TU Dresden, Fakultät Informatik, Institut für Systemarchitektur

Belegarbeit

Entwicklung eines effizienten Nachrichten-Aggregators in Erlang

Ole Rixmann
Matrikelnummer: 3266061

30. September 2011

Betreut durch
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill,
Dr.-Ing. Marius Feldmann &
Dipl.-Medien-Inf. Philipp Katz

Gliederung

1 Einführung

Heutzutage haben Personen, die im Management großer Firmen arbeiten, oft das Problem, sehr viele Informationen aufnehmen zu müssen. Neben der Kommunikation, die innerhalb und außerhalb des Unternehmens stattfindet, werden diese Informationen in Nachrichtenströmen übermittelt.

Diese Arbeit ist ein Teil des PRISMA-Projekts, in dem eine Software entwickelt wird, die auf effiziente Art Informationsströme filtert und verknüpft. Durch das Entfernen von redundanten und unwichtigen Informationen kann die Zeit, die zur Informationsaufnahme benötigt wird, verringert werden und damit eine Produktivitätssteigerung bei der angesprochenen Zielgruppe erreicht werden. Gleichzeitig kann durch moderne KI-Methoden der Erhalt von wichtigen Nachrichten sichergestellt werden.

Das PRISMA-System ist auf verschiedene Komponenten aufgeteilt, die jeweils so ausgelegt sind, dass performance-kritische Teile der Anwendung skaliert werden können.

Die Benutzer des PRISMA-Systems wollen ihre Nachrichten möglichst ohne Verzögerung erhalten, deshalb müssen alle Nachrichtenströme kontinuierlich in das System eingespeist werden.

Die Komponente von PRISMA, die diese Aufgabe übernimmt, ist der Aggregator. Wenn das System unter großer Last steht, können auch mehrere Aggregatoren zum Einsatz kommen. Die Aufgabe der Aggregatoren ist es, kontinuierlich alle Datenströme, die in PRISMA verarbeitet werden, auf neue Nachrichten zu überprüfen bzw. neue Nachrichten in Empfang zu nehmen. Nach dem Empfang werden die Nachrichten in ein einheitliches Format überführt. In diesem Format werden sie gefiltert und an andere Systemkomponenten weitergeleitet. Weitere Filter werden von anderen PRISMA-Komponenten angewendet.

Durch die Vorfilterung der Nachrichten wird nur ein Bruchteil von ihnen an den Rest des Systems weitergeleitet. Daraus ergibt sich eine deutlich höhere Last auf den Aggregatoren als auf den restlichen Systemkomponenten. Der größte Teil der Nachrichten, die nicht an das System weitergeleitet werden, sind bereits im System vorhandene Nachrichten, die keiner wiederholten Verarbeitung bedürfen, denn viele Nachrichtenströme senden immer gleich alle Nachrichten und nicht nur die ungelesenen. Hinzu kommt, dass es wünschenswert ist, viele Nachrichtenströme mit einer möglichst hohen Frequenz abzufragen, um so ohne große Verzögerung neue Nachrichten in das PRISMA-System einzuspeisen. Aufgrund der großen Last, die auf den Aggregatoren liegt, erscheint es besonders lohnenswert, hier viel Aufwand in die Optimierung der Implementierung zu stecken. Außerdem gibt es, nach unseren Recherchen, keine Analyse von Erlang in dieser speziellen Domäne d.h. können auch keine ähnlichen Arbeiten betrachtet werden.

Es existiert bereits eine Implementierung des Aggregators in der Programmiersprache Java, doch in Java ist es umständlich viele Verbindungen zu anderen Rechnern aufrecht zu halten, weshalb der Aggregator schnell ausgelastet ist.

In dieser Arbeit wird der Versuch unternommen, eine effizientere Implementierung des Aggregators zu erstellen. Grundlage für diese Implementierung ist die Programmiersprache Erlang, die auf Parallelität, Verfügbarkeit und Fehlertoleranz ausgelegt ist. Aktuelle, in Erlang geschriebene Webserver, schaffen es, 10.000 Anfragen pro Sekunde zu beantworten. Selbst wenn das Intervall, in dem die Nachrichtenströme abgerufen werden, sehr klein ist, z.B. 5 Sekunden, könnten mit 10.000 gestellten Anfragen pro Sekunde 50.000 Nachrichtenströme kontinuierlich abgerufen werden.

Um eine möglichst effiziente Implementierung des Aggregators zu entwickeln, soll diese Arbeit auch als Grundlage für einen Vergleich mit anderen Implementierungen des Aggregators dienen. Ein Bestandteil ist daher die Konzeption und Durchführung von Testreihen für den Aggregator.

Für die Durchführung dieser Testreihen wird ein zweites Programm benötigt, das gegenüber dem zu testenden Aggregator den Rest des PRISMA-Systems und je nach Test auch die Nachrichtenströme simuliert.

Diskussion der Ziele dieser Arbeit:

- Konzeption eines Nachrichtenaggregators in der Programmiersprache Erlang.
- Konzeption und Durchführung von Testreihen für die Aggregator-Komponente des PRISMA-Systems.
- Umsetzung einer Testumgebung, mit Hilfe derer diese Testreihen auf alternative Implementierungen der Aggregator-Komponente angewendet werden können.

Zunächst werden die Anforderungen, die an die im Rahmen dieser Arbeit umzusetzende Aggregator-Komponente gestellt werden müssen, analysiert. Anschließend folgt ein Kapitel, in dem technische Grundlagen beschrieben werden, die in den folgenden Kapiteln benutzt werden. Darauf folgt eine Beschreibung der Konzeption und Implementierung der Aggregator-Komponente. Im weiteren werden Testreihen konstruiert und spezifiziert. Die zur Durchführung der Testreihen benötigte Software wird konstruiert und implementiert. Auf eine Beschreibung der Testumgebung folgt die Durchführung und Analyse der Testreihen. Abschließend folgt ein Kapitel mit Fazit und Ausblick zu dieser Arbeit.

2 Anforderungsanalyse

Aus der Architektur des PRISMA-Systems und der entsprechenden Rolle des Aggregators, die im Einführungskapitel geschildert wurde, ergeben sich die hier aufgeführten Anforderungen an die Entwicklung der Software, die im Rahmen dieser Arbeit entstehen soll.

[Effizienz] Die Implementierung des Aggregators soll möglichst effizient arbeiten.

Effizienz bedeutet im Kontext des Aggregators, dass die Implementierung darauf optimiert ist, möglichst viele Abonnements in möglichst kurzen zeitlichen Abständen auf neue Nachrichten zu überprüfen. Unter diesem Kriterium findet die Wahl der verwendeten Programmiersprache, wie auch die Wahl der in der Implementierung benutzten Bibliotheken statt.

Der Aggregator soll in einer "typischen" Nutzungssituation eingesetzt werden können, wobei hier von ca. 150 Nutzern pro Instanz ausgegangen wird.

Bei willkürlich angenommenen 30 Abonnements pro Benutzer, ergeben sich damit ca. 4500 Abonnements, die eine Instanz des Aggregators verarbeiten können sollte. Da die Erfüllung dieses Kriteriums von der verwendeten Hardware abhängig ist, sollte diese Zahl nur als Richtwert verstanden werden.

[Integration] Da der Aggregator keine alleinstehende Software, sondern Teil des PRISMA-Systems ist, muss er die definierten Schnittstellen zum Rest des Systems unterstützen.

Die Kommunikation mit den anderen Komponenten des PRISMA-Systems findet über XMPP statt, der Inhalt der einzelnen Nachrichten besteht aus JSON in Form einer Objektstruktur.

Eingehende Nachrichten sind in dieser Objektstruktur als Subscriptions mit einer SourceSpecification und beliebig vielen FilterSpecifications dargestellt.

Ausgehende Nachrichten haben die Struktur einer Message, die aus MessageParts besteht oder die einer einfachen ErrorMessage, mehr dazu im Grundlagenkapitel (3.1) über das PRISMA-System.

[Heterogene Nachrichtenströme] Auf der anderen Seite kommuniziert der Aggregator mit den Nachrichtenströmen über Protokolle, die von der Art des jeweiligen Nachrichtenstroms abhängen.

Vorgesehen ist die Umsetzung von 3 Nachrichtenströmen: RSS, ATOM und XMPP. Damit muss der Aggregator auch Subscriptions für diese drei Nachrichtenströme verarbeiten können. In Kapitel 3.8 werden diese Nachrichtenströme beschrieben.

[Dokumentformat] Des weiteren gehört zu dieser Arbeit das Entwickeln eines einheitlichen Dokumentenformats, in dem eingehende Nachrichten an die anderen Systemkomponenten weitergeleitet werden können, sodass diese kein Wissen über die Art des Nachrichtenstroms benötigen, aus dem eine Nachricht stammt.

Der Prozess der Konzeption und Implementierung des Aggregators wird im Kapitel "Architektur des Aggregators" dargestellt, wo auch die benötigten Änderungen an der Umgebung (Betriebssystem) des Aggregators festgehalten werden.

Dies geschieht, um eine Grundlage für die Verwendung des Aggregators im PRISMA-Projekt zu schaffen (Installationsanleitung) und um zukünftige Änderungen am Quellcode zu erleichtern.

[Testreihen] Das Testen des Aggregators und das Schaffen einer Vergleichsgrundlage sind Bestandteil dieser Arbeit, daher ergibt sich die Notwendigkeit, geeignete Testreihen auszuarbeiten. In den Testreihen wird die Auslastung der "wichtigen" Systemressourcen in unterschiedlichen Szenarien und auf unterschiedlich leistungsfähigen Maschinen untersucht. Die Skalierbarkeit des Aggregators unter Verwendung mehrerer CPU-Kerne ist aufgrund der aktuellen Entwicklung der CPUs ein interessanter Punkt für den Vergleich mit anderen Implementierungen und sollte deshalb großen Einfluss auf die Wahl der Testmaschinen haben.

Um die Tests praktisch durchführen zu können, wird eine Implementierung von Testprogrammen benötigt, deren Entwicklung hier auch beschrieben wird.

3 Grundlagen

Hier werden technische Grundlagen beschrieben, die zum Verständnis der folgenden Kapitel benötigt werden. Zunächst wird auf Details der Konzeption des PRISMA-Systems eingegangen. Anschließend werden Eigenschaften der Programmiersprache Erlang beschrieben, die sie von anderen Programmiersprachen abheben. Eingegangen wird auf Teile der Laufzeitumgebung und Teile des Ökosystems, das sich um Erlang gebildet hat. Ein HTTP-Server (YAWS), mehrere Datenbanken (Couchdb und Mnesia) und ein XMPP-Server (Ejabberd) gehören dazu. Desweiteren werden die Datenströme ATOM, RSS und XMPP beschrieben.

3.1 PRISMA Projekt

Das PRISMA-System dient, wie bereits in der Einführung beschrieben, dem Extrahieren, Klassifizieren und Verdichten von Informationen aus Nachrichtenströmen, mit dem Ziel, die Empfänger der Informationen zu entlasten. Zu diesem Zweck sollen im PRISMA-System unterschiedliche Algorithmen getestet werden. Ein weiteres Design-Ziel ist die Skalierbarkeit von stark belasteten Komponenten. Eine genauere Beschreibung des PRISMA-Systems ist in [prisma-architecture] enthalten.

Die Komponente, die neue Nachrichten in das System einspeist, ist der Aggregator, der Thema dieser Arbeit ist. In Abbildung 3.1 ist die Architektur des gesamten PRISMA-Systems dargestellt. Die Kommunikation zwischen den einzelnen Komponenten des PRISMA-Systems findet über XMPP statt. Einzelne Maschinen im System haben ihren eigenen XMPP-Server, so auch der Aggregator.

Die beiden anderen Komponenten des Systems, mit denen der Aggregator kommuniziert, sind der Coordinator und der Aggregator Accessor. Der Coordinator versorgt die Aggregatoren mit neuen Nachrichtenstrom-Abonnements. Außerdem erhält er Fehlermeldungen von den Aggregatoren, wenn Fehler beim Abrufen der Nachrichtenströme auftreten. Im Falle einer Überlastung einzelner Systemkomponenten kann der Coordinator die Topologie ändern. Bezogen auf den Aggregator bedeutet dies, dass er neue Aggregatoren starten, alte stoppen und Abonnements zwischen Aggregatoren verschieben kann. Der Aggregator Accessor erhält vom Aggregator alle neuen Nachrichten, die der Aggregator aus den Nachrichtenströmen empfängt.

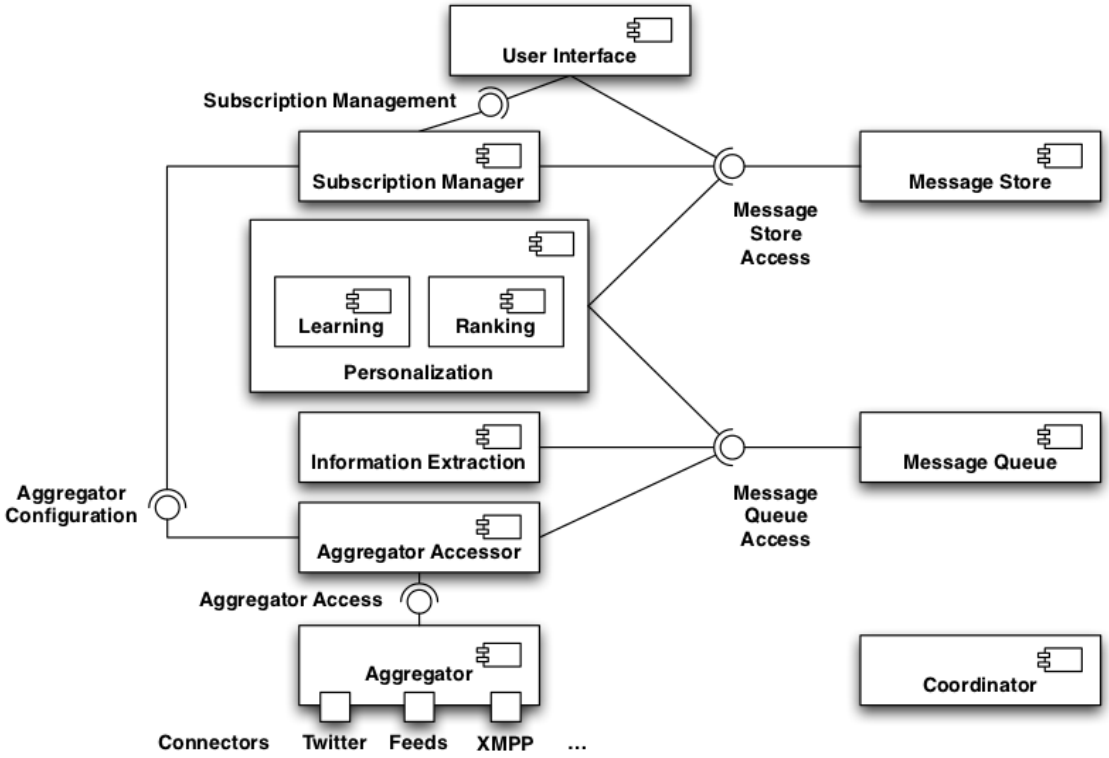

Abbildung 3.1: Architektur des PRISMA-Systems

3.2 Erlang

Erlang ist eine Programmiersprache die ab 1986 bei Ericsson entwickelt wurde. Dort wurden Experimente angestellt, um herauszufinden, wie sich große Telefonanlagen gut programmieren lassen, was schließlich zur Entwicklung von Erlang führte, siehe dazu [doe97].

Eine zentrale Anforderung an Telefonanlagen ist die ständige Verfügbarkeit. Um Sie erfüllen zu können, sind in Erlang verschiedene Konzepte enthalten, die es nur selten in anderen Programmiersprachen gibt. Ein idiomatisches Erlang-Programm ist unter Nutzung dieser Konzepte automatisch ausfallsicher und skalierbar.

Der Prozess ist in Erlang ein zentrales Sprachelement, für das es auch syntaktische Unterstützung gibt. Anstatt wie in anderen Sprachen native Threads oder Betriebssystems-Prozesse als Kapselung für Nebenläufigkeit zu benutzen, bedient in Erlang ein Threadpool (Scheduler) leichtgewichtige Arbeiter (Erlang-Prozesse, im Folgenden auch nur “Prozess” genannt). Dieses Konzept ist sehr viel effizienter als die beiden anderen Ansätze, ein Erlang Prozess verbraucht z.B. nur 512 Byte Speicher während ein Java Thread 512 KByte Speicher verbraucht.

Das Versenden von Nachrichten zwischen Prozessen ist ebenfalls sehr effizient implementiert. Ob der empfangende Prozess für eine Nachricht auf der lokalen Node läuft oder auf einer über das Netzwerk erreichbaren, macht in der Syntax von Erlang keinen Unter-

schied, wodurch sich ein Programm leicht auf mehrere Computer verteilen lässt.

Die nebenläufige Programmierung wird neben dem Prozess-Konzept von Erlang auch durch das Weglassen von veränderbaren Variablen unterstützt. Wenn Variablen unveränderbar sind, bietet dies den Vorteil, dass eine Funktion keine Seiteneffekte haben und damit auch keine Ressourcen verändern kann, die von anderen Prozessen genutzt werden. Natürlich gibt es trotzdem Möglichkeiten, Seiteneffekte auszulösen, etwa durch das Senden einer Nachricht, aber im Gegenteil zu anderen Programmiersprachen ist dies im Sourcecode direkt zu erkennen und eine Ausnahme, nicht die Regel.

Um Ausfallsicherheit garantieren zu können, gibt es, neben der aus anderen Programmiersprachen bekannten Fehlerbehandlung mit “throw”, “try” und “catch”, auch die Möglichkeit, dass Prozesse von anderen Prozessen überwacht werden.

Beliebt ist Erlang aufgrund der genannten Eigenschaften vor allem für die Programmierung von Systemen, die mit sehr vielen Verbindungen umgehen und dabei skalierbar sein müssen. Die bekanntesten Beispiele sind Ejabberd und Couchdb, die beide in der Implementierung des Aggregators in dieser Arbeit eingebunden sind, ausserdem gibt es noch die Webserver Yaws und Mochiweb.

3.3 OTP

Die OTP (Open Telecom Platform) ist ein Teil von Erlangs Standardbibliothek und des Laufzeitsystems. Unter Nutzung von supervisor-Trees (Bäume von Prozessen, bei denen jeder Knoten seine Blätter überwacht), ist es mit OTP möglich, Systeme so zu strukturieren, dass Teile des Systems ausfallen können (z.B. durch den Ausfall einer Node im Cluster) ohne die Verfügbarkeit des Gesamtsystems zu beeinträchtigen.

