\n")
149             self.__LOGGER.debug("\"" + self.__HOSTNAME + "\",\"end\"")
150         except Exception, e:
151             out.write(str(e) + "\n")
152             print str(e)
153         time.sleep(1)
154         print "Logger " + self.__ID + " exited"
155 
156     def add_logmessage(self, msg):
157         """ Add a new message to logger
158         @param msg The message
159         """
160         self.__messages.append([msg])
```

Listing 12: Run-Methode der Klasse TCPLogger (`Logger.py`)

Anschließend werden dem Logger durch den Aufruf der Methode `debug` die Logeinträge übergeben. Durch den Socket-Handler erfolgt dann das sofortige Versenden dieser Logeinträge an den Logging-Server. Wie man auf den Zeilen 145 und 149 sehen kann, werden vor der Übergabe der eigentlichen Logeinträge mit den Strings *start* und *end* klare Marken gesetzt. Sie dienen dem Zweck einer schnellen Überprüfung der Log-Datei, am Logging Server, auf mögliche Fehler während der Übertragung. Auf den Zeilen 146 bis 148 werden alle Strings, aus der Liste, an den Logger übergeben. Um die geforderte Redundanz umzusetzen wird parallel in eine zuvor geöffnete Datei geschrieben Die Liste muss über die, auf den Zeilen 156 bis 161, definierte Methode `add_logmessage` mit Logeinträgen befüllt werden. Sind alle Logeinträge geschrieben, beendet sich der Logger.

Last-Logger Die Klasse LoadLoggerThread wurde ebenfalls als Thread implementiert. Die Init-Methode auf Zeile 22 des Listing 13 zeigt, dass zur Instantiierung des Objektes zwei Parameter übergeben werden. Der erste Parameter legt dabei das Intervall für die Lasterfassung fest und dient der Initialisierung des Timers auf Zeile 31. Als Callback-Funktion des Timers wird dort die Methode `__log` definiert. Der zweite Parameter wird für die Instantiierung des TCPLoggers auf Zeile 28 benötigt und gibt die IP-Adresse des Logging-Servers an. Die auf Zeile 30 definierte Variable `__seccounter` wird als Zähler verwendet und dient der Erzeugung der Zeitstempel in den Logeinträgen. Nachdem auf Zeile 32 eine Queue für die asynchrone Steuerung der Klasse erzeugt wird, erfolgt zuletzt die Initialisierung des Elternobjektes.

```
3  """
4  @package log
5  Logger classes
6  @author Martin Stoffers
7  """
8
9  import time
10 import socket
11 import logging, logging.handlers
12 import Queue
13
14 from threading import Timer
15 from threading import Thread
16
17
18 class LoadLoggerThread(Thread):
19     """
20     A simple thread based load logger
21     """
22     def __init__(self, checkseconds, loggerip):
23         """
24         Init
25         @param checkseconds logging interval
26         @param loggerip log server ip
27         """
28         self.__LOGGER = TCPLogger(loggerip, 'Load')
29         self.__CHECKSECONDS = checkseconds
30         self.__seccounter = 0
31         self.__t = Timer(self.__CHECKSECONDS, self.__log)
32         self.__QUEUE = Queue.Queue()
33         Thread.__init__(self)
```

Listing 13: Init-Methode der Klasse LoadLoggerThread (`Logger.py`)

Wie in Listing 14 zu sehen ist, erfolgt nach dem Aufruf der Run-Methode der Start des Timers auf Zeile 40. Anschließend wird über die blockierende Methode `get` der Queue, innerhalb einer Endlosschleife, auf den Eingang von Steuernachrichten gewartet. Dieses Vorgehen hat den Vorteil, dass der Thread durch die Blockierung vom Betriebssystem schlafen gelegt werden kann. Hingegen würde eine einfache While-Scheife das System wesentlich stärker belasten und könnte damit die Messergebnisse des Tests verfälschen. Die Nachrichten werden, mit Ausnahme des Timers, über externe Ereignisse erzeugt. Dabei wird bei der Nachricht *stop* das Logger-Objekt durch den Aufruf der Run-Methode mit dem Versenden der Logeinträge beauftragt (Z. 47). Wird durch die Methode `join` das erfolgreiche Versenden registriert, wird der Thread beendet. Bei einer Nachricht mit dem Inhalt *error* erfolgt die sofortige Beendigung des Threads. Wird hingegen eine Nachricht mit dem Inhalt *log* empfangen, werden dem Logger-Objekt neue Logeinträge hinzugefügt und der Zähler um ein Intervall erhöht (Z. 52-54).

```
35      def run(self):
36          """
37          Run-Method
38          @brief Controlls the thread state
39          """
40          self.__t.start()
41          while True:
42              data = self.__QUEUE.get()
43              if data == "stop":
44                  # Send logs and quit
45                  self.__t.cancel()
46                  print "Send Logs"
47                  self.__LOGGER.start()
48                  self.__LOGGER.join()
49                  break
50              if data == "log":
51                  # Excecute logging
52                  self.__LOGGER.add_logmessage(str(self.__seccounter) + " CPU:" ↩
                        + self.__getCPUusage())
53                  self.__LOGGER.add_logmessage(str(self.__seccounter) + " RAM:" ↩
                        + self.__getRAMusage())
54                  self.__seccounter += self.__CHECKSECONDS
55              elif data == "error":
56                  # Quit on error
57                  self.__t.cancel()
58                  break
59
60      def __log(self):
61          """
62          Timer callback for logging
63          @brief Put a log message into the queue for logging
64          """
65          self.__QUEUE.put("log")
66          self.__t = Timer(self.__CHECKSECONDS, self.__log)
67          self.__t.start()
```

Listing 14: Run-Methode in der Klasse LoadLoggerThread (`Logger.py`)

Bevor einer der beiden Logeinträge erzeugt werden kann, werden zunächst die Methoden `__getCPUusage` und `__getRAMusage` aufgerufen um die aktuelle Systemlast zu erfassen. Dabei wird, wie in Listing 15 auf Seite 36 für die CPU Lasterfassung zu erkennen ist, über das virtuelle proc-Dateisystem die Datei *loadavg* ausgelesen. Nach einer Bearbeitung der Daten wird das Ergebnis als String zurückgegeben. Dieses Verfahren wurde mit einigen Anpassungen auch in der Methode zur Erfassung der RAM-Belastung gewählt.

```
83      def __getCPUusage(self):
84          """
85          CPU usage logging
86          @brief Logs CPU usage from /proc/loadavg
87          @retval String CPU usage
88          """
89          loadavg = open("/proc/loadavg","r")
90          usage = loadavg.readline().strip()
91          loadavg.close()
92          return usage
93
94      def __getRAMusage(self):
95          """
96          RAM usage logging
97          @brief Logs RAM usage from /proc/meminfo
98          @retval String RAM usage
99          """
100         meminfo = open("/proc/meminfo","r")
101         total = meminfo.readline()
102         free = meminfo.readline()
103         meminfo.close()
104
105         total = float(total.split()[1])
106         free = float(free.split()[1])
107         used = total - free
108         return str(total) + ", " + str(used) + ", " + str(free)
```

Listing 15: Lasterfassung in der Klasse LoadLoggerThread (`Logger.py`)

Anwendung zur Lasterfassung Nach dem Start der Anwendung erfolgt das Auslesen der Systemkonfiguration, gefolgt von einer Überprüfung der Gültigkeit der darin enthaltenen Angaben. Diese erfolgt auf Zeile 26 des Listing 16. Ist dies erfolgreich, wird auf Zeile 28 die Verbindung zum Testcontroller aufgebaut und die Anmeldung mit einem *l hello* auf Zeile 33 durchgeführt.

```
21      CLIENTS = config.getint('Test', 'max_clients')
22      CLIENTSTATIONS = config.items('Client')
23      CNC = None
24
25      # Check configuration
26      if (((CLIENTS % len(CLIENTSTATIONS)) == 0) or len(CLIENTSTATIONS) < ↩
            CLIENTS) and CLIENTS > 0:
27          try:
28              CNC = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
29              # Connect to controller
30              CNC.connect((CONTROLLER_IP, 1337))
31
32              # Say hello to controller
33              CNC.send('l hello')
```

Listing 16: Initalisierung der Lasterfassung (`load.py`)

Anschließend wird, wie in Listing 17 auf Seite 37 auf Zeile 37 zu sehen ist, auf weitere Kommandos des Controllers gewartet. Erreicht die Anwendung das Kommando *l start*, so startet sie den Thread zur Erfassung der Last. Zuvor wird jedoch geprüft, ob überhaupt eine Erfassung stattfinden soll. Mit dem Kommando *l exit* wird auf Zeile 52 das Versenden der Logeinträge durch den Thread veranlasst. Ist dies erfolgt, so wird die Aktion bestätigt und die Anwendung beendet (Z. 53-55) Erfolgt eine Fehlermeldung durch den Testcontroller, so wird die Anwendung sofort beendet (Z. 56-62). Ein Versenden der Logeinträge erfolgt nicht.

```
35              while True:
36                  try:
37                      data = CNC.recv(1024)
38                      if data == 'l start':
39                          # Start logger
40                          if CHECKSECONDS > 0:
41                              LOADLOGGER = LoadLoggerThread(
42                                      CHECKSECONDS,
43                                      LOGGER_IP
44                              )
45                              LOADLOGGER.start()
46                              print 'Logger active'
47                      if data == 'l exit':
48                          # Send logs and quit
49                          print 'Logger stop'
50                          if CHECKSECONDS > 0:
51                              LOADLOGGER.stop()
52                              LOADLOGGER.join()
53                          CNC.send('l exit')
54                          CNC.close()
55                          break
56                      if data == 'error':
57                          # Go offline without sending the logs
58                          if CHECKSECONDS > 0:
59                              LOADLOGGER.error()
60                              LOADLOGGER.join()
61                          CNC.close()
62                          break
```

Listing 17: Implementierung der Kommunikation mit dem Testcontroller (`load.py`)

Lasterfassung RabbitMQ-Broker Wie in Listing 18 zu erkennen ist, wurde die Lasterfassung über eine einfache For-Schleife implementiert. Da zum Zeitpunkt der Implementierung bereits der Zeitraum und die Häufigkeit der Lasterfassung für die Testdurchläufe festgelegt wurde, sind diese Werte im Programm `seq` statisch hinterlegt. Durch die angebenen Parameter 0, 10 und 320 wird die Variable **i** innerhalb der Schleife in Zehner Schritten von 0 bis 320 erhöht. Sicherlich wäre auch das Auslesen des Testzeitraums aus der globalen Systemkonfiguration umsetzbar gewesen. Dies wurde aber aus Zeitgründen nicht in Erwägung gezogen. Das Programm `sleep` auf Zeile 7 sorgt für die Erfassung im vorgegebenen Intervall. Auf Zeile 6 wird bei jedem Schleifendurchlauf über das Monitoringprogramm `rabbitctl` der Status des Brokers abgefragt. Die relevanten Zeilen werden über `grep` gefiltert und in die Log-Datei rabbit.log geschrieben. Das Programm geht von einem manuellen Start durch den Benutzer aus. Da der RabbitMQ-Broker auf dem selben System wie der Logging-Server und der Testcontroller verortet war, konnte das Skript ohne großen Aufwand bei jedem Testlauf gestartet werden.

```
1 #!/bin/bash
2 for i in $(seq 0 10 320)
3 do
4     echo $i
5     echo "$i ========================" >> rabbit.log
6     rabbitmqctl status | grep -e 'total,\|connection_procs\|queue_procs,' >> ↩
          rabbit.log
7     sleep 10
8 done
9 echo "bye"
```

Listing 18: Lasterfassung für den RabbitMQ-Broker (`rabbitlog.sh`)

5 Testaufbau Chat-System

5.1 Anforderungen

Vor Beginn der Konzeption des Chat-Systems wurden aus den allgemeinen Anforderungen und der Konzeption des Testsystems die Anforderungen für das zu implementierende System entwickelt. Diese Anforderungen sind nachfolgend aufgelistet.

1. Anforderungen durch das Testsystem:
   - Die Kommunikation des Servers mit den Clients ist unabhängig vom Messagingsystem
   - Gewährleistung des dynamischen Aufbaus der Systeme durch die Systemparameter
   - Einbeziehung der vorhanden Hardwareressourcen beim Systemaufbau
   - Separate Ausführung für jedes Messaging-System entwerfen
   - Erfassung aller geforderten Messpunkte
   - Umsetzung aller festgelegten Testparameter
   - Belastung des Systemes durch Messwerterfassung gering halten
   - Nachrichten für die Clients sollen immer über das Messagingsystem verteilt werden
   - Die Messagingsysteme sollen vergleichbare Messaging-Pattern nutzen
   - Die Messagingsysteme sollen vergleichbare Übertragungsparadigmen verwenden
2. Anforderungen an das Chat-System:
   - Chat-Clients müssen über mehrere Server miteinander kommunizieren können
   - Alle Nachrichten sollen an alle Clients verteilt werden (Broadcast-Chat)
   - Die Kommunikation ist für die Chat-Clients transparent.

5.2 Konzeption

5.2.1 Chat-Clients

Um einen dynamischen Systemaufbau unter Beachtung der vorhanden Hardwareressourcen zu erreichen, sollen mehrere Chat-Clients auf einer Workstation ausgeführt werden können. Damit kann eine gleichmäßigere Auslastung aller Workstations in der Testumgebung erreicht werden. Dazu soll zunächst bestimmt werden, wieviel Clients pro Workstation zu starten sind und mit welchen Parametern diese Clients arbeiten sollen. Die Anzahl der zu startenden Clients kann dabei über die Summe der bereit gestellten Workstations und dem Wert der Variable *max_clients* aus der globalen Konfigurationsdatei berechnet werden. Ein Start der Chat-Clients hat nur zu erfolgen, wenn die Konfiguration gültig ist. Zur Umsetzung dieses Konzepts soll die Client-Software funktional aufgeteilt werden. Wie in Abbildung 22 auf Seite 39 zu sehen ist, soll die Hauptroutine, hier Client-Starter genannt, die einzelnen Client-Threads starten. Über die Hauptroutine soll auch die Kommunikation mit dem Testcontroller erfolgen. In diesem Punkt soll sich die Implementierung weitestgehend an die Implementierung des Programms für Lasterfassung auf den Workstations halten. Innerhalb eines Client-Threads ist das zuvor implementierte Logging-Modul wieder zu verwenden, um die erfassten Messwerte an den Logging-Server senden zu können. Über einen weiteren Thread soll die Kommunikation mit dem Chat-Server über das Websocket-Protokoll erfolgen.

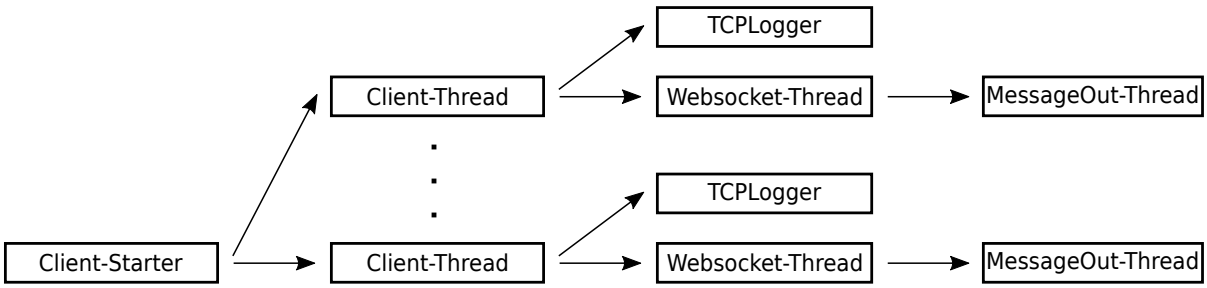

Abbildung 22: Schematischer Aufbau der Chat-Client-Anwendung

Aus den Festlegungen der Testparameter aus Abschnitt 3.2.1 geht hervor, dass die Nachrichtengröße und das Nachrichtenaufkommen im System beeinflussbar sein müssen. In einem Chat-System kann dies nur sinnvoll durch die Chat-Clients erfolgen. Um diese Vorgabe zu erfüllen, muss der Chat-Client in der Lage sein unabhängig von den zu empfangenden Nachrichten kontinuierlich Nachrichten zu versenden. Die Umsetzung soll über einen separaten Thread erfolgen. Dies garantiert, dass sowohl Empfang als auch Versand der Nachrichten über die bidirektional Websocketverbindung gleichzeitig erfolgen kann. Da das Chat-System als simpler Broadcast-Chat ausgeführt werden soll, ist zur Umsetzung des Clients keine Anmeldung des Chat-Clients erforderlich. Die Kommunikation mit dem Chat-Server zum Aufbau einer Verbindung kann daher auf ein Minimum begrenzt werden. Wie in Abbildung 23 zu erkennen ist, soll zu Beginn nur ein Nickname an den Server gesandt werden. Damit entfällt für den Client das Versenden des Nicknames mit jeder weiteren Nachricht. Aus Sicht des Testsystems könnte dieser Vorgang auch entfallen. Er erleichtert aber die Fehlersuche und bildet ein realeres Chat-System ab. Zusätzlich könnte mit seiner Hilfe eine detaillierte Erfassung der Messwerte umgesetzt werden.

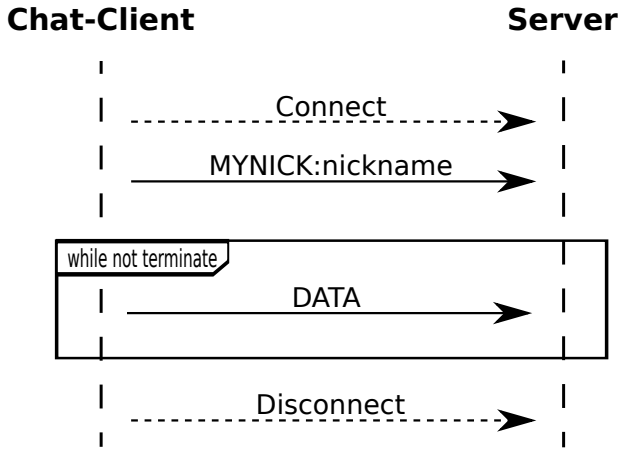

Abbildung 23: Kommunikation zwischen Chat-Client und Chat-Server

5.2.2 Chat-Server

Zur Umsetzung der geforderten Anforderungen soll auch der Server in unterschiedliche Klassen eingeteilt werden. Die Kommunikation mit dem Testcontroller soll ebenso wie das Versenden der Messwerte mit der Klasse TCPLogger über die Hauptroutine erfolgen. Auch die Auswertung der globalen Systemkonfiguration erfolgt in diesem Programmteil. Wie in Abbildung 24 auf Seite 40 zu erkennen, soll durch die Hauptroutine der Start der Threads zur Kommunikation mit den anderen Teilnehmern erfolgen. Der Frontend-Thread

dient dabei der Kommunikation mit den Chat-Clients über das Websocket-Protokoll nach dem bereits beschriebenen Verfahren. Im Backend-Thread soll die Kommunikation über das verwendete Messaging-System erfolgen. Diese Komponente muss für beide Messaging-Systeme implementiert werden. Dabei hat die Kommunikation zwischen Frontend- und Backend-Thread über die gleichen Schnittstellen zu erfolgen, um eine Austauschbarkeit der Messaging-Systeme zu gewährleisten.

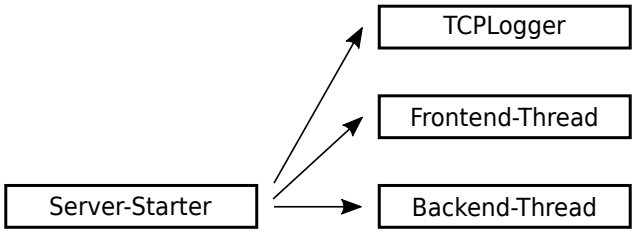

Abbildung 24: Schematischer Aufbau des Chat-Servers

AMQP-Backend Für die Konzeption des Chat-Systems mit AMQP als Messaging-System muss zunächst der Aufbau dieses Backends festgelegt werden. Wie in Abbildung 25 zu erkennen ist, erfolgt die Verteilung der Nachrichten über den zentralen RabbitMQ-Broker. Um zu gewährleisten, dass alle Nachrichten verteilt werden können, sollen alle Server über einen Fanout-Exchange im Broker kommunizieren. Folglich muss jeder Server die Nachrichten seiner Clients an diesen Exchange weiterleiten, und agiert somit als Producer. Um im Gegenzug alle Nachrichten der anderen Server zu erhalten, muss sich ein Server zum selben Exchange über eine Queue verbinden und ist somit auch Consumer. Das verwendete Messaging-Pattern ist also Publish/Subscribe.

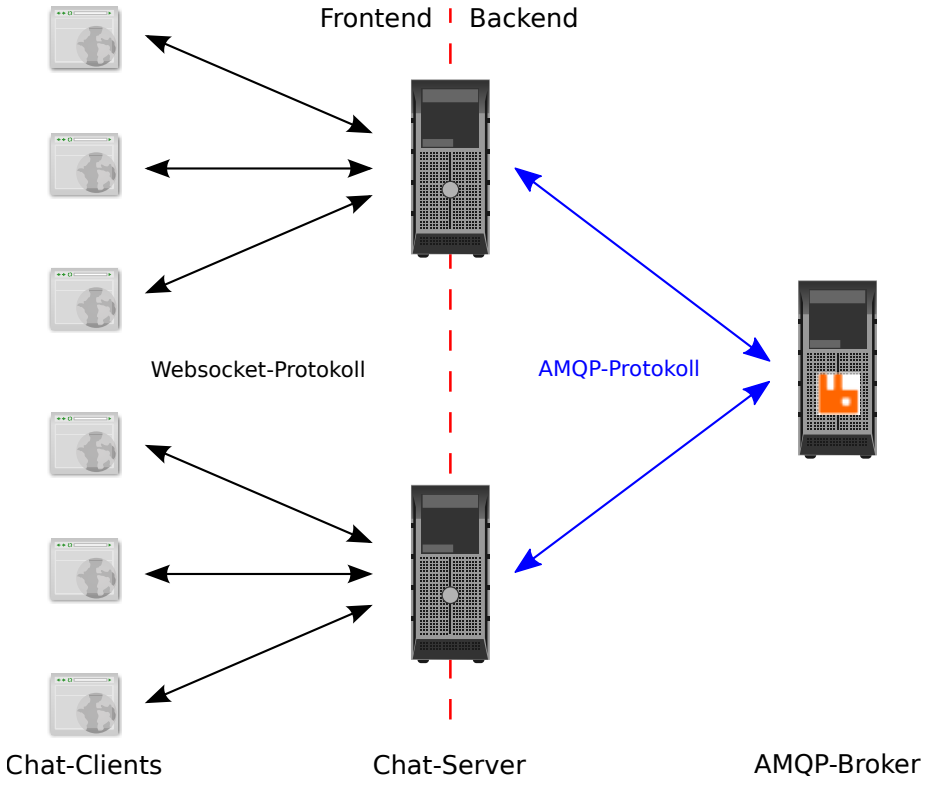

Abbildung 25: Aufbau des Chatsystems mit AMQP

Um die Verbindung der Server über den Broker zu verdeutlichen, soll Abbildung 26 einen detaillierten Überblick liefern. Wie hier deutlich zu sehen ist baut jeder Server zwei Verbindungen zum AMQP-Server auf. Um das Chat-System im Broker getrennt von anderen Systemen zu halten, soll ein Virtualhost mit Zugangsbeschränkungen verwendet werden. Auf die Möglichkeit der transaktionalen Nachrichten soll, mit Blick auf die Vergleichbarkeit mit ZeroMQ, verzichtet werden.

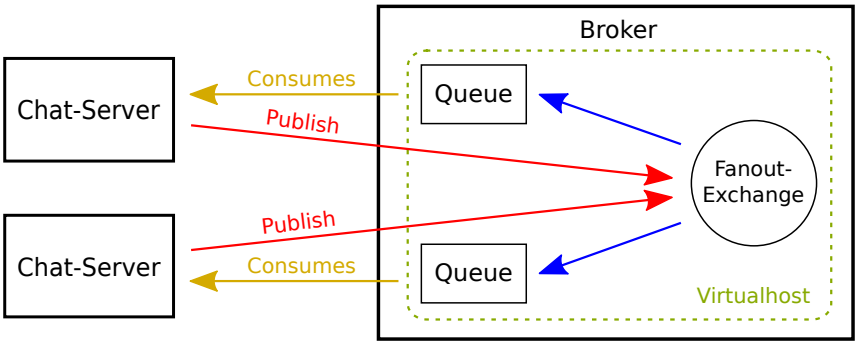

Abbildung 26: Detailansicht des Chatsystems mit AMQP

ZeroMQ-Backend Um eine Umsetzung des Backend mit ZeroMQ zu ermöglichen, muss auch hier zunächst der Aufbau des Backends festgelegt werden. Da ZeroMQ über keinen zentralen Broker verfügt, soll für den Test eine direkte Verbindung der Server untereinander erfolgen. Auf die Implementierung eines Proxys oder Routers soll somit verzichtet werden. Wie Abbildung 27 zeigt bietet jeder Server sowohl einen Pub-Socket als auch einen Sub-Socket an. Damit wird auch im ZeroMQ-System das Messaging-Pattern Publish/Subscribe verwendet. Da jeder Server eine Verbindung zu allen anderen Servern aufbauen muss, ist es unumgänglich der Anwendung alle anderen Server bekannt zu geben. Durch die Informationen aus der Konfigurationsdatei ist dies gegeben und soll für die späteren Test ausreichen.

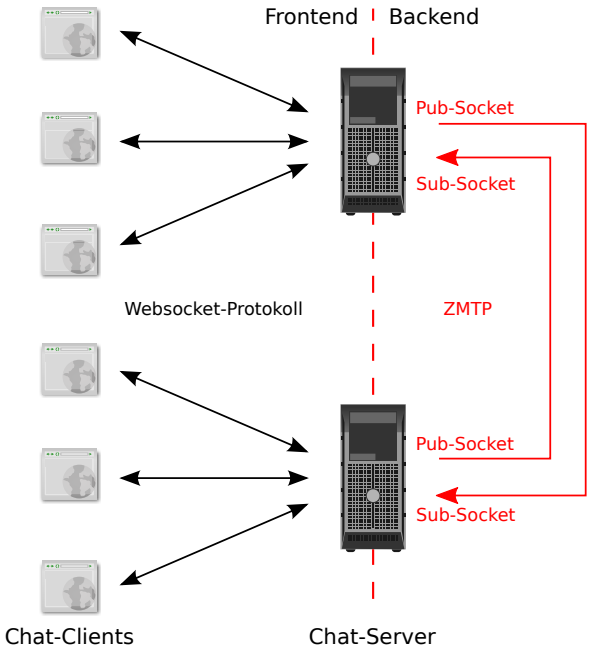

Abbildung 27: Aufbau des Chatsystem mit ZeroMQ

5.3 Implementierung

5.3.1 Chat-Client

Nach dem Start des Chat-Clients erfolgt zunächst das Auslesen aller relevanten Werte, für den Chat-Client, aus der Konfigurationsdatei. Dies sind insbesondere die Werte *max_clients*, *sleep_seconds* und *data_size_byte*. Für die Berechnung der zu startenden Client-Threads werden zusätzlich alle Workstations über die Methode `items` in eine Liste eingelesen. Die Berechnung erfolgt dann, wie in Listing 19 auf Zeile 32 zu sehen, durch eine einfache Division der insgesamt zu startenden Chat-Clients durch alle verfügbaren Workstations. Da das Ergebnis dieser Berechnung ganzzahlig sein muss, erfolgt zunächst eine Überprüfung der Konfigurationsdatei. Schlägt die Prüfung fehl, so werden keine Client-Threads gestartet und das Programm sofort beendet. Im Erfolgsfall wird auf Zeile 36 versucht eine Verbindung zum Testcontroller aufzubauen. Mit dem Exception-Handling, welches auf Zeile 33 eingeleitet wird, ist sichergestellt, dass die Anwendung sich beendet wenn kein Testcontroller erreicht werden kann. Wurde die Verbindung zum Controller erfolgreich aufgebaut, so erfolgt das Erzeugen der Client-Threads auf den Zeilen 39-48. Innerhalb der dortigen Schleife sei besonders auf Zeile 40 verwiesen. Dort erfolgt, mit Hilfe des Moduls *random*, eine zufällige Auswahl des anzusprechenden Chat-Servers aus der Liste SERVER_IPS. Sind alle Threads fehlerfrei erzeugt worden, wird das Kommando *c hello* an den Testcontroller gesendet. Anschließend tritt die Anwendung in eine Endlosschleife ein und wartet auf die nächsten Befehle durch den Testcontroller. Wird das Kommando zum Starten des Tests empfangen, so wird es über den Aufruf der Methode `test` innerhalb einer For-Schleife an alle Threads weitergegeben. Da diese Implementierung sich an den Testablauf aus Abbildung 20 auf Seite 23 hält und die Art der Implementation bereits durch die Anwendung zur Lasterfassung der Workstations bekannt ist, soll an dieser Stellt nicht weiter darauf eingegangen werden.