Die Einbindung eigener Komponenten in ein OTP-System erfolgt durch die Implementierung von Behaviors. Diese haben Ähnlichkeiten mit einer Java-Klasse die noch abstrakte Methoden hat. Es muss also ein Teil der Funktionalität implementiert werden, während ein anderer Teil schon von der Behavior bereitgestellt wird. Dazu kann die Behavior auch Funktionen bereitstellen, die beim Implementieren der “abstrakten” Methoden helfen.

In der OTP sind verschiedene Behaviors enthalten, im Aggregator wird vor allem Gen-Server verwendet. Ein Gen-Server ist eine Abstraktion für einen allgemeinen Erlang-Server, dieser empfängt und sendet Erlang-Messages. Im Vergleich zu einem normalen Erlang-Prozess ist ein Gen-Server stärker integriert: er kann in Supervisor-Trees eingebunden werden, hat Vorkehrungen zum Einspielen neuer Versionen des Programms zur Laufzeit und bietet Unterstützung zum asynchronen Antworten auf Anfragen und mehr.

Jede Erlang-Laufzeitumgebung hat einen Cookie, der als gemeinsames Geheimnis für ein Cluster benutzt werden kann, d.h. alle Erlang-Instanzen, die sich über das Netzwerk sehen können und den gleichen Cookie haben, befinden sich in einem gemeinsamen Cluster.

3.4 Mnesia

Mnesia ist eine Datenbank, die in Erlang/OTP enthalten ist. Durch die starke Verknüpfung von Mnesia und Erlang ist sie sehr einfach aus Erlang zu benutzen.

Sie unterliegt jedoch einigen Beschränkungen, die sie für die Verwendung in einigen Situationen, in denen konventionelle Rdbm Systeme Verwendung finden, disqualifiziert.

Eine Tabelle kann z.B. höchstens 2GB groß werden und durch locking über mehrere Maschinen hinweg (bei Schreibzugriffen) kann die Performanz der Mnesia-Datenbank, im Vergleich zu anderen Datenbanken, abfallen.

Die Stärken der Mnesia-Datenbank sind Ausfallsicherheit und die Möglichkeit, das System zur Laufzeit der (verteilten) Anwendung umzukonfigurieren [erldoc:mnesia].

3.5 Ejabberd

Die einzelnen Komponenten des PRISMA-Systems sind über XMPP (Extensible Messaging and Presence Protocol) miteinander verbunden, bei XMPP wird XML als Format für die Nachrichten benutzt. Unterschiedliche Komponenten eines verteilten Systems über XMPP miteinander zu verbinden ist, aufgrund der guten Unterstützung für XML, in den meisten Programmiersprachen sinnvoll.

Die Anbindung des Aggregators wird durch ein Modul für den XMPP-Server Ejabberd umgesetzt.

Ejabberd ist genauso wie der Aggregator in Erlang geschrieben und gilt als sehr effiziente und gut skalierbare XMPP-Server-Implementierung. Aufgrund der guten Erweiterbarkeit durch Module kann der Ejabberd als Grundlage für verteilte Systeme genutzt werden.

Dieser Ansatz wird bei der Implementierung des Aggregators gewählt, neben dem Senden und Empfangen von XMPP-Nachrichten benutzt der Aggregator den XML-Parser und- Generator, der im Ejabberd enthalten ist. Der Parser ist als C-Programm umgesetzt und daher effizienter als andere XML-Parser-Bibliotheken, die in Erlang verfügbar sind.

3.6 Couchdb

Jede Nachricht, die der Aggregator an den Aggregator Accessor weiterleitet, wird zusätzlich in einer Datenbank abgespeichert. Aufgrund von Erfahrungen in anderen Projekten wird hier Couchdb eingesetzt.

Die Couchdb ist ein Key-Value-Store, der in Erlang geschrieben ist. Neben dem einfachen Abspeichern und Auslesen von Key-Value-Paaren, kann die Couchdb mit Hilfe von sogenannten “views” auch Queries umsetzen. Views werden mit Hilfe von Map-Reduce aufgebaut, was schon beim Einfügen der Dokumente geschehen kann und zu Queries führt, die nur noch ein einfacher Tabellen-Lookup sind.

Die Couchdb bietet kein natives Erlang-Interface an, sondern muss, wie in jeder anderen Programmiersprache, über HTTP angesprochen werden.

Skalierbarkeit erreicht die Couchdb durch zwei Techniken. Zum einen können Dokumente anhand ihrer Id im Sinne eines konsistenten Hash-Rings auf unterschiedliche Instanzen verteilt werden. Zum anderen kann die Couchdb Instanzen der gleichen Datenbank auf unterschiedlichen Rechnern synchronisieren und so für Redundanz sorgen.

Aufgrund der Beschränkung auf eine HTTP-Rest-API kann die Couchdb bereits vorhandene Infrastruktur wie Loadbalancer und Caches zur effizienten Skalierung nutzen.

3.7 YAWS

YAWS (Yet Another Web Server) ist ein in Erlang geschriebener Webserver.

Er ist minimalistisch, bietet das Konzept der Serverpages, kann aber auch, über sogenannte Module, frei programmiert werden. YAWS ist einer der performantesten Webserver, die es gibt [webserver-comparison].

3.8 Unterschiedliche Informationsflüsse

3.8.1 RSS

RSS (Real Simple Syndication) ist ein Format, das im Web zur Anwendung kommt, um Neuerungen auf Webseiten, Nachrichten oder andere aktuelle Informationen zu verbreiten.

Der RSS-Stream ist auf einem HTTP-Server hinterlegt, im Gegenteil zu einer Webseite besteht er jedoch nicht aus HTML, sondern aus XML. Auf Nutzerseite kann eine Vielzahl sogenannter Feed-Reader zum Abrufen von RSS-Streams genutzt werden. Feed-Reader gibt es in jeder erdenklichen Form, etwa als Webseite (Saas) oder Smartphone-Application. Wenn ein Feed-Reader dazu in der Lage ist, unterschiedliche Nachrichtenströme abzurufen, wird er auch Aggregator genannt.

Durch regelmässiges Abrufen des HTTP-Servers (pollen) auf dem der RSS-Stream liegt, kann ein Feed-Reader zeitnah neue Nachrichten aus dem Stream erhalten.

Im Laufe der Zeit haben sich unterschiedliche Versionen von RSS entwickelt. Diese unterscheiden sich in der Struktur des generierten XML-Codes, wobei neuere Versionen nicht unbedingt abwärtskompatibel sein müssen [wiki:rss].

Da RSS über HTTP verteilt wird und der Betreiber eines Streams ihn in der Regel der Öffentlichkeit zugänglich machen will, ist eine Authentisierung zum Abrufen des Streams meistens nicht nötig. Technisch sind aber alle Authentisierungsmethoden, die auf Webseiten Verwendung finden, denkbar.

RSS-Streams bieten die Möglichkeit, über eine Rest-API auch neuen Inhalt oder Veränderungen an den Stream zu schicken, in diesem Fall ist es nötig, ein Authentifizierungsverfahren zu benutzen.

3.8.2 Atom

Atom (Atom Syndication Format) gilt als der Nachfolger von RSS. In Atom sollen sich die Vorteile von verschiedenen RSS-Versionen vereinen.

Bis auf die Struktur des XML gleicht ein Atom- einem RSS-Feed. Entscheidender Unterschied ist die Angabe des Typs bei inhaltstragenden Elementen, wodurch ein Feed-Reader z.B. einfacher erkennen kann, ob der Inhalt eines Eintrags HTML oder einfacher Text ist [wiki:atom] und ihn dementsprechend darstellen kann.

3.8.3 XMPP

Ursprünglich ist XMPP ein Protokoll für Instant Messaging, das früher unter der Bezeichnung Jabber bekannt war [wiki:xmpp].

Über den im Aggregator genutzten XMPP-Server Ejabberd gibt es bereits eine Schnittstelle, durch die XMPP-Nachrichten an den Aggregator gesendet werden können.

Im Gegenteil zu RSS und ATOM muss XMPP nicht gepollt werden, die neuen Nachrichten werden vom Client selbstständig an den Aggregator gesendet. Dadurch ergibt sich ein geringerer Ressourcenverbrauch für XMPP-Abonnements.

4 Architektur des Aggregators

In diesem Kapitel wird die Entwicklung der Aggregator-Komponente in Konzeption und Implementierung beschrieben. Dabei sollen die Anforderungen aus Kapitel 2 erfüllt werden.

4.1 Konzeption

Die Hauptaufgabe des Aggregators, das Abrufen von Nachrichtenströmen, kann auf zwei unterschiedliche Arten umgesetzt werden.

Zum einen unter Nutzung von synchronem I/O und vielen Threads, wobei jeder Thread eine Netzwerkverbindung verwaltet.

Zum anderen unter Nutzung von asynchronem I/O und einem Threadpool, in dem Arbeiter-Prozesse die Callback-Methoden für den I/O abarbeiten. Die asynchrone Variante kann auch einen Thread für jede Verbindung benutzen, doch durch die Nutzung eines Threadpools gibt es insgesamt weniger Context-Switches im Programmablauf, welche sehr "teuer" sind und damit die zweite Variante effizienter machen.

Anstatt Bibliotheken zu benutzen oder den "Scheduler" selber zu schreiben, wird hier die Programmiersprache Erlang benutzt, deren Nebenläufigkeitsmodell einem solchen Scheduler mit kleinen Arbeiter-Prozessen und asynchronem I/O entspricht.

Um die API des PRISMA-Systems umzusetzen, muss der Aggregator viele XMPP-Nachrichten verschicken, dies wird durch die Nutzung eines XMPP-Servers erreicht.

Wichtig ist dabei, dass der Aggregator ein XMPP-Server ist und kein Client, denn die Konzeption als XMPP-Client ist weniger effizient. Das XMPP-Protokoll ist geteilt in einen C2S (Client to Server) und einen S2S (Server to Server) Teil, wobei das effiziente Verschicken und Empfangen von vielen Nachrichten nur über das S2S Protokoll möglich ist. Der Grund dafür ist, dass ein XMPP-Server für S2S-Verbindungen einen weniger komplexen Protokoll-Stack hat. Dinge wie bekannte Kontakte, in Abwesenheit erhaltene Nachrichten und ähnliches sind für S2S-Verbindungen nicht vorgesehen [jabber-draft]. Um das S2S-Protokoll nutzen zu können, ist die Eingliederung des Aggregators in einen XMPP-Server notwendig.

Der XMPP-Server Ejabberd bietet diese Möglichkeit und ist in Erlang geschrieben, er gilt als sehr effizient [xmpp-scalability] und kommt daher hier zur Verwendung.

Die grundlegende Architektur des Aggregators ist in Abbildung 4.1 dargestellt. Die eckigen Elemente stellen Erlang-Applications dar, während die runden Elemente Gen-Server darstellen.

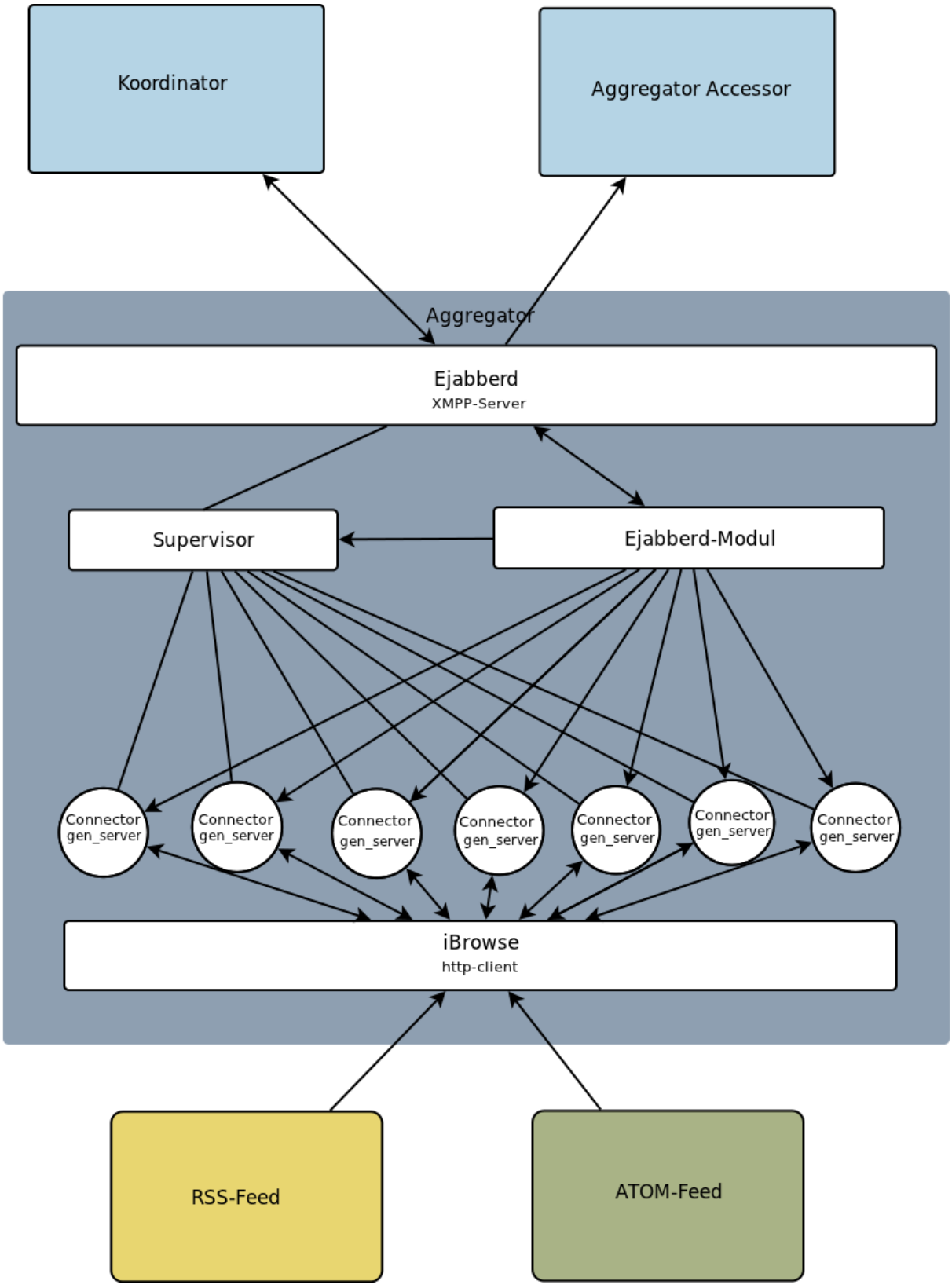

Abbildung 4.1: Architektur des Aggregators

Innerhalb des Aggregators wird für jedes Abonnement ein Erlang-Prozess gestartet (Connector), der sich anschliessend um das Aggregieren der Nachrichten aus dem zugewiesenen Abonnement kümmert.

Abonnements, die push-basierte Protokolle verwenden, brauchen keinen Connector pro Abonnement, da diese ihre neuen Nachrichten selbstständig an den Aggregator schicken und nicht gepollt werden müssen. Eine Interpretation und Transformation der Nachrichten in das intern (im PRISMA-System) genutzte Dokumentformat ist aber auch hier nötig. Im Fall von XMPP-Nachrichten kann der Ejabberd-Server diese Nachrichten empfangen. Bei Nachrichtenströmen, die andere push-basierte Protokolle verwenden, muss zumindest ein Prozess für alle Abonnements dieser Protokollart Nachrichten empfangen und weiterverarbeiten können.

Wenn die Menge der Nachrichten, die über eines dieser Protokolle in das System gelangt, groß genug ist, kann es auch notwendig werden, einen Prozess-Pool für den Empfang einer Protokollart zu benutzen. Im Extremfall (bei der Nutzung eines Prozesses pro Abonnement), funktioniert der Empfang push-basierter Protokolle genauso wie der pull-basierter.

Das Abrufen der pull-basierten Nachrichtenströme erfolgt über das HTTP-Protokoll. Andere Nachrichtenströme müssen eventuell andere Protokolle verwenden, aber die beiden pull-basierten Nachrichtenströme, die hier abgerufen werden (RSS und ATOM), benötigen nur HTTP.

Nach dem Abrufen eines Nachrichtenstroms wählt der entsprechende Connector die neuen Nachrichten aus und schickt diese an den Aggregator-Accessor, der diesem Abonnement zugeordnet ist, weiter.

Zusätzlich werden alle neuen Nachrichten in einer Datenbank persistiert. Da diese zu groß für eine Mnesia-Tabelle werden kann, muss hier auf eine externe Datenbank zurückgegriffen werden. Die Couchdb ist dafür gut geeignet, da sie effizient arbeitet und über das HTTP-Protokoll eine Schnittstelle anbietet, über die auch andere Komponenten des PRISMA-Systems, die nicht in Erlang geschrieben sind, Zugriff auf die persistierten Dokumente erhalten können. Nach Systemabstürzen könnten so z.B. Dokumente, die sich gerade in Bearbeitung befanden als das System abstürzte, wieder neu eingespielt werden.

Die Konfiguration des Aggregators (z.B. die Daten, die zum Abrufen von Abonnements benötigt werden) wird in der Mnesia-Datenbank gespeichert. Diese ist Bestandteil von Erlang und wird auch vom Ejabberd verwendet.

Wie bereits in den Grundlagen erwähnt, verwendet Erlang (und OTP) supervisor-Trees, um die Stabilität von Programmen sicherzustellen. Um dieses Konzept zu nutzen, wird ein eigener Supervisor in den Top-Level-Supervisor des Ejabberd eingehängt. Dieser überwacht die Connectoren und startet sie im Fehlerfall neu. Die Art von Supervisor ist hier "simple-one-to-one", dieser Supervisor kann nur einen Typ von Child-Elementen überwachen - hier Connectoren.

Andere Systemkomponenten, deren Ausfallsicherheit gewährleistet werden muss, werden direkt in den Ejabberd-Top-Level-Supervisor eingehängt.