```
31      if (((CLIENTS % len(CLIENTSTATIONS)) == 0) or len(CLIENTSTATIONS) < ↩
            CLIENTS) and CLIENTS > 0:
32          CLIENTS_TO_EXECUTE = CLIENTS / len(CLIENTSTATIONS)
33          try:
34              # Make socket and connect to controller
35              CNC = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
36              CNC.connect((CONTROLLER_IP, 1337))
37
38              # Generate Client-Threads
39              for num in range(CLIENTS_TO_EXECUTE):
40                  server_ip = random.choice(SERVER_IPS)
41                  thread = ClientThread(
42                          str(num+1),
43                          MSGSPEED,
44                          DATASIZE,
45                          LOGGER_IP,
46                          server_ip[1]
47                  )
48                  THREADS += [thread]
49
50              # Say hello to controller
51              CNC.send('c hello')
52              while True:
53                  try:
54                      data = CNC.recv(20)
55                      if data == 'c start':
56                          # On start send a message to each threads
57                          for thread in THREADS:
58                              thread.test()
```

Listing 19: Hautroutine des Chat-Clients (`clientstarter.py`)

Bezüglich der gerade beschriebenen Implementierung soll angemerkt werden, dass die Kommunikation mit dem Testcontroller zunächst durch jeden Client-Thread erfolgte. Dies hat den Vorteil, dass mögliche Fehler besser einem einzelnen Chat-Client zugeordnet werden können. Jedoch wurde bei Testläufen des Systems festgestellt, dass dieses Konzept die maximale Anzahl an Clients beschränkte, da der Testcontroller nicht mehr als 1000 Dateizugriffe und damit Verbindungen verwalten konnte. Das Problem lies sich auf die Beschränkung der Dateizugriffe durch das Betriebssystem zurückführen. Um den Test flexibel zu halten, wurde deshalb auf das bereits beschriebene Konzept zurückgegriffen.

Client-Thread Zur Implementierung des eigentlichen Chat-Clients wurde, wie in Listing 20 auf Zeile 15 zu sehen ist, der Thread von der Thread-Klasse abgleitet. Die Implementierung folgt im Wesentlichen dem Last-Logger aus Listing 13 auf Seite 34. Jedoch werden über die Init-Methode auf Zeile 19 neben der IP-Adresse des Logging-Servers weitere Parameter übergeben. Der Parameter *number* dient als fortlaufende Nummer dazu, die einzelnen Clients in der Log-Datei des Testlaufs identifizieren zu können. Dazu wird sie bei der Erzeugung des TCP-Loggers auf Zeile 33 als Teil der ID übergeben. Die bereits erwähnte IP-Adresse des Servers wird direkt im Anschluss an den Logger als Logeintrag übergeben. Damit ist nachvollziehbar mit welchem Server jeder Client während des Testlaufs verbunden war. Auf Zeile 37 erfolgt die Initialisierung der Klasse WebsocketThread. Sie dient später der Kommunikation mit dem Chat-Server. Abschließend erfolgt mit dem Aufruf der Methode `start` der Start des Threads.

```
15 class ClientThread(Thread):
16     """
17     Websocket clientthread
18     """
19     def __init__(self, number, msgspeed, datasize, loggerip, serverip):
20         """
21         Init
22         @param number Client number
23         @param msgspeed Frequenz for message output
24         @param datasize Maximum datasize
25         @param loggerip Logger ip
26         @param serverip Server ip
27         """
28         Thread.__init__(self)
29         self.__NUMBER = number
30         self.__MSGSPEED = msgspeed
31         self.__DATASIZE = datasize
32         self.__SERVERIP = serverip
33         self.__LOGGER = TCPLogger(loggerip, 'Client ' + number)
34         self.__QUEUE = Queue.Queue()
35         self.add_logmessage("server:" + serverip)
36         #Websockt object which controlls the websocketconnection
37         self.__WS = Websocket(self, self.__NUMBER, self.__MSGSPEED, ↩
               self.__DATASIZE, self.__SERVERIP)
38         self.start()
```

Listing 20: Client-Thread (`ClientThread.py`)

Die Run-Methode arbeitet dabei, wie schon im Last-Logger erläutert, mit einer blockierenden Queue. Durch den Aufruf einer Methode des Client-Threads wird eine Nachricht in diese Queue gelegt und anhand von Bedingungen abgearbeitet. In den Listings 21 und 22 auf Seite 44 ist dies auszugsweise für die Nachricht *test* zu sehen. Sie dient dem Starten des Testlaufs. Innerhalb der Nachrichtenverarbeitung in Listing 21 wird die Klasse zur Verwaltung der Websocket-Verbindung über die Methode `runtest` auf Zeile 55 zur Aufnahme einer Verbindung mit dem Chat-Server aufgefordert.

```
51          while True:
52              data = self.__QUEUE.get()
53              if data == "test":
54                  # Connect client to server
55                  self.__WS.runtest()
```

Listing 21: Verarbeitung der Nachricht *test* im Client-Thread (`ClientThread.py`)

```
71      def test(self):
72          """
73          Excecute the test
74          @brief Puts a test message into the queue
75          """
76          self.__QUEUE.put("test")
```

Listing 22: Methode `test` zum Starten des Testlaufs (`ClientThread.py`)

Websocketverbindung Wie Listing 23 auf Zeile 14 zeigt, wurde die Klasse WebsocketThread von der Klasse WebSocketClient abgeleitet. Diese Klasse ist Bestandteil des Moduls *ws4py* und bietet eine vollständige Implementation des Websocket-Protokolls. Durch das Ableiten der Klasse und dem anschließenden Überschreiben der als Prototypen ausgeführten Methoden `opened`, `closed` und `received_message` kann relativ schnell ein einfacher Websocket-Client entworfen werden[39].

```
 7  import random as rnd
 8  import time
 9  from faker import Factory
10  from ws4py.client.threadedclient import WebSocketClient
11  import wsaccel
12  from threading import Thread
13
14  class WebsocketThread(WebSocketClient):
15      """
16      Websocket-Thread
17      """
18      def __init__(self, parent, num, msgspeed, datasize, serverip):
19          """
20          Init
21          @param parent Parent thread
22          @param num Client number
23          @param msgspeed Frequenz of message output
24          @param datasize Maximum datasize
25          @param serverip Server ip
26          """
27          self.__PARENT = parent
28          self.__NUMBER = num
29          self.__timeup = False
30          self.errors = 0
31          self.msg_in_counter = 0
32          self.msg_out_counter = 0
33          self.msg_out_fail = 0
34          self.__runtime = None
35          self.__MSGOUT = MsgOutThread(self, msgspeed, datasize)
36          rnd.seed(time.time())
37          self.__nick = self.__NUMBER + "-" + Factory.create().user_name()
38          wsaccel.patch_ws4py()
39          WebSocketClient.__init__(self, "ws://" + serverip + ":9090/ws")
```

Listing 23: Definition der Klasse WebsocketThread (`WebsocketThread.py`)

Während der Initialisierung der abgeleiteten Klasse ab Zeile 18 in Listing 24 werden auf den Zeilen 30 bis 34 alle Variablen für die Messwerterfassung erzeugt. Auf Zeile 35 wird der Thread zur Generierung der Chat-Nachrichten initialisiert. Auf 37 wird mit Hilfe des Moduls *faker* und in Kobination mit der Nummer des Threads ein eindeutiger Nickname erzeugt. Dieser dient, wie vorgesehen, der Anmeldung am Chat-Server. Durch den Aufruf der Init-Methode der Elternklasse erfolgt abschließend die Festlegung der anzusprechenden Ressource auf dem ausgewählten Chat-Server. Der Aufruf der Methode `patch_ws4py` aus dem Modul *wsaccel* verbessert die Leistungsfähigkeit des Moduls *ws4py* und wird durch deren Entwickler ausdrücklich empfohlen[43, 44].

```
14 class WebsocketThread(WebSocketClient):
15     """
16     Websocket-Thread
17     """
18     def __init__(self, parent, num, msgspeed, datasize, serverip):
19         """
20         Init
21         @param parent Parent thread
22         @param num Client number
23         @param msgspeed Frequenz of message output
24         @param datasize Maximum datasize
25         @param serverip Server ip
26         """
27         self.__PARENT = parent
28         self.__NUMBER = num
29         self.__timeup = False
30         self.errors = 0
31         self.msg_in_counter = 0
32         self.msg_out_counter = 0
33         self.msg_out_fail = 0
34         self.__runtime = None
35         self.__MSGOUT = MsgOutThread(self, msgspeed, datasize)
36         rnd.seed(time.time())
37         self.__nick = self.__NUMBER + "-" + Factory.create().user_name()
38         wsaccel.patch_ws4py()
39         WebSocketClient.__init__(self, "ws://" + serverip + ":9090/ws")
```

Listing 24: Initialisierung der Klasse WebsocketThread (`WebsocketThread.py`)

Wie bereits in Listing 21 auf Seite 44 zu sehen war, erfolgt der Aufruf der Methode `runtest` aus Listing 25 zum Startzeitpunkt des Testlaufs. Innerhalb dieser Methode wird zunächst der genaue Startzeitpunkt des Chat-Clients ermittelt und der Nickname über die Methode `add_logmessage` der Elternklasse an den TCP-Logger übergeben. Auf Zeile 94 wird versucht eine Websocket-Verbindung zum Server aufzubauen. Schlägt der Verbindungsaufbau fehl, so wird die Fehlermeldung über die geworfene Exception in Zeile 96 ebenfalls an den TCP-Logger übergeben.

```
86     def runtest(self):
87         """
88         Run-method
89         @brief Sets runtime and connect to server
90         """
91         try:
92             self.__runtime = time.time()
93             self.__PARENT.add_logmessage("nick:" + self.__nick)
94             self.connect()
95         except Exception, e:
96             self.__PARENT.add_logmessage("Connect-Error:" + str(e))
```

Listing 25: Start des Testlaufs in der Klasse WebsocketThread (`WebsocketThread.py`)

Ist die Verbindung zum Chat-Server erfolgreich aufgebaut, so wird die in Listing 26 auf Zeile 41 gezeigte Methode `opened` aufgerufen. Hier wird zunächst der Nickname an den Chat-Server gesandt. Anschließend erfolgt der Start des Threads zur Generierung der Chat-Nachrichten. Wie ab Zeile 61 in der Methode `received_message` zu sehen ist, werden die eintreffenden Chat-Nachrichten lediglich gezählt. Der Aufruf der Methode `closed` ab Zeile 50 erfolgt nach dem Schließen der Websocket-Verbindung. Die darin enthaltende Bedingung prüft, ob der Testlauf bereits abgeschlossen wurde. Ist dies nicht der Fall wird angenommen, dass die Verbindung zum Chat-Server vorzeitig abgebrochen wurde. Die in der Methode übergebenen Gründe für den Fehler werden zur besseren Analyse an den TCP-Logger übergeben. Zusätzlich wird der Thread für das Versenden der Chat-Nachrichten vorzeitig beendet.

```
41      def opened(self):
42          """
43          Callback on open connection
44          @brief Send a nickname on open connection
45          """
46          self.send("MYNICK:" + self.__nick)
47          time.sleep(0.1)
48          self.__MSGOUT.start()
49
50      def closed(self, code, reason=None):
51          """
52          Callback if connection is closed
53          @param code exitcode
54          @param reason reason for close
55          """
56          if not self.__timeup:
57              self.__MSGOUT._stop()
58              self.errors += 1
59              self.__PARENT.add_logmessage("Error:" + str(self.errors) + " ↩
                    code:" + str(code) + " reason:" + str(reason) + " time:" + ↩
                    str(time.time()-self.__runtime))
60
61      def received_message(self, message):
62          """
63          Callback on message
64          @brief Simply counts every message
65          @param message a message object
66          """
67          self.msg_in_counter += 1
```

Listing 26: Nachrichtenverarbeitung im WebsocketThread (`WebsocketThread.py`)

Erhält der Chat-Client, durch den Testcontroller, die Aufforderung die bis zu diesem Zeitpunkt vorliegenden Messwerte zu erfassen, so wird durch diesen die Methode `log` der Klasse WebsocketThread aufgerufen. Dabei wird, wie in Listing 27 auf Seite 47 auf den Zeilen 74 und 75 zu sehen, zunächst der Testlauf als abgeschlossen markiert und der Thread für den Versand der Chat-Nachrichten beendet. Danach wird die Gesamtlaufzeit des Tests berechnet. Anschließend erfolgt die Berechnung aller geforderten Messwerte und die Übergabe an den TCP-Logger (Z. 77-84).

```
69      def log(self):
70          """
71          Log states
72          @brief Logs current states of all counters on excecute
73          """
74          self.__timeup = True
75          self.__MSGOUT._stop()
76          self.__runtime = time.time() - self.__runtime
77          self.__PARENT.add_logmessage("runtime:" + str(self.__runtime))
78          self.__PARENT.add_logmessage("msg_in_count:" + str(self.msg_in_counter))
79          self.__PARENT.add_logmessage("msg_out_count:" + ↩
                str(self.msg_out_counter))
80          msgs_in = float(self.msg_in_counter)/float(self.__runtime)
81          msgs_out = float(self.msg_out_counter)/float(self.__runtime)
82          self.__PARENT.add_logmessage("msgs_out/s:" + str(msgs_out))
83          self.__PARENT.add_logmessage("msgs_in/s:" + str(msgs_in))
84          self.__PARENT.add_logmessage("msg_out_fail:" + str(self.msg_out_fail))
```

Listing 27: Erfassung der Messwerte (`WebsocketThread.py`)

Generierung der Nachrichten Die Generierung der Chat-Nachrichten des Chat-Clients erfolgt, wie bereits angedeutet, durch einen separaten Thread. Dieser Thread versendet innerhalb einer While-Schleife in einem festgelegten Intervall Nachrichten, bis dies durch das Setzen der Variable *__stop* abgebrochen wird. Dazu wird mit Hilfe des Moduls *faker*, auf Zeile 132 in Listing 28, vor dem Versenden der Nachricht ein zufälliger Text fester Größe generiert. Anschließend erfolgt das Versenden der Nachricht und die Erhöhung des Nachrichtenzählers. Schlägt das Versenden fehl, so wird im Exception-Handling der Zähler für die nicht erfolgreich versandte Nachrichten erhöht.

```
106 class MsgOutThread(Thread):
107     """
108     Thread for message generation
109     """
110     def __init__(self, parent, msgspeed, datasize):
111         """
112         Init
113         @param parent Parent object
114         @param msgspeed Frequenz of message output
115         @param datasize Maximum datasize
116         """
117         Thread.__init__(self)
118         self.__PARENT = parent
119         self.__MSGSPEED = msgspeed
120         self.__DATASIZE = datasize
121         self.__FAKE = Factory.create()
122         self.__stop = False
123 
124     def run(self):
125         """
126         Run-methode
127         @brief Sends a every x seconds
128         """
129         while not self.__stop:
130             time.sleep(self.__MSGSPEED)
131             try:
132                 data = self.__FAKE.text(self.__DATASIZE)
133                 self.__PARENT.send(data)
134                 self.__PARENT.msg_out_counter += 1
135             except Exception:
136                 self.__PARENT.msg_out_fail += 1
```

Listing 28: Klasse MsgOutThread (`WebsocketThread.py`)

5.3.2 Chat-Server

Für die Implementierung der Chat-Server wurde zur Trennung der beiden Messagingsysteme für jeden Server eine eigene Anwendung zum Starten entworfen. Beide Anwendungen sind funktional identisch aufgebaut, jedoch wird je Test eine andere Klasse für das Backend verwendet. Für die Beschreibung dieser Anwendung soll sich deshalb auf den Aufbau der Anwendung zum Starten des ZeroMQ-Servers beschränkt werden. Die Umsetzung der beiden Backends und des einheitlichen Frontends wird in getrennten Abschnitten erläutert.

Wie auch in der Hauptroutine des Chat-Clients wird im Chat-Server zunächst die Konfigurationsdatei ausgelesen und überprüft. Da je Workstation nur ein Chat-Server ausgeführt werden soll, erfolgt hier auch die Initialisierung des TCP-Loggers. Auf eine Abbildung wurde aufgrund des identischen Aufbaus verzichtet. Nach dem anschließenden anmelden am Testcontroller wartet der Server auf den Empfang seiner IP-Adresse. Diese wird im Backend für das Schreiben der Logeinträge und den Aufbau des Messaging-Systems benötigt. Auf die Ermittlung der IP-Adresse auf Seiten des Servers wurde verzichtet, da eine Ermittlung der IP-Adresse über alle verfügbaren Netzwerkinterfaces einen erheblichen Mehraufwand bedeutet hätte. Hat der Chat-Server die IP-Adresse durch den Testcontroller erhalten, werden auf den Zeilen 42 bis 46 in Listing 29 sowohl Backend- als auch Frontend-Thread gestartet. Der Backend-Thread erhält, wie auf Zeile 44 zusehen die zuvor empfangene IP-Adresse und eine Liste aller Chat-Server im Testsystem aus der Konfigurationsdatei. Dem Frontend-Thread wird als einziger Parameter die Referenz auf den Backend-Thread übergeben. Damit wird eine Kommunikation zwischen beiden Threads ermöglicht.

```
34          if (((CLIENTS % len(CLIENTSTATIONS)) == 0) or len(CLIENTSTATIONS) < ↩
                CLIENTS) and CLIENTS > 0:
35              # Try to set up the controller connection
36              s.connect((CONTROLLER_IP, 1337))
37              #Say hello to controller
38              s.send('s hello')
39              IP = s.recv(1024)
40
41              #Fire up backend
42              BACKEND = ZmqThread(IP, SERVERSTATIONS)
43              BACKEND.start()
44              # Fire up tornado server
45              FRONTEND = WebsocketThread(BACKEND)
46              FRONTEND.start()
```

Listing 29: Starter für den ZeroMQ-Chat-Server (`zmq-server.py`)

Innerhalb der While-Schleife in Listing 30 auf Seite 49 erfolgt die Verarbeitung der Kommandos vom Testcontroller nach der Konzeption aus der Teststeuerung. Empfängt der Chat-Server das Kommando *s log* zur Erfassung der Messwerte, so wird ab Zeile 55 das Backend durch die Methode `log` angewiesen, die Laufzeit des Tests zu ermitteln. Anschließend erfolgt der Abruf und die Berechnung der geforderten Messwerte über fest definierte Methoden im Backend-Thread. Die zurückgegebenen Messwerte werden dann als neue Logeinträge an den TCP-Logger übergeben. Abschließend wird die Verarbeitung der Messwerte an den Testcontroller bestätigt und auf das nächste Kommando gewartet. Der Konzeption der Teststeuerung folgend, wird als nächstes Kommando *l getlog* empfangen und ab Zeile 76 verarbeitet. Wie auch im Chat-Client wird der TCP-Logger nun angewiesen alle Logeinträge an den Logging-Server zu schicken. Abschließend erfolgt auch hier eine Bestätigung der Aktion an den Testcontroller. Wird das Kommando *s exit* empfangen, werden auf den Zeilen 81-89 sowohl Frontend- als auch Backend-Thread beendet. Danach erfolgt die Bestätigung der Aktion an den Testcontroller und die Anwendung terminiert.

```
50                      if data == 's log':
51                          # Log the current state
52                          BACKEND.log()
53                          LOGGER.add_logmessage('backend_out_msg_count:' + ↩
                                str(BACKEND.get_backend_out_msgcounter()))
54                          LOGGER.add_logmessage('backend_out_msg_total:' + ↩
                                str(BACKEND.get_backend_out_msgtotal()))
55                          LOGGER.add_logmessage('backend_in_msg_count:' + ↩
                                str(BACKEND.get_backend_in_msgcounter()))
56                          LOGGER.add_logmessage('backend_out_fail_count:' + ↩
                                str(BACKEND.get_backend_out_failcounter()))
57                          LOGGER.add_logmessage('backend_in_msg_total:' + ↩
                                str(BACKEND.get_backend_in_msgtotal()))
58                          LOGGER.add_logmessage('frontend_out_msg_count:' + ↩
                                str(BACKEND.get_frontend_out_msgcounter()))
59                          LOGGER.add_logmessage('frontend_out_msg_total:' + ↩
                                str(BACKEND.get_frontend_out_msgtotal()))
60                          LOGGER.add_logmessage('frontend_in_msg_count:' + ↩
                                str(BACKEND.get_frontend_in_msgcounter()))
61                          LOGGER.add_logmessage('frontend_in_msg_total:' + ↩
                                str(BACKEND.get_frontend_in_msgtotal()))
62                          LOGGER.add_logmessage('frontend_out_fail_count:' + ↩
                                str(BACKEND.get_frontend_out_failcounter()))
63                          LOGGER.add_logmessage('frontend_out_fail_total:' + ↩
                                str(BACKEND.get_frontend_out_failtotal()))
64                          LOGGER.add_logmessage('frontend_out_keyerror_count:' + ↩
                                str(BACKEND.get_frontend_out_keyerror()))
65                          LOGGER.add_logmessage('frontend_out_keyerror_total:' + ↩
                                str(BACKEND.get_frontend_out_keyerrortotal()))
66                          LOGGER.add_logmessage('runtime:' + ↩
                                str(BACKEND.get_runtime()))
67                          LOGGER.add_logmessage('backend_in_msgs/s:' + ↩
                                str(BACKEND.get_backend_in_msgs_per_second()))
68                          LOGGER.add_logmessage('frontend_out_msgs/s:' + ↩
                                str(BACKEND.get_frontend_out_msgs_per_second()))
69                          LOGGER.add_logmessage('avg_processtime_msg:' + ↩
                                str(BACKEND.get_process_time_min_max()))
70                          LOGGER.add_logmessage('sub_count:' + ↩
                                str(BACKEND.get_sub_count()))
71                          LOGGER.add_logmessage('unsub_count:' + ↩
                                str(BACKEND.get_unsub_count()))
72                          s.send('s logdone')
73                      elif data == 's getlog':
74                          # Transmit all available logs to log-server
75                          LOGGER.start()
76                          LOGGER.join()
77                          s.send('s logsend')
78                      elif data == 's exit':
79                          # stop all threads and go offline
80                          FRONTEND.stop()
81                          FRONTEND.join()
82                          BACKEND.stop()
83                          BACKEND.join()
84                          s.send('s exit')
85                          s.close()
86                          break
87                      elif data == 'error':
88                          # Abort test
89                          s.close()
```

Listing 30: Steuerung durch den Testcontroller im ZeroMQ-Chat-Server (`zmq-server.py`)

Websocket-Frontend Wie in Listing 31 erfolgte die Implementierung des Websocket-Frontends mit Hilfe des Tornado-Frameworks[38]. Dazu wird in der Init-Methode des Frontend-Threads auf Zeile 82 das Framework durch den Aufruf der Methode `tornado.web.Application` initialisiert. Dabei werden über eine Liste die Pfade der Ressourcen und die zuständigen Handler übergeben. Diese Handler wurden in einer eigenen Klasse implementiert und werden im nächsten Abschnitt beschrieben. Zusätzlich wird in einem Dictionary die Referenz auf den Backend-Thread übergeben. Anschließend wird über die Methode `listen` unter Angabe des Ports die Initialisierung abgeschlossen. Innerhalb der Run-Methode wird, ab Zeile 94, beim Starten des Frontends-Threads die Tornado-Applikation durch den Aufruf der Methode `tornado.ioloop.instance().start` gestartet. Dieser Aufruf ist blockierend und kehrt erst durch den Aufruf der Methode `stop` zurück. Die Beendigung der laufenden Instanz des Tornado-Servers erfolgt dabei durch den Aufruf in Zeile 109.

```
72  class WebsocketThread(Thread):
73
74      def __init__(self, backendthread):
75          """
76          Init
77          @brief Setup for the tornando server
78          @param backendthread backendthread object
79          """
80          Thread.__init__(self)
81          self.__BACKENDTHREAD = backendthread
82          self.__APPLICATION = tornado.web.Application([
83              (r'/ws',
84               WSHandler,
85               dict(
86                   backendthread=self.__BACKENDTHREAD,
87               )
88              ),
89              (r"/", MainHandler),
90              (r"/(.*)", tornado.web.StaticFileHandler, {"path": "./resources"}),
91          ])
92          self.__APPLICATION.listen(9090)
93
94      def run(self):
95          """
96          Run-method
97          @brief Start the torando serverthread
98          """
99          print "Frontend is up"
100         tornado.ioloop.IOLoop.instance().start()
101         print "Frontend is down"
102
103     def stop(self):
104         """
105         Stop-Method
106         @brief Stop the torando serverthread and send close to all connected ↩
                clients
107         """
108         ioloop = tornado.ioloop.IOLoop.instance()
109         ioloop.add_callback(lambda x: x.stop(), ioloop)
110         print "Ask fronted for graceful shutdown"
```

Listing 31: Frontend-Implementierung durch die Klasse WebsocketThread (`Frontend.py`)

Der innerhalb der Instantiierung der Tornando-Servers übergebene Websocket-Handler mit der Bezeichnung *wshandler* wurde dabei, wie in Listing 32 auf Seite 51 zu sehen, implementiert. Erfolgt durch die Chat-Clients die Verbindung zum Chat-Server, so wird für jeden dieser Clients ein eigener Websocket-Handler erzeugt. Damit erfolgt die Bearbeitung aller Nachrichten eines Chat-Clients immer über diesen Handler. Bei einer neuen Verbindung wird zunächst die Methode `initialize` aufgerufen. Der Methode wird dabei

die Referenz auf den Backend-Thread übergeben und dort für die spätere Verwendung im Websocket-Handler abgespeichert. Zusätzlich erfolgt auf Zeile 41 die Kompilierung des regulären Ausdrucks zur Erfassung des Nicknames. Die auf Zeile 45 zu sehende Methode `open` wird aufgerufen, wenn die Verbindung zum Chat-Client endgültig aufgebaut wurde. Sie wird in dieser Implementierung aber nicht verwendet.

Nach der Konzeption und der Implementierung des Chat-Clients erfolgt direkt nach dem Aufbau der Verbindung das Versenden des Nicknames. Auf den Zeilen 57 bis 61 wird dieser innerhalb der Methode `on_message` über einen regulären Ausdruck ausgewertet und im Websocket-Handler gespeichert. Gleichzeitig erfolgt die Registrierung des Chat-Clients am Backend-Thread durch den Aufruf der Methode `subscribe`. Hierbei wird der Websocket-Handler an den Backend-Thread übergeben. Trifft die Überprüfung der Nachricht nicht auf den regulären Ausdruck zu, wird die Nachricht auf Zeile 63 um den Nicknamen des Chat-Clients erweitert und über die Methode `send` an den Backend-Thread übergeben.

Durch den Aufruf der Methode `on_close` auf Zeile 65 erfolgt während des Schließens der Verbindung zum Chat-Client das Abmelden des Websocket-Handlers beim Backend-Thread durch die Methode `unsubscribe`.

```
30 class WSHandler(tornado.websocket.WebSocketHandler):
31     """
32     Websocket handler
33     @brief Handler for each client which is connectet to the server
34     """
35     def initialize(self, backendthread):
36         """
37         Init
38         @brief Setup websocket handler object
39         @param backendthread backendthread object
40         """
41         self.__NICKREGEX =  re.compile(r"^MYNICK:(.*)$")
42         self.__BACKENDTHREAD = backendthread
43         self.nick = None
44 
45     def open(self):
46         """
47         Callback for a new websocket connections
48         """
49         pass
50 
51     def on_message(self, message):
52         """
53         Callback for messages
54         @brief Subscribe a client on backendthread if message contains ↩
               MYNICK:[nickname] or directly send the message to backend
55         @param message message object
56         """
57         mo = self.__NICKREGEX.search(str(message))
58         if mo:
59             self.nick = mo.group(1)
60             self.__BACKENDTHREAD.subscribe(self)
61             return
62 
63         self.__BACKENDTHREAD.send(self.nick + " " + str(message))
64 
65     def on_close(self):
66         """
67         Callback on close
68         @brief Unsubscribe the client on backendthread
69         """
70         self.__BACKENDTHREAD.unsubscribe(self)
```

Listing 32: Implementierung der Klasse WSHandler (`Frontend.py`)

AMQP-Backend Im Backend-Thread für das Messaging-System AMQP erfolgt zunächst die Initialisierung eines Sets für die Websocket-Handler (Zeile 38 von Listing 33). Da ein Set innerhalb von Python als Repräsentation einer Menge gilt ist sichergestellt, dass niemals zwei gleiche Referenzen auf einen Websocket-Handler existieren können. Die durch das Modul *RLock* initialisierten Objekte *___INLOCK* und *___OUTLOCK*, auf Zeile 36 und 37, dienen dem synchronisieren der Zugriffe auf die später erläuterten Methoden `subscribe`, `unsubscribe` und `send`. Ferner werden alle benötigten Variablen zur Erfassung der Messwerte auf den Zeilen 39 bis 48 initialisiert. Die Initialisierung mit geschweiften Klammern erzeugt dabei eine Variable vom Typ Dictionary und stellt damit eine Hashwert-Tabelle dar. Auf den Zeilen 51 bis 59 werden diese Hash-Tabellen zur genaueren Erfassung und Einteilung des Nachrichtenflusses mit den IP-Adressen als Key aus der Liste aller Chat-Server gefüllt und vorinitialisiert. Durch die Verwendung der IP-Adressen als Routingkey im Chat-System kann so die genaue Anzahl aller verarbeiteten Nachrichten eines Servers ermittelt werden. Die Bedingung auf Zeile 56 grenzt dabei die Generierung von Einträgen auf die eigene IP-Adresse ein.

```
21 class AmqpThread(Thread):
22     """
23     AMQP-backendthread
24     """
25     def __init__(self, selfip, brokerip, serverstations):
26         """
27         Init
28         @brief Setup the Backendthread and connections to the broker
29         @param selfip Server ip
30         @param brokerip Broker ip
31         @param serverstations list with all serverips
32         """
33         Thread.__init__(self)
34         self.__IP = selfip
35         self.__CREDENTIALS = pika.PlainCredentials('chat','chat01')
36         self.__INLOCK = RLock()
37         self.__OUTLOCK = RLock()
38         self.__WSHANDLER = set()
39         self.__subscribe_count = 0
40         self.__unsubscribe_count = 0
41         self.__backend_in_msgcounter = {}
42         self.__frontend_in_msgcounter = {}
43         self.__backend_out_msgcounter = {}
44         self.__backend_out_failcounter = {}
45         self.__frontend_out_msgcounter = {}
46         self.__frontend_out_failcounter = {}
47         self.__frontend_out_keyerror = {}
48         self.__runtime = None
49 
50         # Setup counter for logging
51         for ip in serverstations:
52                 self.__backend_in_msgcounter[ip[1]] = 0
53                 self.__frontend_out_msgcounter[ip[1]] = 0
54                 self.__frontend_out_failcounter[ip[1]] = 0
55                 self.__frontend_out_keyerror[ip[1]] = 0
56                 if ip[1] == self.__IP:
57                     self.__frontend_in_msgcounter[ip[1]] = 0
58                     self.__backend_out_msgcounter[ip[1]] = 0
59                     self.__backend_out_failcounter[ip[1]] = 0
```

Listing 33: Implementierung der Klasse AmqpThread (`Backend.py`)

Anschließend erfolgt, wie in Listing 34 abgebildet ist, die Verbindung zum AMQP-Broker unter Nutzung des Moduls *pika*[45]. Dazu werden, auf den Zeilen 62 bis 77, unter Angabe aller erforderlichen Parameter zwei getrennte Verbindungen zum Broker aufgebaut. Ist der Aufbau erfolgreich, wird auf den Zeilen 79 und 83 je ein Kanal innerhalb jeder Verbindung erzeugt. Laut den Spezifikationen des AMQP und der Dokumentation des Moduls *pika* hätte die Erzeugung der Kanäle auch innerhalb einer Verbindung erfolgen können. Dem Autor der Arbeit gelang es aber nicht eine stabile Verarbeitung bei einer solchen Konfiguration zu erreichen. Auf Zeile 80 und 84 erfolgt für beide Kanäle die Erzeugung des Fanout-Exchange im Broker. Der eingehende Kanal für den Empfang der Nachrichten erzeugt, wie in den Zeilen 86 bis 88 zu sehen, zusätzlich eine temporäre Queue im Broker und bindet diese an den Exchange. Die Konfiguration wird durch den Aufruf der Methode `basic_consume`, auf Zeile 89, unter Angabe der Queue und eines Callbacks abgeschlossen.