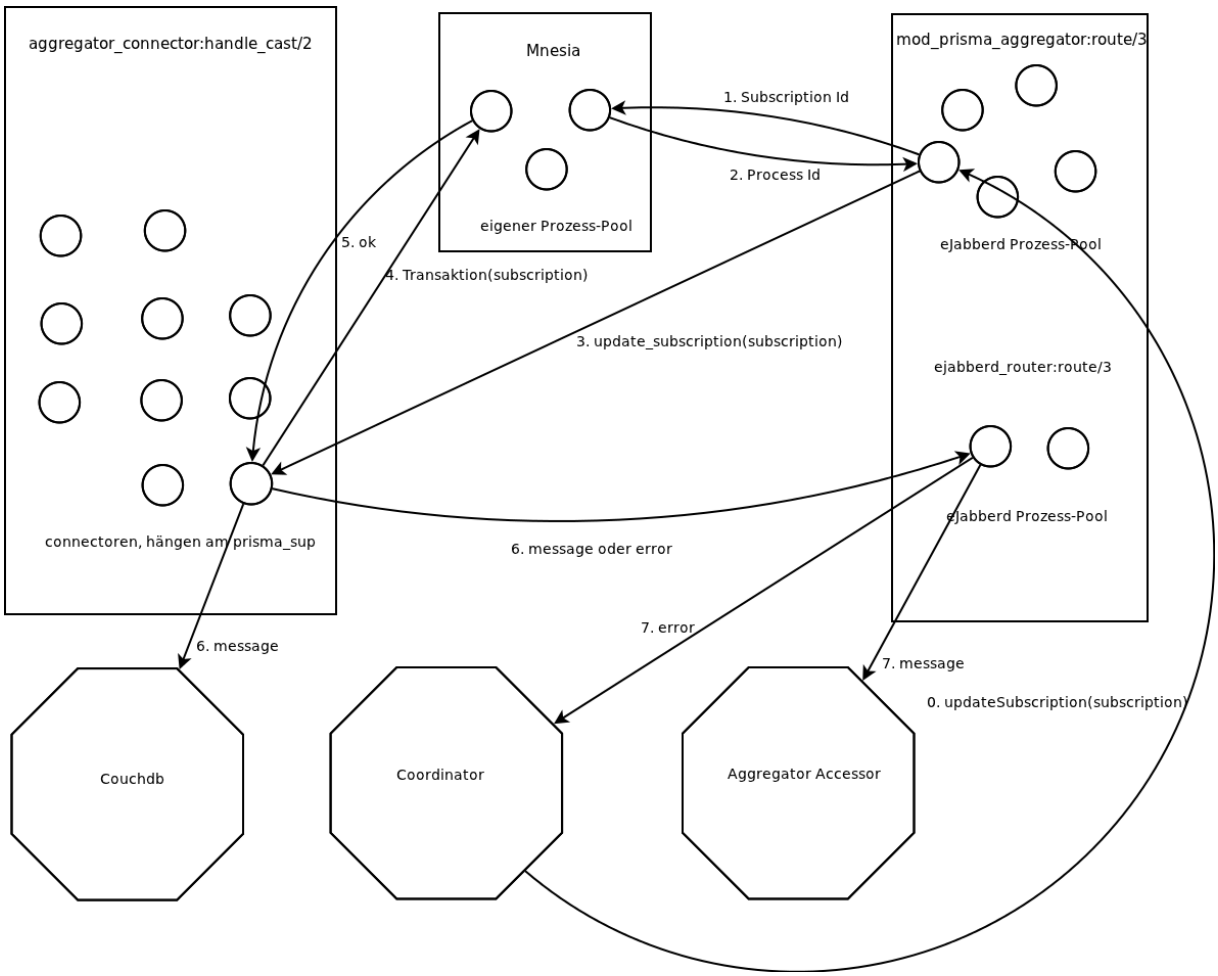

Abbildung 4.2: Ablauf eines updateSubscription-RPCs

Die Coordinator-Komponente des PRISMA-Systems ist unter anderem für die Skalierung der einzelnen Komponenten von PRISMA zuständig. Um dieser Aufgabe nachgehen zu können, benötigen sie Informationen über die Auslastung der entsprechenden Komponenten zum aktuellen Zeitpunkt.

Diese wird ihm über eine XMPP-Nachricht von den Komponenten mitgeteilt. Für diese Aufgabe wird ein gen_server benötigt, der regelmässig auslastungs-relevante Informationen sammelt und (bei Überlastung) an den Coordinator sendet.

Um die in Kapitel 5 konzipierten Testreihen auswerten zu können, werden Informationen aus dem laufenden System benötigt. Diese werden vom gleichen gen_server gesammelt. Interessante Informationen sind z.B. CPU- und Speicherauslastung sowie Anzahl der abgerufenen Nachrichtenströme und Erlang-spezifische Laufzeitparameter.

Die internen Schnittstellen des Aggregators werden durch die Nutzung von OTP, insbesondere des gen_servers weitgehend vorbestimmt. Idiomatisch definiert jeder gen_server seine eigene API, die widerum von anderen Komponenten aufgerufen wird. In Tabelle 4.1 ist die API des Connectors aufgeführt, die Namen der API-Funktionen entsprechen größtenteils den im Inneren benutzten Signalen des gen_servers.

In Abbildung 4.2 ist die Abarbeitung eines updateSubscription-RPCs im Aggregator abgebildet. Die rechteckigen Kästen stellen Prozess-Pools dar, die 8-eckigen-Elemente sind außerhalb des Aggregators und die runden Elemente sind Prozesse.

Die Schritte 0 bis 5 beschreiben das Aktualisieren eines Abonnements, wärend die Schritte 4 bis 7 nach Erhalt von neuen Nachrichten auf einem Nachrichtenstrom ausgeführt werden.

Bei Erhalt einer neuen Nachricht ist das Schreiben in die Datenbank in den Schritten 4 und 5 notwendig, um zu speichern, welche Nachricht die aktuellste auf einem Nachrichtenstrom ist.

4.1.1 Datenstrukturen

Die Informationen, die ein Abonnement repräsentieren, werden im Aggregator in einer Datenstruktur mit dem Namen `subscription` verwaltet. Diese Datenstruktur sollte sowohl in der Mnesia-Datenbank persistiert Systemausfälle überleben als auch dazu geeignet sein, über das Netzwerk auf einen anderen Aggregator verschoben zu werden.

Der Coordinator kann so neue Aggregatoren hinzufügen und die Last zwischen den Aggregatoren verteilen.

4.1.2 Filter

In dem Format für die Abonnements sind Filter, die im Aggregator zur Anwendung kommen, vorgesehen. Diese können nicht, wie in anderen Programmiersprachen, in die Objektstruktur übersetzt werden, aus der sie entstanden sind, sondern müssen in Erlang interpretiert werden. Zu diesem Zweck werden sie zunächst in ein Zwischenformat übersetzt, dass in einer Subscription und damit in der Mnesia-Tabelle, gespeichert werden kann. Mit Hilfe dieses Zwischenformates kann der Connector beim Erhalt neuer Nachrichten überprüfen, ob Nachrichten an den Accessor weitergeleitet werden oder nicht.

4.2 Implementierung

Der Sourcecode für das Ejabberd-Modul liegt in der datei `mod_prisma_aggregator.erl`, um den Aggregator benutzen zu können muss das Kompilat der Datei, `mod_prisma_aggregator.beam`, in das ebin-Verzeichnis des Ejabberd kopiert werden.

Ejabberd wird in der Datei /etc/ejabberd/ejabberd.cfg konfiguriert, hier muss das Modul eingetragen werden. Konfigurationsparameter für das Modul können hier übergeben werden, ein Beispiel ist in Quellcodebeispiel 4.1 angeführt.

Quellcodebeispiel 4.1: Beispielkonfiguration für mod_prisma_aggregator

```
{mod_prisma_aggregator, [
    {accessor, "ole@rixmann"},
```

```
{coordinator, "aggregatortester.rixmann"},
{debugLvl, 10},
{polltime, 6000}]}
```

Alle anderen Kompilate und Bibliotheken müssen in dem Bibliothekspfad des Betriebssystems, der für Erlang vorgesehen ist, abgelegt werden. Zur Vereinfachung des Kompilierens und der Installation befindet sich im Projekt eine make-Datei. Diese ist allerdings nur ein einfaches Shell-Script, keine "richtige" make-Datei. Sie funktioniert nur in Systemen, bei denen das Ejabberd-Bibliotheks-Verzeichnis in /usr/lib/ejabberd liegt.

Es kann zu Fehlern bei der Programmausführung kommen, wenn Systemparameter falsch eingestellt sind, die Standardeinstellungen des Betriebssystems müssen zumeist angepasst werden. Die Anzahl der Datei-Deskriptoren für den Ejabberd-Prozess sollte deutlich über 1024 liegen (Standardeinstellung unter Ubuntu-Linux), da ansonsten die iBrowse-Bibliothek diese Ressource auslastet. Wenn keine Datei mehr geöffnet werden kann (alle Datei-Deskriptoren sind aufgebraucht), beendet sich der Ejabberd bei dem Versuch, die Mnesia-Datenbank auf der Festplatte zu aktualisieren.

In der Datei "/etc/default/ejabberd" können Parameter der Erlang-Laufzeitumgebung des Ejabberd konfiguriert werden. Falls in dem Aggregator mehr als ca. 1400 Subscriptions verarbeitet werden sollen, ist es notwendig, die maximale Anzahl an Ets-Tabellen hochzusetzen. Jede Instanz des Connectors benutzt eine Ets-Tabelle, um Callbacks für asynchronen I/O zwischenzuspeichern.

Ein Ejabberd-Modul muss eine Behavior implementieren ("`gen_mod`"). Bei den zu implementierenden Methoden handelt es sich um start und stop. Durch Einhängen einer neuen Komponente in der start-Methode, die eine Subdomain des Servers verwaltet (mit Hilfe der Methode `ejabberd_router:register_route`), bekommt das Modul die Möglichkeit, XMPP-Nachrichten effizient zu empfangen und zu senden. Dazu wird eine Methode vom Modul bereitgestellt, die hier die Bezeichnung "route" trägt.

4.2.1 Interpretieren und Generieren von Nachrichten im PRISMA-System

In der route-Methode ankommende XMPP-Nachrichten werden in der XML-Darstellung des Ejabberd-eigenen XML-Parser bereitgestellt, ein Beispiel dafür ist in Quellcodebeispiel 4.2 zu sehen.

Quellcodebeispiel 4.2: Erlang-XML

```
{xmlelement, "message",
  [{"type", <<"type">>},
    {"from", <<"koordinator@host1">>)},
    {"to", <<"aggregator@host2">>}],
  [{xmlelement, "body", [],
      [{xmlcdata, <<"Inhalt der Nachricht">>}]}]}
```

Der vom Ejabberd bereitgestellte XML-Parser basiert auf der Expat-Bibliothek (in C geschrieben) und ist nach unseren Recherchen der effizienteste XML-Parser, der in Erlang verfügbar ist.

Da der Ejabberd mehrere Prozesse benutzt, um Nachrichten zu verarbeiten (die route-Methode wird aus unterschiedlichen Prozessen aufgerufen), können in der route-Methode Nachrichten verarbeitet werden, ohne dass ein Flaschenhals an dieser Stelle entsteht.

Der Inhalt des Body-Elements eingehender Nachrichten, die von anderen Komponenten des PRISMA-Systems kommen (diese sollten beim Aggregator immer vom Koordinator stammen), ist in JSON kodiert.

Die Dekodierung erfolgt in der route-Methode unter Nutzung der `json_eep`-Bibliothek.

Ähnlich wie die Expat-Bibliothek benutzt auch die `json_eep`-Bibliothek die Erlang-Datenstrukturen Tupel und Liste zur Abbildung von JSON in Erlang. Dadurch können die spracheigenen Mittel zur Manipulation und Darstellung von JSON benutzt werden.

In Java ist dies nicht möglich, da Java keine Sprachunterstützung für Datentypen bietet, die mächtig genug sind, um JSON strukturäquivalent abzubilden. Die Abbildung von Java-Objekten in JSON ist außerdem umständlich, da viel redundante Information enthalten ist. Ein Beispiel dafür ist Quellcodebeispiel 4.3, das abgebildete Objekt ist eine einfache Subscription.

Quellcodebeispiel 4.3: Erlang-JSON Subscription für tagesschau.de ATOM-Feed

```
{[{<<"class">>,<<"de.prisma.datamodel.subscription.Subscription">>},
  {<<"filterSpecification">>,null},
  {<<"id">>,null},
  {<<"sourceSpecification">>,
   {[{<<"accessProtocol">>,
      {[{<<"accessParameters">>,
         [{[{<<"class">>,
             <<"de.prisma.datamodel.subscription.source.AccessParameter">>},
            {<<"id">>,null},
            {<<"parameterType">>,<<"feeduri">>},
            {<<"parameterValue">>,<<"http://tagesschau.de/xml/atom">>}]}]},
        {<<"authenticationData">>,null},
        {<<"class">>,
         <<"de.prisma.datamodel.subscription.source.AccessProtocol">>},
        {<<"id">>,null},
        {<<"protocolType">>,null}]}},
    {<<"class">>,
     <<"de.prisma.datamodel.subscription.source.SourceSpecification">>},
    {<<"id">>,null},
    {<<"sourceType">>,<<"ATOM">>}]}},
  {<<"subscriptionID">>,<<"tagesschau">>}]}
```

Nach der Dekodierung wird die Art des RPCs ermittelt (subscribe, unsubscribe, updateSubscription). Bei unsubscribe und updateSubscription muss noch ermittelt werden, welcher Connector für die entsprechende Subscription zuständig ist, woraufhin dieser die Daten zur Weiterverarbeitung erhält (dies geschieht durch einen Aufruf von `gen_server:cast`, wird aber hinter API-Methoden des Connectors verborgen, siehe Tabelle 4.1).

Die Verknüpfung zwischen einer Subscription und dem Prozess, der diese repräsentiert, findet mit Hilfe einer ETS-Tabelle statt: `subscription_process_table`.

4.2.2 Zufalls-Server

Beim Start des Aggregators und beim Hinzufügen von neuen Subscriptions wurde zuerst der naive Ansatz gewählt, alle Connectoren für die entsprechenden Abonnements gleichzeitig zu starten. Dies führte zu einer Überlastung der Mnesia-Datenbank, da jeder Connector zu Anfang seinen Eintrag aus der ?PST (`prisma_subscription_table`) liest und bei neuen Nachrichten auf einem Nachrichtenstrom die entsprechende Subscription gespeichert werden muss. Bei späterem Abrufen des Nachrichtenstroms muss in der Subscription stehen, welche Nachrichten bereits ins System eingespeist wurden, daher muss diese Information bei jedem Empfangen von neuen Nachrichten gespeichert werden.

Um zu verhindern, dass alle Connectoren gleichzeitig arbeiten, wird eine Verzögerung vor dem ersten Abrufen des Nachrichtenstroms eingebaut.

Erste Versuche, bei denen die benötigten Zufallszahlen (um einen zufälligen Offset für die erste Anfrage an den Nachrichtenstrom zu haben) in den Connectoren berechnet wurden, führten leider dazu, dass es Häufungspunkte im Pollingintervall gab, an denen sehr viele Connectoren ihre Anfragen starteten. Der Grund dafür ist die Abhängigkeit des verwendeten Zufallszahlengenerators von der Systemzeit, wodurch Connectoren, die am gleichen Zeitpunkt gestartet wurden (die Systemzeit ist diskret), den gleichen Offset berechneten - dies konnten bei 10000 Subscriptions bis zu 2000 Connectoren sein.

Durch einen Gen-Server (Quellcode: `prisma_random_generator.erl`), der Zufallszahlen ausgibt, konnte eine gleichmässige Verteilung über das Pollingintervall erreicht werden. Ein Flaschenhals entsteht dadurch nicht. Bei 10.000 gestarteten Connectoren konnte keine merkliche Verzögerung festgestellt werden, was zeigt, wie effizient Nachrichten zwischen Erlang-Prozessen ausgetauscht werden.

Beim prisma_random_generator wäre eine Tabelle, wie die für den Connector in Tabelle 4.1 abgebildete, sehr kurz, es gibt nur zwei API-Funktionen, `start_link` und `uniform`. Die Funktion `start_link` ist in jedem gen_server vorhanden, auch wenn sie eventuell eine andere Bezeichnung trägt. Sie ruft die Funktion gen_server:start_link auf und startet damit den gen_server. Die Funktion uniform ruft in dem Prozess des gen_servers die Funktion random:uniform auf und gibt das Ergebnis zurück, während die Nachrichten zum und vom gen_server verschickt werden, schläft der aufrufende Prozess (Verhalten bei einem Call, bei einem Cast gibt es keine Rückgabe und der aufrufende Prozess läuft

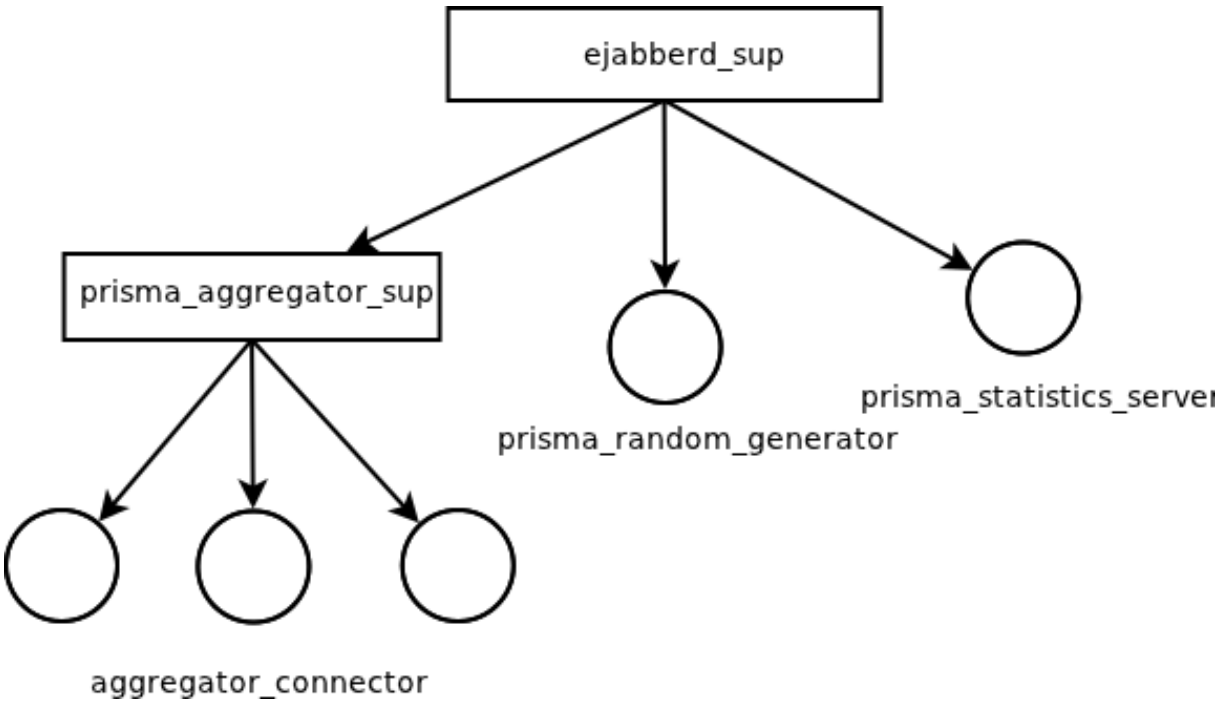

Abbildung 4.3: Vollständiger Supervisor-Tree des Aggregators

direkt weiter).

4.2.3 Supervisor-Tree

In Abbildung 4.3 ist der Supervisor-Tree des Aggregators vollständig abgebildet, die eckigen Elemente sind gen_supervisor, während die runden Elemente gen_server sind. Ein Pfeil bedeutet beobachtet/überwacht. Der Supervisor des Ejabberd läuft schon wenn das Modul geladen wird, in diesen werden alle weiteren Komponenten eingehängt. Der prisma_aggregator_sup hat als Kinder nur aggregator_connectoren. Er ist ein spezieller Supervisor, der darauf spezialisiert ist eine einzige Art von Kindern dynamisch zu verwalten.

4.2.4 Connector-API

Tabelle 4.1 vermittelt einen Überblick über die Nachrichten, die ein Connector interpretieren kann. Nachrichten werden entweder über gen_server:cast/call abgesetzt und entsprechend in handle_cast/call empfangen oder sie werden als normale Erlang-Messages verschickt und von handle_info interpretiert (handle_cast(Cast, State), handle_call (Call, State) und handle_info(Info, State) sind "abstrakte" Methoden der gen_server behavior).

4.2.5 HTTP-Client

Am Anfang der Entwicklung benutzten die Connectoren die Erlang-eigene HTTP-Client-Bibliothek (inets:httpc), um Nachrichtenströme abzurufen. Diese erwies sich jedoch als

wenig leistungsfähig. Nach unserer Recherche ist die iBrowse-Bibliothek für die Menge der Anfragen, die der Aggregator machen muss, geeignet [erlang-http-clients] und wird im Folgenden benutzt.

Die iBrowse-Bibliothek lässt sich sehr detailliert konfigurieren, z.B. lassen sich Einstellungen für die Verbindung zu jedem einzelnen Server festlegen (wie viele parallele Verbindungen, Länge der Warteschlangen für Anfragen zu diesem Server, eigene Verwaltung des Prozess-Pools usw.).

Der Aufwand, der in der Implementierung für die Nutzung der iBrowse-Bibliothek getrieben werden muss, ist jedoch größer als der, der für die Erlang-eigene HTTP-Client-Bibliothek benötigt wird. So kommt z.B. jede Antwort in Teilen (sog. chunks) an und muss im Connector zu einer kompletten Antwort zusammengebaut werden (in Tabelle 4.1 ibrowse_async_headers, ibrowse_async_response und ibrowse_async_end).

4.2.6 Statistik-Server

Zum Zwecke der Erfassung statistischer Daten gibt es im Aggregator einen weiteren gen_server, die Implementierung befindet sich in der Datei `prisma_statistics_server.erl`.

Dieser schreibt mit einem Intervall von 100 Millisekunden einige Informationen über die Systemauslastung in die Datei "/var/log/ejabberd/runtimestatistics.dat".

Die Connectoren schicken Nachrichten (API-Funktionen signal_httpc_overload und signal_httpc_ok) an den Statistik-Server, wenn die iBrowse-Bibliothek überlastet ist bzw. wenn Anfragen erfolgreich gestartet wurden.

Informationen, die vom Statistik-Server in runtimestatistics.dat geschrieben werden

- Laufzeit in zehntel Sekunden seit Start des Statistik-Servers
- Anzahl der Prozesse im Aggregator, in deren Mailbox neue Nachrichten auf Verarbeitung warten
- CPU-Auslastung, zu diesem Zeitpunkt (100 entspricht der vollen Auslastung eines Kerns)
- 1 falls iBrowse zu diesem Zeitpunkt überlastet ist, ansonsten 0
- Anzahl der Abonnements, die dieser Aggregator verwaltet
- Anzahl der erfolgreich verarbeiteten HTTP-Responses seit dem letzten Intervall, hochgerechnet auf eine Sekunde
- Menge des benutzten virtuellen Speichers, mit Hilfe des unix-Befehls "ps -p <PID> -o vmx=" erhalten

Informationen über den aktuellen Speicherverbrauch des Programmes sind in Erlang nicht leicht zu erhalten. Da Erlang Garbage-Collection auf Prozesslevel macht, gibt es keine zentrale Stelle, an der der aktuelle Speicherverbrauch bekannt ist [erlfaq:memory].

Mit der Funktion system_info(allocated_areas) lässt sich die Größe einiger statischer Konstrukte im Erlang-Laufzeitsystem abfragen, aber um den Speicherverbrauch der Prozesse zu erhalten, muss jeder Prozess einzeln gefragt werden - dieser Vorgang dauert einige Zeit und ist aufwändig (2*(Anzahl Prozesse im System) Nachrichten müssen versendet werden).

Aus diesem Grund wird als Speicherauslastung der Wert genommen, den das Betriebssystem für den Erlang-Prozess ermittelt, dieser ist jedoch als obere Schranke für die tatsächliche Auslastung zu interpretieren [erlfaq:memory].

Der prisma_statistics_server schickt im Falle einer Systemüberlastung Nachrichten an den Koordinator. Dieser hat die Möglichkeit, Abonnements vom überlasteten Aggregator zu einem Aggregator zu migrieren, der noch genügend freie Ressourcen hat.

Die genauen Vorraussetzungen, unter denen eine solche Überlastungsnachricht abgesetzt wird, werden in den Tests ermittelt. Sie sollten so gewählt werden, dass eine Überlastung unabhängig von der verwendeten Hardware erkannt wird.

Tabelle 4.1: Beschreibung der Funktionen des Connectors

Api-Funktion	Cast/Info	Beschreibung
new_subscription	init	new_subscription löst init über die Funktion add_child des Supervisors aus.
start_link	init	Wird vom Supervisor aufgerufen und benutzt gen_server:start_link, um den neuen Connector zu starten.
unsubscribe	unsubscribe	Löscht die Subscription des Connectors aus der ?PST und beendet den Connector.
update_ subscription	update_ subscription	Ändert die Subscription in der ?PST und im Zustand des Connectors.
stop	stop	Beendet den Connector, die Subscription bleibt aber in der ?PST. Bei einem Neustart des Aggregators startet der Connector wieder.
-	go_get_messages	Dieser Cast wird von einem Timer ausgelöst, die Polltime dieses gen_servers ist abgelaufen, es muss ein neuer Request abgesetzt werden.
-	ibrowse_async _headers	Info, die das Eintreffen eines HTTP-header signalisiert.
-	ibrowse_async _response	Info, die das Eintreffen eines HTTP-Bodyparts signalisiert.
-	ibrowse_async _end	Info, die das Ende eines HTTP-Requests signalisiert. Die empfangenen Nachrichten werden interpretiert und an den Accessor verschickt. Hier wird ein neuer Timer aufgesetzt, der nach Ablauf von Polltime go_get_messages an den Connector schickt.
collapse	collapse	Simuliert einen Fehler des Connectors.
stop_all_and_ delete_mnesia	stop	Löscht die Mnesia-Tabelle ?PST und castet anschliessend stop an alle Connectoren.
emigrate	emigrate	Beginnt diese Subscription auf einen anderen Aggregator zu verschieben. Stoppt den Connector, löscht aber den Eintrag in der ?PST nicht. Sendet immigrate an den anderen Aggregator.
immigrate	immigrate	Ruft new_subscription auf und sendet anschliessend unsubscribe an den Aggregator, von dem die Subscription immigriert.

5 Konzeption und Implementierung der Testreihen

5.1 Use case Analyse

In diesem Kapitel werden Anwendungsfälle für den Aggregator entwickelt. Sie versuchen performanz-kritische Situationen aus dem realen Betrieb des Aggregators nachzubilden. Gleichzeitig kann die korrekte Funktionsweise des Aggregators untersucht werden.

Im ersten Anwendungsfall, "Effizienz des Aggregators", wird mit Hilfe von zwei Szenarien versucht, eine obere und eine untere Schranke für die Effizienz des Aggregators zu finden.

Der zweite Anwendungsfall, "Auslasten und Skalieren", versucht möglichst realistisch den Normalbetrieb des Aggregators zu simulieren.

Ein Nachrichtenstrom heißt kontrolliert, wenn seine Eigenschaften im Anwendungsfall definiert werden. Um kontrollierte Nachrichtenströme zu verwenden, wird ein Programm benötigt, das diese gegenüber dem Aggregator bereitstellt. Dieses Programm wird in Kaptitel 5.3.2 beschrieben.

5.1.1 Effizienz des Aggregators

Die Effizienz des Aggregators wird hier definiert als die Menge an Nachrichtenströmen, die innerhalb einer Zeiteinheit auf neue Nachrichten überprüft werden können.

Die Anzahl der pull-basierten Nachrichtenströme, die ein Aggregator bewältigen kann, hängt neben der Leistungsfähigkeit des Computers, der Bandbreite der Internetanbindung und der Effizienz der Implementierung des Aggregators auch von dem Verhalten der einzelnen Nachrichtenströme ab.

Unterschiedliche Parameter, z.B. Anzahl der Nachrichten (die bei jedem Abrufen geschickt werden), Größe der einzelnen Nachrichten, Antwortzeit des Nachrichtenstroms beim Abrufen, haben Einfluss auf die Effizienz des Aggregators.

Bei push-basierten Nachrichtenströmen hängt die Effizienz des Gesamtsystems, da hier die Nachrichtenströme nicht regelmäßig abgerufen werden müssen, vor allem von dem Nachrichtenaufkommen ab.

Daher soll der Einfluss, den einzelne Parameter, die das Verhalten von Nachrichtenströmen bestimmen, auf die Effizienz des gesamten Aggregators haben, ermittelt werden. Zu diesem Zweck müssen die Nachrichtenströme kontrolliert werden.

Um Rückschlüsse auf die Effizienz des Aggregators bei Verwendung realistischer Nachrichtenströme ziehen zu können, werden hier eine obere und eine untere Schranke für die Effizienz des Aggregators gesucht. Es werden mehrere Fälle betrachtet, in denen jeweils die Änderung unterschiedlicher Parameter untersucht wird.

Hohes Nachrichtenaufkommen

In diesem Anwendungsfall wird überprüft, wie der Aggregator sich verhält, wenn sehr viele neue Nachrichten auf den aggregierten Nachrichtenströmen generiert werden.

Jede eingehende neue Nachricht löst eine entsprechende Nachricht vom Aggregator an seinen Accessor aus. Jeder Teil des Aggregators ist an der Verarbeitung solcher Nachrichten beteiligt. Daher ist dieser Anwendungsfall dazu geeignet, die Effizienz des Aggregators als Gesamtsystem zu untersuchen.

Da das Nachrichtenaufkommen zwischen Aggregator und Accessor maximiert wird, kann auch die Integrität und Verfügbarkeit der Kommunikationsverbindung mit dem Accessor untersucht werden.

Niedriges Nachrichtenaufkommen

Wenn die Nachrichtenströme statisch (wenn nie neue Nachrichten auf dem Nachrichtenstrom generiert werden) sind, werden nur beim ersten Abrufen Nachrichten vom Aggregator an den Accessor geschickt. Nach dieser Einlaufphase kommuniziert der Aggregator nur noch mit den Nachrichtenströmen und interpretiert deren (immer gleiche) Antworten. Da keine neuen Nachrichten mehr ankommen, sind auch keine Datenbankzugriffe mehr nötig, um die Wiedererkennungsmerkmale der neuen Nachrichten zu speichern.

Die Teile des Aggregators, die für das Abrufen von Nachrichtenströmen und für das Interpretieren der Antworten verantwortlich sind, sind in diesem Anwendungsfall isoliert und können daher besser untersucht werden als in dem Anwendungsfall 5.1.1.

Durch eine große Anzahl von Nachrichten oder sehr lange Nachrichten in den einzelnen Nachrichtenströmen kann untersucht werden, wie effizient der Aggregator die Antworten der Nachrichtenströme interpretiert.

Indem die Nachrichtenströme ihre Antworten gar nicht oder mit einer Verzögerung verschicken, kann untersucht werden, wie effizient und fehlertolerant die Kommunikationsverbindung zu den Nachrichtenströmen ist.

5.1.2 Auslasten und Skalieren

Dieser Anwendungsfall spiegelt den normalen Betrieb des Aggregators wieder. Daher sollten hier keine kontrollierten Nachrichtenströme zum Einsatz kommen, sondern Nachrichtenströme mit möglichst realistischen Eigenschaften.

In dem vollständigen PRISMA-System schickt ein Aggregator bei Überlastung eine Nachricht an den Coordinator, der daraufhin eine Migration eines Teils der Abonnements

des überlasteten Aggregators veranlasst.

Dem Aggregator werden zunächst solange neue Abonnements übergeben, bis dieser überlastet ist. Anschließend wird ein Teil der Abonnements auf einen zweiten, noch leeren, Aggregator migriert.

Mit Hilfe dieses Anwendungsfalls soll ermittelt werden, unter welchen Bedingungen der Aggregator so stark wie möglich ausgelastet und trotzdem noch in der Lage dazu ist, seine Abonnements fehlerfrei auf einen anderen Aggregator zu migrieren. Diese Bedingungen sollen Einfluss auf die Implementierung des Aggregators haben, um die Effizienz des PRISMA-Systems, bei Einsatz mehrerer Aggregatoren, zu verbessern.

Die Aggregatoren tauschen untereinander nur Nachrichten aus, wenn Abonnements migrieren. Integrität und Verfügbarkeit der Kommunikationsverbindung zwischen den Aggregatoren kann in diesem Anwendungsfall untersucht werden. Dazu muss überprüft werden, ob alle Migrationen erfolgreich waren und ob sich doppelte Abonnements auf den Aggregatoren befinden.

5.2 Spezifikation der Testreihen

Die hier beschriebenen Testreihen werden analog zu den Anwendungsfällen im letzten Kapitel konstruiert.

Aufgrund der höheren Belastung beim ersten Abrufen eines Nachrichtenstroms kommt es zu einer Überlastung der Mnesia-Datenbank, wenn zu viele Nachrichtenströme gleichzeitig vom Aggregator gestartet werden. Aus diesem Grund werden dem Aggregator neue Abonnements in 1000'er Paketen übergeben, zwischen deren Übergabe 120 Sekunden vergehen. 120 Sekunden entsprechen der doppelten Abrufzeit, nach deren Ablauf der Aggregator alle neuen Nachrichtenströme einmal verarbeitet hat und die Datenbank sollte wieder entlastet ist.

Das Programm, das gegenüber dem Aggregator den Coordinator und den Accessor simuliert, soll auf einem zweiten Computer laufen, da dies im PRISMA-System auch der Fall ist.

5.2.1 Effizienz des Aggregators

Alle in diesem Kapitel beschriebenen Testreihen benutzen kontrollierte Nachrichtenströme, deren Parameter in der jeweiligen Testreihe angegeben werden.

In jedem Test wird die Anzahl der Abonnements so lange erhöht, bis der Aggregator überlastet ist.

Um die Latenz, die durch den Netzwerkverkehr entsteht, zu minimieren, sollte das Programm, das gegenüber dem Aggregator die Nachrichtenströme simuliert, auf der selben Maschine laufen wie der Aggregator.

Hohes Nachrichtenaufkommen

In dieser Testreihe generiert jeder Nachrichtenstrom eine bestimmte Anzahl neuer Nachrichten pro Minute.

Die Anzahl neuer Nachrichten pro Minute wird in den Stufen 0, 1, 5 und 10 getestet. Jeder Nachrichtenstrom kann höchstens 15 Nachrichten beinhalten. Die Nachrichtenströme antworten ohne Verzögerung.

Niedriges Nachrichtenaufkommen

Alle Nachrichtenströme sind statisch, es gibt keine neuen Nachrichten.

Wenn im Test nicht anders aufgeführt, enthält jeder Nachrichtenstrom 15 Nachrichten und antwortet ohne Verzögerung.

[Verzögerung der Nachrichtenströme] In dieser Testreihe wird die Verzögerung in fünf-Sekunden-Schritten bis über die Zeit erhöht, an der der Aggregator einen Timeout erkennt und den Versuch, Nachrichten von diesem Nachrichtenstrom abzurufen, abbricht. Der Timeout des Aggregators ist für HTTP-Verbindungen bei 30 Sekunden. Die Tests erfolgen also mit 0, 5, 10, ..., 30 Sekunden Verzögerung.

Da die HTTP-Client-Bibliothek für jedes Server/Port-Paar maximal zehn Verbindungen gleichzeitig benutzt, müssen in dieser Testreihe mehrere Nachrichtenströme simuliert werden. Es werden 1500 Nachrichtenströme simuliert. Dies ermöglicht, bei der maximalen Verzögerung von 30 Sekunden das Abrufen von 30.000 Nachrichtenströmen pro Minute.

[Anzahl der Nachrichten] Durch Erhöhen der Anzahl an Nachrichten, die die Nachrichtenströme bei jedem Abrufen versenden, müssen mehr Daten durch das Netzwerk übertragen werden. Dadurch hat in dieser Testreihe der HTTP-Client größeren Einfluss als in anderen Testreihen.

Außerdem hängt die Menge an XML, die interpretiert werden muss, direkt von der Größe der Antwort des Nachrichtenstroms ab.

Die Menge an Nachrichten in jedem Strom wird in den Größen 1, 15 und 100 getestet.

[Anzahl der unterschiedlichen Nachrichtenströme] In dieser Testreihe wird die Anzahl der Server, von denen die Nachrichtenströme kommen, variiert. In den anderen Testreihen zur Effizienz des Aggregators werden, wenn nicht anders beschrieben, alle Abonnements vom gleichen Server abgerufen.

Die Anzahl der Server wird in den Größen 1, 500, 1000 und 1500 getestet.

[Effizienz auf unterschiedlicher Hardware] Das PRISMA-System ist mit dem Ziel der Skalierbarkeit konstruiert worden. In dieser Testreihe soll überprüft werden, wie die Abhängigkeit der Effizienz des Aggregators von der eingesetzten Hardware ist (d.h. wie gut der Aggregator auf einer Maschine “skaliert”).

Zu diesem Zweck wird das Testsetup "umgedreht". Die Programme, die sonst auf dem Testaggregator laufen, laufen auf dem Testsimulator und umgekehrt. Siehe dazu Kapitel 6.1.

5.2.2 Auslasten und Skalieren

In diesem Test sind die Nachrichtenströme unkontrolliert. Die Adressen der Nachrichtenströme werden einer Liste entnommen, die mit Hilfe eines Web-Crawlers erstellt wurde. Sie enthält sowohl RSS- als auch Atom-Ströme.

Da diese Nachrichtenströme zu ca. 15% fehlerhaft sind, in dem Sinne, dass sie entweder Netzwerkfehler liefern (kein DNS-Eintrag, Timeout, nimmt die Verbindung an, aber sendet keine Daten etc.) oder von dem Aggregator nicht richtig interpretiert werden, werden Nachrichtenströme, die eine Fehlermeldung an den Controller auslösen, aus dem Aggregator entfernt.