```
61          try:
62              # Setup connection to broker for outgoing traffic
63              self.__OUTCONNECTION = pika.BlockingConnection(
64                      pika.ConnectionParameters(
65                          host=brokerip,
66                          virtual_host='/chat',
67                          credentials=self.__CREDENTIALS
68                      )
69              )
70              # Setup connection to broker for incomming traffic
71              self.__INCONNECTION = pika.BlockingConnection(
72                      pika.ConnectionParameters(
73                          host=brokerip,
74                          virtual_host='/chat',
75                          credentials=self.__CREDENTIALS
76                      )
77              )
78              # Get a channel and choose an exchangepoint for outgoing traffic
79              self.__OUTCHANNEL = self.__OUTCONNECTION.channel()
80              self.__OUTCHANNEL.exchange_declare(exchange='chat', ↩
                    exchange_type='fanout')
81
82              # Get a channel and choose an exchangepoint for incomming traffic
83              self.__INCHANNEL = self.__INCONNECTION.channel()
84              self.__INCHANNEL.exchange_declare(exchange='chat', ↩
                    exchange_type='fanout')
85              # Bind a broker side queue to exchange
86              result = self.__INCHANNEL.queue_declare(exclusive=True)
87              self.__QUEUE = result.method.queue
88              self.__INCHANNEL.queue_bind(exchange='chat', queue=self.__QUEUE)
89              self.__INCHANNEL.basic_consume(self.callback, queue=self.__QUEUE, ↩
                    no_ack=True)
90          except Exception, e:
91              raise TestErrorException("Problems while initiate connection to ↩
                    Broker: " + str(e))
```

Listing 34: Aufbau der Verbindung zum AMQP-Broker (`Backend.py`)

Wird der Thread durch den Aufruf der Methode `start` gestartet, beginnt mit dem Aufruf der Methode `start_consuming`, in Listing 35 auf Seite 54 auf Zeile 100, der Empfang der Nachrichten vom Broker. Diese Methode ist blockierend und kehrt erst nach dem Aufruf der Methode `stop_consuming` auf Zeile 117 innerhalb der Methode `stop` zurück. Erfolgt dies, so wird der Kanal geschlossen und der Thread beendet sich. Das Schließen der ausgehenden Verbindung erfolgt hingegen schon beim Aufruf der Methode `stop`. Dies soll vermeiden, dass weitere Nachrichten nach Abschluss des Testlaufs an den Broker gesendet werden. Ebenso wird die Verbindung zwischen Queue und Exchange des eingehenden Kanals getrennt, um den weiteren Empfang von Nachrichten aus dem Backend zu unterbinden.

```
93      def run(self):
94          """
95          Run-method
96          @brief Start up Backendthread an consume incomming traffic
97          """
98          print "Backend is up"
99          try:
100             self.__INCHANNEL.start_consuming()
101         except BodyTooLongError:
102             pass
103         self.__INCHANNEL.close()
104         self.__INCONNECTION.close()
105         print "Backend is down"
106
107     def stop(self):
108         """
109         Stop AMQP-backendthread
110         @brief Closes all channels and connections to broker
111         """
112         print "Shutdown Backend"
113         try:
114             self.__OUTCHANNEL.close()
115             self.__OUTCONNECTION.close()
116             self.__INCHANNEL.queue_unbind(exchange='chat', queue=self.__QUEUE)
117             self.__INCHANNEL.stop_consuming()
118         except Exception:
119             pass
```

Listing 35: Verarbeitung der Nachrichten vom Broker (`Backend.py`)

Wie bereits angedeutet, erfolgt die Bearbeitung der empfangenen Nachrichten vom Broker über einen Callback. Diese Callback-Methode ist in Listing 36 abgebildet. Wird sie aufgerufen, so wird der Zähler im Dictionary für die Nachrichten aus dem Backend an der Stelle des Routingskeys erhöht (Z. 130). Anschließend wird eine Kopie des Dictionarys mit den Websocket-Handlern erzeugt (Z. 132). Dazu wird das Objekt zuvor durch das RLock-Objekt für andere Zugriffe blockiert und nach dem Kopiervorgang wieder freigegeben (Z. 131 und 133). Das Kopieren minimiert dabei die Zeitspanne in der das Objekt blockiert ist. Innerhalb der folgenden For-Scheife wird die Nachricht an alle verfügbaren Websocket-Handler, und damit Chat-Clients, verteilt. Dabei erfolgt eine Zählung aller erfolgreich versandten Nachrichten nach dem Routingkey. Durch das Ansprechen der Websocket-Verbindung über den direkten Zugriff auf die WebsocketHandler, wird der Zugriff über den Frontend-Thread vermieden und die Erfassung aller relevanten Messwerte innerhalb des Backend-Threads ermöglicht. Schlägt das Versenden einer Nachricht fehl oder kann auf den Websocket nicht zugegriffen werden, so werden mit Hilfe des Exception-Handlings entsprechende Fehlerzähler erhöht. Dies ermöglicht in der späteren Log-Datei eine bessere Beurteilung des Testlaufs.

```
130         self.__backend_in_msgcounter[method.routing_key] += 1
131         self.__INLOCK.acquire()
132         websocket = set(self.__WSHANDLER)
133         self.__INLOCK.release()
134         try:
135             for ws in websocket:
136                 try:
137                     ws.write_message(method.routing_key + " - " + body)
138                     self.__frontend_out_msgcounter[method.routing_key] += 1
139                 except Exception:
140                     self.__frontend_out_failcounter[method.routing_key] += 1
141         except KeyError:
142             self.__frontend_out_keyerror[method.routing_key] += 1
```

Listing 36: Verarbeitung der Nachrichten vom Broker (`Backend.py`)

Die bereits genannte Registrierung der Chat-Clients am Backend-Thread durch die Methoden `subscribe` und `unsubscribe` wurde wie in Listing 37 gezeigt umgesetzt. Da der Aufruf dieser Methoden durch mehrere Websocket-Handler gleichzeitig erfolgen kann, mussten die Methoden mit Hilfe der RLock-Objekte vollständig synchronisiert werden. Beim ersten Aufruf der Methode `subscribe` erfolgt, durch die Bedingung in Zeile 151, das Setzen des Startzeitpunktes des Testlaufs. Wie auf der Zeile 150 zu sehen ist, wird innerhalb der Methode `subscribe` jeder Aufruf gezählt. In Kombination mit dem Zähler auf Zeile 163 kann so in der späteren Log-Datei die Fehlerrate der Clients ermittelt werden. Dies liegt darin begründet, dass zum Zeitpunkt der Messwerterfassung durch den TCP-Logger noch alle Chat-Clients mit dem Chat-Server verbunden sein sollten. Ist dies nicht der Fall, so wurde die Methode `unsubscribe` mindestens einmal aufgerufen und der Wert der Variable *__unsubscribe_count* damit erhöht. Auf den Zeilen 154 und 164 wird der Websocket-Handler dem Set *__WSHANDLER* hinzugefügt beziehungsweise entfernt.

```
144     def subscribe(self, wshandler):
145         """
146         Add a wshandler object to set
147         @param wshandler websocket handler object from frontend
148         """
149         self.__INLOCK.acquire()
150         self.__subscribe_count += 1
151         if not self.__runtime:
152             # Log the runtime on first client subscribe
153             self.__runtime = time.time()
154         self.__WSHANDLER.add(wshandler)
155         self.__INLOCK.release()
156 
157     def unsubscribe(self, wshandler):
158         """
159         Delete a wshandler object from the set
160         @param wshandler websocket handler object from frontend
161         """
162         self.__INLOCK.acquire()
163         self.__unsubscribe_count += 1
164         self.__WSHANDLER.discard(wshandler)
165         self.__INLOCK.release()
```

Listing 37: Registrierung und Abmeldung der Chat-Clients (`Backend.py`)

Der Aufruf der Methode `send`, wie er bereits in Listing 32 auf Seite 51 in Zeile 63 zu sehen war, ermöglicht dem Frontend-Thread das Weiterleiten der Nachrichten der Chat-Clients an den Broker. Da auch hier ein mehrfacher Aufruf der Methode durch die Websocket-Handler möglich ist, wird auch diese Methode durch ein RLock-Objekt synchronisiert. Dies ist in Listing 38 auf Seite 56 auf den Zeilen 173 und 180 zu erkennen. Neben dem Erfassen jeder versandten beziehungsweise nicht versandten Nachricht in den Zeilen 174, 177 und 179 wird in Zeile 176 die Nachricht an den Broker weitergeleitet. Dies geschieht durch den Aufruf der Methode `basic_publish` auf dem ausgehenden Kanal. Dabei werden als Parameter der Exchange und der Routingkey sowie die vom Websocket-Handler übergebenen Daten angegeben. Der Routingkey besteht dabei immer aus der IP-Adresse des sendenden Chat-Servers.

Ab Zeile 182 ist die Methode `log` zu sehen, welche wie bereits gezeigt vor der Messwert-Erfassung durch die TCP-Logger aufgerufen wird. Sie ermittelt lediglich die Gesamtlaufzeit des Tests für den Server. Mit ihrer Hilfe werden wie zuvor im Chat-Client alle benötigten Messwerte berechnet. Dies geschieht wie in Listing 30 auf Seite 49 gezeigt mit Hilfe mehrerer Methoden. Wegen der großen Anzahl dieser Methoden, sollen an dieser Stelle nur die Methoden für die Ermittlung des Nachrichtendurchsatzes aus dem Backend gezeigt werden. Wie in Listing 39 zu erkennen ist, wird mit Hilfe der insgesamt verarbeiteten Nachrichten aus dem Backend und der Testlaufzeit der Nachrichtendurchsatz pro Sekunde errechnet und als Float-Wert zurückgeben.

```
167     def send(self, data):
168         """
169         Send a message from frontend to broker
170         @brief The broker will distribute it over all servers
171         @param data data to be send
172         """
173         self.__OUTLOCK.acquire()
174         self.__frontend_in_msgcounter[self.__IP] += 1
175         try:
176             self.__OUTCHANNEL.basic_publish(exchange='chat', ↩
                routing_key=self.__IP, body=data)
177             self.__backend_out_msgcounter[self.__IP] += 1
178         except:
179             self.__backend_out_failcounter[self.__IP] += 1
180         self.__OUTLOCK.release()
181 
182     def log(self):
183         """
184         Calculate the current runtime
185         """
186         self.__runtime = time.time() - self.__runtime
```

Listing 38: Zustellung der Nachrichten an den Broker (`Backend.py`)

```
297     def get_backend_in_msgs_per_second(self):
298         """
299         Returns AMQP messages processed per second
300         @retval msgs float
301         """
302         if self.__runtime:
303             return float(self.get_backend_in_msgtotal())/float(self.__runtime)
304         else:
305             return 0.0
306 
307     def get_backend_in_msgtotal(self):
308         """
309         Return the amount of AMQP messages from broker in total
310         @retval sum int
311         """
312         sum = 0
313         for msgs in self.__backend_in_msgcounter.values():
314             sum += msgs
315         return sum
316 
317     def get_frontend_out_msgs_per_second(self):
318         """
319         Returns messages to clients processed per second
320         @retval msgs float
321         """
322         if self.__runtime:
323             return float(self.get_frontend_out_msgtotal())/float(self.__runtime)
324         else:
325             return 0.0
```

Listing 39: Erfassung empfangener Nachrichten pro Sekunde vom Broker (`Backend.py`)

ZeroMQ-Backend Da die Implementierung des Backend-Threads für das Messaging-System ZeroMQ in weiten Teilen der Implementierung für das AMQP-System gleicht, soll an dieser Stelle nur noch auf die wesentlichen Änderungen eingegangen werden. Wie bereits an den Zeilen 380 bis 387 in Listing 40 zu erkennen ist, folgt die Implementierung der ZeroMQ-Sockets stark den gewöhnlichen Unix-Sockets. Nach dem Erzeugen eines Contexts werden auf den Zeilen 381 und 385 ein Sub-Socket und ein Pub-Socket erzeugt. Auf Zeile 383 wird dem Sub-Socket durch die Socket-Option *zmq.SUBSCRIBE* und dem Parameter *b""* mitgeteilt alle Nachrichten zu empfangen. Damit erfolgt keine Filterung anhand eines Routingkeys mehr. Auf Zeile 387 wird der Pub-Socket an den Port 5550 gebunden. Damit können sich bereits alle anderen Server auf diesen Socket verbinden. Um auch Nachrichten empfangen zu können, muss der Sub-Socket noch mit den anderen Chat-Servern verbunden werden. Dies erfolgt mit dem Aufruf der Methode `connect` auf Zeile 393 innerhalb einer For-Schleife über alle IP-Adressen aus der Liste *serverstations*. Dabei wird das Ziel als String, bestehend aus Protokoll, IP-Adresse und Port übergeben. Da in der Liste auch die eigene IP-Adresse enthalten ist verbindet sich der Sub-Socket folglich auch auf den eigenen Pub-Socket. Somit wird jede Nachricht aus dem Frontend-Thread auch an diesen Socket zugestellt, aber im Unterschied zur AMQP-Implementation nicht physikalisch über das Netzwerkinterface verschickt.

```
378         try:
379             # Create ZMQ context and publish socket
380             self.__CONTEXT = zmq.Context()
381             self.__SUBSOCKET = self.__CONTEXT.socket(zmq.SUB)
382             self.__SUBSOCKET.setsockopt(zmq.LINGER,0)
383             self.__SUBSOCKET.setsockopt(zmq.SUBSCRIBE, b"")
384 
385             self.__PUBSOCKET = self.__CONTEXT.socket(zmq.PUB)
386             self.__PUBSOCKET.setsockopt(zmq.LINGER,0)
387             self.__PUBSOCKET.bind("tcp://*:5550")
388         except Exception, e:
389             raise TestErrorException("Problems while setting up ZMQ-Context or ↩
                Socket " + str(e))
390 
391         # Setup counter for logging and and connect to all other publish sockets
392         for ip in serverstations:
393                 self.__SUBSOCKET.connect("tcp://" + ip[1] + ":5550")
394                 self.__backend_in_msgcounter[ip[1]] = 0
395                 self.__frontend_out_msgcounter[ip[1]] = 0
396                 self.__frontend_out_failcounter[ip[1]] = 0
397                 self.__frontend_out_keyerror[ip[1]] = 0
398                 if ip[1] == self.__IP:
399                     self.__frontend_in_msgcounter[ip[1]] = 0
400                     self.__backend_out_msgcounter[ip[1]] = 0
401                     self.__backend_out_failcounter[ip[1]] = 0
```

Listing 40: Init-Methode des Backend-Threads für das ZeroMQ-System (`Backend.py`)

Ein weiterer Unterschied innerhalb der Implementation ist die Art der Verarbeitung der Nachrichten, welche über den Sub-Socket empfangen werden. Sie werden, wie in Listing 41 auf Seite 58 auf Zeile 414 zu sehen ist, innerhalb einer Endlosschleife mit Hilfe der Methode `recv_multipart` abgerufen. Diese ist blockierend. Die empfangenen Frames werden sofort in die Variablen *address* und *data* geschrieben. Die Variable *address* enthält dabei den Routingkey. Dieser wird, wie auch in der AMQP-Implementation, zum Zählen der verarbeiteten Nachrichten nach dem Routingkey verwendet. Das Zustellen der Nachrichten an die Chat-Clients erfolgt nach dem gleichen Prinzip wie in der vorhergehenden Implementation.

```
403     def run(self):
404         """
405         Run-method
406         @brief Start up Backendthread an consume incomming traffic.
407         Counts every message on in an out.
408         The message will be routet to each connect/subscribed websocket on the ↩
                frontend
409         """
410         print "Backend is up"
411         while True:
412             try:
413                 # Receive adress und data from subcribe socket
414                 [address, data] = self.__SUBSOCKET.recv_multipart()
415                 self.__backend_in_msgcounter[address] += 1
416                 self.__INLOCK.acquire()
417                 websocket = set(self.__WSHANDLER)
```

Listing 41: Verarbeitung der empfangenen Nachrichten des SUB-Sockets (`Backend.py`)

Das Beenden des Backend-Threads im ZeroMQ-System erfolgt, wie in Listing 42 zu sehen, durch das Schließen der beiden ZeroMQ-Sockets und dem anschließenden Terminieren des Contexts. Durch dieses Verfahren wird die Bearbeitung der Endlosschleife aus Listing 41 durch das Auslösen einer Exception der Receive-Methode beendet. Leider wurde trotz längerer Recherche keine zweckmäßige Lösung gefunden. Die einzige Möglichkeit zur sauberen Terminierung der Sockets scheint das Schicken einer entsprechenden Nachricht zu sein, welche nach dem Empfang und der Verarbeitung den Socket schließt. Da das Erfassen der Messwerte bereits vor dem Aufruf der Methode `stop` erfolgt, wurde dennoch die unsaubere Lösung gewählt, weil sie keinen Einfluss auf die Testergebnisse hat.

```
432     def stop(self):
433         """
434         Stop Zmq-backendthread
435         @brief Closes all sockets and ZMQ context
436         """
437         print "Shutdown Backend"
438         self.__PUBSOCKET.close()
439         self.__SUBSOCKET.close()
440         self.__CONTEXT.term()
```

Listing 42: Beenden des Backend-Threads im ZeroMQ-System (`Backend.py`)

Wie in Listing 43 zu sehen ist, entspricht der Aufbau der Methode `send` weitgehend der Implementation aus dem Backend-Thread des AMQP-Systems. Das Zustellen über den Pub-Socket erfolgt aber mit Hilfe einer Multipart-Nachricht, welche über die Methode `send_multipart` verschickt wird. Dabei wird im ersten Frame die IP-Adresse des Chat-Servers als Routingkey angeben. Der zweite Frame enthält die Chat-Nachricht.

```
470     def send(self, data):
471         """
472         Send a message from frontend to broker
473         @brief The broker will distribute it over all servers
474         @param data data to be send
475         """
476         self.__OUTLOCK.acquire()
477         self.__frontend_in_msgcounter[self.__IP] += 1
478         self.__PUBSOCKET.send_multipart([bytearray(self.__IP), bytearray(data)])
479         self.__backend_out_msgcounter[self.__IP] += 1
```

Listing 43: Zustellung der Nachrichten über den Pub-Socket (`Backend.py`)

5.4 Tests

5.4.1 Testplanung

Für die folgenden Testläufe wurden vor Beginn der Testphase die Parameter festgelegt, mit denen die Systeme getestet werden sollen. Der Einfluss der Komplexität auf das Lastverhalten, sollte dabei über die Veränderung der verwendeten Server im System ermittelt werden. Hierfür wurden 10 Workstations aus dem Computerpool A1.26 ausgewählt, da diese über die höchste Rechenleistung verfügen. Zusätzlich sollte das Lastverhalten der Chat-Systeme bei unterschiedlicher Nachrichtengröße und unterschiedlichem Nachrichtenaufkommen betrachtet werden. Die Anzahl der Chat-Clients soll dabei in allen Tests stabil bei 500 Clients gehalten werden. Damit bleibt die maximale Anzahl an Nachrichten, die während eines Testlaufs verschickt werden können, immer konstant. Aus der Testumgebung wurden für diese Chat-Clients fünf Workstations aus dem Computerpool A1.27 gewählt. Somit muss jede dieser Workstations 100 Chat-Clients als Threads verwalten. Diese zusätzliche Einschränkung der Testumgebung erfolgte aufgrund der starken Nutzung der Workstations in den Computerpools zum Testzeitpunkt. Der AMQP-Broker wurde in allen Tests auf einem privaten Laptop betrieben, da dieser über die nötige Leistungsfähigkeit und eine aktuelle Version des verwendeten RabbitMQ-Brokers verfügte. Eine Installation des Brokers innerhalb der Testumgebung kam auch aufgrund der geringen Hardwareressourcen und mit Blick auf die noch folgenden Tests nicht in Betracht.

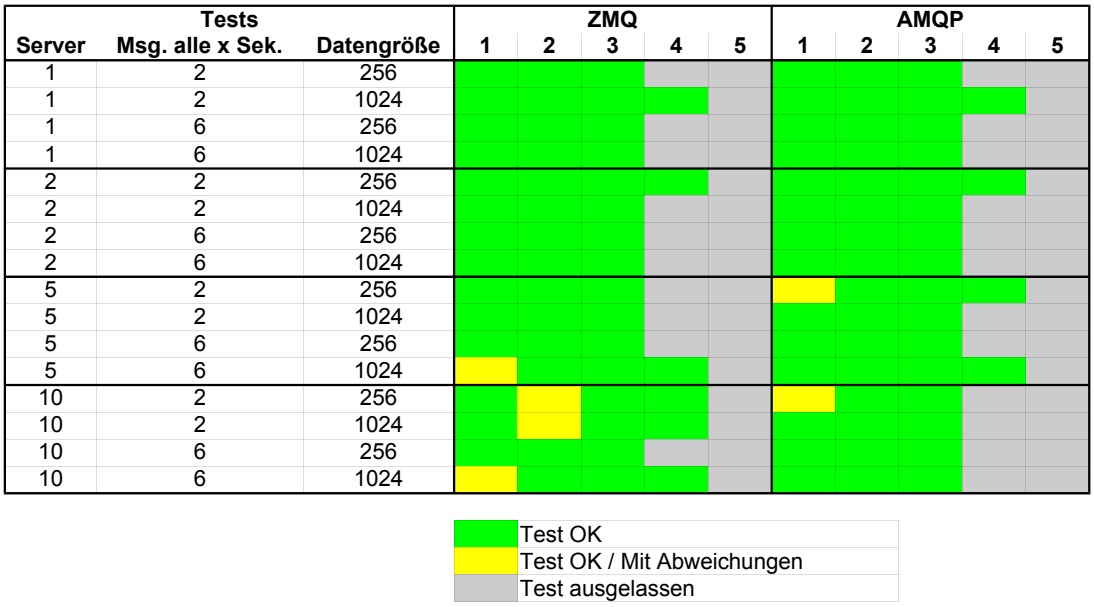

Abbildung 28: Checkliste für die Testläufe im Chat-System

Die Laufzeit der Tests wurde wegen der großen Zahl an Testläufen auf 300 Sekunden gesetzt. Die Erfassung des Speicherverbrauchs und der CPU-Auslastung auf den Workstations erfolgte alle 10 Sekunden. Zur Erfassung verlässlicher Messwerte wurde zudem festgelegt, dass die Tests je Testkonfiguration mindestens dreimal auszuführen sind. Dabei erfolgte nach jedem Testlauf eine Einschätzung der erfassten Werte auf Plausibilität. Mit den in Abbildung 28 gezeigten Testkonfigurationen ergeben sich so die mindestens 96 auszuführenden Testläufe. Die Parameter für das Nachrichtenaufkommen und Nachrichtengröße wurden bewusst auf hohe Werte gesetzt, um trotz der geringen Anzahl an Teilnehmern eine nennenswerte Last zu erzeugen.

5.4.2 Durchführung und Auswertung

Durchführung Die Bewertung der Testläufe erfolgte mit Hilfe des Bash-Skriptes `prf.sh`, welches die eintreffenden Logeinträge live analysierte. Wie in Abbildung 29 zu erkennen ist, wurden dabei alle Werte erfasst, welche für einen erfolgreichen Testlauf sprechen. Während eines Testlaufs wurde insbesondere darauf geachtet, dass alle Logeinträge der einzelnen Systemkomponenten erfasst wurden. Dies wurde mit der Zählung der Logeinträge *start* und *end* jeder Komponente erreicht. Weiterhin wurde darauf geachtet, dass im Test die Fehlerrate von einem Prozent nicht überschritten wurde. War dies der Fall, so wurde der Testlauf wiederholt.

```
Testzusammenfassung
===================

Server_start_end:       1 = 1
Client_start_end:       500 = 500
Load_start_end:         6 = 6
CNC_start_end:          1 = 1
Client errors:          1
Errorsrate in per.:     1/500*100 = .20000000000000000000

"Client 93","ws09a127","Error:1 code:1006 reason:Going away time:47.5718212128"

Srv all front. out:     8537612
Client all in:          8036220
Diff Server/Client:     501392

Srv avg. front. out/s:  28459
Cl avg. front. in/s:    53


Timestamps
================

Test start:             21:00:59 [24.Jul 2014]
Test end:               21:06:14 [24.Jul 2014]
Test runtime:           314 seconds

Logger exit:            320 seconds
Server exit:            324 seconds
Client exit:            425 seconds
```

Abbildung 29: Ausgabe des Skriptes prf.sh (Chat-System)

Auswertung Nach der Durchführung aller benötigten Testläufe wurden, mit Hilfe einiger Python-Skripte, die benötigten Daten aus den entstanden Log-Dateien ausgelesen. Dabei wurden die in Abbildung 28 gelb markierten Testläufe ausgeklammert und flossen nicht in die Ergebnisse, da es bei diesen Testläufen zu Abstürzen oder zu einer gehäuften Anzahl an Verbindungsabbrüchen durch die Chat-Clients kam.

Die Auswertung der Messergebnisse erfolgte unter anderem anhand der Messwerte für die erfolgreich verarbeiteten Nachrichten pro Sekunde, welche den Chat-Server über das verwendete Messaging-System erreichten. Durch den Testaufbau wurde davon ausgegangen, dass jeder Chat-Server über eine im Mittel gleiche Anzahl an Verbindungen von Chat-Clients verfügte. Mit der ausgewählten Hardware für die Chat-Server und Chat-Clients war zudem sichergestellt, dass für jede Komponente des Systems eine Workstation mit identischer Leistungsfähigkeit genutzt wurde. Aus diesen Gründen wurden die Messwerte über alle Testläufe und Server gemittelt. Das heißt der Messwert berechnet sich bei einem

System bestehend aus 2 Servern mit je drei Testläufen aus insgesamt 6 Messwerten. Zur weiteren Bewertung des System wurde der Speicherverbrauch der Chat-Server über den Testzeitraum ausgewertet. Dabei wurde wie zuvor der einfache Mittelwert über alle Server und Testläufe einer Konfiguration gebildet. Eine Auswertung des Speicherverbrauchs des AMQP-Brokers erfolgte aufgrund der fehlenden Messwerte im Chat-System nicht. Die Auswertung der CPU-Last konnte ebenfalls nicht erfolgen, da der kleinste erfasste Durchschnittswert des Betriebssystems auf einen Zeitraum von fünf Minuten beschränkt ist. Durch die gewählte Laufzeit der Tests von ebenfalls fünf Minuten sind die erfassten Messwerte folglich nicht sinnvoll auswertbar.

Die Auswertung für die Testläufe mit einem Nachrichtenaufkommen von einer Nachricht je sechs Sekunden und einem Datenvolumen von 256 Byte pro Nachricht zeigte, dass die Chat-Server nicht überlastet waren. In beiden Varianten des Chat-Systems ist zu erkennen, dass die Belastung der Chat-Server unabhängig von den verwendeten Messaging-Systemen und der Anzahl der Chat-Server ist, da sich der Speicherverbrauch über die Zeit identisch entwickelt. Dies kann am gleichmäßig verlaufenden Speicherverbrauch in Abbildung 30 festgestellt werden. Da auch der Speicherverbrauch bei den Testläufen mit 2 und 10 Chat-Servern identisch verläuft, kann die Abweichung bei den Testläufe mit 5 Chat-Servern nur mit einer zusätzlichen Belastung durch andere Prozesse auf den Workstations erklärt werden. Da aufgrund der fehlenden Messwerte zum Startzeitpunkt des Tests nicht auf die Belastung der Workstations vor dem Testlauf geschlossen werden kann, ist dies aber nicht mit Sicherheit festzustellen.

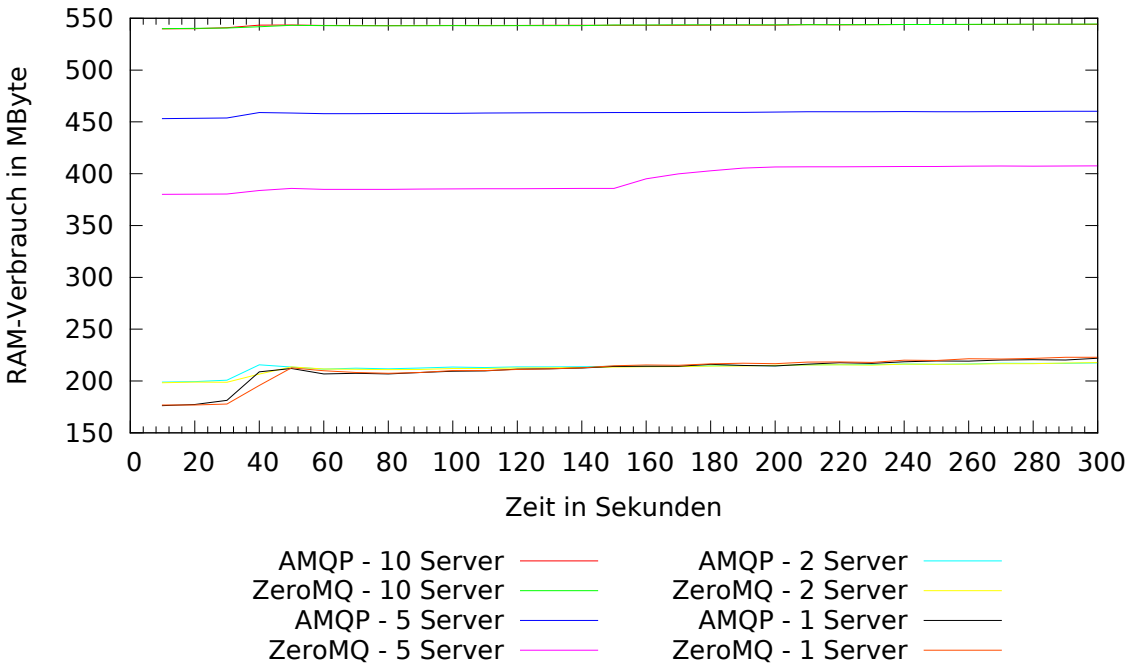

Abbildung 30: Speicherverbrauch bei 0,166 Msg/s und 256 Byte Daten

Diese Annahme wird durch die Messwerte aus Listing 44 auf Seite 62 unterstützt. Alle empfangenen Nachrichten durch die Chat-Clients wurden an das Backend weitergereicht und in gleicher Anzahl empfangen. Dieses Verhalten ist auch bei mehreren Chat-Servern zu beobachten ist. Jedoch reduziert sich die Anzahl Nachrichten mit der Aufteilung der Chat-Clients auf die Server. In Listing 45 auf Seite 62 ist dies an der um ein Fünftel kleineren Zahl der empfangenden Nachrichten für ein System mit 5 Chat-Servern zu erkennen. Die Anzahl der Nachrichten, welche über das Backend empfangen wurden, bleibt aber nahezu konstant.