Zunächst wird der Aggregator, analog zu den Testreihen zur Effizienz des Aggregators, mit Nachrichtenströmen versorgt bis er überlastet ist. Aus den daraus entstandenen Messdaten soll ermittelt werden, unter welcher Bedingung der Aggregator noch Abonnements an andere Aggregatoren migrieren kann.

Anschließend wird die Hälfte der Abonnements auf einen zweiten Aggregator migriert. Dieser sollte zu Anfang keine Abonnements bearbeiten und die Kapazitäten haben, die Hälfte der Abonnements aufnehmen zu können.

5.3 Entwicklung der Testumgebung

Um Testreihen mit der hier entwickelten Implementierung des Aggregators durchführen zu können, werden Programme benötigt, die die den Aggregator umgebende Infrastruktur simulieren, Testreihen starten und Ergebnisse der Testreihen erfassen.

Zu dieser Infrastruktur gehören die Komponenten des PRISMA-Systems, die in unmittelbarem Kontakt zu dem Aggregator stehen. Der Coordinator ist eine solche Komponente, er versorgt den Aggregator mit neuen Abonnements und verschiebt diese gegebenenfalls auf andere Aggregatoren. Eine zweite Komponente ist der Aggregator-Accessor. Wenn der Aggregator neue Nachrichten erhält, werden diese an den Accessor zur Verarbeitung weitergereicht. Die Simulation von Coordinator und Accessor wird von dem gleichen Programm übernommen. Konstruktion und Implementierung dieses Programmes werden in Unterkapitel 5.3.1 beschrieben.

Nach außen kommuniziert der Aggregator mit unteschiedlichen Nachrichtenströmen. Diese werden ebenfalls von einem Programm gegenüber dem Aggregator simuliert. Da das Abrufen von pull-basierten Nachrichtenströmen im Vergleich zu push-basierten Nachrichtenströmen sehr "billig" ist, werden nur pull-basierte Nachrichtenströme simuliert. Konstruktion und Implementierung des Nachrichtenstrom-Simulators werden in Unterkapitel 5.3.2 beschrieben.

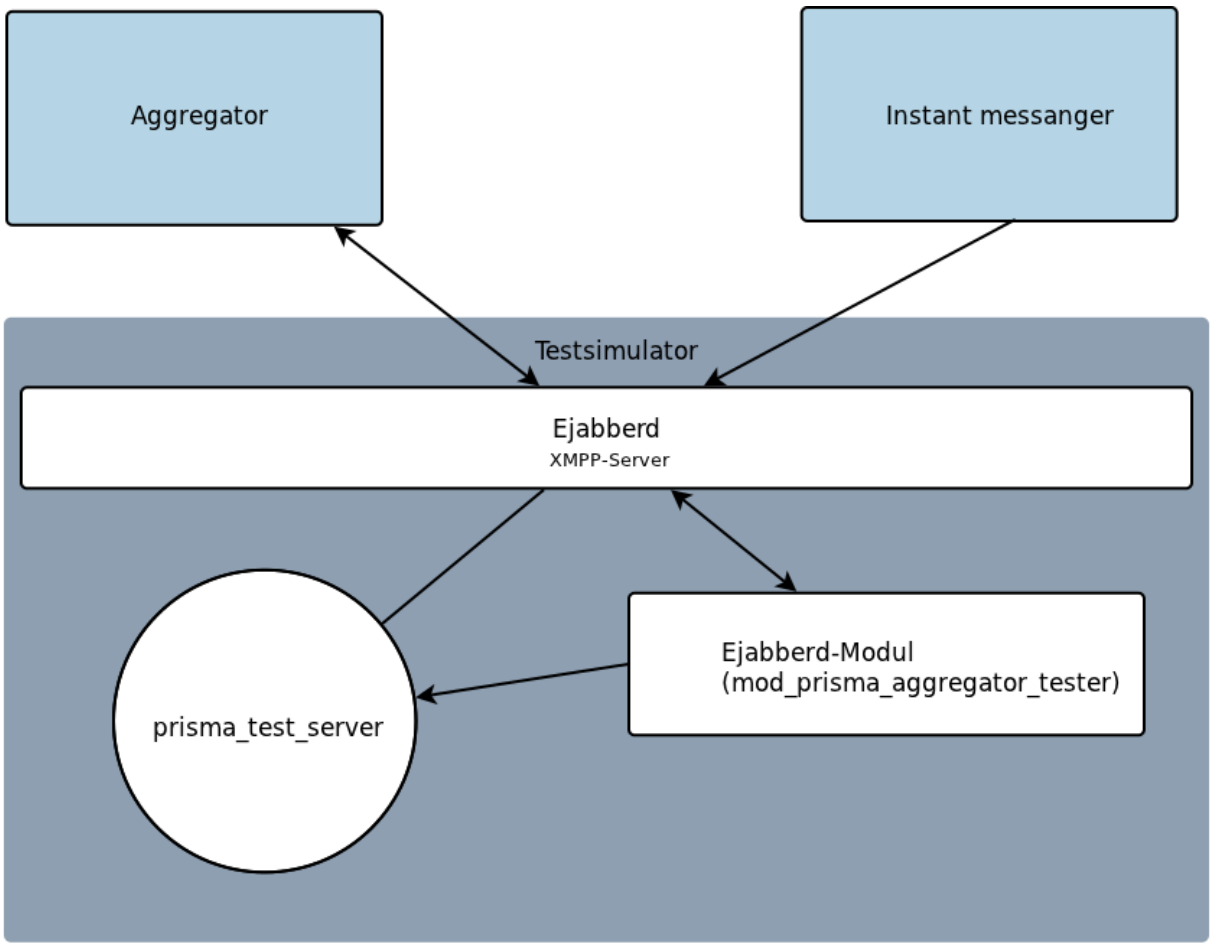

Abbildung 5.1: Architektur des mod_prisma_aggregator_tester

5.3.1 mod_prisma_aggregator_tester

Der mod_prisma_aggregator_tester simuliert Accessor und Coordinator gegenüber dem Aggregator. Da er über XMPP mit dem Aggregator kommuniziert, ist die Struktur des Programms ähnlich der des Aggregators.

Konstruktion

Um in der bereits bekannten Umgebung zu arbeiten und bereits geschriebenen Sourcecode wiederverwenden zu können, wird der Tester ähnlich dem Aggregator in Erlang als Modul des Ejabberd konstruiert.

In der Abbildung 5.1 wird der Aufbau des mod_prisma_aggregator_tester dargestellt.

Da der Test "Auslasten und Skalieren" von dem Coordinator nach dem Auslasten ein anderes Verhalten erwartet als vor dem Auslasten, muss dieser Zustandswechsel im Tester umgesetzt werden. Zu diesem Zweck wird ein gen_server benutzt, der prisma_test_server.

Eintreffende Nachrichten für den Accessor oder Coordinator werden vom Modul an den prisma_test_server weitergeleitet. Dieser schreibt statistische Daten (Anzahl der Empfangenen Nachrichten, Anzahl der empfangenen Fehlermeldungen) in eine Datei. Zusam-

men mit den Daten, die der prisma_statistics_server produziert, können so die Testreihen ausgewertet werden.

Der mod_prisma_aggregator_tester wird über XMPP-Nachrichten gesteuert. So kann ein einfacher XMPP-Client benutzt werden, um Tests zu starten und einzelne Funktionen des Coordinators durchzuführen.

In der Ejabberd-Konfigurationsdatei wird ein Aggregator angegeben, dieser wird bei einigen Befehlen als Ziel(-Aggregator) für den Befehl benutzt.

Die folgenden Befehle kann der mod_prisma_aggregator_tester interpretieren:

[test <Aggregator> <URL>] Startet einen Test, bei dem dem Aggregator unter der Adresse <Aggregator> alle 120 Sekunden 1000 Abonnements des Nachrichtenstroms unter der Addresse <URL> hinzugefügt werden.

[emigrate <Source> <Destination> <Id>] Emigriert die Subscription mit der Id <Id> vom Aggregator unter der Addresse <Source> zum Aggregator unter der Addresse <Destination>.

[subs_from_file <Start> <Count> <Accessor> <Batchname>] Sendet für jede Zeile von <Start> bis <Start> + <Count> aus `/usr/lib/ejabberd/testfeeds.txt` ein Abonnement an den Aggregator, wobei alle Abonnements ihre neuen Nachrichten an den Accessor unter <Accessor> weiterreichen. Die Id der jeweiligen Subscription wird aus <Batchname> + "-" + Zeilennummer zusammengesetzt.

[subs_from_file_bulk <Start> <Count> <Accessor> <Batchname>] Wie subs_from_file, sendet jedoch eine Bulk-Subscription mit allen Abonnements anstatt für jedes Abonnement eine einzelne Nachricht zu schicken.

[update_subscription <Url> <Accessor> <Feed> <Id>] Aktualisiert das Abonnement unter der Id <Id> auf dem Aggregator.

[unsubscribe_bulk <Name> <Start> <Stop>] Entfernt die Abonnements mit den Ids <Name> + "-" + Start bis <Name> + "-" + Stop vom Aggregator.

Implementierung

Der Tester wird als Ejabberd-Modul implementiert. Wie der mod_prisma_aggregator, registriert auch der Tester eine Subdomain am Ejabberd, um deren Verwaltung er sich kümmert.

Die Methode, in der eintreffende Nachrichten ankommen, ist wie beim Aggregator die "route". Hier wird der Inhalt der Nachrichten mit Hilfe der json_eep Bibliothek in eine Erlang-Term-Struktur überführt.

Nach der Interpretierung des Nachrichteninhalts wird entweder ein neuer Test gestartet oder die entsprechende Nachricht zur weiteren Auswertung bzw. statistischen Erfassung

an den “prisma_test_server” weitergeleitet. Erfasst werden die statistischen Daten in der Datei `/var/log/ejabberd/teststats.dat`.

Der prisma_test_server hat, wie der prisma_statistics_server ein Device (die offene Datei, in die die statistischen Daten geschrieben werden) in seinem Zustand. Das Schreiben der Daten wird, analog zum prisma_statistics_server, durch einen Timer, der einen gen_server:call auslöst, veranlasst. Die API des gen_servers ist in Tabelle 5.1 beschrieben.

Der Zustand des gen_servers wird in einem Erlang-Record geführt, das aus den Feldern test, error_count, message_count, device und walltime_init besteht. Walltime_init wird benötigt, um jeder Datenerfassung einen Zeitpunkt relativ zur Laufzeit des Testservers zuordnen zu können. In dem test-Feld wird eine Referenz auf den Timer geführt, der den Callback auf run_test ausführt. Durch diese Referenz kann der Timer und damit der entsprechende Test gestoppt werden.

Tabelle 5.1: Beschreibung der API des prisma_test_servers

Api-Funktion	Cast/Info	Beschreibung
start_link	init	Wird vom Supervisor aufgerufen, benutzt gen_server:start_link um den test-Server zu starten.
message_received	message_received	Wird aus der route des Moduls aufgerufen, bei neu eintreffender Nachricht - erhöht einen Zähler.
error_received	error_received	Wird aus der route des Moduls aufgerufen, bei neu eintreffendem Fehler - erhöht einen Zähler.
start_test	start_test	Startet einen neuen Test.
stop_test	stop_test	Stoppt einen laufenden Test.
-	run_test	Führt den nächsten Schritt in einem laufenden Test aus.
-	collect_stats	Schreibt eine neue Zeile mit statistischen Daten in die Datei.

5.3.2 stream_simulator

In diesem Kapitel wird die Konstruktion und Implementierung des Nachrichtenstrom-Simulators beschrieben.

Pull-basierte Nachrichtenströme brauchen nicht simuliert zu werden, da diese in den Testreihen nicht vorkommen. Unter den push-basierten beschränkt er sich auf die Simulation von Atom-Strömen. Da RSS- und Atom-Ströme sehr ähnlich sind (insbesondere was die Verarbeitung im Aggregator anbelangt), würde die zusätzliche Simulation von RSS-Strömen auf die Testergebnisse nur geringen Einfluss haben.

Konstruktion

Atom-Ströme werden über HTTP abgerufen, daher muss der Simulator ein HTTP-Server sein.

Da die Anzahl der offenen Verbindungen zu einem Server, die die HTTP-Client-Bibliothek des Aggregators gleichzeitig unterhalten kann, beschränkt ist, muss der stream_simulator in der Lage sein, auf mehreren Ports Anfragen zu beantworten.

Um den Simulator nicht bei jedem neuen Testlauf neu konfigurieren zu müssen, werden die speziellen Eigenschaften des Nachrichtenstroms bei jeder Anfrage in den Request-Parametern übergeben.

Diese Parameter sind: message_interval (Intervall, nach dem der Simulator eine neue Nachricht erzeugt, in Sekunden), resp_time (Verzögerung, bevor der HTTP-Server auf eine Anfrage antwortet, zur Simulation von Latenz im Netzwerk, in Millisekunden), max_entries (Anzahl der neuesten Nachrichten, die der Nachrichtenstrom höchstens enthält).

Implementierung

Die Implementierung benutzt den YAWS-Webserver. Dieser lässt sich in andere Erlang-Programme integrieren und ist in der Lage, in mehreren Instanzen auf mehreren Ports zu laufen.

YAWS benutzt das verbreitete Modell der Server-Pages, bei dem eine Webseite im HTML-Code eingebetteten Erlang-code enthält, der vor der Auslieferung ausgeführt wird und so den Inhalt dynamisch erzeugt.

Der stream_simulator interpretiert die Kommandozeilen-Parameter start_port, instances (Anzahl der Ports, ab dem start_port, auf denen Instanzen gestartet werden) und yaws_dir (Verzeichnis, in dem die Server-Pages liegen).

Der stream_simulator startet die YAWS-Instanzen und übergibt jeder Einzelnen den Zeitpunkt des Programmstarts. Bei einem eingehenden Request wird unter Zuhilfenahme der aktuellen Systemzeit berechnet, welche Einträge der Nachrichtenstrom haben muss, die Nachrichten werden generiert und der Strom ausgeliefert.

Falls der Parameter resp_time übergeben wurde, führt der Server ein `timer:sleep` aus, bevor er den Nachrichtenstrom ausliefert.

6 Durchführung und Ergebnisse der Testreihen

In diesem Kapitel wird zunächst der genaue Aufbau der Testumgebung beschrieben. Anschließend werden die Ergebnisse und deren statistische Weiterverarbeitung vorgestellt.

6.1 Beschreibung der Testumgebung

Der Aufbau der Testumgebung ist in Abbildung 6.1 dargestellt.

Es stehen zwei virtuelle Maschinen zum Testen des Aggregators zur Verfügung. Beide laufen auf dem selben Host-System.

Die CPU des Host-Systems ist ein Intel Xeon ES620 mit 2,4GHz Taktfrequenz. Die virtuellen Maschinen laufen auf einem Teil der Kerne des Host-Systems.

Das erste System "Testaggregator" hat vier Kerne und 16GB Arbeitsspeicher. Das zweite System "Testsimulator" hat zwei Kerne und 8GB Arbeitsspeicher.

Auf beiden Systemen läuft als Betriebssystem Ubuntu Linux 11.04 mit einer Couchdb in der Version 1.0.1 und einem Ejabberd in der Version 2.1.4.

Die Kernelparameter, die die Größe der Buffer zur Verwaltung von Netzwerkverbindungen bestimmen, wurden auf beiden Systemen verändert. Die Anfangsgröße des Buffers wurde von 4MB auf 512Byte verringert. Die Effizienz des Aggregators hat sich dadurch allerdings nur wenig verbessert. Unsere Vermutung war, dass bei sehr vielen Verbindungen diese Buffer sehr viel Arbeitsspeicher belegen würden.

Wenn in der Spezifikation der Testreihe keine Änderung an diesem Setup beschrieben ist, läuft der Aggregator auf dem Testaggregator und sowohl das Testprogramm als auch das Programm zur Simulation der Nachrichtenströme laufen auf dem Testsimulator. Die Nachrichtenströme werden auf Port 8000 des Testsimulators simuliert.

Ausgelöst werden die Tests mit Hilfe des XMPP Messangers Gajim, durch Senden von Textnachrichten, die in Kapitel 5.3.1 beschrieben sind.

Nach der Durchführung eines Tests wird vom Testaggregator die Datei `/var/log/ejabberd/runtimestats.dat` und vom Testsimulator die Datei `/var/log/ejabberd/teststats.dat` heruntergeladen. Das Format dieser Dateien ist in den Kapiteln 4.2.6 und 5.3.1 beschrieben. Eine Abbildung der Rohdaten ist in 6.19 zu sehen.

Um den Testaggregator und den Testsimulator auf die Durchführung eines neuen Tests vorzubereiten müssen beide Ejabberds neu gestartet werden. Zuvor wird dem Testag-

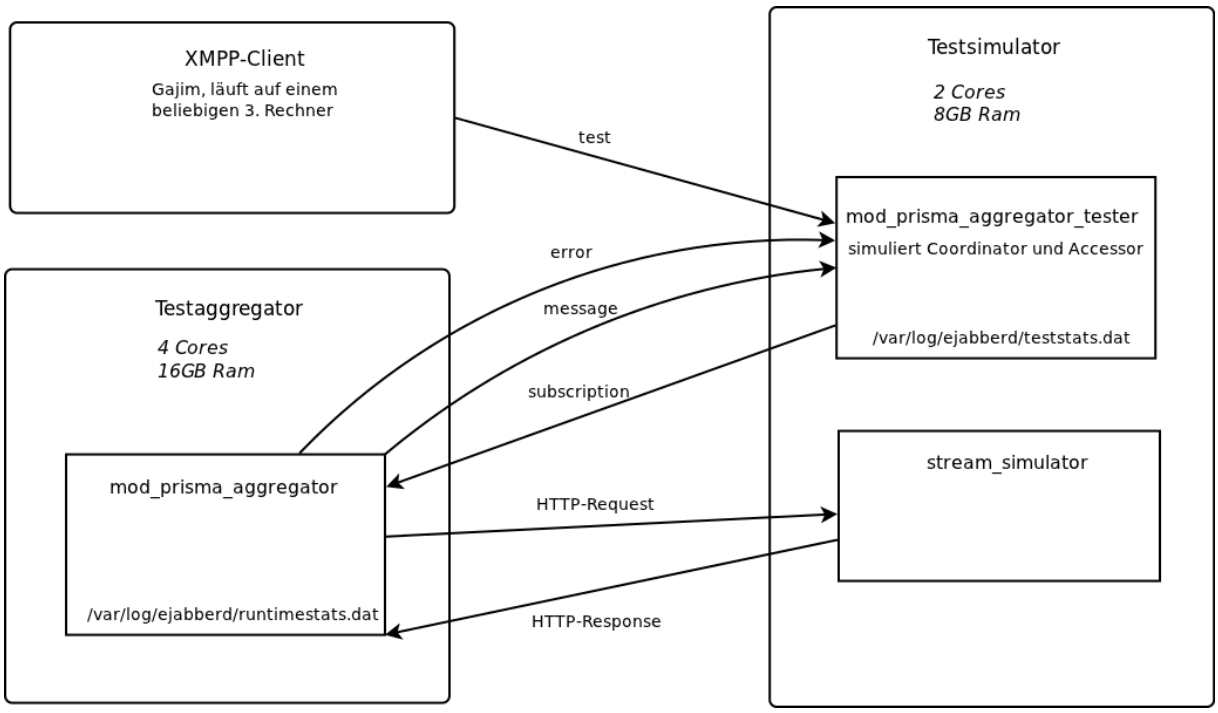

Abbildung 6.1: Übersicht der Testumgebung

gregator die Nachricht "stop_all_and_delete_mnesia" gesendet, um die nach dem Test verbliebenen Abonnements zu löschen.