```
389 "Server","ws17a126","backend_in_msg_total:17001"
390 "Server","ws17a126","frontend_in_msg_total:17001"
```

Listing 44: Testlauf mit einem Server bei 0.166 Msg/s und 256 Byte

```
645 "Server","ws16a126","backend_in_msg_total:16835"
646 "Server","ws16a126","frontend_in_msg_total:2665"
```

Listing 45: Testlauf mit fünf Servern bei 0.166 Msg/s und 256 Byte

Anhand der Messwerte der verarbeiteten Nachrichten pro Sekunde, welche über das Backend empfangen wurden wird zusätzlich bestätigt, dass keine Überlastung der Chat-Server erfolgte. Sie blieben über alle Testläufe konstant. In Abbildung 31 ist aber zu erkennen, dass die Verarbeitung der Nachrichten durch das ZeroMQ-System etwas schneller erfolgte. Im AMQP-System ist sogar eine leichte Abnahme der verarbeiteten Nachrichten zu beobachten, was wiederum auf eine performantere Verarbeitung der Nachrichten im ZeroMQ-System schließen lässt.

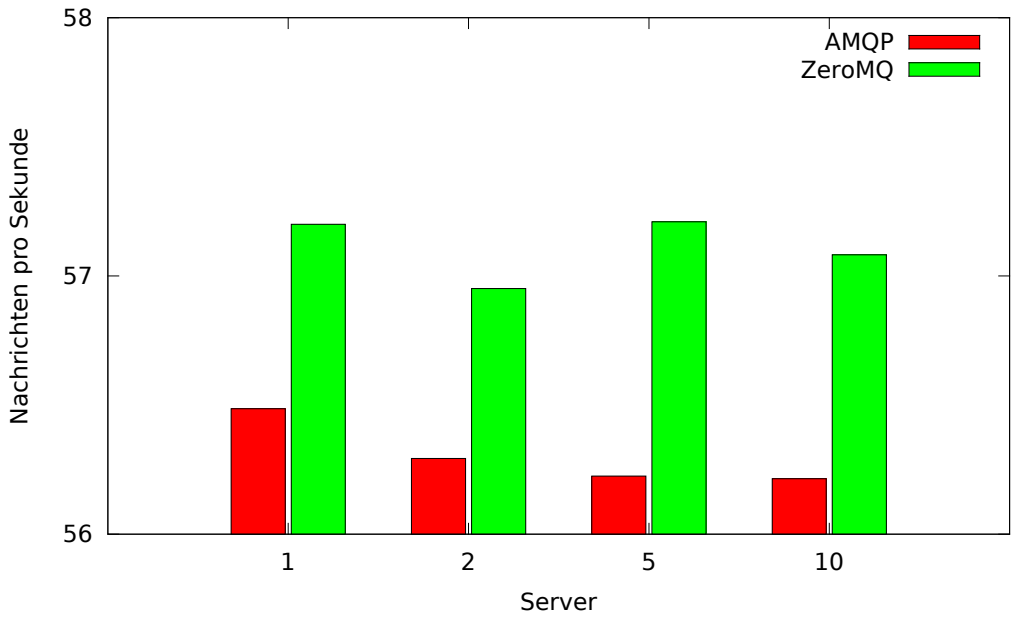

Abbildung 31: Empfangene Nachrichten bei 0,166 Msg/s und 256 Byte Daten

Eine Erhöhung des Datenvolumens brachte ähnliche Ergebnisse, wie in den gerade beschriebenen Testläufen. Wie in der Abbildung 32 auf Seite 63 zu sehen ist, verläuft auch hier die Auslastung des Speichers an den Chat-Servern für beide Messaging-Systeme gleich. Jedoch ist die Belastung der Server durch die Nachrichtengröße erheblich gestiegen. Der Anstieg des Speicherverbrauchs bei allen Testläufen ab Sekunde 240 kann allerdings nicht erklärt werden. Eine Vermutung ist, dass die Nachrichten aufgrund ihrer Größe nicht mehr schnell genug an die Chat-Clients zugestellt werden können. Dies kann aber nur mit Testläufen über einen längeren Zeitraum sicher festgestellt werden.

Die höhere Belastung der Server zeigt sich auch, wie in den Listing 46 und Listing 47 auf Seite 63 zu sehen, an der geringeren Anzahl der insgesamt verarbeiteten Nachrichten innerhalb des Testzeitraum. Dennoch wurde bei einem Vergleich der Logeinträge festgestellt, dass die Chat-Server unabhängig von ihrer Anzahl im System die Nachrichten vollständig

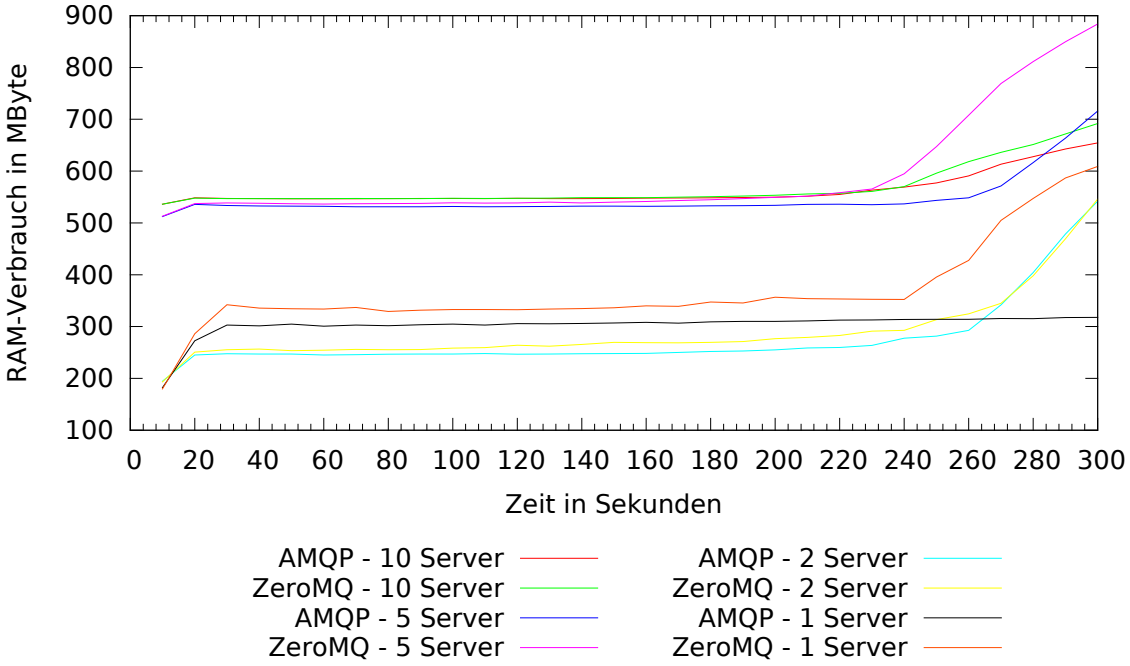

Abbildung 32: Speicherverbrauch bei 0,166 Msg/s und 1024 Byte Daten

verteilen konnten. Bei mehr als einem Chat-Server zeigte sich wieder die Aufteilung der Last durch die Verteilung der Chat-Clients auf die verfügbaren Chat-Server. Die Steigerung der maximal verarbeiteten Nachrichten im Backend kann dabei ebenfalls auf die Aufteilung der Last zurückgeführt werden. Sie begünstigt die schnellere Abarbeitung der Nachrichten im Backend durch die verringerte Last am Frontend.

```
393 "Server","ws17a126","backend_in_msg_total:14365"
394 "Server","ws17a126","frontend_in_msg_total:14365"
```

Listing 46: Testlauf mit einem Server bei 0.166 Msg/s und 1024 Byte

```
649 "Server","ws16a126","backend_in_msg_total:16162"
650 "Server","ws16a126","frontend_in_msg_total:3380"
```

Listing 47: Testlauf mit fünf Servern bei 0.166 Msg/s und 1024 Byte

Anhand der Anzahl der verarbeiteten Nachrichten pro Sekunde aus dem Backend konnte festgestellt werden, dass die Chat-Systeme mit ZeroMQ-Backend eine leicht schnellere Verarbeitung der Nachrichten ermöglichten. Diese höhere Verarbeitungsgeschwindigkeit könnte an der direkten Verbindung der Chat-Server im ZeroMQ-System liegen. Ferner zeigte sich, wie in Abbildung 33 auf Seite 64, die zunehmende Entlastung der einzelnen Chat-Server durch die steigende Anzahl der verarbeiteten Nachrichten mit der Erhöhung der verfügbaren Server. Die damit verbundene Entlastung des Frontends, bedingt durch die geringere Anzahl an Chat-Clients, begünstigte die performantere Verarbeitung der Nachrichten aus dem Backend. Es fand also eine Verschiebung der Last hin zum Backend statt.

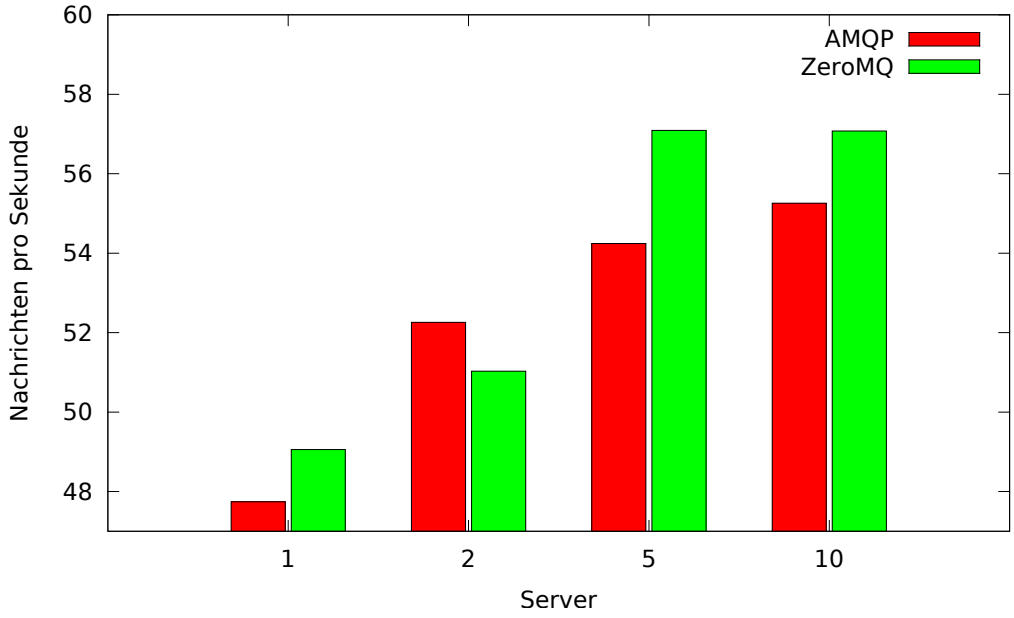

Abbildung 33: Empfangene Nachrichten bei 0,166 Msg/s und 1024 Byte Daten

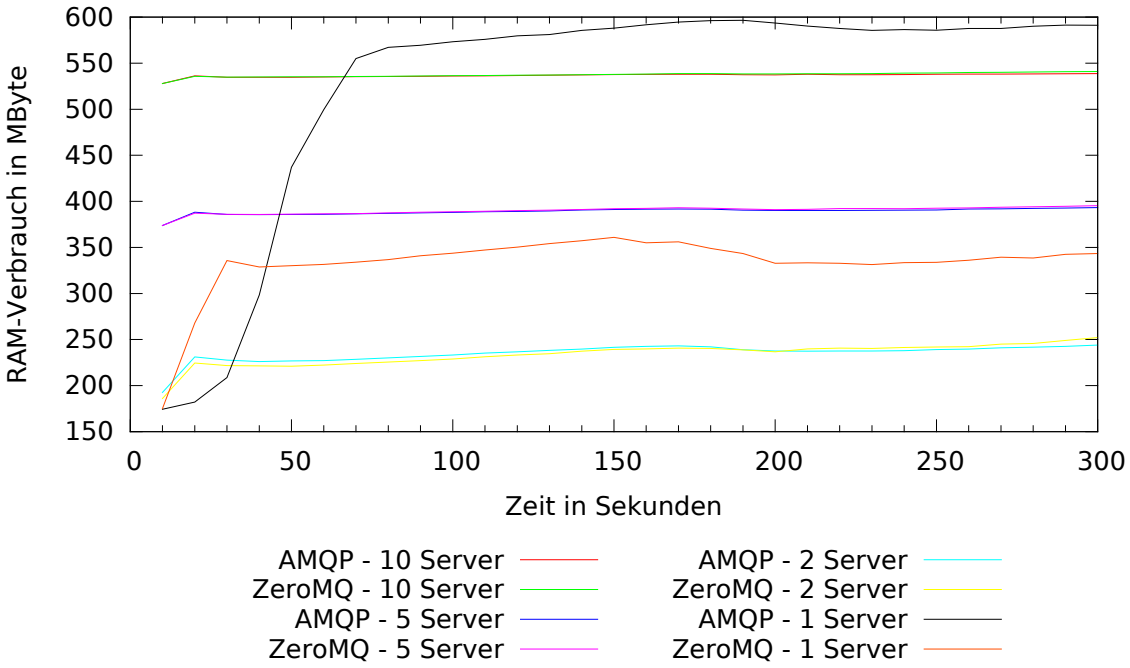

Abbildung 34: Speicherverbrauch bei 0,5 Msg/s und 256 Byte Daten

Die Testläufe mit einem verdreifachten Nachrichtenaufkommen und einem Datenvolumen von je 256 Byte pro Nachricht zeigten ein ähnliches Verhalten, wie die Testläufe mit einem geringeren Nachrichtenaufkommen und dem selben Datenvolumen. Bedingt durch die größere Anzahl an Nachrichten stieg der genutzte Arbeitsspeicher, wie in Abbildung 34 zu sehen, in den Chat-Servern aber wie erwartet stark an. Interessant ist dabei das Verhalten der Chat-Systeme bei den Testläufen mit nur einem Server. Beide Systeme weisen hier einen starken Anstieg des Speicherverbrauchs auf. Gerade das AMQP-System weicht hier stark von den Messwerten mit einer größeren Anzahl an Chat-Servern ab. Wie die Abbildungen 48 und 49 auf Seite 65 zeigen, liegt dies an einer Überlastung der Chat-Server im Websocket-Frontend. Die Nachrichten der Chat-Clients konnten innerhalb des Testzeit-

raums nicht mehr vollständig durch das Backend verarbeitet werden. Dabei kam es zu einem Nachrichtenstau und folglich zu einer höheren Speicherauslastung.

```
393 "Server","ws17a126","backend_in_msg_total:17079"
394 "Server","ws17a126","frontend_in_msg_total:17179"
```

Listing 48: Testlauf mit einem Server bei 0.5 Msg/s und 256 Byte

```
649 "Server","ws16a126","backend_in_msg_total:16875"
650 "Server","ws16a126","frontend_in_msg_total:2886"
```

Listing 49: Testlauf mit fünf Servern bei 0.5 Msg/s und 256 Byte

Wie Abbildung 35 zeigt, wurden die Server mit steigender Anzahl der verfügbaren Chat-Server wieder entlastet. Dieser Effekt tritt bereits ab zwei Chat-Servern im Chat-System auf. Es zeigt sich deutlich, dass nach der Entlastung die bereits mehrfach zu sehende Steigerung der Nachrichten im ZeroMQ-Backend eintritt. Das identische Verhalten beider Varianten des Chats-Systems mit einem Server weißt darauf hin, dass bedingt durch die Synchronisierung der Methoden im Backend-Thread eine vorrangige Verarbeitung der Nachrichten aus dem Frontend erfolgt sein muss. Durch die größere Anzahl der durch das Frontend verschickten Nachrichten und der identischen Ausführung des Frontends wird diese Annahme unterstützt.

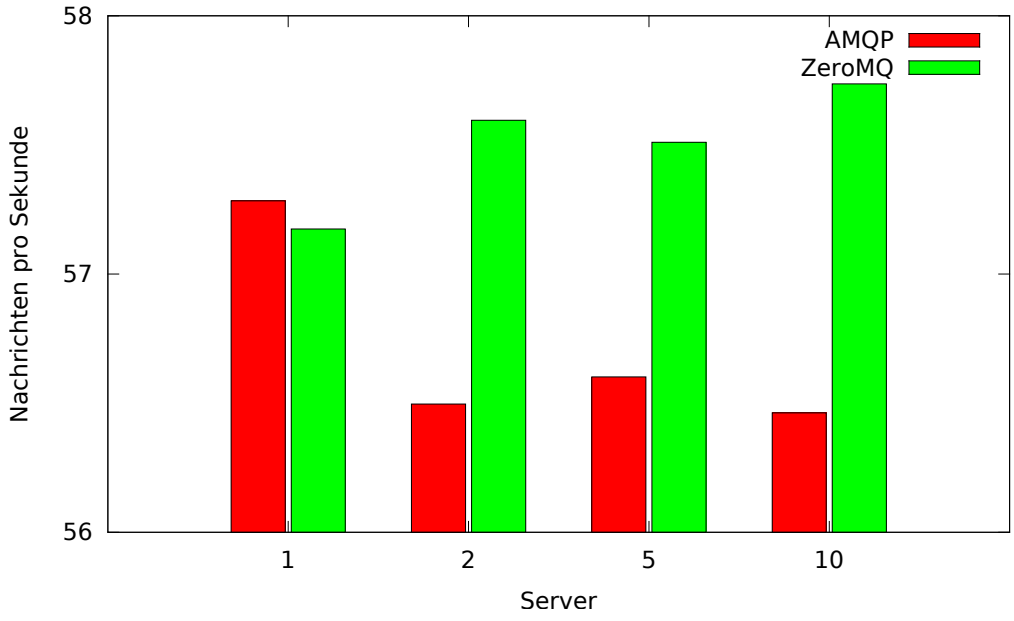

Abbildung 35: Empfangene Nachrichten bei 0,5 Msg/s und 256 Byte Daten

Da bereits die Auswertung der vorherigen Testläufe eine zeitweise Überlastung der Chat-Server zeigte, wurde in den Ergebnisse der Testläufe mit einem vervierfachten Datenvolumen bei gleichem Nachrichtenaufkommen bereits mit einer vollständigen Überlastung der Server gerechnet. Diese Vermutung bestätigte sich bei der Auswertung des Speicherverbrauchs der Chat-Server in Abbildung 36. Wie bei allen Konfigurationen zu erkennen ist, steigt der Speicherverbrauch in allen Systemen sehr stark an. Dabei zeigte sich, dass die Chat-Server mit einem ZeroMQ-Backend, unabhängig von der Anzahl der verfügbaren Server, wesentlich stärker ausgelastet wurden. Nach Ansicht des Autors liegt dies an der lokalen

Speicherung aller empfangenen Nachrichten über das Backend in der Queue des ZeroMQ-Sockets. Interessant ist, dass der Anstieg erst nach ca. 60 Sekunden erfolgt. Dieser Anstieg zeigt deutlich, dass ab diesem Zeitpunkt die Verarbeitung der Nachrichten einbricht. Das Verhalten ähnelt dabei dem Verhalten im Abbildung 32 auf Seite 63 im hinteren Teil der Testläufe. Die dort gemachte Vermutung, dass die Nachrichten aus dem Backend aufgrund ihrer Größe nicht mehr schnell genug an alle Chat-Clients im Frontend zugestellt werden können, bestätigen die aktuell betrachteten Testläufe.

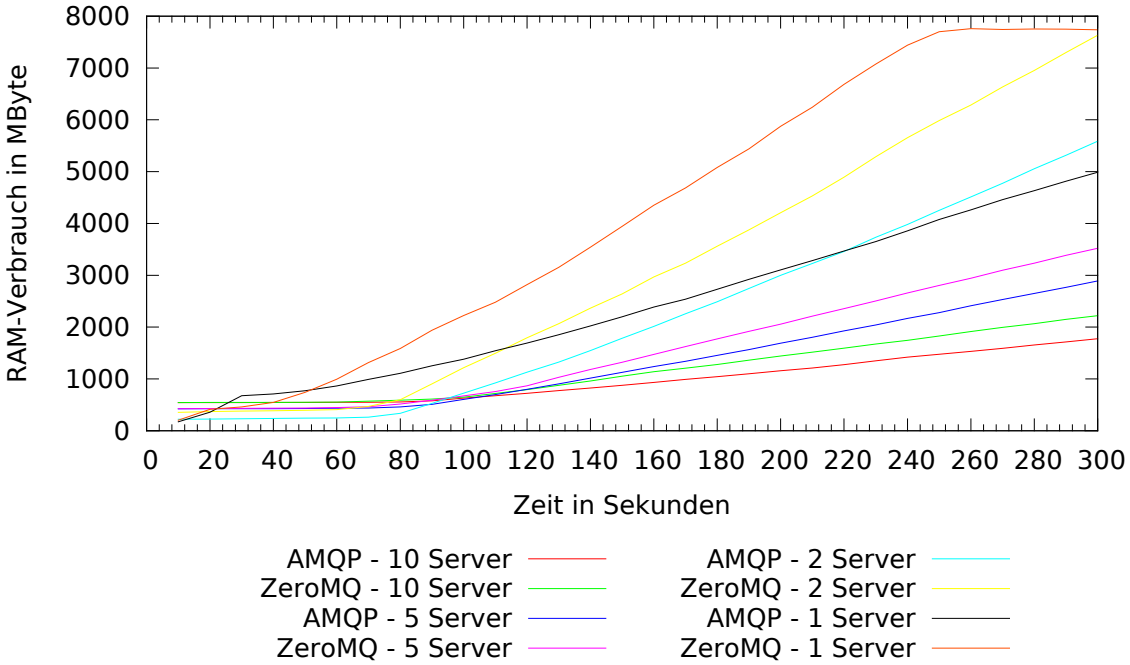

Abbildung 36: Speicherverbrauch bei 0,5 Msg/s und 1024 Byte Daten

Eine weitere interessante Feststellung betrifft den Vergleich des Speicherverbrauchs im Chat-System mit einem Chat-Server. Die wesentlich flachere Verbrauchskurve des AMQP-Systems gegenüber der des ZeroMQ-Systems deutet darauf hin, dass die Queue im Broker als Nachrichtenpuffer agiert und so die Chat-Server entlastet. Aufgrund der fehlenden Messwerte zum AMQP-Broker kann dies an dieser Stelle aber nur vermutet werden. Wie an der oberen Verbrauchskurve des ZeroMQ-Systems zu erkennen ist, ist der Arbeitsspeicher dieses System bereits nach 240 Sekunden voll ausgelastet. Eine weitere mögliche Ursache für die schnellere Auslastung ist, dass durch den einzigen Chat-Server die Verbindung zwischen dem Sub- und Pub-Socket nur innerhalb dieses Servers erfolgt. Damit kann eine schnellere Zustellung der Nachrichten erfolgen als im AMQP-System, da dieses die Nachrichten zunächst tatsächlich an den Broker versenden muss.

Die Auswertung von Listing 50 auf Seite 67 zeigt ein ähnliches Ergebnis wie im vorhergehenden Testlauf. Die Nachrichten der Chat-Clients, welche über das Backend empfangen wurden, konnten innerhalb der Zeit des Testlaufs nicht vollständig verarbeitet werden. Nach der Entlastung des Frontends durch die Verwendung von weiteren Chat-Servern zeigt sich in Listing 51 auf Seite 67, dass wieder mehr Nachrichten im Backend verarbeitet wurden. Durch die größere Anzahl an verschickten Nachrichten ist der Effekt hier aber besser zu erkennen.

```
393 "Server","ws17a126","backend_in_msg_total:22558"
394 "Server","ws17a126","frontend_in_msg_total:22629"
```

Listing 50: Testlauf mit einem Server bei 0.5 Msg/s und 1024 Byte