6.2 Verarbeitung der Testdaten

Um Aussagen über die Effizienz des Aggragators machen zu können, ist eine Visualisierung der Auslastung unterschiedlicher Ressourcen in Abhängigkeit von den im Aggregator befindlichen Abonnements nötig. Die rohen Testdaten aus der Datei /var/log/ejabberd/runtimestats.dat geben jedoch die Ressourcenauslastung in Abhängigkeit von der Laufzeit des Aggregators wieder.

Die Verarbeitung der Testdaten und das Produzieren der Graphiken (Plotten) wird in der Programmiersprache Octave durchgeführt.

Zunächst müssen die Zeilen der Datei, die sich auf einen Zeitpunkt beziehen, an dem noch kein Test läuft, entfernt werden. Anschließend müssen die Zeitstempel so angepasst werden, dass das erste Abonnement bei Zeitpunkt Null im Aggregator verarbeitet wird. Analog muss mit der Datei /var/log/ejabberd/teststats.dat verfahren werden, mit dem Unterschied, dass hier ab dem Eintreffen der ersten PRISMA-Nachricht, bzw. dem Eintreffen des ersten PRISMA-Fehlers (abhängig davon, was früher eintritt) gezählt wird. Dadurch kommen die Daten des Aggregators und die des Aggregator-Testers in einen zeitlichen Zusammenhang.

Das Octave-Programm nimmt nun den Mittelwert einer gewählten Ressource über 1000 Abonnements (da Abonnements in fast allen Testreihen in 1000'er Paketen hinzugefügt

werden) und stellt die Ergebnisse mehrerer Tests in einem Graphen dar.

Der Zeitpunkt, an dem der Aggregator-Tester zum ersten Mal einen Fehler vom Aggregator übermittelt bekommt, wird in jeder einzelnen Kurve mit der Markierung "Error" hervorgehoben. Falls kein Fehler auftritt, befindet sich diese Markierung am letzten Punkt der jeweiligen Kurve. Wenn der Aggregator durch einen Fehler bei Abrufen eines Nachrichtenstroms abstürzen würde, könnte er die Fehlernachricht nicht mehr an den Coordinator senden. Daher bedeutet die Error-Markierung am letzten Punkt der Kurve, dass der Testlauf ohne Fehler beendet wurde.

6.3 Testergebnisse

In diesem Kapitel werden die Plots der Testreihen dargestellt. Alle Abbildungen wurden mit Octave und Gnuplot erstellt.

Nicht alle erfassten Daten, die mit Hilfe des Programms zur Verarbeitung der Testdaten erstellt werden können, werden zu jeder Testreihe dargestellt, da sie teilweise keine Signifikanz haben oder ihre wiederholte Darstellung zuviel Platz einnehmen würde.

Die folgenden Ressourcen werden abhängig von der Anzahl der Abonnements, die der Aggregator verarbeitet, dargestellt:

[CPU] Auslastung der CPU in % (100 entspricht der vollständigen Auslastung eines Kerns).

[SPS] Anzahl der korrekt verarbeiteten (abgerufen, XML interpretiert, neue Nachrichten verarbeitet) Abonnements pro Sekunde.

[runque] Anzahl der Erlang-Prozesse im Aggregator, die eine neue Nachricht in ihrem Briefkasten haben. Diese Prozesse müssen vom Scheduler verarbeitet werden. Wenn diese Zahl sehr hoch ist, kann es lange dauern, bis eine Nachricht von dem Prozess, an den sie gesendet wurde, verarbeitet wird.

[time] Der Zeitpunkt, an dem die Anzahl an Abonnements im Aggregator vorhanden sind. Diese Kurve sollte eine Gerade sein, denn der Aggregatortester fügt in regelmässigen Abständen immer die gleiche Menge an neuen Abonnements hinzu.

6.3.1 Effizienz des Aggregators

Die Testreihen, deren Ergebnisse hier dargestellt werden, sind in Kapitel 5.2.1 definiert.

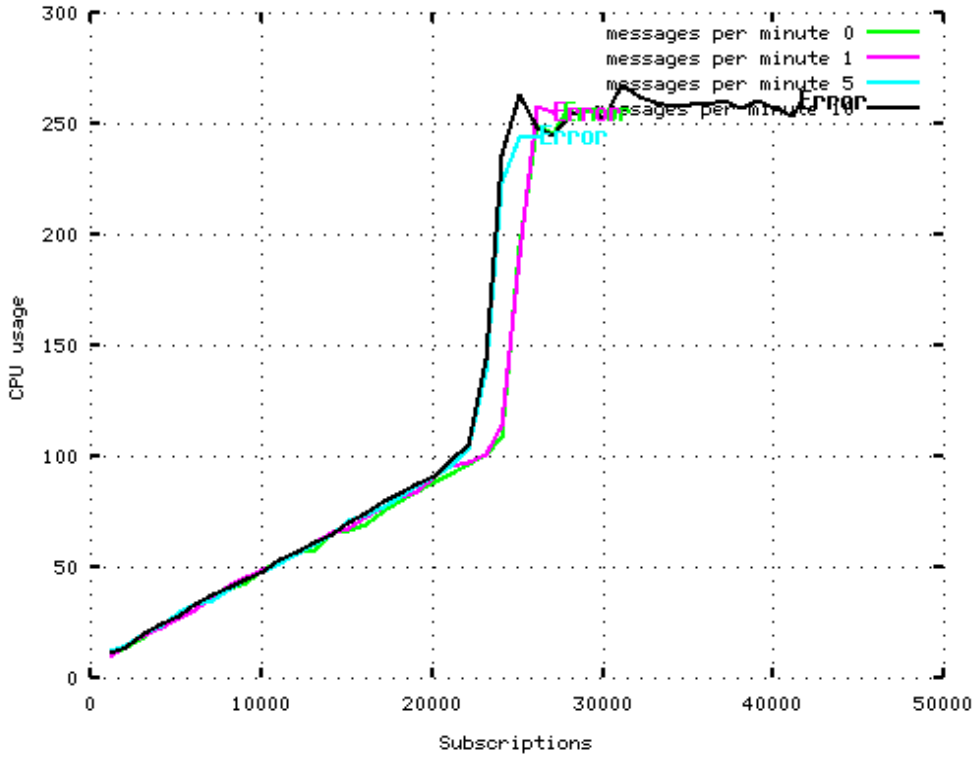

Abbildung 6.2: Hohes Nachrichtenaufkommen: CPU-Last

6.3.2 Auslasten und skalieren

In den Abbildungen 6.19 und 6.20 sind die Rohdaten (runtimestats) der beiden Aggregatoren abgebildet. Die Abbildung 6.21 zeigt die Auslastung des Arbeitsspeichers des Haupt-Aggregators.

Die restlichen Abbildungen sind analog zu den Abbildungen im vorhergehenden Kapitel entstanden. Es wurde jedoch nur der Teil der /var/log/ejabberd/runtimestats.dat verwendet, in dem der Aggregator noch neue Abonnements entgegennimmt.

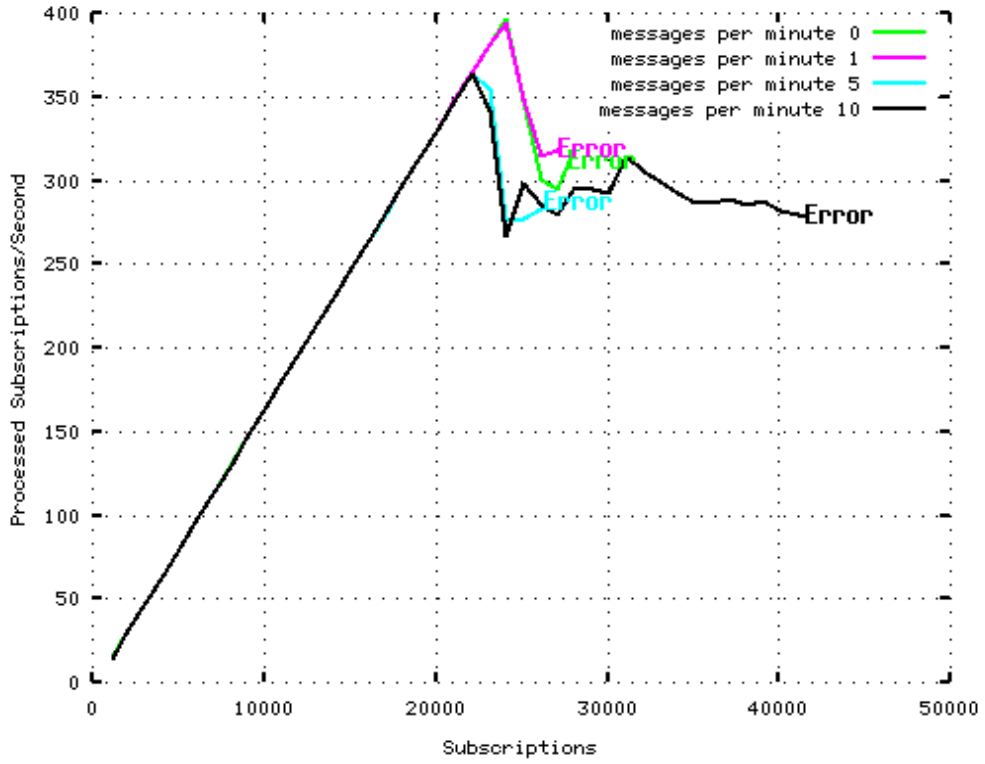

Abbildung 6.3: Hohes Nachrichtenaufkommen: Anzahl verarbeiteter Abonnements pro Sekunde

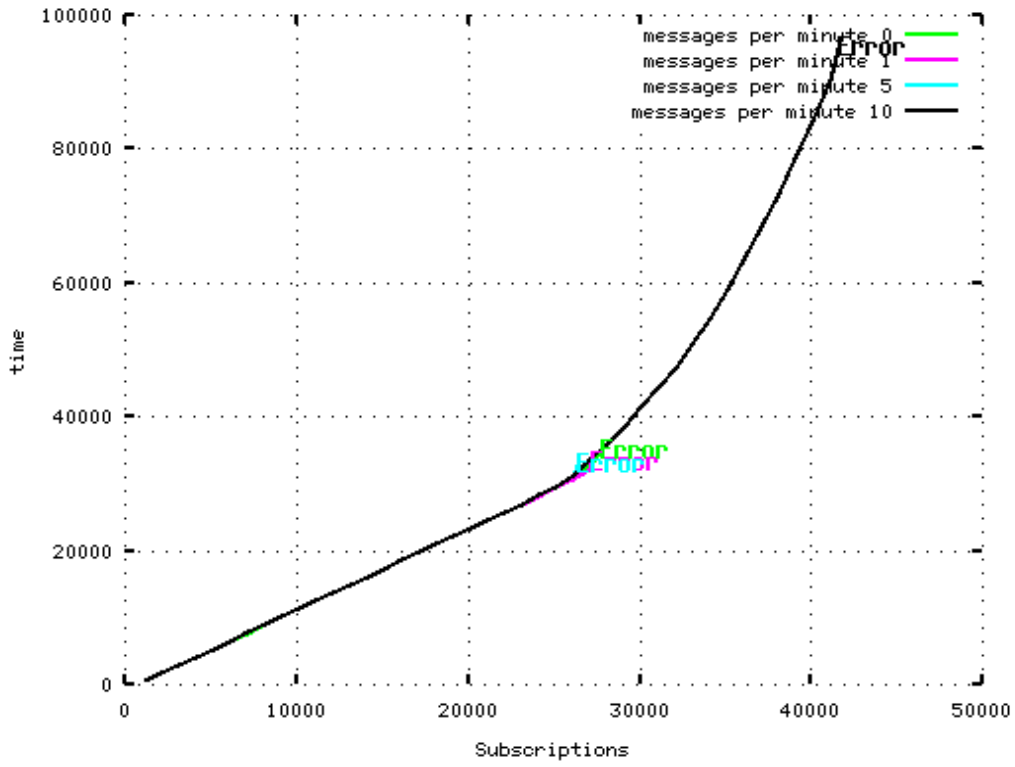

Abbildung 6.4: Hohes Nachrichtenaufkommen: Zeitpunkt, an dem die Anzahl an Abonnements im Aggregator verarbeitet werden

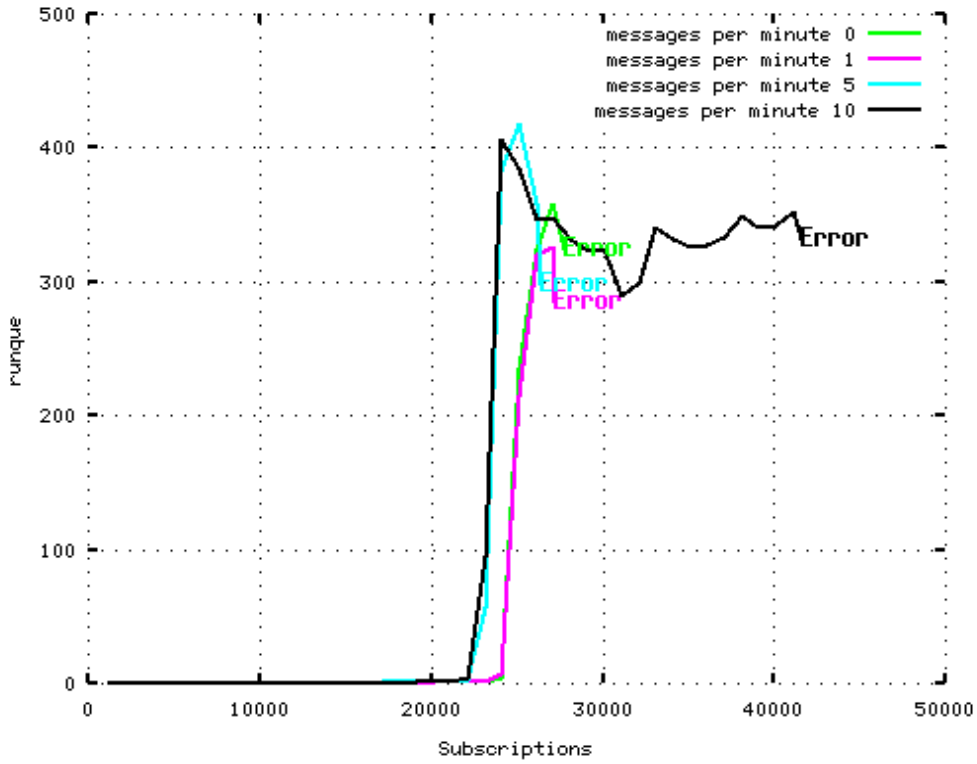

Abbildung 6.5: Hohes Nachrichtenaufkommen: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

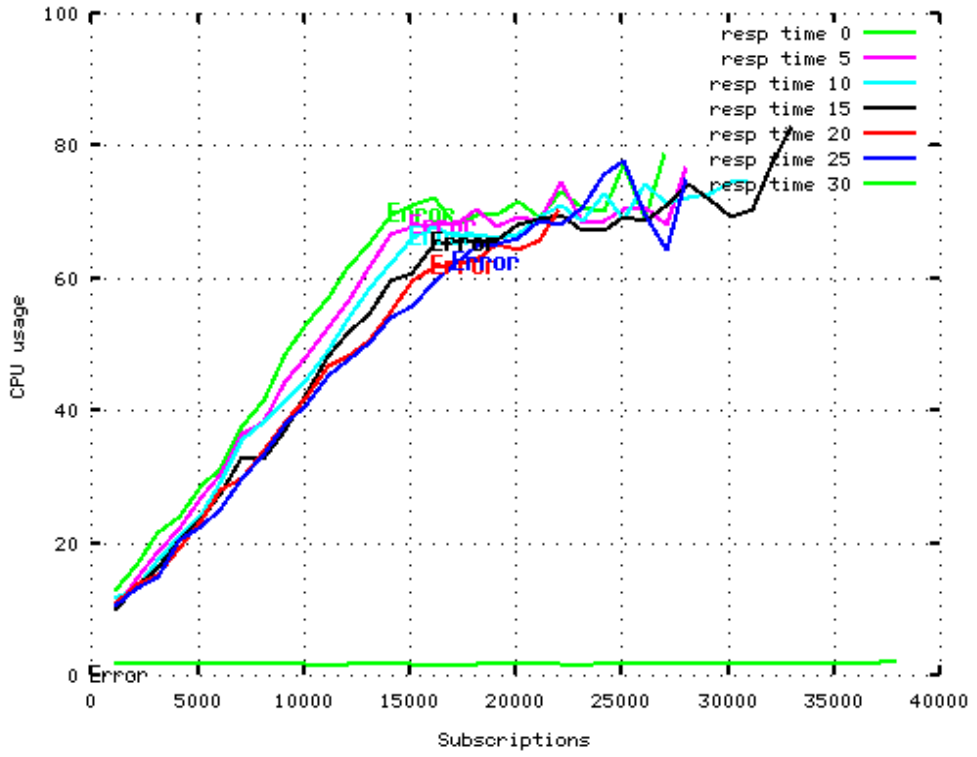

Abbildung 6.6: Verzögerung der Nachrichtenströme: CPU-Last

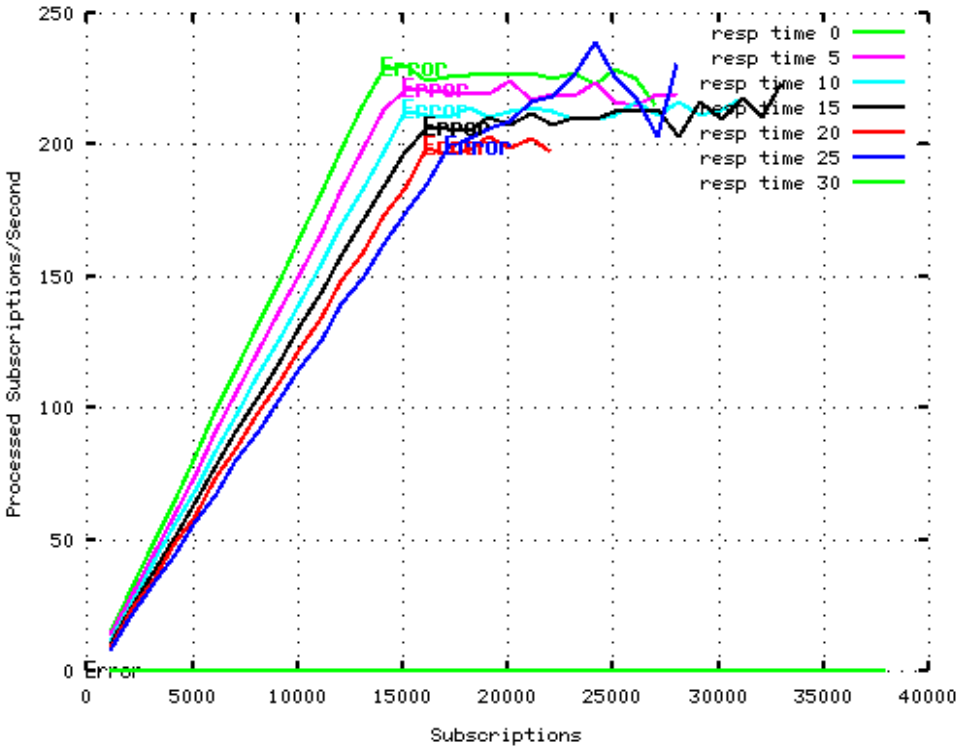

Abbildung 6.7: Verzögerung der Nachrichtenströme: Anzahl verarbeiteter Abonnements pro Sekunde

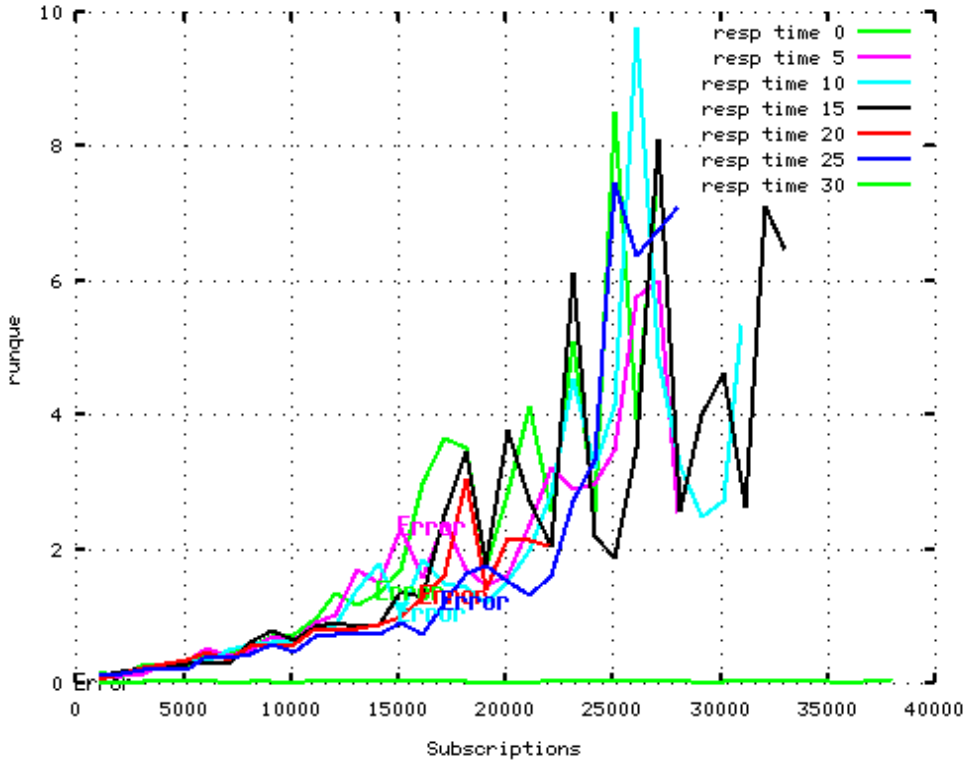

Abbildung 6.8: Verzögerung der Nachrichtenströme: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

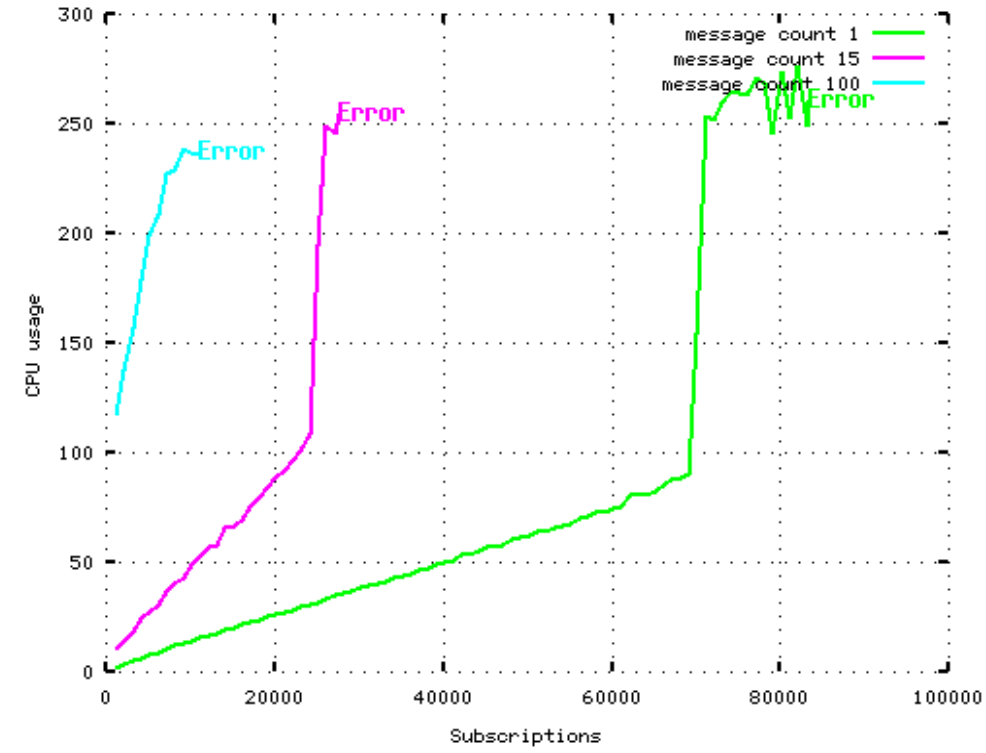

Abbildung 6.9: Anzahl der Nachrichten: CPU-Last

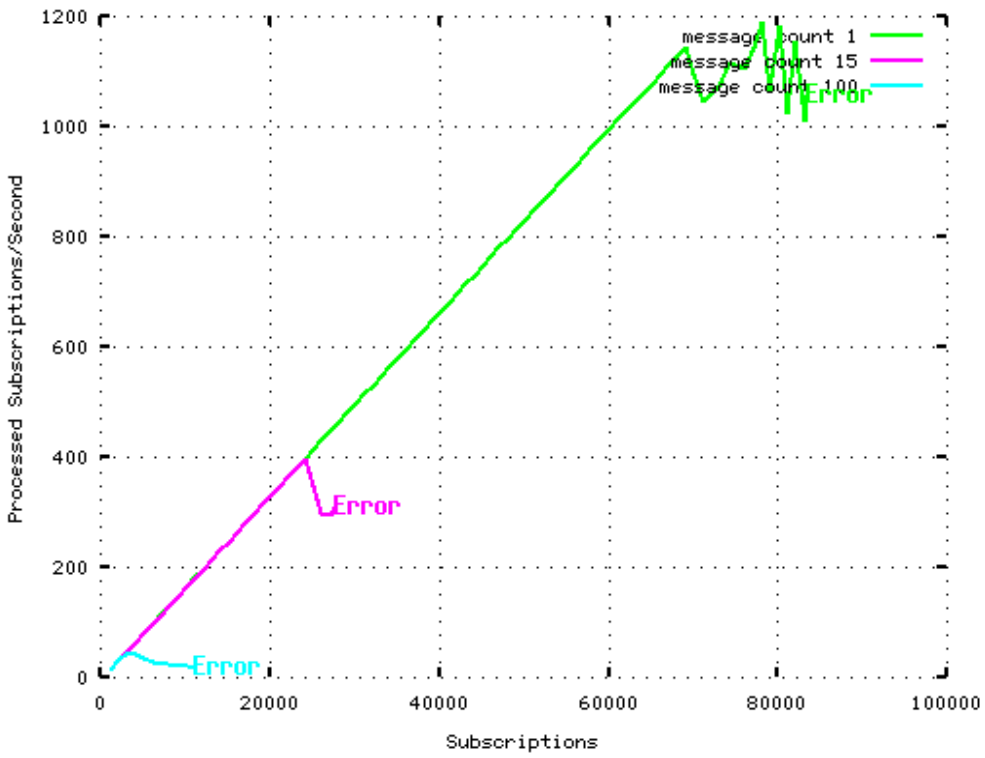

Abbildung 6.10: Anzahl der Nachrichten: Anzahl verarbeiteter Abonnements pro Sekunde

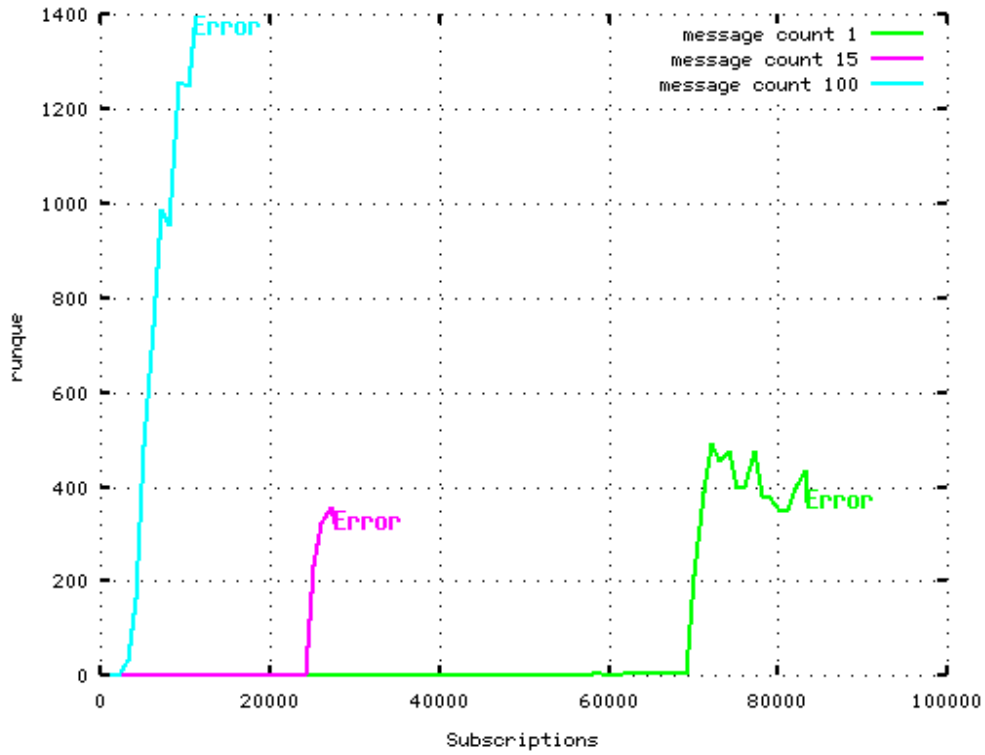

Abbildung 6.11: Anzahl der Nachrichten: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

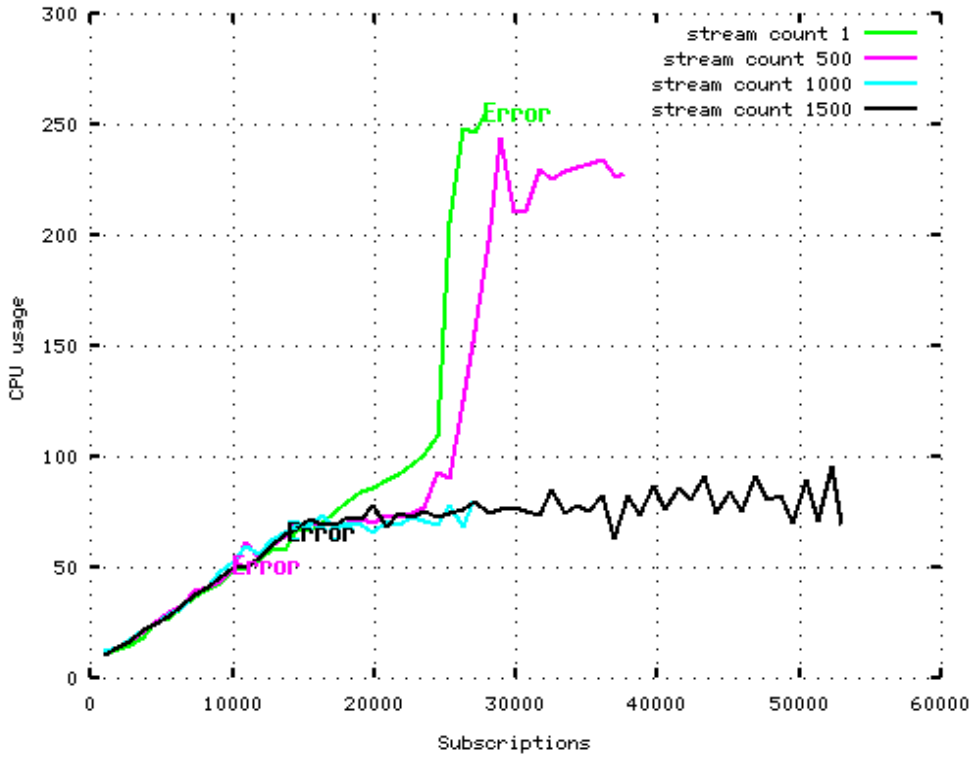

Abbildung 6.12: Anzahl der Nachrichtenströme: CPU-Last

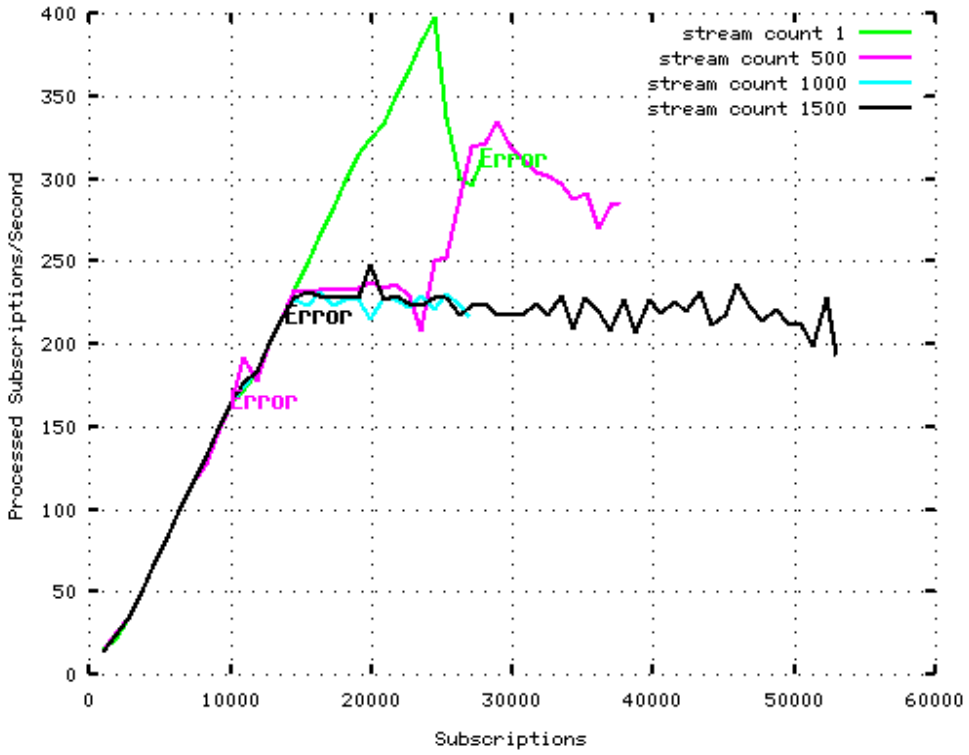

Abbildung 6.13: Anzahl der Nachrichtenströme: Anzahl verarbeiteter Abonnements pro Sekunde

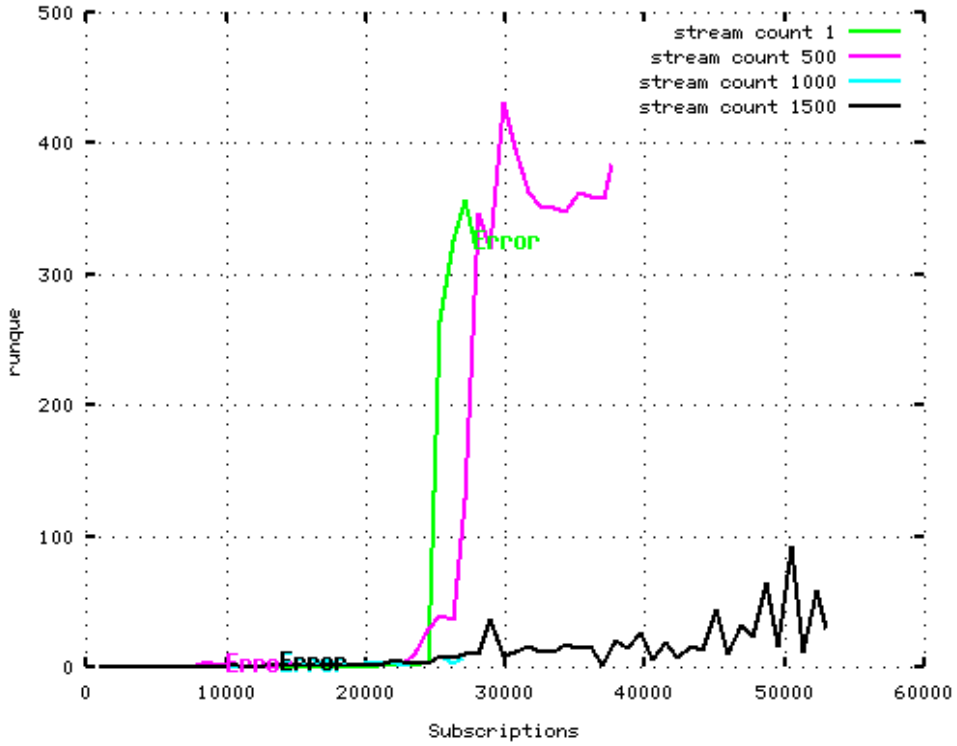

Abbildung 6.14: Anzahl der Nachrichtenströme: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