```
649 "Server","ws16a126","backend_in_msg_total:37718"
650 "Server","ws16a126","frontend_in_msg_total:7162"
```

Listing 51: Testlauf mit fünf Servern bei 0.5 Msg/s und 1024 Byte

In Bezug auf die verarbeiteten Nachrichten pro Sekunde im Backend, aus Abbildung 37 zeigt sich der gleiche Effekt wie in Abbildung 33 auf Seite 64. Auch hier erfolgt eine bessere Verarbeitung der Nachrichten aus dem Backend bei einer Erhöhung der Anzahl der Chat-Server. Dies begründet sich mit der Entlastung des Frontends durch eine geringe Zahl an Chat-Clients. Dabei zeigt sich auch wieder die etwas performantere Verarbeitung der Nachrichten durch das ZeroMQ-System. Die große Anzahl der verarbeiteten Nachrichten bei 2, 5 und 10 Servern gegenüber den anderen Testläufen, lässt sich auf das höhere Nachrichtenaufkommen im Backend zurückführen. Der maximale Wert kann durch konstante Überlastung der Chat-Server in allen Testläufen jedoch nie erreicht werden.

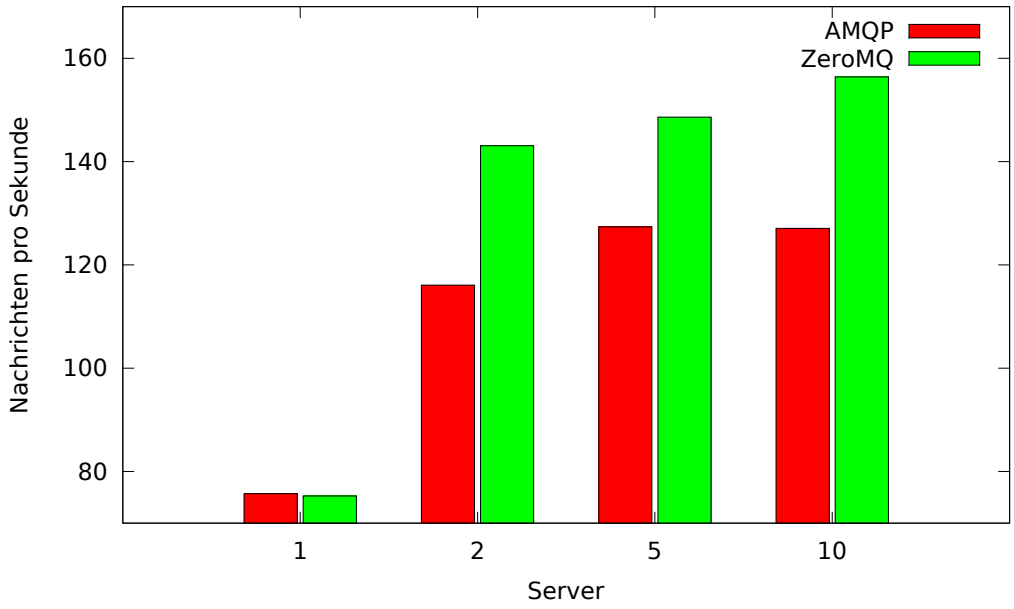

Abbildung 37: Empfangene Nachrichten bei 0,5 Msg/s und 1024 Byte Daten

5.5 Zusammenfassung

Die Auswertung der Tests der Chat-Systeme zeigte, dass die direkte Verbindung der Chat-Server im ZeroMQ-Messagingsystem zur schnelleren Verteilung der Nachrichten führte. Dabei hatte die vollständige Vermaschung mehrerer Chat-Server einen deutlich positiveren Einfluss auf die Verteilung der Nachrichten als zunächst vermutet. Eine Verringerung der Leistungsfähigkeit durch die damit verbundene höhere Komplexität konnte im Rahmen der Tests nicht festgestellt werden. Jedoch wurde bestätigt, dass bei zunehmender Last im Backend die Speicherung der noch nicht verarbeiteten Nachrichten im ZeroMQ-Socket zu Problemen führen kann. Dies wurde insbesondere bei den letzten Testläufen deutlich. Dabei zeigte sich auch, dass ein AMQP-System durch das Halten der Queues im Broker eine wesentlich bessere Entlastung der Chat-Server ermöglichte. Durch den zusätzlichen Broker sinkt aber auch der Durchsatz im Backend dieses Systems.

Die Ergebnisse lassen den Schluss zu, dass beide Messaging-Systeme für den Einsatz im Backend eines webbasierten Chat-Systems geeignet sind, da die Belastung der Chat-Server in weiten Teilen der Tests identisch war. Mit der schnelleren Übertragung und Verarbeitung der Nachrichten der ZeroMQ-Sockets ist der Einsatz eines ZeroMQ-Backends aber gerade bei kleineren Chat-Systemen mit einem geringen Datenvolumen und einem hohen Nachrichtenaufkommen durchaus sinnvoll. Dies trifft vor allem dann zu, wenn damit zu rechnen ist, dass das System keinen größeren Veränderungen unterworfen ist. Durch die schnell wachsende Komplexität dieses Messaging-Systems, bei einer größeren Anzahl an Chat-Servern, wird es aber schwer das hier umgesetzte System sinnvoll zu verwalten, da das Hinzufügen eines neuen Chat-Servers den Neustart aller Server erforderlich macht. Folglich sollte gerade in Chat-Systemen mit geringerem Nachrichtenaufkommen und einer noch unbekannten Anzahl an Chat-Servern bereits während der Planung auf AMQP gesetzt werden. Auch die bessere Entlastung der Chat-Server durch die Pufferung in den Queues des Brokers unterstützt dieses Vorgehen, wenn mit einer Zunahme der Nachrichtengröße zu rechnen ist.

6 Testaufbau Nachrichtenticker

6.1 Anforderungen

Vor Beginn der Konzeption des Nachrichtenticker-Systems wurden aus den allgemeinen Anforderungen und der Konzeption des Testsystems die Anforderungen für das zu implementierende System entwickelt. Diese Anforderungen sind nachfolgend aufgelistet.

1. Anforderungen durch das Testsystem:
   - Die Kommunikation des Servers mit den Subscribern ist unabhängig vom Messagingsystem
   - Gewährleistung des dynamischen Aufbaus der Systeme durch die Systemparameter
   - Einbeziehung der vorhanden Hardwareressourcen beim Systemaufbau
   - Separate Ausführung für jedes Messaging-System entwerfen
   - Erfassung aller geforderten Messpunkte
   - Umsetzung aller festgelegten Testparameter
   - Belastung des Systemes durch Messwerterfassung gering halten
   - Die Messagingsysteme sollen vergleichbare Messaging-Pattern nutzen
   - Die Messagingsysteme sollen vergleichbare Übertragungsparadigmen verwenden
2. Anforderungen an das Nachrichtenticker-System:
   - Die Subscriber können beliebig viele Publisher abonnieren
   - Die Auswahl der Publisher durch die Subscriber soll zufällig erfolgen

6.2 Konzeption

6.2.1 Subscriber

Um einen dynamischen Systemaufbau unter Beachtung der vorhanden Hardwareressourcen zu erreichen, sollen auch im Nachrichtenticker-System mehrere Subscriber auf einer Workstation ausgeführt werden können. Zur Umsetzung soll, so weit möglich, die Codebasis des Chat-Clients genutzt werden. Die Ermittlung der je Workstation zu startenden Subscriber und die Prüfung der Konfigurationsdatei soll deshalb in gleicher Weise wie bei den Chat-Clients erfolgen. Wie in Abbildung 38 zu sehen ist, ergibt sich damit ein ähnlicher Aufbau wie bei den Chat-Clients aus dem Chat-System. Jedoch entfällt der Thread zum versenden von Nachrichten, da der Subscriber nur zu Beginn einer Verbindung Daten an den Server senden muss.

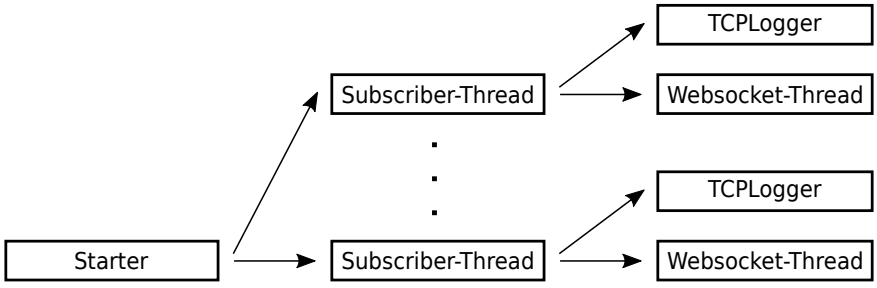

Abbildung 38: Schematischer Aufbau der Subscriber-Anwendung

Zur Kommunikation mit den Servern soll weiterhin auf das Websocket-Protokoll gesetzt werden. Trotz des Streamingcharakters des vorliegenden Systems, ermöglicht die Websocket-Verbindung eine einfache Kommunikation in beide Richtungen. Zur Übertragung von Daten an den Server muss so nicht auf andere Protokolle zurückgegriffen werden. Wie in Abbildung 39 zu erkennen ist, soll nach dem öffnen der Verbindung eine *SUB*-Nachricht durch den Server empfangen werden. Aus der Menge der darin angegebenen Publisher soll der Subscriber nun selbst eine Untermenge auswählen und diese zurück an den Server schicken. Anschließend werden, bis zum Ende des Testlaufs, alle Nachrichten des Servers entgegen genommen und gezählt.

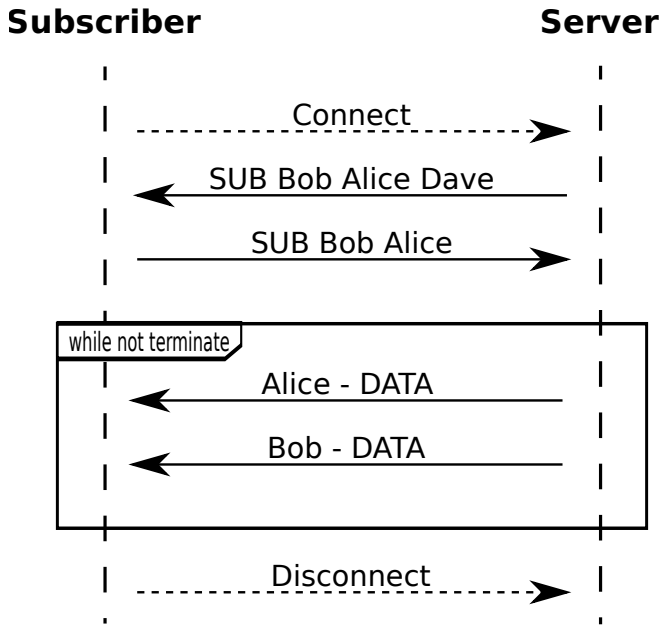

Abbildung 39: Kommunikation zwischen Subscriber und Server

6.2.2 Publisher

Wie bereits in Abbildung 40 zu erkennen ist, soll auch der Publisher im Aufbau dem Konzept des Chat-Clients folgen. Das heißt, dass unter Beachtung des Systemaufbaus und der vorhanden Hardwareressourcen ebenfalls mehrere Publisher pro Workstation gestartet werden sollen, wenn dies erforderlich ist. Dazu soll, genau wie beim Chat-Client, die Anzahl der zu startenden Publisher über die Konfigurationsdatei berechnet werden. Dies erfolgt mit Hilfe des Parameters *max_workers* und der Anzahl der zu startenden Publisher im ganzen System.

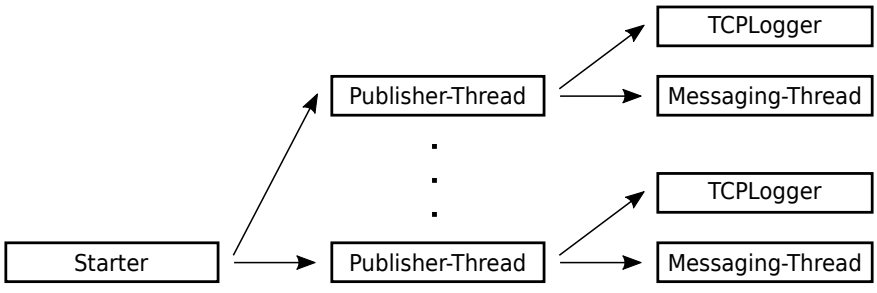

Abbildung 40: Schematischer Aufbau der Publisher-Anwendung

Da der Publisher für die Generierung der Nachrichten im System zuständig ist, müssen die Parameter *sleep_seconds* und *data_size_byte* ebenfalls aus der Konfigurationsdatei ausgelesen werden. Durch die Verortung des Publishers im Messaging-System ist es erforderlich, den als Messaging-Thread bezeichneten Block in Abbildung 40 für die Tests austauschbar zu machen. Das heißt, für jedes Messaging-System ist eine eigene Klasse mit gleichen Methoden zu implementieren. Da das Tickersystem ein Routing-Pattern abbildet und dieses sich nur marginal von der Umsetzung des Publish/Subscribe-Pattern innerhalb des Chat-Servers unterscheidet, soll in der folgenden Implementierung auf die dort verwendeten Techniken zurückgegriffen werden. Die Beschreibung an welcher Stelle des Messaging-Systems der Publisher eingebunden wird, folgt in der Konzeption zum jeweiligen Backend im Abschnitt Server.

6.2.3 Server

Auch die Umsetzung des Servers für das Nachrichtenticker-System soll in weiten Teilen der des Servers im Chat-System entsprechen. Wie in Abbildung 41 zu sehen ist, gleicht der konzeptuelle Aufbau des Servers daher auch dem des Chat-Servers. Die Kommunikation mit dem Testcontroller und das Erfassen der Messwerte soll deshalb ebenfalls wie im Chat-Server erfolgen. Um die Implementierung zu erleichtern, sollen die bereits vorhandenen Klassen für den Frontend- und Backend-Thread aus dem Chat-Server wiederverwendet werden, da zur Umsetzung des Tickersystem nur Änderungen in der Art der Nachrichtenverarbeitung vorgenommen werden müssen.

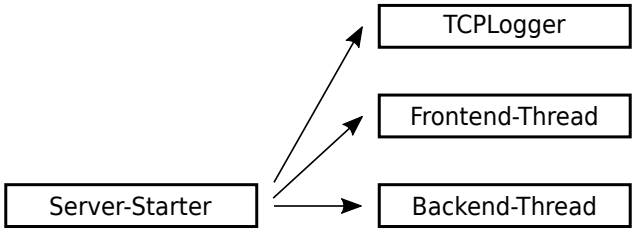

Abbildung 41: Schematischer Aufbau des Servers

AMQP-Backend Für eine vollständige Konzeption des Tickersystems mit AMQP im Backend muss zunächst dessen Aufbau festgelegt werden. Wie in Abbildung 42 auf Seite 72 zu erkennen ist, erfolgt die Verteilung der Nachrichten wieder über den zentralen RabbitMQ-Broker. Der Server agiert in diesem System jedoch nur noch als Consumer der Nachrichten. Die Generierung der Nachrichten erfolgt durch den bereits beschriebenen Publisher in der Funktion des Producers. Die Verteilung der Nachrichten im Broker soll über einen Direct-Exchange innerhalb einer Virtualhost-Umgebung erfolgen. Damit entspricht das System dem in Abbildung 3 auf Seite 5 gezeigten Aufbau. Zur besseren Vergleichbarkeit der beiden Messaging-Systeme, soll auch in der folgenden Implementierung des AMQP-Backends von der Verwendung der transaktionalen Nachrichten abgesehen werden.

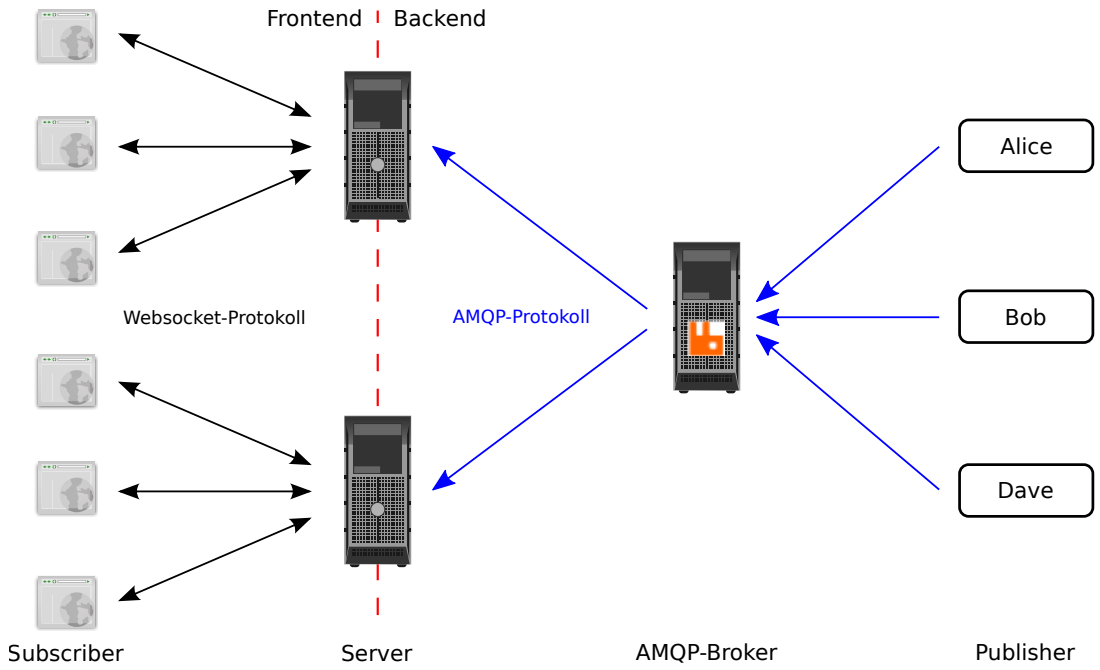

Abbildung 42: Aufbau des Nachrichtentickersystems mit AMQP

ZeroMQ-Backend Wie Abbildung 43 der Konzeption des ZeroMQ-Backends zeigt, entsteht durch die direkte Anbindung der Publisher an die Server eine voll vermaschte Netzwerkstruktur. Sie ähnelt der Struktur aus dem Chat-System. Im Unterschied zu dieser, steigt die Komplexität des System aber nicht nur mit der Anzahl der Server an, sondern auch mit jedem zusätzlichen Publisher. Da jeder Server eine Verbindung zu allen anderen Publishern aufbauen muss, müssen der Anwendung alle Publisher bekannt gegeben werden. Durch die Informationen aus der Konfigurationsdatei ist dies aber gegeben.

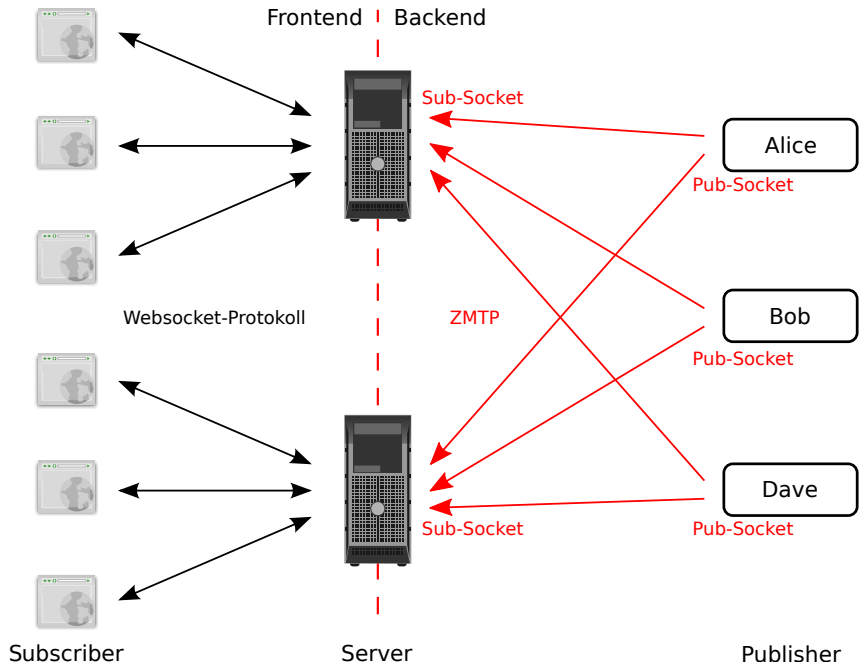

Abbildung 43: Aufbau des Nachrichtentickersystems mit ZeroMQ

6.3 Implementierung

6.3.1 Subscriber

Da sich die Implementierung des Subscribers nur im Kommunikationsablauf mit den Servern von der des Chat-Clients unterscheidet, soll auf eine Beschreibung des Subscriber-Threads und der Hauptroutine weitgehend verzichtet werden. Im nachfolgend beschriebenen Websocket-Thread soll nur auf die angepassten Programmteile eingegangen werden.

Wie in Listing 52 zu sehen ist, beschränken sich die Änderungen einzig auf die Methode `received_message`. Wird eine Nachricht vom Server empfangen, so wird mit Hilfe eines regulären Ausdrucks auf Zeile 58 geprüft, ob es sich um die Nachricht mit der Auflistung aller verfügbaren Publisher im System handelt. Der reguläre Ausdruck matcht dabei auf den Präfix *SUB*. Ist dies der Fall, so wird mit der Methode `split` der gefangene String mit allen Publishern in eine Liste umgewandelt, dabei wird am Leerzeichen getrennt. Anschließend erfolgt mit der Methode `sample` des Moduls *random* eine zufällige Auswahl von Publishern aus dieser Liste (Z. 61). Der Methode werden zu diesem Zweck sowohl die Liste selbst, als auch die minimale und maximale Anzahl der zu wählenden Publisher als Parameter übergeben. Mit den verwendeten Werten 1 und len(workers) kann die Methode also minimal einen Publisher auswählen oder im Maximum die ganze Liste zurückgeben. Die Auswahl erfolgt dabei ohne Gewichtung. Damit kann davon ausgegangen werden, dass bei vielen Subscribern im Mittel jeder Publisher ausgewählt wird. Aus dieser neu erzeugten Liste wird, auf den Zeilen 64 und 65, der String mit den Subscriptions generiert. Als Präfix wird, wie in der Konzeption definiert, ebenfalls *SUB* verwendet. Abschließend erfolgt das versenden der Subscriptions an den Server (Z. 66). In den Zeilen 62 und 67 ist außerdem das Erfassen der insgesamt abonnierten Subscriptions sowie der vollständigen Liste der Subscriptions zu sehen. Empfangende Nachrichten, welche nicht auf den regulären Ausdruck matchen, werden auf Zeile 70 gezählt.

```
51      def received_message(self, message):
52          """
53          Callback on message
54          @brief Simple count every message
55          @param message a message object
56          """
57          # Send choosen subs to server
58          mo = self._subregex.search(str(message))
59          if mo:
60              workers =  mo.group(1).split()
61              subs = rnd.sample(workers, rnd.randint(1,len(workers)))
62              self.__PARENT.add_logmessage("subcount:" + str(len(subs)))
63              self.__subs = "SUB"
64              for sub in subs:
65                  self.__subs += " " + sub
66              self.send(self.__subs)
67              self.__PARENT.add_logmessage("subs:" + str(self.__subs))
68              return
69          # count any other message that doesn't match the previous regex
70          self.__msgcounter += 1
71          return
```

Listing 52: Methode `receive_message` (`WebsocketThread.py`)

6.3.2 Publisher

Durch die Ableitung der Implementation des Publishers aus der Implementation des Chat-Clients ergab sich ein ähnlicher Aufbau des Programms. Um jedoch beide Messaging-Systeme zu unterstützen, wurde er in zwei Varianten implementiert. Dabei wurde in den Varianten lediglich der Thread für die Kommunikation mit dem Messaging-System ausgetauscht. Die Ausführung in zwei Versionen wurde gewählt, um die Übergabe von Parametern während des Starts der Tests zu vermeiden. Dieses Vorgehen entspricht im übrigen der bisherigen Umsetzung der Server.

Listing 53 zeigt einen Ausschnitt aus der Hauptroutine des Publishers. Wie zu erkennen ist, erfolgt zunächst die obligatorische Überprüfung der ausgelesenen Werte aus der Konfigurationsdatei und die anschließende Berechnung der zu startenden Publisher (Z. 29-30). Dann folgt der Verbindungsaufbau zum Testcontroller. Wie Zeile 38 zeigt, wird nach der Anmeldung am Testcontroller auf den Empfang der IP-Adresse vom Testcontroller gewartet. Dieser Vorgang entspricht dem des Chat-Servers und dient dem gleichen Zweck. Wurde die IP-Adresse empfangen werden die benötigten Publisher-Threads erzeugt. Der im Listing gezeigte Publisher-Thread für das AMQP-System wird, im Unterschied zum Publisher-Thread des ZeroMQ-Systems, zusätzlich die IP-Adresse des Brokers übergeben. Sind alle Threads erzeugt worden, wartet die Hauptroutine wie gewohnt auf weitere Kommandos des Testcontrollers.

```
29      if (((WORKERS % len(WORKERSTATIONS)) == 0) or len(WORKERSTATIONS) < ↩
            WORKERS) and WORKERS > 0:
30          WORKERS_TO_EXECUTE = WORKERS / len(WORKERSTATIONS)
31
32          try:
33              CNC = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
34              # Connect to controller
35              CNC.connect((CONTROLLER_IP, 1337))
36
37              # Say hello to controller
38              CNC.send("w hello")
39              selfip = CNC.recv(1024)
40
41              # Generate worker-threads
42              for num in range(WORKERS_TO_EXECUTE):
43                  thread = AmqpWorkerThread(
44                           str(num+1),
45                           selfip,
46                           BROKER_IP,
47                           LOGGER_IP,
48                           MSGSPEED,
49                           DATASIZE
50                  )
51                  THREADS += [thread]
```

Listing 53: Hauptroutine des AMQP-Publishers (`amqp-worker.py`)

Wie in Listing 54 auf Seite 75 innerhalb der Init-Methode des AMQP-Publishers zu erkennen ist, wird wie schon im Chat-Client ein TCP-Logger erzeugt. Diesem wird als eindeutige ID die IP-Adresse des Publishers in Kombination mit der Thread-Nummer übergeben. Durch das spätere Hinzufügen des Hostnamens im TCP-Logger kann damit jeder Publisher innerhalb der Log-Datei eindeutig identifiziert werden. Die auf Zeile 38 zu sehende Erzeugung des AMQP-Threads, welcher der eigentlichen Kommunikation mit dem Backend dient, wird innerhalb der Implementation des ZeroMQ-Publishers entsprechend ersetzt. In allen anderen Punkten der Implementation unterscheiden sich AMQP-Publisher und ZeroMQ-Publisher nicht voneinander.

```
16 class AmqpWorkerThread(Thread):
17     """
18     Controls a worker with AMQP connection
19     """
20     def __init__(self, number, selfip, brokerip, loggerip, msgspeed, datasize):
21         """
22         Init
23         @param number Worker number
24         @param selfip Worker ip
25         @param brokerip Broker IP
26         @param loggerip log server IP
27         @param msgspeed Frequenz for message output
28         @param datasize datasize to send
29         """
30         Thread.__init__(self)
31         self.__NUMBER = number
32         self.__IP = selfip
33         self.__BROKER_IP = brokerip
34         self.__LOGGER_IP = loggerip
35         self.__MSGSPEED = msgspeed
36         self.__DATASIZE = datasize
37         self.__LOGGER = TCPLogger(self.__LOGGER_IP, "Worker " + self.__IP + ↩
               "-" + self.__NUMBER)
38         self.__AMPQ = Amqp(self, self.__BROKER_IP, self.__IP, self.__NUMBER, ↩
               self.__MSGSPEED, self.__DATASIZE)
39         self.__QUEUE = Queue.Queue()
40         self.__messages = []
41         self.start()
```

Listing 54: Init-Methode des AMQP-Publishers (`WorkerThread.py`)

AMQP-Thread Wie Listing 55 als Ausschnitt der Init-Methode des AMQP-Threads zeigt, wird die Verbindung zum Broker in gleicher Weise aufgebaut, wie es schon im Chat-Server erfolgte. Lediglich der Virtualhost in Zeile 47 hat sich verändert. Da die Verteilung der Nachrichten über ein Routing-Pattern erfolgen soll, wurde auf Zeile 59 der Exchange-Typ auf *direct* geändert.

```
42         # Connect to Broker
43         try:
44             self.__CONNECTION = pika.BlockingConnection(
45                     pika.ConnectionParameters(
46                         host=self.__BROKERIP,
47                         virtual_host='/ticker',
48                         credentials=self.__CREDENTIALS
49                     )
50             )
51             self.__CHANNEL = self.__CONNECTION.channel()
52         except Exception, e:
53             self.__PARENT.add_logmessage("setuperror:" + str(e))
54             print str(e)
55             raise TestErrorException("Problems while connect to Broker: " + ↩
                   str(e))
56 
57         self.__msgcounter = 0
58         self.Stop = False
59         self.__CHANNEL.exchange_declare(exchange='ticks', ↩
               exchange_type='direct')
```

Listing 55: Init-Methode im AMQP-Thread (`MessageBackends.py`)

Erfolgt, wie in Listing 56 auf Zeile 66 zu sehen, der Start des Threads über die Run-Methode, so wird zunächst der Startzeitpunkt des Testlaufs gespeichert. Anschließend werden, in einer While-Schleife, die Nachrichten nach den Vorgaben der Konfigurationsdatei erzeugt (Z. 69-75). Das Versenden geschieht über die Methode `basic_publish`. Ihr wird neben den Daten und dem Exchange auch der Routingkey als String übergeben. Dieser setzt sich aus der IP-Adresse und der Nummer des Publishers zusammen.

```
61      def run(self):
62          """
63          Run-method
64          @brief Simply send messages to broker with the given speed and datasize
65          """
66          self.__runtime = time.time()
67          self.__FAKE.seed(self.__runtime)
68          while (not self.Stop):
69              data = str(self.__FAKE.text(self.__DATASIZE))
70              try:
71                  self.__CHANNEL.basic_publish(exchange='ticks', ↩
                        routing_key=self.__KEY, body=data)
72                  self.__msgcounter += 1
73                  time.sleep(self.__MSGSPEED)
74              except Exception as e:
75                  print "Problems while sending data to Broker", e
76          self.__CONNECTION.close()
```

Listing 56: Run-Methode im AMQP-Thread (`MessageBackends.py`)

ZeroMQ-Thread Innerhalb der Init-Methode des ZeroMQ-Threads, in Listing 57, wird zunächst ein neuer ZeroMQ-Context auf Zeile 114 erzeugt. Anschließend erfolgt das Erzeugen und Binden des Pub-Sockets auf den Zeilen 116 und 117. Die Portnummer wird dabei dynamisch über die Nummer des Publishers berechnet. Ist dies erfolgreich, können Nachrichten über den Socket versandt werden, sobald ein Server mit ihm verbunden ist.

```
113         # Open a publish socket
114         try:
115             self.__CONTEXT = zmq.Context()
116             self.__ZSOCKET = self.__CONTEXT.socket(zmq.PUB)
117             self.__ZSOCKET.bind("tcp://*:" + str(5550+int(self.__NUMBER)))
118         except Exception, e:
119             self.__PARENT.add_logmessage("setuperror:" + str(e))
120             print str(e)
121             raise TestErrorException("Problems while setting up ZMQ-Context or ↩
                    Socket " + str(e))
```

Listing 57: Init-Methode im ZeroMQ-Thread (`MessageBackends.py`)

Wie Listing 58 der Run-Methode des Threads auf Seite 77 zeigt, erfolgte der Aufbau in ähnlicher Weise wie beim AMQP-Threads. Innerhalb der While-Schleife wird zunächst die Nachricht generiert (Z. 131). Sie besteht dabei aus einer Liste von Bytearrays, welche die Frames der Multipart-Nachricht darstellt. Der erste Eintrag entspricht dabei dem Routingkey. Dieser weist den selben Aufbau auf, wie der Routingkeys aus dem AMQP-Thread. Anschließend wird diese Liste an die Methode `send_multipart` übergeben und damit dem Socket zum verschicken übergeben.

```
123     def run(self):
124         """
125         Run-method
126         @brief Simply send messages to broker with the given speed and datasize
127         """
128         self.__runtime = time.time()
129         self.__FAKE.seed(self.__runtime)
130         while (not self.Stop):
131             data = [bytearray(self.__KEY), ↩
                        bytearray(str(self.__FAKE.text(self.__DATASIZE)))]
132             try:
133                 self.__ZSOCKET.send_multipart(data)
134                 self.__msgcounter += 1
135                 time.sleep(self.__MSGSPEED)
136             except Exception as e:
137                 print "Problems while sending data", e
138         self.__ZSOCKET.close()
139         self.__CONTEXT.term()
```

Listing 58: Run-Methode im ZeroMQ-Thread (`MessageBackends.py`)

6.3.3 Server

Da die Implementierung des Servers in weiten Teilen der Implementation des Chat-Servers gleicht, sollen auch in diesem Abschnitt nur die wesentlichen Änderungen erläutert werden. Wie in der Hauptroutine in Listing 59 zu erkennen ist, erfolgt nach dem Einlesen der Konfigurationsdatei zunächst die Überprüfung der ermittelten Werte auf Gültigkeit. Im Gegensatz zum Chat-Server werden aber auch die Werte für die Publisher geprüft (Z. 35-41). Damit wird sichergestellt, dass der Start der Threads für das Frontend und Backend nur erfolgt, wenn die Prüfung erfolgreich war. Für den Server mit ZeroMQ-Backend werden auf Zeile 37 nur die Anzahl der Workstations sowie die Liste der Publisher im System übergeben. Im Falle des AMQP-Backends wird zusätzlich noch die IP-Adresse des Brokers übergeben.

```
30  WORKERSTATIONS = config.items('Worker')
31  LOGGER = TCPLogger(LOGGER_IP, 'Server')
32  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
33  try:
34      # Check the worker configuration
35      if (((WORKERS % len(WORKERSTATIONS)) == 0) or len(WORKERSTATIONS) < ↩
            WORKERS) and WORKERS > 0:
36          WORKERCOUNT = WORKERS / len(WORKERSTATIONS)
37          BACKEND = ZmqThread(WORKERCOUNT, WORKERSTATIONS)
38          BACKEND.start()
39
40      # Check the client configuration
41      if (((CLIENTS % len(CLIENTSTATIONS)) == 0) or len(CLIENTSTATIONS) < ↩
            CLIENTS) and CLIENTS > 0:
42
43          # Fire up Tornado server
44          FRONTEND = WebsocketThread(BACKEND, WORKERCOUNT, WORKERSTATIONS)
45          FRONTEND.start()
46
47          # Try to set up the connection to controller
48          s.connect((CONTROLLER_IP, 1337))
49
50          #Say hello to controller
51          s.send('s hello')
52          while True:
```

Listing 59: Hauptroutine des ZeroMQ-Servers (`zmq-server.py`)

Nach dem Starten von Frontend- und Backend-Thread erfolgt der Aufbau einer Verbindung zum Testcontroller, gefolgt von der Anmeldung über das vordefinierte Kommando. Anschließend wird auf weitere Kommandos des Testcontrollers gewartet und in der bereits beschriebenen Weise darauf reagiert.

Websocket-Frontend Die Änderungen im Frontend-Thread betreffen hauptsächlich die Methoden für die Kommunikation mit den Subscribern. Damit wurden die meisten Veränderungen in der Klasse *WSHandler* vorgenommen, da diese für den Nachrichtenaustausch mit den Subscribern verantwortlich ist. In der Klasse *WebsocketThread*, welche dem Start der Tornado-Applikation dient wurden über das Dictionary zusätzlich die Parameter für die Anzahl der zu startenden Publisher sowie der Liste aller Publisher eingepflegt. Da im Nachrichtenticker, zur Vereinfachung der Testumgebung, keine direkte Anmeldung der Publisher am Server vorgesehen wurde, benötigt das Frontend diese Informationen um den Subscribern die verfügbaren Publisher anbieten zu können. Diese zusätzlichen Einträge im Dictionary sind auf den Zeilen 101 und 102 in Listing 60 zu sehen.

```
97              (r'/ws',
98               WSHandler,
99               dict(
100                  backendthread=self.__BACKENDTHREAD,
101                  workercount=self.__WORKERCOUNT,
102                  workerstations=self.__WORKERSTATIONS,
103              )
104             ),
105             (r"/", MainHandler),
106             (r"/(.*)", tornado.web.StaticFileHandler, {"path": "./resources"}),
107         ])
108         self.__APPLICATION.listen(9090)
```

Listing 60: Initialisierung der Tornado-Applikation (`Frontend.py`)

Wie bereits in der Methode `initialize` in Listing 61 zu erkennen ist, wird neben dem Kompilieren des Patterns für den regulären Ausdruck auch ein Set für die Subscriptions erzeugt (Z. 44 und 48). Damit enthält jeder Websocket-Handler zu jedem Zeitpunkt alle Subscriptions des mit ihm assoziierten Subscribers.

```
36      def initialize(self, backendthread, workercount, workerstations):
37          """
38          Init
39          @brief Setup websocket handler object
40          @param backendthread backendthread object
41          @param workercount Number of workers each Workstation offer
42          @param workerstations List with all other serverips
43          """
44          self.__SUBREGEX =  re.compile(r"^SUB (.*)$")
45          self.__BACKENDTHREAD = backendthread
46          self.__WORKERCOUNT = workercount
47          self.__WORKERSTATIONS = workerstations
48          self.subscriptions = set()
```

Listing 61: Initialisierung des Websocket-Handlers (`Frontend.py`)

Nach der Initialisierung der Websocket-Handlers wird die, in Listing 62 auf Seite 79 gezeigte, Methode `open` gerufen. Dort wird, auf den Zeilen 56 bis 59, zunächst der String mit den verfügbaren Publishern aus den in der Methode `initialize` übergebenen Werten generiert. Anschließend wird dieser String an den Subscriber versandt.

```
49
50      def open(self):
51          """
52          Callback for a new websocket connections
53          @brief Primaly send all availabe Publishers to all clients
54          """
55          # Sende subscriptions
56          sub = "SUB"
57          for station in self.__WORKERSTATIONS:
58              for i in range(self.__WORKERCOUNT):
59                  sub += " " + station[1] + "-" + str(i+1)
60          self.write_message(sub)
```

Listing 62: Initialisierung des Websocket-Handlers (`Frontend.py`)

Die darauf folgende Antwort des Subscribers wird anschließend in der Methode `on_message` ausgewertet. Dies geschieht auf der Zeile 69 in Listing 63. Die Kompilierung des regulären Ausdrucks in der Methode `initialize` bietet hier den Vorteil, dass die Anwendung des Pattern auf jede empfangene Nachricht wesentlich schneller erfolgen kann. Innerhalb der folgenden For-Schleife werden die angeforderten Subscriptions aus der Nachricht ausgelesen und in das vordefinierte Set eingefügt. Die Verwendung eines Sets garantiert dabei, dass ein doppeltes Abonnement eines Publishers unterbunden wird. Sind alle Subscriptions dem Set hinzugefügt worden, wird die Methode `subscribe` des Backend-Threads mit dem Websocket-Handler als Parameter aufgerufen.

Die auf den Zeilen 76 bis 81 definierte Methode `on_close` zeigt, dass bei einer Terminierung der Verbindung der Subscriber über die Methode `unsubscribe` des Backend-Threads abgemeldet wird.

```
62      def on_message(self, message):
63          """
64          Callback for messages
65          @brief Subscribe a client on backendthread if message contains ↩
                SUB:[subscriptionlist]
66          @param message message object
67          """
68          # Get subscriptions from the client and subscribe him
69          mo = self.__SUBREGEX.search(str(message))
70          if mo:
71              for sub in set(mo.group(1).split()):
72                  self.subscriptions.add(sub)
73              self.__BACKENDTHREAD.subscribe(self)
74              return
75
76      def on_close(self):
77          """
78          Callback on close
79          @brief Unsubscribe the client on backendthread
80          """
81          self.__BACKENDTHREAD.unsubscribe(self)
```

Listing 63: Nachrichtenverarbeitung im Websocket-Handler (`Frontend.py`)

AMQP-Backend Wie die Init-Methodes des AMQP-Threads in Listing 64 zeigt, wurde der Aufbau an den Chat-Server angelehnt. Da der Nachrichtenfluss nur aus dem Backend an die Subscriber erfolgt, wird nur ein RLock-Objekt zur Synchronisierung der Methoden `subscribe` und `unsubscribe` benötigt (Z. 34). Eine Änderung betrifft die Variable *___WSHANDLER_DICT* auf Zeile 35. Anders als im Chat-Server wurde hier für die spätere Speicherung der Websocket-Handler ein Dictionary benutzt. Eine größere Änderung erfolgte bei der Vorbelegung der Dictionarys zur Erfassung der Messwerte. Als Keys für die Dictionary-Einträge wurden die Routingkeys gewählt, um eine getrennte Aufschlüsselung der verarbeiteten Nachrichten zu erhalten. Um diese zu generieren werden, auf den Zeilen 45 bis 50, alle Kombinationen von Routingkeys über eine For-Schleife erzeugt. Dieser Vorgang vermeidet die spätere Generierung innerhalb der Methode zur Nachrichtenverarbeitung. Dies sorgt schlussendlich für eine schnellere Verarbeitung der Nachrichten und reduziert damit den Einfluss der Messwerterfassung auf das System.

```
24      def __init__(self, brokerip, workercount, workerstations):
25          """
26          Init
27          @brief Setup the Backendthread and connections to the broker
28          @param brokerip Broker ip
29          @param workercount Number of workers each Workstation offer
30          @param workerstations List with all other serverips
31          """
32          Thread.__init__(self)
33          self.__CREDENTIALS = pika.PlainCredentials('ticker','ticker01')
34          self.__SUBLOCK = RLock()
35          self.__WSHANDLER_DICT = {}
36          self.__subscribe_count = 0
37          self.__unsubscribe_count = 0
38          self.__backend_msgcounter = {}
39          self.__frontend_msgcounter = {}
40          self.__frontend_failcounter = {}
41          self.__frontend_keyerror = {}
42          self.__runtime = None
43
44          # Setup counter for logging
45          for ip in workerstations:
46              for num in range(workercount):
47                  self.__backend_msgcounter[ip[1] + "-" + str(num+1)] = 0
48                  self.__frontend_msgcounter[ip[1] + "-" + str(num+1)] = 0
49                  self.__frontend_failcounter[ip[1] + "-" + str(num+1)] = 0
50                  self.__frontend_keyerror[ip[1] + "-" + str(num+1)] = 0
```

Listing 64: Init-Methode des AMQP-Backend-Threads (`Backend.py`)

Der anschließende Verbindungsaufbau zum Broker erfolgt in der gewohnten Weise. Jedoch wurde, wie in Listing 65 auf Seite 81 in Zeile 62 zu sehen, beim Aufruf der Methode `exchange_declare` der Typ des Exchange auf *direct* geändert. Anders als im Chat-Server, erfolgt nach dem Erzeugen der Queue in Zeile 65 aber kein Binding zwischen Exchange und eben jener Queue. Viel mehr wird direkt die Methode `basic_consume` unter Angabe der Queue aufgerufen. Dies hat zur Folge, dass der Server noch keine Nachrichten über den Broker empfangen kann.

```
51          try:
52              # Setup connection to broker for incomming traffic
53              self.__CONNECTION = pika.BlockingConnection(
54                      pika.ConnectionParameters(
55                          host=brokerip,
56                          virtual_host='/ticker',
57                          credentials=self.__CREDENTIALS
58                      )
59              )
60              # Get a channel and choose an exchangepoint for incomming traffic
61              self.__CHANNEL = self.__CONNECTION.channel()
62              self.__CHANNEL.exchange_declare(exchange='ticks', ↩
                    exchange_type='direct')
63              # Bind a broker side queue to exchange
64              result = self.__CHANNEL.queue_declare(exclusive=True)
65              self.__QUEUE = result.method.queue
66              self.__CHANNEL.basic_consume(self.callback, queue=self.__QUEUE, ↩
                    no_ack=True)
67          except Exception, e:
68              raise TestErrorException("Problems while initiate connection to ↩
                    Broker: " + str(e))
```

Listing 65: Verbindungsaufbau des AMQP-Backend-Threads (`Backend.py`)

Innerhalb der Methode `subscribe`, in Listing 66, werden alle Subscriber und ihre Subscriptions registriert. Da die Methode durch mehrere Websocket-Handler des Frontend-Threads gleichzeitig aufgerufen werden kann, wurde sie synchronisiert. Wie Zeile 109 zeigt, wird jeder Aufruf der Methode gezählt. Dieser Zähler dient später dazu Fehler im Testlauf zu erkennen. Wird die Methode zum ersten Mal aufgerufen so wird dies, wie beim Chat-Server, als Beginn des Testlaufs gewertet und ein Zeitstempel erzeugt. Anschließend werden die Websocket-Handler in ein Set innerhalb des Dictionary eingetragen. Dabei dienen die Subscriptions als Key im Dictionary und ermöglichen später den schnellen Zugriff auf alle Websocket-Handler anhand des Routingkeys. In den Zeilen 116 und 117 ist zu sehen, dass ein neues Set erzeugt wird, wenn zur angeforderten Subscription noch kein Key im Dictionary vorliegt. Danach erfolgt, mit der Methode `queue_bind`, das Binding der Queue mit dem Exchange unter der Angabe des gerade behandelten Abonnements als Routingkey. Ab diesem Zeitpunkt werden Nachrichten mit diesem Routingkey durch den Broker an die Queue ausgeliefert.

```
103     def subscribe(self, wshandler):
104         """
105         Add a wshandler object to set
106         @param wshandler websocket handler object from frontend
107         """
108         self.__SUBLOCK.acquire()
109         self.__subscribe_count += 1
110         if not self.__runtime:
111             self.__runtime = time.time()
112         for sub in wshandler.subscriptions:
113             if sub in self.__WSHANDLER_DICT:
114                 self.__WSHANDLER_DICT[sub].add(wshandler)
115             else:
116                 self.__WSHANDLER_DICT[sub] = set([wshandler,])
117                 self.__CHANNEL.queue_bind(exchange='ticks', ↩
                        queue=self.__QUEUE, routing_key=sub)
118                 #print "--> sub " + str(sub)
119             #print "sub " + str(sub)
120         self.__SUBLOCK.release()
```

Listing 66: Methode `subscribe` des AMQP-Backend-Threads (`Backend.py`)

Die in Listing 67 gezeigte Methode `unsubscribe` besitzt einen ähnlichen Aufbau, löscht aber die Websocket-Handler wieder aus den Sets im Dictionary. Auch sie muss synchronisiert werden. Um alle Einträge des Websocket-Handlers im Dictionary zu löschen, durchläuft die Methode alle Subscriptions dieses Handlers (Z. 129-134). Dabei wird auf Zeile 132 die Länge des Sets geprüft. Befinden sich im Set keine Einträge mehr so wird es aus dem Dictionary gelöscht. Da kein Subscriber mehr an den Nachrichten dieses Publishers interessiert ist, wird über die Methode `queue_unbind` unter Angabe Routingkeys das Abonnement gekündigt.

```
122     def unsubscribe(self, wshandler):
123         """
124         Delete a wshandler object from the set
125         @param wshandler websocket handler object from frontend
126         """
127         self.__SUBLOCK.acquire()
128         self.__unsubscribe_count += 1
129         for sub in wshandler.subscriptions:
130             self.__WSHANDLER_DICT[sub].discard(wshandler)
131             #print "unsub " + str(sub)
132             if len(self.__WSHANDLER_DICT[sub]) == 0:
133                 del self.__WSHANDLER_DICT[sub]
134                 self.__CHANNEL.queue_unbind(exchange='ticks', ↩
                        queue=self.__QUEUE, routing_key=sub)
135                 #print "--> unsub " + str(sub)
136         self.__SUBLOCK.release()
```

Listing 67: Methode `unsubscribe` des AMQP-Backend-Threads (`Backend.py`)

Wie in Listing 68 zu sehen ist, erfolgt die Verarbeitung der Nachrichten wieder über die Methode `callback`. Die Nachrichtenzähler in den Zeilen 87, 95, 97 und 99 zeigen, dass die Erfassung dieser Messwerte anhand des Routingkeys erfolgt. Da auch in dieser Methode auf das Dictionary mit den Websocket-Handlern zugegriffen wird, muss auch hier eine teilweise Synchronisierung erfolgen. Um die damit eintretende Blockierung des Dictionarys möglichst kurz halten, wird auf Zeile 90 eine Kopie des benötigten Sets erzeugt. Dieses wird mit Hilfe des Routingkeys ausgewählt. Anschließend erfolgt das Versenden der Nachrichten an alle Subscriber über die Websocket-Handler. Dabei werden mögliche Fehler beim Versenden oder Fehlzugriffe auf das Dictionary für eine spätere Analyse erfasst.

```
80      def callback(self, ch, method, properties, body):
81          """ Callback for incomming messages on INCHANNEL
82          @brief The message will be routet to each subscribed websocket
83          @param ch Channel
84          @param method Flags in current message
85          @param properties Some message headers
86          @param body Messagebody """
87          self.__backend_msgcounter[method.routing_key] += 1
88          try:
89              self.__SUBLOCK.acquire()
90              websockets = set(self.__WSHANDLER_DICT[method.routing_key])
91              self.__SUBLOCK.release()
92              for ws in websockets:
93                  try:
94                      ws.write_message(method.routing_key + " - " + body)
95                      self.__frontend_msgcounter[method.routing_key] += 1
96                  except Exception:
97                      self.__frontend_failcounter[method.routing_key] += 1
98          except KeyError:
99              self.__frontend_keyerror[method.routing_key] += 1
```

Listing 68: Nachrichtenverarbeitung im AMQP-Backend-Thread (`Backend.py`)

ZeroMQ-Backend Die Implementierung der Init-Methode des ZeroMQ-Threads, wie in Listing 69 zu sehen, unterscheidet sich im Umfang wesentlich von der des AMQP-Threads. Zu Beginn der Methode werden zunächst der ZeroMQ-Context und der Sub-Socket auf Zeile 284 und 285 erzeugt. Die Verbindung mit den Publishern erfolgt innerhalb der For-Schleifen ab Zeile 298. Dabei werden die Verbindungsparameter der Methode `connect` dynamisch aus der Liste der Workstations und der Anzahl der darauf laufenden Publisher generiert (Z. 300). Dabei wird der Port auf die gleiche Weise, wie im Publisher berechnet. Nach Abschluss der Init-Methode sind alle Publisher mit dem Server verbunden.

```
283         Thread.__init__(self)
284         self.__CONTEXT = zmq.Context()
285         self.__ZSOCKET = self.__CONTEXT.socket(zmq.SUB)
286         self.__ZSOCKET.setsockopt(zmq.LINGER,0)
287         self.__SUBLOCK = RLock()
288         self.__WSHANDLER_DICT = {}
289         self.__subscribe_count = 0
290         self.__unsubscribe_count = 0
291         self.__backend_msgcounter = {}
292         self.__frontend_msgcounter = {}
293         self.__frontend_failcounter = {}
294         self.__frontend_keyerror = {}
295         self.__runtime = None
296 
297         # Setup counter for logging and and connect to all other publish sockets
298         for ip in workerstations:
299             for num in range(workercount):
300                 self.__ZSOCKET.connect("tcp://" + ip[1] + ":" + str(5550 + ↩
                        int(num+1)))
301                 self.__backend_msgcounter[ip[1] + "-" + str(num+1)] = 0
302                 self.__frontend_msgcounter[ip[1] + "-" + str(num+1)] = 0
303                 self.__frontend_failcounter[ip[1] + "-" + str(num+1)] = 0
304                 self.__frontend_keyerror[ip[1] + "-" + str(num+1)] = 0
```

Listing 69: Init-Methode des ZeroMQ-Backend-Threads (`Backend.py`)

Wie auch beim AMQP-Thread werden innerhalb der Methode `subscribe`, in Listing 70, alle Subscriber und ihre Subscriptions registriert. Der einzige Unterschied in der Abarbeitung besteht im Aufruf der Methode `setsockopt` mit den Parametern *zmq.SUBCRIBE* und der Subscriptions als Routingkey, auf Zeile 364. Durch diesen Aufruf wird der Sub-Socket angewiesen alle Nachrichten mit diesem Routingkey an die Recv-Methode weiter zureichen.

```
350     def subscribe(self, wshandler):
351         """
352         Add a wshandler object to set
353         @param wshandler websocket handler object from frontend
354         """
355         self.__SUBLOCK.acquire()
356         self.__subscribe_count += 1
357         if not self.__runtime:
358             self.__runtime = time.time()
359         for sub in wshandler.subscriptions:
360             if sub in self.__WSHANDLER_DICT:
361                 self.__WSHANDLER_DICT[sub].add(wshandler)
362             else:
363                 self.__WSHANDLER_DICT[sub] = set([wshandler,])
364                 self.__ZSOCKET.setsockopt(zmq.SUBSCRIBE, b"" + sub + "")
365         self.__SUBLOCK.release()
```

Listing 70: Methode `subscribe` des ZeroMQ-Backend-Threads (`Backend.py`)

Die in Listing 71 zu sehende Methode `unsubscribe` arbeitet ebenfalls wie die selbe Methode aus dem AMQP-Thread. Jedoch erfolgt hier das Entfernen eines Abonnements wieder über die Methode `setsockopt`. Dabei wird allerdings die Option *zmq.UNSUBSCRIBE* gefolgt vom Routingkey als Parameter übergeben.

```
367     def unsubscribe(self, wshandler):
368         """
369         Delete a wshandler object from the set
370         @param wshandler websocket handler object from frontend
371         """
372         self.__SUBLOCK.acquire()
373         self.__unsubscribe_count += 1
374         for sub in wshandler.subscriptions:
375             self.__WSHANDLER_DICT[sub].discard(wshandler)
376             if len(self.__WSHANDLER_DICT[sub]) == 0:
377                 del self.__WSHANDLER_DICT[sub]
378                 self.__ZSOCKET.setsockopt(zmq.UNSUBSCRIBE, b"" + sub + "")
379         self.__SUBLOCK.release()
```

Listing 71: Methode `unsubscribe` des ZeroMQ-Backend-Threads (`Backend.py`)

Wie in Listing 72 zu erkennen ist, erfolgt die Verarbeitung der Nachrichten im ZeroMQ-Thread innerhalb einer While-Schleife (Z. 315-333). Damit gleicht die Implementierung in dieser Beziehung der Implementierung des ZeroMQ-Threads aus dem Chat-Server. Auf Zeile 318 werden über die Methode `recv_multipart` die Nachrichten der Publisher empfangen. Dabei werden die Daten und der Routingkey in getrennte Variablen zwischengespeichert. Die anschließende Verarbeitung der Nachrichten, auf den Zeilen 319 bis 331, erfolgt wieder nach dem gleichen Prinzip wie im AMQP-Thread.

```
306     def run(self):
307         """
308         Run-method
309         @brief Start up Backendthread an consume incomming traffic.
310         Counts every message on in an out.
311         The message will be routet to each connect/subscribed websocket on the ↩
                frontend
312         """
313         print "Backend is up"
314         websockets = None
315         while True:
316             try:
317                 # Receive adress und data from subcribe socket
318                 [address, data] = self.__ZSOCKET.recv_multipart()
319                 self.__backend_msgcounter[address] += 1
320                 try:
321                     self.__SUBLOCK.acquire()
322                     websockets = set(self.__WSHANDLER_DICT[address])
323                     self.__SUBLOCK.release()
324                     for ws in websockets:
325                         try:
326                             ws.write_message(address + " - " + data)
327                             self.__frontend_msgcounter[address] += 1
328                         except Exception:
329                             self.__frontend_failcounter[address] += 1
330                 except KeyError:
331                     self.__frontend_keyerror[address] += 1
332             except ZMQError:
333                 break;
```

Listing 72: Nachrichtenverarbeitung im ZeroMQ-Backend-Thread (`Backend.py`)

6.4 Tests

6.4.1 Testplanung

Vor Beginn der Testphase wurden, unter Einbeziehung der Erfahrungen aus den Test des Chat-Systems, die Parameter für das Nachrichtenticker-System festgelegt. Um die Lastverteilung innerhalb des Backend beurteilen zu können, wurde zunächst festgelegt, dass die Anzahl der Subscriber analog zur Anzahl der Server steigen sollte. Das heißt, dass bei allen Testläufen im Mittel immer die gleiche Anzahl an Subscribern mit den Servern verbunden sein soll. Durch dieses Vorgehen konnte davon ausgegangen werden, dass die Last im Backend bei einer konstanten Zahl von Servern nur von der Anzahl der Publisher abhängt. Um dennoch eine Beurteilung des Einflusses der Komplexität auf das Lastverhalten im Backend abgeben zu können, erfolgten die Testläufe zusätzlich mit einer unterschiedlichen Anzahl an Servern. Für die Server wurden, wie zuvor beim Chat-System, maximal zehn Workstation aus dem Computerpool A1.26 ausgewählt.

Für die Publisher wurden die verblieben 10 Workstations im selben Computerpool eingesetzt. Sie sollten, so weit genutzt, je 25 Publisher starten und somit im Maximum 250 Publisher erzeugen. Durch die Aufteilung konnte von einer konstanten Auslastung der verwendeten Workstations in allen Testläufen ausgegangen werden. Die Auswahl der Workstations, für Server und Publisher, erfolgte anhand ihrer Leistungsfähigkeit.

Da in der Auswertung der Testläufe des Chat-Systems festgestellt wurde, dass eine hohe Anzahl an Chat-Clients pro Server einen starken Einfluss auf das Verhalten im Backend hatte, sollte dies im Nachrichtenticker vermieden werden. Um eine Überlastung des Frontends des Servers zu vermeiden, wurde die Anzahl der Subscriber pro Server auf 50 reduziert. Aufgrund dieser Entscheidung wurden im Maximum 40 Workstations benötigt um eine konstante Belastung von zehn Servern mit je 50 Subscribern zu erreichen. Die Subscriber mussten somit auf die verblieben Workstations in den Computerpools A1.27 und A1.28 aufgeteilt werden. Dabei war in der Auswertung der Testläufe mit zehn Servern zu beachten, dass die Workstations in A1.28 wesentlich leistungsstärker sind als die Workstations in A1.27.

Die Größe der zu versenden Nachrichten durch die Publisher wurde für alle Testläufe auf 256 Byte festgelegt. Ferner wurde der zeitliche Abstand zwischen zwei Nachrichten in den Tests auf eine Sekunde festgelegt. Dies sollte zum einen die zuverlässige Beurteilung des Lastverhaltens ermöglichen und zum anderen die Überlastung der Server im Frontend vermeiden. Der AMQP-Broker wurde, wie im Chat-System, auf einem privaten Laptop betrieben

Die Gesamtlaufzeit der einzelnen Testläufe wurde, wie im Chat-System, auf 300 Sekunden gesetzt. Die Erfassung des Speicherverbrauchs und der CPU-Auslastung auf den Workstations erfolgte ebenfalls alle 10 Sekunden. Zu jeder Testkonfiguration erfolgten wieder mindestens drei Testläufe. Mit den in Abbildung 44 auf Seite 86 gezeigten Testkonfigurationen ergeben sich so die mindestens 144 auszuführenden Testläufe.

Tests		ZMQ					AMQP				
Server	Worker	1	2	3	4	5	1	2	3	4	5
1	25										
1	50										
1	100										
1	150										
1	200										
1	250										
2	25										
2	50										
2	100										
2	150										
2	200										
2	250										
5	25										
5	50										
5	100										
5	150										
5	200										
5	250										
10	25										
10	50										
10	100										
10	150										
10	200										
10	250										

Test OK
Test OK / Mit Abweichungen
Test ausgelassen

Abbildung 44: Checkliste für die Testläufe im Nachrichtenticker-System

6.4.2 Durchführung und Auswertung

Durchführung Die Bewertung der Testläufe erfolgte ebenfalls mit Hilfe des Bash-Skriptes `prf.sh`. Wie in Abbildung 45 auf Seite 87 auszugsweise zu erkennen ist, wurde es angepasst um auch die korrekte Anzahl der Logeinträge der Publisher zu erfassen. Diese Erweiterung ist in der Abbildung am Eintrag *Worker_start_end* zu erkennen. Die Auswertung der Fehlerrate erfolgte auch hier.

Während der Durchführung der Testläufe im ZeroMQ-System kam es gehäuft zu Fehlern in der Abarbeitung der Nachrichten aus dem Backend. Bedingt durch die ansteigende Zahl der Publisher in den Testläufen trat der Fehler zudem immer dann gehäuft auf, wenn das System mehr als 150 Publisher besaß. Während der Fehlersuche stellte sich heraus, dass die Filterung der Nachrichten anhand der Routingkeys nicht zuverlässig funktionierte. Das heißt, es wurden Nachrichten an die Receive-Methode weitergereicht, die von keinem Subscriber abonniert wurden. Bei der Analyse der Log-Dateien wurde klar, dass der Fehler immer dann in einem Server auftrat, wenn nicht alle verfügbaren Publisher abonniert wurden. Die Analyse zeigte im Weiteren, dass sich die Anzahl der nicht abonnierten Publisher immer im einstelligen Bereich bewegte. Leider konnte aufgrund der

```
Testzusammenfassung
===================

Server_start_end:       10 = 10
Worker_start_end:       250 = 250
Client_start_end:       1000 = 1000
Load_start_end:         60 = 60
CNC_start_end:          1 = 1

Client errors:          1
Errorsrate in per.:     1/1000*100 = .10000000000000000000

"Client 8","ws17a127","Error:1 code:1006 reason:Going away time:9.38332390785"
```

Abbildung 45: Ausgabe des Skriptes prf.sh (Nachrichtenticker)

knappen Zeit dieser Fehler nicht vollständig eingegrenzt und damit behoben werden. Der Autor vermutet jedoch einen Zusammenhang mit der verwendeten Version der *libzmq*, da Versuche mit einer aktuelleren Version, während der Einarbeitung in das Messaging-System diesen Fehler nicht aufwiesen. Der Changelog zur ZeroMQ in Version 3.0.0 bestätigt diese Vermutung[46].

Da, wie bereits in der Testplanung angedeutet, der Fokus der Auswertung auf die Lastverteilung im Backend gelegt werden sollte, wurden schlussendlich die Subscriber angepasst um die weiteren Tests nicht zu gefährden. Wie in Listing 73 zu sehen ist, wurde die Methode `received_message` so angepasst, dass jeweils der zuerst gestartete Subscriber alle Publisher abonniert (Z. 10-16). Dabei wurde ein kleines Ungleichgewicht innerhalb der Verteilung der Nachrichten an die Subscriber bewusst in Kauf genommen. Da der Fehler voraus setzt, dass in den bereits absolvierten Testläufen immer alle Publisher abonniert wurden kann davon ausgegangen werden, dass diese Veränderung zu keinem signifikanten Unterschied in den Messwerten des Backends führen. Die Veränderung im Subscriber bewirkt zudem, dass beide Messaging-Systeme identisch beeinflusst werden.

```
7     if mo:
8         if not self.__subs:
9             workers =  mo.group(1).split()
10            if self.__NUM == "1":
11                subs = workers
12                self.__PARENT.add_logmessage("subcount:" + str(len(subs)))
13                self.__subs = "SUB"
14                for sub in subs:
15                    self.__subs += " " + sub
16                self.__PARENT.add_logmessage("subs:" + str(self.__subs))
17            else:
18                subs = rnd.sample(workers, rnd.randint(1,len(workers)))
19                self.__PARENT.add_logmessage("subcount:" + str(len(subs)))
20                self.__subs = "SUB"
21                for sub in subs:
22                    self.__subs += " " + sub
23                self.__PARENT.add_logmessage("subs:" + str(self.__subs))
24        self.send(self.__subs)
25        return
26
27    self.__msgcounter += 1
28    return
```

Listing 73: Fix der Methode `received_message`im Subscriber (`WebsocketThread.py`)

Auswertung Wie auch im Chat-System wurden nach Durchführung aller Testläufe die Daten mit Hilfe von Skripten gezielt gefiltert und ausgewertet. Auch hier wurden die, in Abbildung 44 auf Seite 86, gelb markierten Testläufe von der Auswertung ausgenommen. Die Auswertung erfolgte mit den erfassten Messwerten von Publisher und Server. Dabei wurden die insgesamt verteilten Nachrichten der Publisher und die insgesamt verarbeiteten Nachrichten in den Servern über den Testzeitraum ausgewertet. Zusätzlich wurde, wie schon im Chat-Server, die Anzahl der verarbeiteten Nachrichten pro Sekunde in den Servern ausgewertet. Um das Lastverhalten des Backends besser beurteilen zu können, wurden zudem die Speicherverbräuche der Server und des AMQP-Brokers ausgewertet. Diese Messwerte wurden wieder über die Anzahl der Server beziehungsweise Publisher und die erfolgten Testläufe einer Testkonfiguration gemittelt. Für die Darstellung der Speicherverbräuche der Server wurden die Messwerte nachträglich auf den ersten Messwert normiert. Somit zeigen die Kurven nur die relative Änderung der Speicherbelegung ab der 10 Sekunde des Testlauf. Die Grundlast, wie sie noch im Chat-System zu erkennen war, kann also nicht abgelesen werden. Eine Auswertung der CPU-Last konnte, aus den selben Gründen wie im Chat-System, ebenfalls nicht vorgenommen werden.

Die Auswertung der Messwerte, in Abbildungen 46 und 47 auf den Seiten 88 und 89, zeigte für die Testläufe mit 25 und 50 Publishern einen relativ flachen Speicherverbrauch für alle Kurven. Wie zu erwarten, scheint die Anzahl der Server bei konstanter Publisherzahl keinen Einfluss auf den Speicherverbrauch zu haben, da die Last durch die Anzahl der Publisher bestimmt wird. Vergleicht man beide Abbildungen, so scheint der Speicherverbrauch bei 50 Publishern etwas schneller anzusteigen. Auch verlaufen die Verbrauchskurven der Testsysteme mit AMQP-Backend etwas unterhalb der des ZeroMQ-Backends. Dies weist auf eine Entlastung durch den Broker hin.

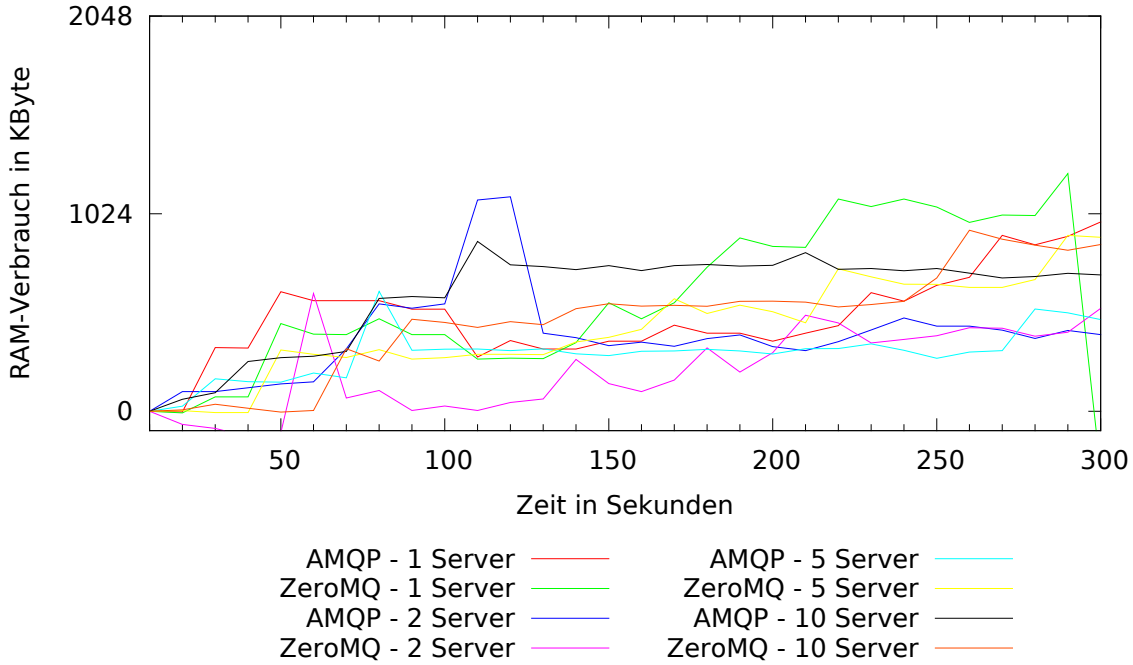

Abbildung 46: Speicherverbrauch bei 25 Publishern

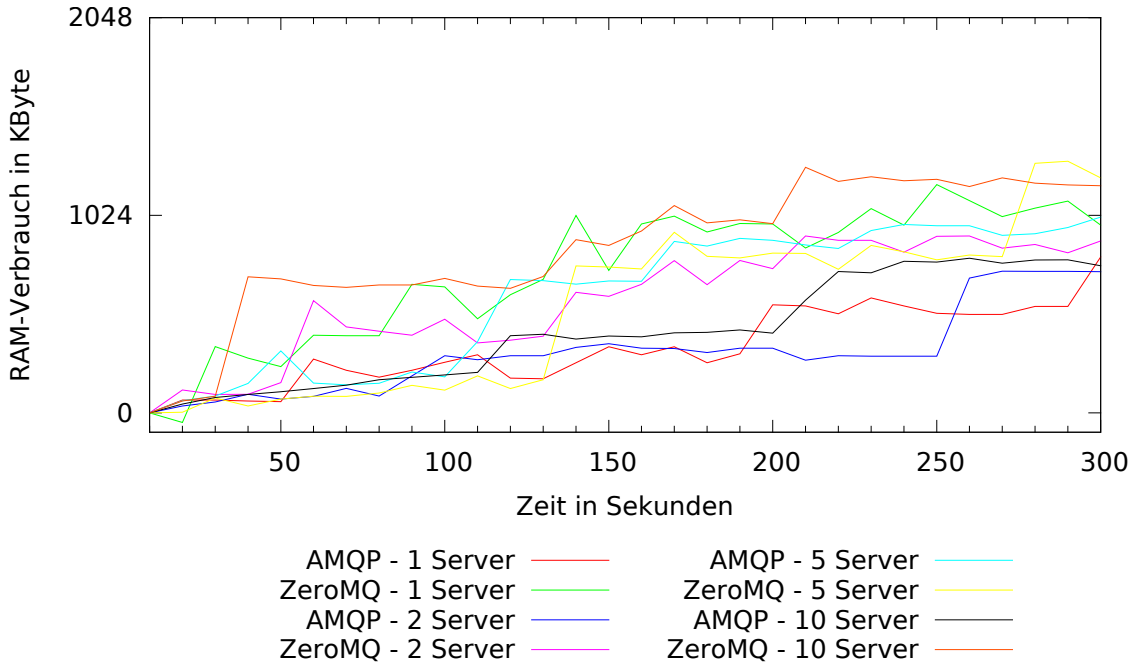

Abbildung 47: Speicherverbrauch in den Servern bei 50 Publishern

Die Verbrauchskurven der RabbitMQ-Broker, in den Abbildungen 48 und 49 auf den Seiten 89 und 90, bestätigen diese Feststellung. Sowohl für die Testläufe mit 25 Publishern als auch für die Testläufe mit 50 Publishern ist zu erkennen, dass der Speicherverbrauch während der Testläufe erhöht ist. Zudem fällt auf, dass die Belastung des Brokers mit der Anzahl der verbunden Server steigt.

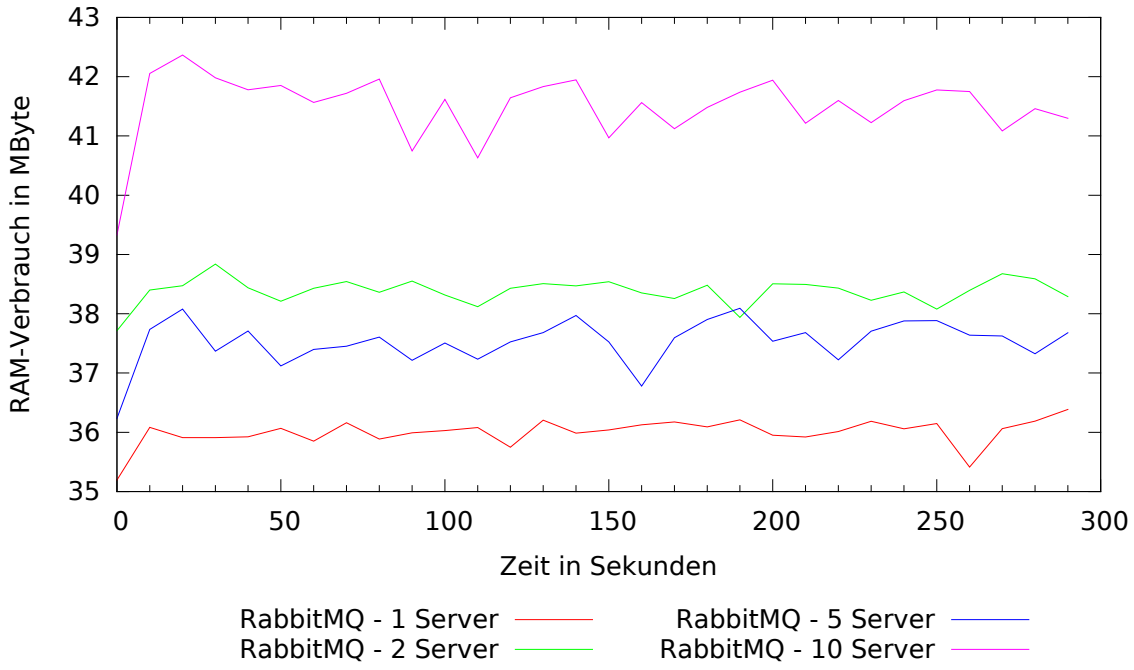

Abbildung 48: Speicherverbrauch im RabbitMQ-Broker bei 25 Publishern

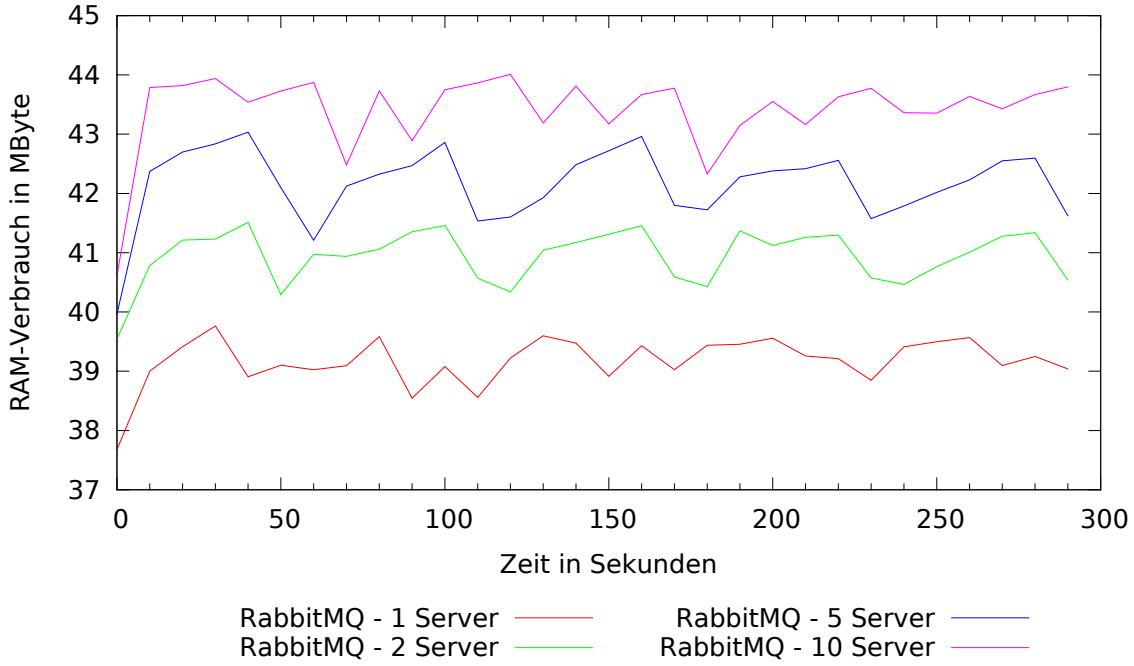

Abbildung 49: Speicherverbrauch im RabbitMQ-Broker bei 50 Publishern

Betrachtet man die Verbrauchskurven für die Testläufe mit 100 Publishern in Abbildung 50, so kann man eine deutliche Lastzunahme in den Systemen mit ZeroMQ-Backend festgestellt werden. Der Speicherverlauf für die Systeme mit AMQP-Backend verläuft jedoch relativ gleichmäßig. Aus dem Verlauf der Kurven für das ZeroMQ-Backend lässt sich mit Blick auf das Chat-System und dem Verhalten von ZeroMQ-Sockets annehmen, dass die Verarbeitung der Nachrichten im Backend der Server aufgrund der Verteilung der Nachrichten an die Subscriber blockiert wird. Aus der deutlichen höheren Last des AMQP-Brokers, in Abbildung 51 auf Seite 91, kann dies auch für das AMQP-Backend angenommen werden.

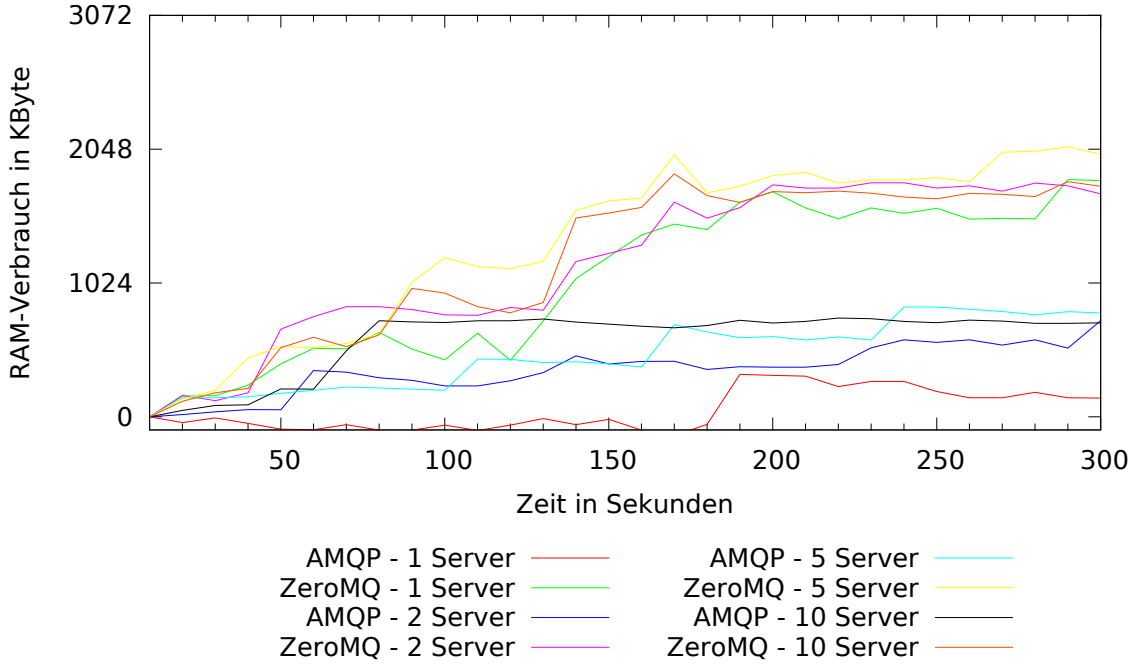

Abbildung 50: Speicherverbrauch in den Servern bei 100 Publishern

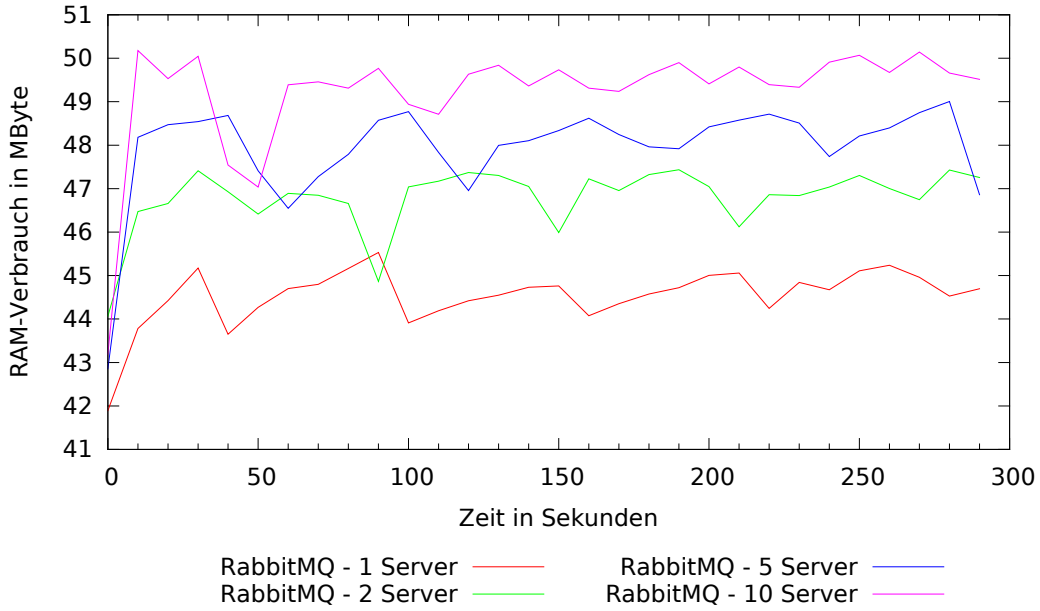

Abbildung 51: Speicherverbrauch im RabbitMQ-Broker bei 100 Publishern

Die Auswertung für 150 und 200 Publisher bestätigt, dass das beschriebene Verhalten unter Zunahme der Last immer deutlicher hervortritt. Die Verbrauchskurven der Server mit ZeroMQ-Backend, in Abbildung 52 auf Seite 91 und Abbildung 53 auf Seite 92, zeigen eine deutliche Zunahme des Speicherverbrauchs von zirka einem MByte je 50 Publisher. Die Kurven der Server mit AMQP-Backend blieben hingegen unter einem MByte stabil. Dafür erfolgte aber eine weitere Erhöhung der Last am RabbitMQ-Broker. Wie in den Abbildungen 54 und 55 auf Seite 92 gut zu erkennen ist, wirken sich die Testläufe mit 10 Serven am stärken auf den Broker aus. Eine Veränderung der Speicherverbräuche der Server unter Beachtung der Serveranzahl mit ZeroMQ-Backend ist jedoch nicht feststellbar. Da mit jedem neuen Server im ZeroMQ-Backend nur die Publisher belastet werden, ist dies ein zu erwartendes Ergebnis. Für alle Testläufe mit 150 und 200 Publisher lässt sich schlussfolgern, dass die Last im Frontend der Server zugenommen hat.

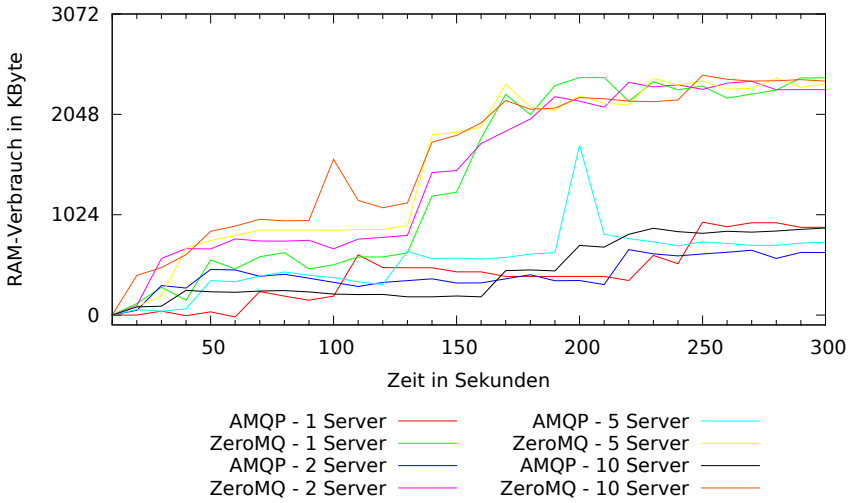

Abbildung 52: Speicherverbrauch in den Servern bei 150 Publishern

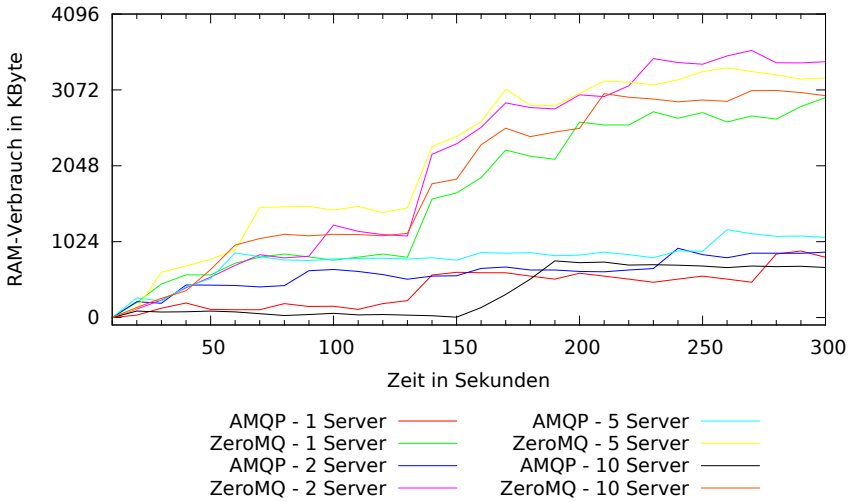

Abbildung 53: Speicherverbrauch in den Servern bei 200 Publishern

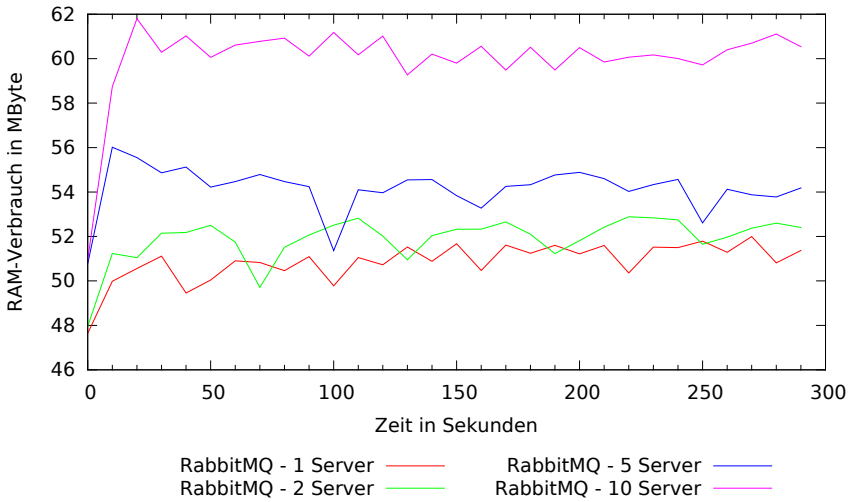

Abbildung 54: Speicherverbrauch im RabbitMQ-Broker bei 150 Publishern

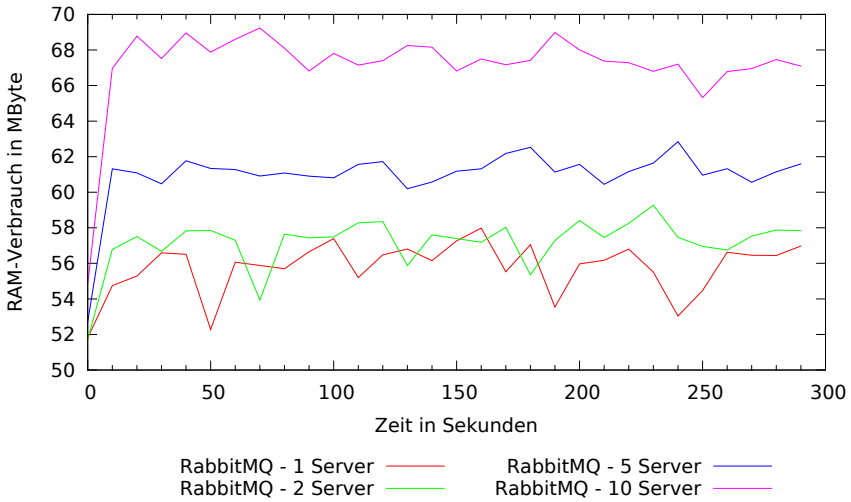

Abbildung 55: Speicherverbrauch im RabbitMQ-Broker bei 200 Publishern

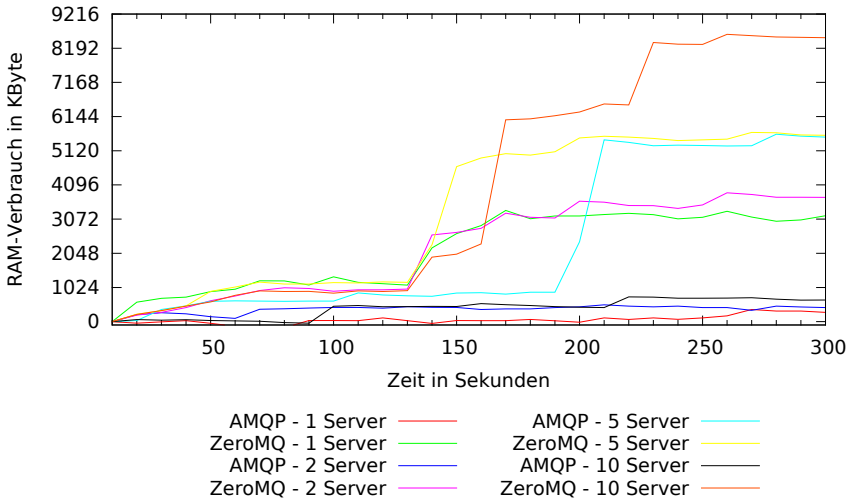

Abbildung 56: Speicherverbrauch in den Servern bei 250 Publishern

Die Testläufe mit 250 Publishern zeigen, in Abbildung 56, dass der ZeroMQ-Socket in den Servern sehr stark belastet wurde. Interessant ist, dass die Auswertung eine Abhängigkeit des Speicherverbrauchs von der Anzahl der Server ausweist. Gerade die Speicherverläufe mit fünf und zehn Servern zeigen eine deutlichen Anstieg ab Sekunde 140. Betrachtet man den Aufbau des ZeroMQ-Backend, so dürfte es keine Abhängigkeit dieser Art geben. Eine Überlastung der Publisher konnte ausgeschlossen werden, da diese eher zu einer Minimierung der Last an allen Servern führen würde. Wie eine genauere Analyse der berechneten Speicherverbräuche zeigte, kam es während des zweiten Testlaufs mit 10 Servern zu einer leicht erhöhten Belastung aller Server im System. Aufgrund der einfachen Mittelwertbildung und der Darstellungsart, kam es deshalb zu der extremen Überhöhung der Kurve. Ein Auszug aus der fehlerhaften Messung mit 10 Servern ist in Listing 74 zu sehen. Die Analyse der Messwerte für die Testläufe mit 5 Servern zeigte den gleichen Fehler. Für die hellblaue Verbrauchskurve der Testläufe mit 5 Servern für das AMQP-Backend wurde ebenfalls auf diesen Fehler zurückgeführt. Wie es zu diesen Abweichungen kam, lies sich abschließend nicht mehr herausfinden.

```
1  zeit ws15a126-zmq2 ws15a126-zmq3 ws15a126-zmq1 ws11a126-zmq2
2   10       0.0           0.0           0.0           0.0
3   20     248.0         504.0         248.0         248.0
4   30     248.0         496.0         124.0         372.0
5   40     248.0         496.0         116.0         744.0
6   50     620.0         612.0         364.0         752.0
7   60     984.0         860.0         612.0         868.0
8   ..       ..            ..            ..            ..
9  140    2100.0        2224.0        1596.0        1976.0
10 150    2224.0        2100.0        1712.0        2100.0
11 160    2224.0        2844.0        1968.0        2472.0
12 170   12772.0        3092.0        2340.0       14632.0
13 180   13020.0        3092.0        2588.0       14136.0
14 190   13020.0        2968.0        2712.0       14508.0
15 200   13028.0        3092.0        2588.0       14384.0
16 210   13144.0        3852.0        3084.0       14260.0
17 220   12896.0        3604.0        2836.0       14260.0
18 230   18800.0        3480.0        2960.0       20164.0
19 240   18684.0        3596.0        2960.0       20164.0
20 250   18652.0        3596.0        3084.0       20156.0
21 260   18528.0        3472.0        4464.0       20032.0
```

Listing 74: Auszug der fehlerbehafteten Messung mit 10 Servern im ZeroMQ-System

Betrachtet man die Abbildung und lässt die Ungenauigkeiten außer acht, so zeigt sich wieder das selbe Lastverhalten wie in den vorherigen Testläufen. Dabei zeigt das AMQP-Backend ein unverändertes Verhalten im Speicherverbrauch. Wie Abbildung 57 zeigt, wird die steigende Last hier weiterhin durch die Queues im Broker aufgefangen.

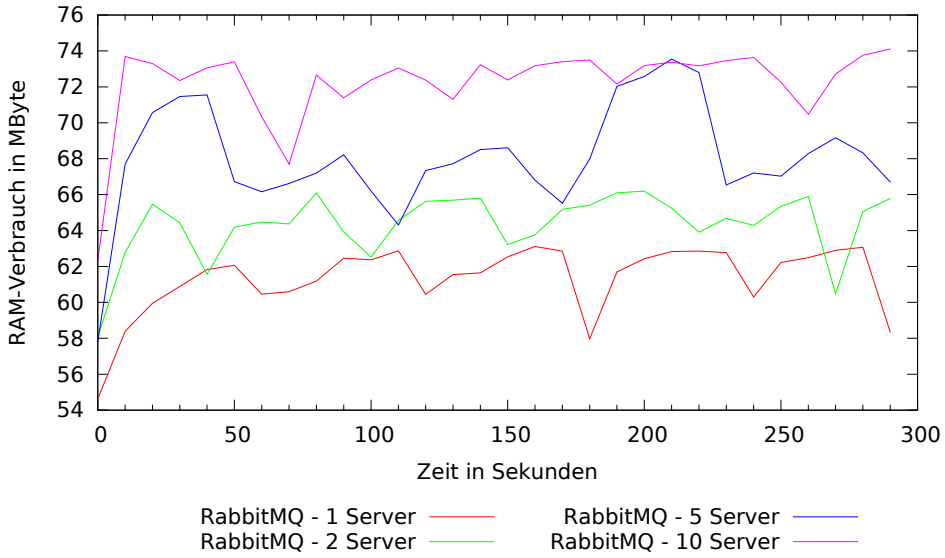

Abbildung 57: Speicherverbrauch im RabbitMQ-Broker bei 250 Publishern

Die Auswertung der verarbeiteten Nachrichten pro Sekunde aus dem Backend zeigte, dass die erzeugte Last in den Servern linear zur Anzahl der Publisher steigt. Wie die Abbildungen 58 und 59 auf den Seiten 94 und 95 für einen und fünf Server zeigen, ist dieses Verhalten unabhängig von der Anzahl der Server. Aus Gründen der Übersichtlichkeit wurde auf die Abbildung für zwei und zehn Server verzichtet, da sie das gleiche Verhalten zeigen.

Neben dem linearen Anstieg der verarbeiteten Nachrichten ist bei genauer Betrachtung auch zu erkennen, dass das ZeroMQ-Backend eine minimal schnellere Verarbeitung der Nachrichten ermöglichte. Wie auch im Chat-System wurde zirka eine Nachricht pro Sekunde mehr verarbeitet.

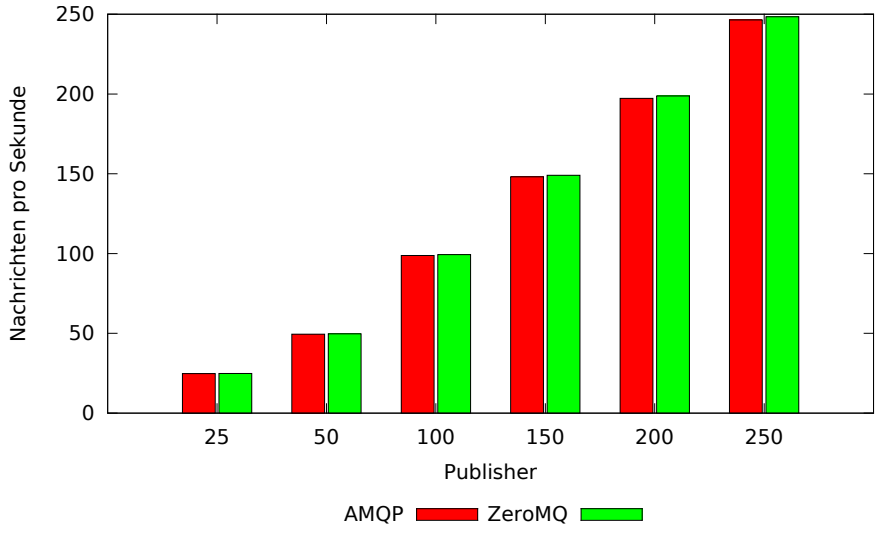

Abbildung 58: Verarbeitete Nachrichten aus dem Backend bei einem Server

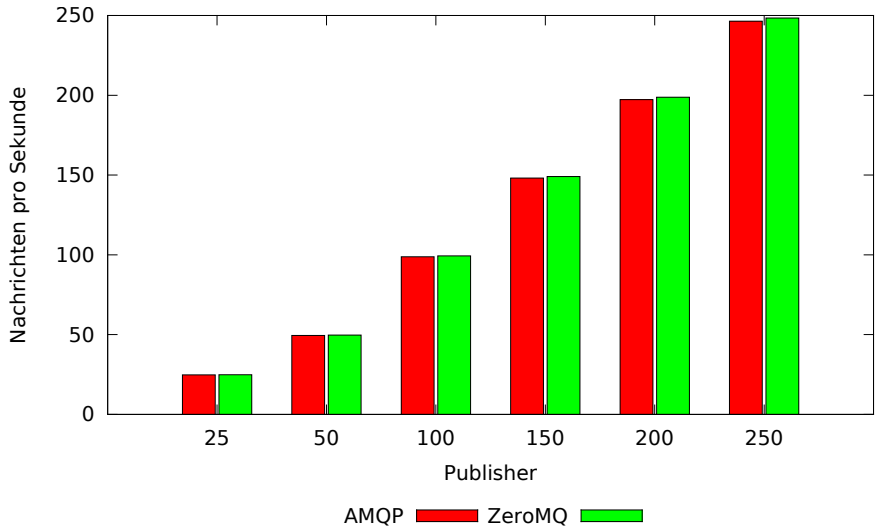

Abbildung 59: Verarbeitete Nachrichten aus dem Backend bei fünf Servern

Abschließend erfolgte eine Betrachtung der insgesamt erfolgreich verarbeiteten Nachrichten im Server in Bezug auf die verschickten Nachrichten aller Publisher des Systems. Dabei sollte festgestellt werden, wie viel Prozent der verschickten Nachrichten innerhalb des Testzeitraums verarbeitet werden konnten. Zur Berechnung wurde der Mittelwert über die verarbeiteten Nachrichten aller Server gebildet, die in einem Testlauf am System beteiligt waren. Dies erfolgte ebenso für die verschickten Nachrichten der Publisher. Danach wurde aus beiden Mittelwerten der prozentuale Anteil gebildet.

Die Auswertung für das ZeroMQ-System in Abbildung 60 auf Seite 96 zeigt, dass der prozentuale Anteil der verarbeiteten Nachrichten weder von der Anzahl der Server noch von der Anzahl der Publisher abhängt. Die leichte Variation ab einer Anzahl von zwei Servern ist auf die leicht unterschiedliche Verteilung der Subscriber im Frontend zurückzuführen und kann deshalb vernachlässigt werden. Die Abbildung zeigt deutlich, dass die Anzahl der verwalteten Verbindungen im Sub-Socket der Server in keiner Testkonfiguration einen Einfluss auf die Anzahl der verarbeiteten Nachrichten hatte. Die gleichmäßig gute Verarbeitung kann dabei auf die Nachrichtenverarbeitung im ZeroMQ-Socket zurückgeführt werden. Da die Abarbeitung der Nachrichten über die verbundenen Sockets erfolgt und nicht nach der zeitlichen Reihenfolge des Empfangs einer Nachricht kann davon ausgegangen werden, dass unabhängig von der Anzahl der Publisher immer eine faire Verarbeitung der Nachrichten erfolgt.

Wie in Abbildung 61 auf Seite 96 zu erkennen ist, besteht im AMQP-Backend ein Zusammenhang zwischen dem prozentualen Anteil der verarbeiteten Nachrichten und der Anzahl der Publisher im Backend. Mit zunehmender Anzahl der Publisher nimmt der Anteil der erfolgreich verarbeiteten Nachrichten im Backend der Server ab. Da die Speicherverbräuche der Server nur eine minimale Auslastung zeigten, deutet dies auf den Broker als Verursacher hin. Da die Verteilung der Nachrichten im Broker über einen Exchange erfolgt und dieser die Nachrichten in der Reihenfolge des Empfangs verteilt, werden die Nachrichten der zu erst abonnierten Publisher bevorzugt versandt. Damit erreichen die Nachrichten der zuletzt abonnierten Publisher die Server nicht mehr innerhalb des Testzeitraums.

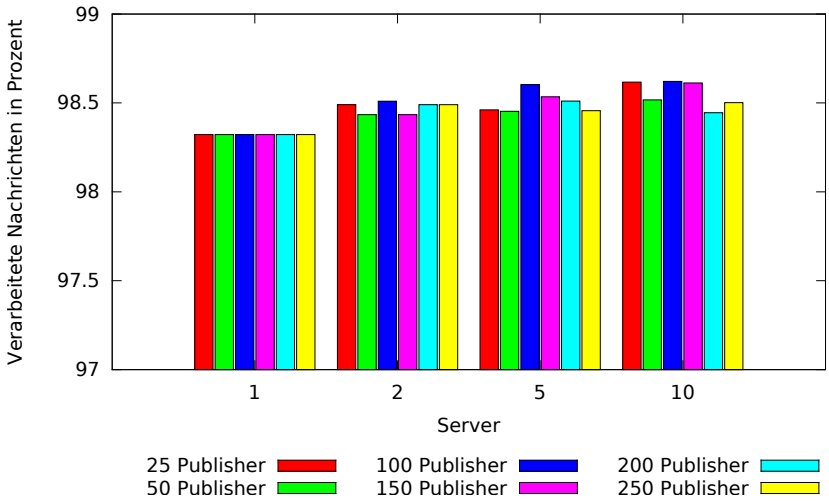

Abbildung 60: Verhältnis - Verarbeitete/Verschickten Nachrichten (ZeroMQ-Backend)

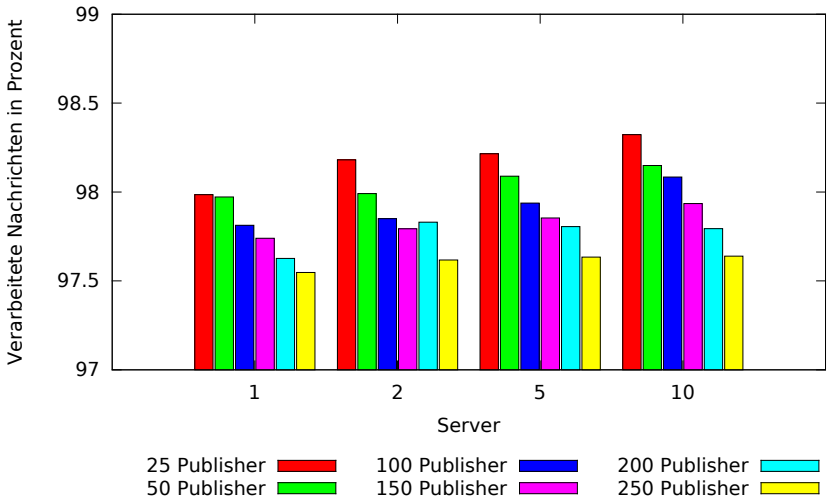

Abbildung 61: Verhältnis - Verarbeite/Verschickte Nachrichten (AMQP-Backend)

6.5 Zusammenfassung

Die Versuche im Tickersystem mit AMQP-Backend zeigten deutlich das eine Lastverschiebung zum RabbitMQ-Broker bei größerer Belastung des Systems erfolgte. Damit konnte in allen Versuchen nachgewiesen werden, dass der Broker maßgeblich an der Begrenzung der Auslastung der Server beteiligt war. Die Verschiebung der Last kann darauf zurückgeführt werden, dass die Server über den voreingestellten Prefetch-Count immer die gleiche Zahl an Nachrichten zwischenspeichern. Dies hat aber auch eine etwas langsamere Verarbeitung der Nachrichten zur Folge. Die Pufferung der Nachrichten im Broker kann somit im AMQP-System das Abfangen von möglichen Lastspitzen ermöglichen, wenn dieser die nötige Leistungsfähigkeit besitzt und über ausreichend Arbeitsspeicher verfügt. Aus der Verschiebung der Last hin zum Broker folgt außerdem, dass eine Skalierung des Systems auf Ebene der Server erst zu einem späteren Zeitpunkt erfolgen muss, als in einem ZeroMQ-System. Ein Nachteil des hier gezeigten Tickersystems mit AMQP-Backend ist, dass die Verteilung der Nachrichten der Publisher nicht fair erfolgt. Dies kann zumindest in Systemen mit vielen Subscribern und einer hohen Fluktuation an Subscriptions zu einer verzögerten Auslieferung führen.

Betrachtet man die Versuche mit ZeroMQ-Backend so zeigen die Ergebnisse, dass ähnlich wie im Chat-System die Last der Server mit der Anzahl Publisher steigt. Dieser Lastanstieg kann im gezeigten Systemaufbau nur durch eine Skalierung der Server ausgeglichen werden. Die Skalierung kann in diesem ZeroMQ-Backend aber, im Unterschied zum ZeroMQ-Backend im Chat-System, mit ähnlichem technischen Aufwand umgesetzt werden, wie im AMQP-Backend. Bei einer horizontalen Skalierung der Server ist zu beachten, dass eine zusätzliche Belastung der Publisher erfolgt. Auch wenn die Ergebnisse zeigen, dass nicht mit Problemen zu rechnen ist, wenn nicht mehr als 10 Server mit den Publishern verbunden sind. Ein Vorteil der ZeroMQ-Lösung ist die faire und unabhängige Verarbeitung der Nachrichten aus den Publishern in den einzelnen Servern. Durch die Funktionsweise der ZeroMQ-Sockets und dem Aufbau des Systems wird sie nicht durch eine hohe Fluktuation der Subscriptions beeinflusst. Es muss aber festgestellt werden, dass ein Nachrichtenticker-System mit ZeroMQ-Backend immer auf Server mit einer höheren Leistungsfähigkeit angewiesen ist, als ein vergleichbares Nachrichtentickersystem mit AMQP-Backend.

Die Ergebnisse lassen den Schluss zu, dass beide Messaging-Systeme in einem webasierten Nachrichtenticker zum Einsatz kommen könnten. Dies liegt im wesentlichen darin begründet, dass diese meist eine niedrigere Nachrichtenfrequenz aufweisen als in den Testläufen. Zielt das System auf eine möglichst gleichmäßige und faire Verteilung der Nachrichten ab, so kann dies mit ZeroMQ zuverlässig umgesetzt werden. Betrachtet man das System unter Beachtung der zu erwartenden Kosten bei einer Skalierung, so kann ein AMQP-System flexibler erweitert werden. Die Skalierung kann sowohl im Broker als auch in den Servern erfolgen.

7 Abschließende Betrachtungen

Mit Hilfe des entwickelten Testsystems konnte das Lastverhalten beider Messaging-Systeme in Bezug auf die Anwendungen analysiert werden. Dabei bestätigten sich die vorab gemachten Annahmen über das Lastverhalten und die Skalierbarkeit der Messaging-Systeme. Bezüglich des ZeroMQ-Systems konnte zudem festgestellt werden, dass die größere Anzahl an Verbindungen einen geringen Einfluss auf das Lastverhalten der ZeroMQ-Sockets hatte als zunächst vermutet. Vergleicht man die mit diesem Messaging-System getesteten Anwendungen so zeigte sich deutlich, dass das Lastverhalten im ZeroMQ-System eng mit dem Aufbau des Systems verknüpft ist. Im Zusammenspiel mit dem flexiblen Nachrichtenaufbau und der großen Vielfalt an ZeroMQ-Sockets kann das Last- und Skalierungsverhalten eines ZeroMQ-Systems also viel genauer gesteuert werden, als dies in einem AMQP-System möglich wäre. Diese Flexibilität ist aber auch der größte Nachteil des ZeroMQ-Systems, da durch sie der Entwicklungsaufwand gerade in der Planungsphase massiv erhöht wird. Werden hier die falschen Entscheidungen bezüglich des Systemaufbaus getroffen ist es möglich, dass eine gute Lastverteilung und Skalierung nicht erreicht werden kann. Die Entwicklung eines leistungsfähigen ZeroMQ-Systems setzt zudem ein tiefere Einarbeitung in die Thematik der verteilten Anwendungen und Lastverteilung voraus. Genau in diesem Punkt scheint das AMQP-System seine größten Vorteile zu haben. Die geringe Komplexität und die Zentralisierung des System ermöglichen den schnellen und stabilen Aufbau eines Messaging-Systems ohne eine tiefere Einarbeitung in die zuvor genannten Thematiken.

Die gute Dokumentation beider Messagingsysteme unterstützten sowohl die Einarbeitung als auch die Implementation. Jedoch zeigte sich, dass die Wahl der verwendeten Module einen großen Einfluss auf die Performance der Systeme hatte. Dies betraf vor allem die Implementierung der Server. Die Auswahl der blockierenden API für beide Messaging-Systeme sorgte insbesondere im Chat-System für eine geringere Leistungsfähigkeit. Auch die synchronisierten Methoden und Codeabschnitte trugen erheblich zur Einschränkung der Leistungsfähigkeit bei. Deshalb wurde versucht einen Dispatcher zu implementieren, der die Synchronisierung teilweise hätte aufheben können. Der Versuch wurde aber aufgrund von Problemen und Zeitmangel nicht vollständig umgesetzt und kam deshalb nicht zur Anwendung. In einer zukünftigen Arbeit wäre dies ein Ansatzpunkt um Tests mit einer größeren Anzahl an Clients im Frontend umsetzen zu können. Dabei könnte die Implementierung von Frontend und Backend vollständig durch eine asynchrone Tornado-Applikation ersetzt werden. Dies würde die synchrone Verarbeitung der Nachrichten aufheben und so für eine bessere Performance der Server sorgen.

Betrachtet man das Testsystem so kann festgestellt werden, dass es für die getesteten Anwendungen seinen Zweck vollständig erfüllte. Die Steuerung über einen zentralen Testcontroller und die globale Konfigurationsdatei erlaubten einen sehr flexiblen Aufbau des Testsystem und der Anwendungen. Die Konzeption und Implementierung ermöglicht zudem, dass der Test von anderen Anwendungen ohne eine wesentliche Änderung am Testsystem erfolgen kann. Das zentrale Erfassen der Messwerte trug erheblich dazu bei, dass die Auswertung der Ergebnisse ohne großen Aufwand erfolgen konnte. Es zeigte sich aber, dass die Stabilität des Testcontrollers durch eine ausführlichere Fehlerbehandlung für die verbundenen Sockets noch erhöht werden könnte. Eine weitere Möglichkeit zur Verbesserung des Testsystem liegt in der Abarbeitung der verbundenen Anwendungen des zu testenden Systems. Durch die serielle Abarbeitung der einzelnen Anwendungstypen entstanden zeitliche Ungenauigkeiten bei der Messwerterfassung. Diese könnten reduziert werden, wenn das Versenden der Kommandos über einzelne Threads innerhalb eines

Dispatchers erfolgen würde. Ferner wäre es möglich die Genauigkeit der Messwerte und die Steuerung der Abläufe durch eine Synchronisation der Zeit zwischen den einzelnen Anwendungen weiter zu verbessern.

Die Wahl der Testumgebung hatte ebenfalls großen Einfluss auf die Testläufe. Die als Testumgebung gewählten Computerpools boten zwar einen schnellen Zugriff auf die Hardware und einen reduzierten Konfigurationsaufwand. Jedoch zeigte sich, dass mit dem freien Zugang zu den Computerpools durch andere Studenten eine schwer zu kontrollierende Fehlerquelle entstand. Insbesondere wegen dieser externen Eingriffe traten zu Beginn der Testphase gehäuft Probleme und Fehler auf. Für zukünftige Tests sollte deshalb eine geschlossene Testumgebung auf virtuellen Maschinen genutzt werden. Dies würde auch das Testen aktuellerer Versionen der Messaging-Systeme ermöglichen, da in der Wahl der Software nicht mit Einschränkungen zu rechnen ist.

Mit Blick auf zukünftige Arbeiten lassen die Tests selbst den größten Spielraum zu. Da aufgrund der zeitlichen Gegebenheiten keine vollständige Auswertung der Daten erfolgen konnte, könnte eine zukünftige Arbeit diese unter anderen Gesichtspunkten auswerten. Insbesondere eine genauere Untersuchung der Daten zum Websocket basierten Frontend scheint dem Autor hier interessant. Ferner könnten neue Tests über eine längere Laufzeit weitere Ergebnisse liefern. Diese würden auch eine Beurteilung der CPU-Auslastung ermöglichen. Mit einer Erweiterung der Anwendung für die Lasterfassung könnten dabei auch zusätzliche Messwerte über die genauen Zustände im Prozess der zu testenden Anwendung erfasst werden. Von den untersuchten Messaging-Systemen bietet sich gerade das ZeroMQ-System für weitere Tests an. Die Kombinationsmöglichkeiten der ZeroMQ-Sockets bieten zusammen mit den nicht betrachteten Messaging-Pattern ein großes Potenzial für weitere Untersuchungen. Letztlich kann aber auch das Testsystem selbst zur Untersuchung andere verteilter Systeme herangezogen werden.

8 Fazit

Die Ausführung der vorliegenden Arbeit zeigte dem Autor deutlich, welchen Aufwand die Konzeption und Umsetzung eines lauffähigen und verlässlichen Testsystems verursacht. Die zusätzliche Entwicklung der zu testenden Systeme und die folgenden Tests ließen den gegebenen Zeitrahmen daher teilweise als viel zu kurz erscheinen. Dennoch konnte der Autor während der Arbeit tiefergehende Kenntnisse über den Aufbau und das Lastverhalten von Messaging-Systemen erlangen. Dabei zeigte gerade das Messaging-System ZeroMQ neue Konzepte und Herangehensweisen auf, die sicher in zukünftige Arbeiten einfließen werden. Die Möglichkeit ZeroMQ auch innerhalb eines Prozesses nutzen zu können stößt dabei auf besonderes Interesse.

Der Autor kommt zum persönlichen Schluss, dass beide Messagingsysteme unabhängig vom Lastverhalten einen vollständig anderen Zweck verfolgen. Dabei scheint ihm ZeroMQ besonders für eher untypische Einsatzgebiete mit sehr speziellen Ansprüchen konzipiert zu sein. Die große Flexibilität des Protokolls ermöglicht in diesem Fall eine genauere Anpassung an die Umgebung. Dies zeigt, nach Meinung des Autors, auch der Einsatz des ZeroMQ-System im CERN. AMQP hingegen ist für die gängigen Einsatzgebiete im Businessbereich bestens gerüstet und bietet alle dort verlangten Eigenschaften. Nach Meinung des Autors ist es daher die bessere Wahl, wenn ein stabiles und leicht wartbares System mit den üblichen Messaging-Pattern benötigt wird. Deshalb würde der Autor bei der Entwicklung von webbasierten Anwendungen, welche auf die üblichen Messaging-Pattern zurückgreifen, in Zukunft AMQP den Vorzug geben. Bei spezielleren Problemstellungen würde er aber jeder Zeit auf ZeroMQ als mögliche Lösung setzen.

Literatur

[Hin13] Pieter Hintjens. *ZeroMQ - Messaging for Many Applications*. O'Reilly Media, Inc., Sebastopol, 2013. ISBN 978-1-449-33406-2.

[Roe11] Kevin Roebuck. *Advanced Message Queuing Protocol (AMQP): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Pty Limited, 2011. ISBN 978-1-743-04837-5.

Quellen

[1] OASIS: *The Problem We Solve*, Abrufdatum: 04.08.2014. `http://www.amqp.org/product/solve`.

[2] Group, AMQP Working: *AMQP Spezification Version 1.0*, Abrufdatum: 06.08.2014. `http://www.amqp.org/sites/amqp.org/files/amqp.pdf`.

[3] Wikipedia: *Advanced Message Queuing Protocol*, Abrufdatum: 28.07.2014. `https://en.wikipedia.org/wiki/AMQP`.

[4] Hintjens, Pieter: *Whats wrong with AMQP*, Abrufdatum: 08.08.2014. `http://www.imatix.com/articles:whats-wrong-with-amqp/`.

[5] OASIS: *AMQP Working Group Transitions to OASIS Member Section*, Abrufdatum: 29.07.2014. `http://www.amqp.org/node/54`.

[6] OASIS: *OASIS Forms AMQP Technical Committee to Advance Business Messaging Interoperability Within Middleware, Mobile, and Cloud Environments*, Abrufdatum: 29.07.2014. `http://www.amqp.org/node/58`.

[7] JTC1: *ISO/IEC 19464:2014*, Abrufdatum: 28.07.2014. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64955&commid=45020`.

[8] OASIS: *Members*, Abrufdatum: 29.07.2014. `http://www.amqp.org/about/members`.

[9] OASIS: *Products and Success Stories*, Abrufdatum: 06.08.2014. `http://www.amqp.org/product/realworld`.

[10] Roebuck, Kevin: *Advanced Message Queuing Protocol (AMQP): High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Pty Limited, 2011.

[11] T. Dierks und E. Rescorla: *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*, 2008. `http://www.ietf.org/rfc/rfc5246.txt`.

[12] A. Melnikov und K. Zeilenga: *RFC 4422 - Simple Authentication and Security Layer (SASL)*, 2006. `http://www.ietf.org/rfc/rfc4422.txt`.

[13] Pivotal Software, Inc.: *High-level Overview of AMQP 0-9-1 and the AMQP Model*, Abrufdatum: 06.08.2014. `https://www.rabbitmq.com/tutorials/amqp-concepts.html`.

[14] iMatix Corporation: *Welcome to ØMQ for AMQP users*, Abrufdatum: 06.08.2014. `http://zeromq.org/docs:welcome-from-amqp`.

[15] RabbitMQ: *http://www.rabbitmq.com/*, Abrufdatum: 06.08.2014. `https://www.rabbitmq.com`.

[16] Pivotal Software, Inc.: *hello-world-example-routing.png*, Abrufdatum: 06.08.2014. http://www.rabbitmq.com/img/tutorials/intro/hello-world-example-routing.png.

[17] Pivotal Software, Inc.: *python-one-overall.png*, Abrufdatum: 09.08.2014. http://www.rabbitmq.com/img/tutorials/python-one-overall.png.

[18] Pivotal Software, Inc.: *direct-exchange.png*, Abrufdatum: 09.08.2014. https://www.rabbitmq.com/img/tutorials/direct-exchange.png.

[19] Pivotal Software, Inc.: *python-five.png*, Abrufdatum: 09.08.2014. https://www.rabbitmq.com/img/tutorials/python-five.png.

[20] RabbitMQ: *Consumer Prefetch*, Abrufdatum: 10.08.2014. http://www.rabbitmq.com/consumer-prefetch.html.

[21] RabbitMQ: *Consumer Priorities*, Abrufdatum: 10.08.2014. http://www.rabbitmq.com/consumer-priority.html.

[22] iMatix Corporation: *Distributed systems are in our blood*, Abrufdatum: 08.08.2014. http://zeromq.org.

[23] iMatix Corporation: *Historical Highlights*, Abrufdatum: 08.08.2014. http://zeromq.org/blog:historical-highlights.

[24] iMatix Corporation: *ØMQ Licensing*, Abrufdatum: 08.08.2014. http://zeromq.org/area:licensing.

[25] iMatix Corporation: *ØMQ Language Bindings*, Abrufdatum: 10.08.2014. http://zeromq.org/bindings:_start.

[26] Dworak, A und Ehm,F und Charrue, P und Sliwinski, W: *The new CERN Controls Middleware*, Abrufdatum: 14.08.2014. https://www-acc.gsi.de/wiki/pub/Frontend/MeetingCMW130425/CHEP-2012-NewCERNcontrolsMiddleware.pdf.

[27] Dworak, A und Ehm,F und Charrue, P und Sliwinski, W: *MIDDLEWARE TRENDS AND MARKET LEADERS 2011*, Abrufdatum: 14.08.2014. http://zeromq.wdfiles.com/local--files/intro%3Aread-the-manual/Middleware%20Trends%20and%20Market%20Leaders%202011.pdf.

[28] iMatix Corporation: *Download*, Abrufdatum: 10.08.2014. http://zeromq.org/area:download.

[29] Hintjens, Pieter: *ØMQ - The Guide*, Abrufdatum: 10.08.2014. http://zguide.zeromq.org/page:all.

[30] iMatix Corporation: *ZMTP - The Protocol of Things*, Abrufdatum: 10.08.2014. http://zmtp.org/page:read-the-docs.

[31] iMatix Corporation: *ZeroMQ Feature List*, Abrufdatum: 08.08.2014. http://zeromq.org/docs:features.

[32] Hintjens, Pieter: *ZeroMQ Certificates, Design Iteration 1*, Abrufdatum: 08.08.2014. http://hintjens.com/blog:62.

[33] iMatix Corporation: *Broker vs. Brokerless*, Abrufdatum: 09.08.2014. http://zeromq.org/whitepapers:brokerless.

[34] Pivotal Software, Inc.: *fig24.png*, Abrufdatum: 09.08.2014. https://raw.githubusercontent.com/imatix/zguide/master/images/fig24.png.

[35] Pivotal Software, Inc.: *fig17.png*, Abrufdatum: 09.08.2014. https://raw.githubusercontent.com/imatix/zguide/master/images/fig16.png.

[36] Pivotal Software, Inc.: *fig18.png*, Abrufdatum: 09.08.2014. https://raw.githubusercontent.com/imatix/zguide/master/images/fig18.png.

[37] Pivotal Software, Inc.: *fig19.png*, Abrufdatum: 09.08.2014. https://raw.githubusercontent.com/imatix/zguide/master/images/fig19.png.

[38] Facebook: *tornado.websocket — Bidirectional communication to the browser*, Abrufdatum: 17.08.2014. http://www.tornadoweb.org/en/stable/websocket.html#tornado.websocket.websocket_connect.

[39] Sylvain Hellegouarch: *Requirements; ws4py 0.3.5 documentation*, Abrufdatum: 17.08.2014. https://ws4py.readthedocs.org/en/latest/sources/requirements/.

[40] Ian Bicking: *Introduction ; virtualenv 1.11.6 documentation*, Abrufdatum: 17.08.2014. http://virtualenv.readthedocs.org/en/latest/.

[41] Python Software Foundation: *logging — Logging facility for Python*, Abrufdatum: 17.08.2014. https://docs.python.org/2.6/library/logging.html?highlight=logging#module-logging.

[42] Python Software Foundation: *Logging Cookbook; Python v2.7.8 documentation*, Abrufdatum: 17.08.2014. https://docs.python.org/2/howto/logging-cookbook.html#logging-cookbook.

[43] INADA Naoki: *Accelerator for ws4py and AutobahnPython*, Abrufdatum: 20.08.2014. https://github.com/methane/wsaccel.

[44] Sylvain Hellegouarch: *Performances*, Abrufdatum: 20.08.2014. https://ws4py.readthedocs.org/en/latest/sources/performance/.

[45] Garnock-Jones, T. Roy, G., VMWare und Weitere: *Introduction to Pika*, Abrufdatum: 20.08.2014. https://pika.readthedocs.org/en/latest/intro.html.

[46] iMatix Corporation: *0MQ version 3.0.0 (alpha), released on 2011/07/12*, Abrufdatum: 15.08.2014. https://raw.githubusercontent.com/zeromq/zeromq3-x/master/NEWS.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir an der Hochschule für Telekommunikation Leipzig eingereichte Bachelorarbeit zum Thema

Vergleich von Lastverteilungsmechanismen unter Einsatz der Messaging-Protokolle ZeroMQ und AMQP

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Abbildungen in dieser Arbeit sind von mir selbst erstellt oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Hochschule/Universität eingereicht worden.

Leipzig, den 27. August 2014

Martin Stoffers