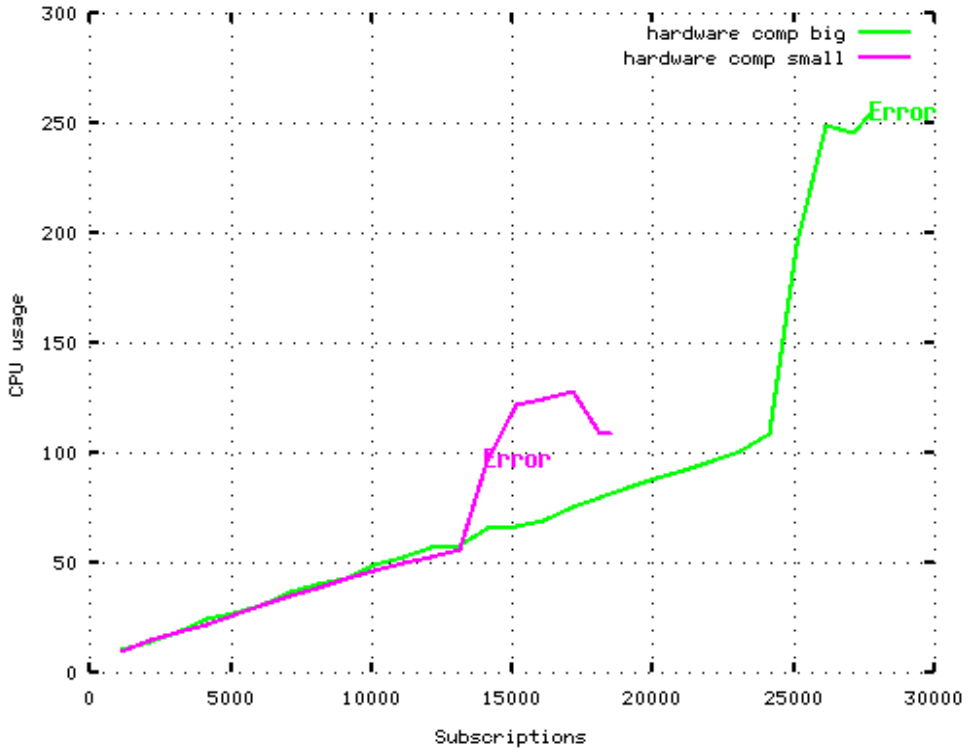

Abbildung 6.15: Unterschiedliche Hardware: CPU-Last

Abbildung 6.16: Unterschiedliche Hardware: Anzahl verarbeiteter Abonnements pro Sekunde

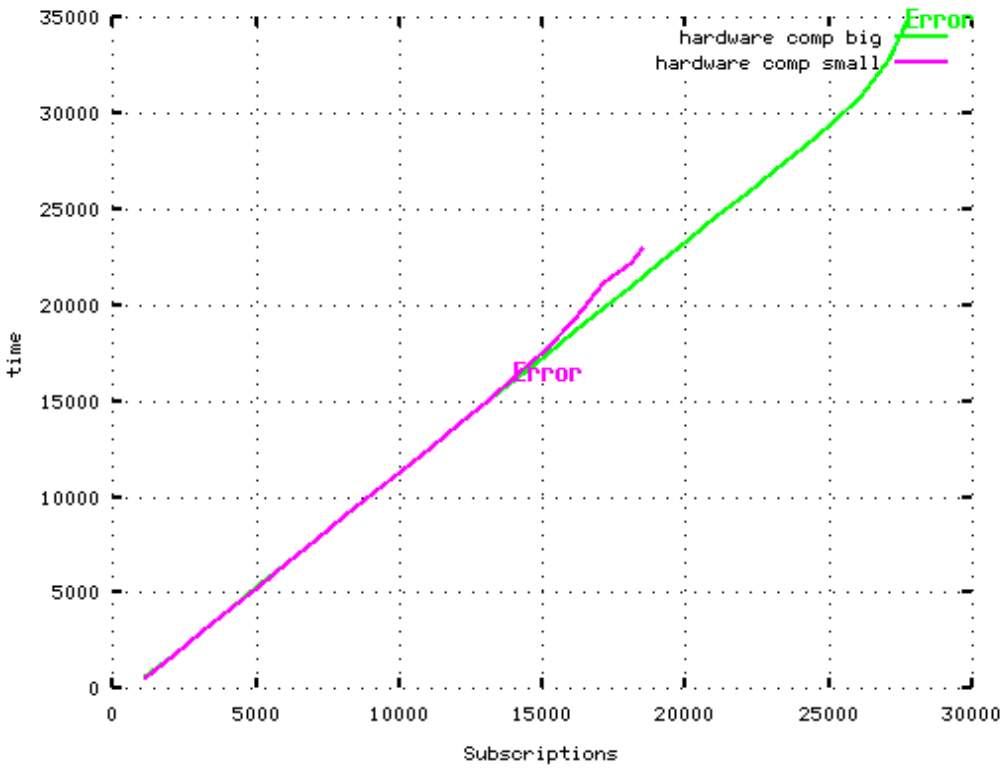

Abbildung 6.17: Unterschiedliche Hardware: Zeitpunkt, an dem die Anzahl an Abonnements im Aggregator verarbeitet werden

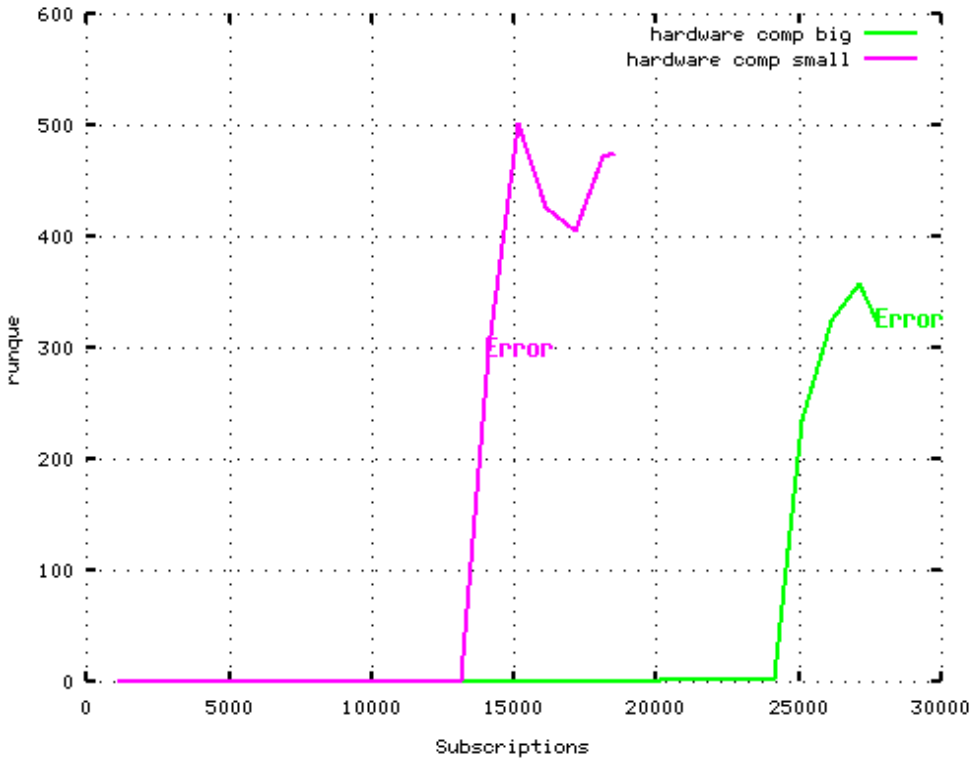

Abbildung 6.18: Unterschiedliche Hardware: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

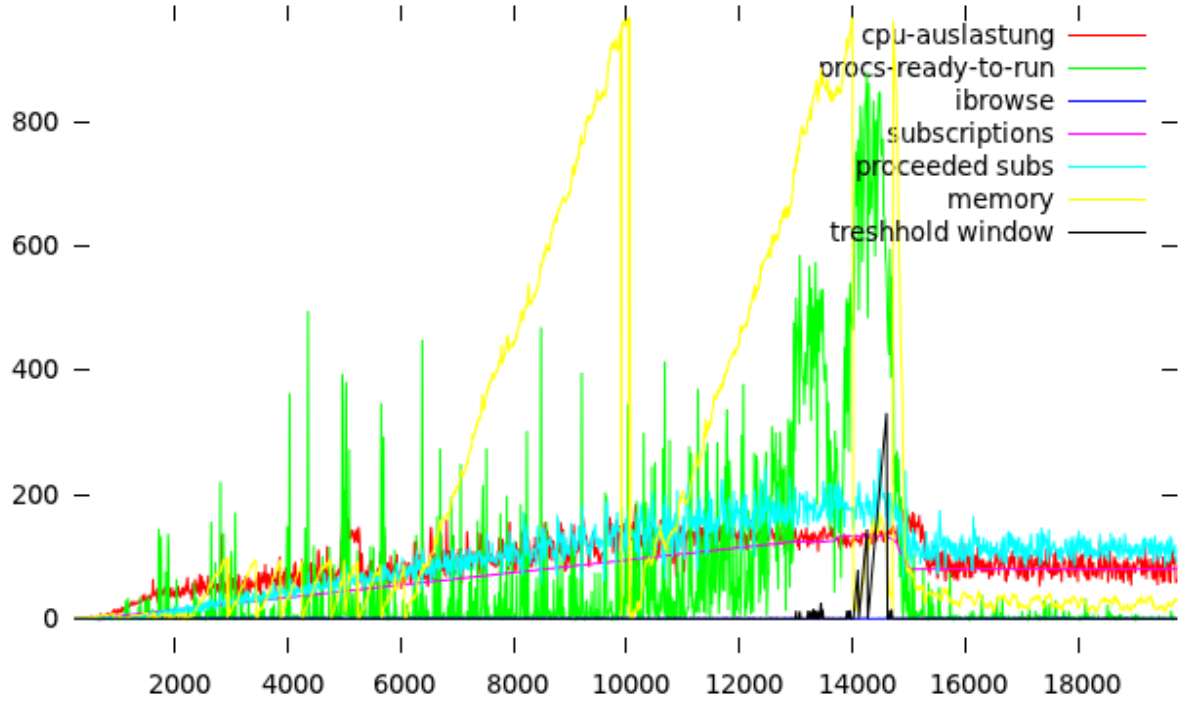

Abbildung 6.19: Aulasten und Skalieren: Rohdaten des Haupt-Aggregators

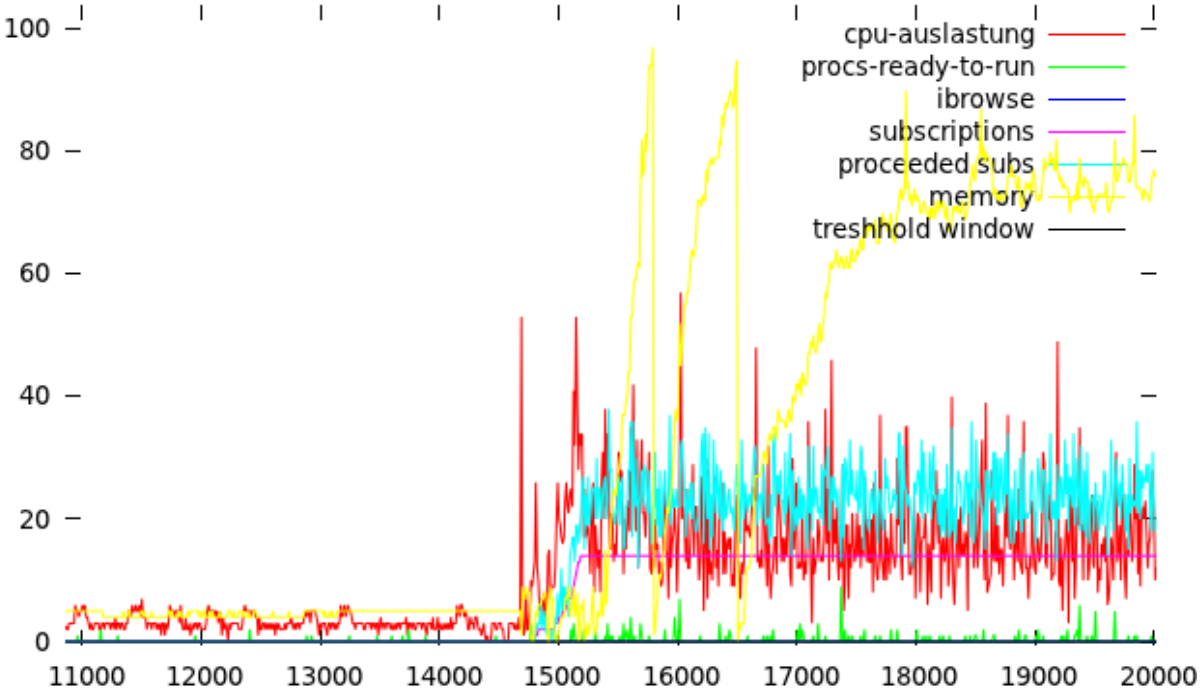

Abbildung 6.20: Auslasten und Skalieren: Rohdaten des Fallback-Aggregators

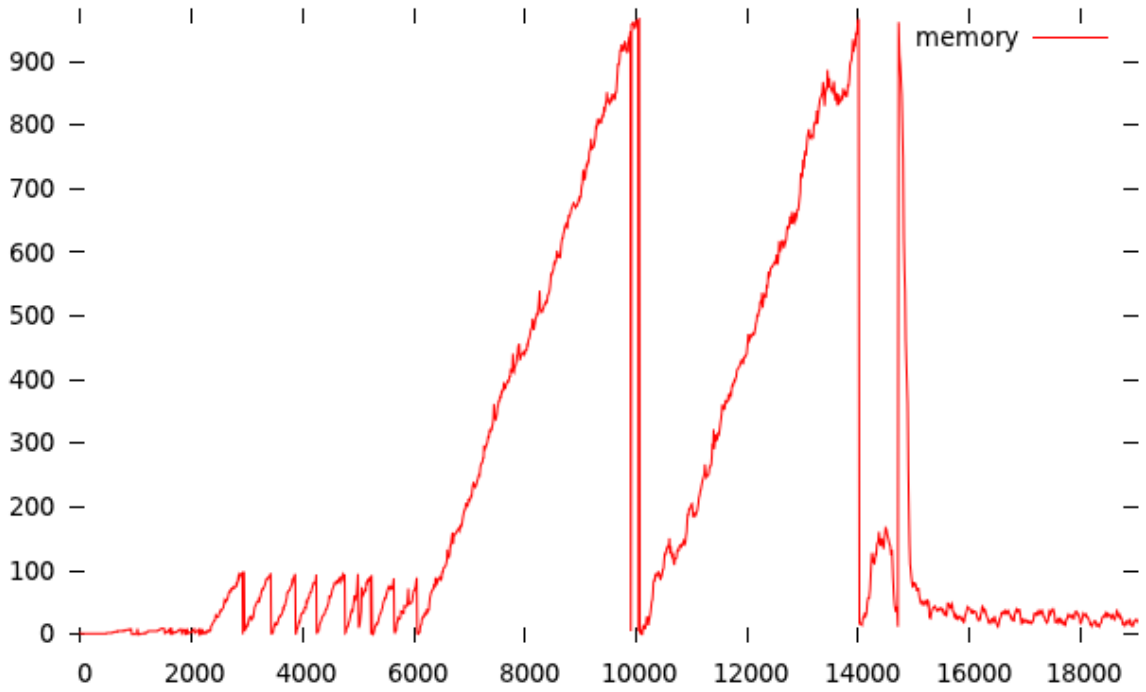

Abbildung 6.21: Auslasten und Skalieren: Speicherauslastung des Haupt-Aggregators (Rohdaten: Zeit=Zehntelsekunden, RAM=10M)

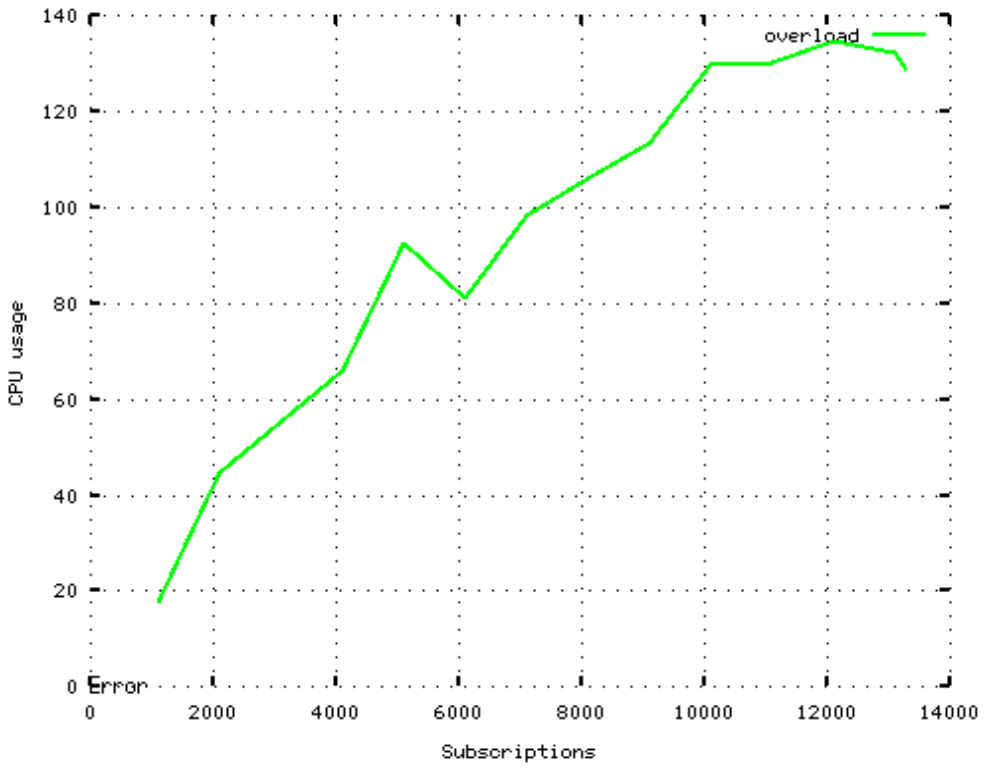

Abbildung 6.22: Auslasten und Skalieren: CPU-Last

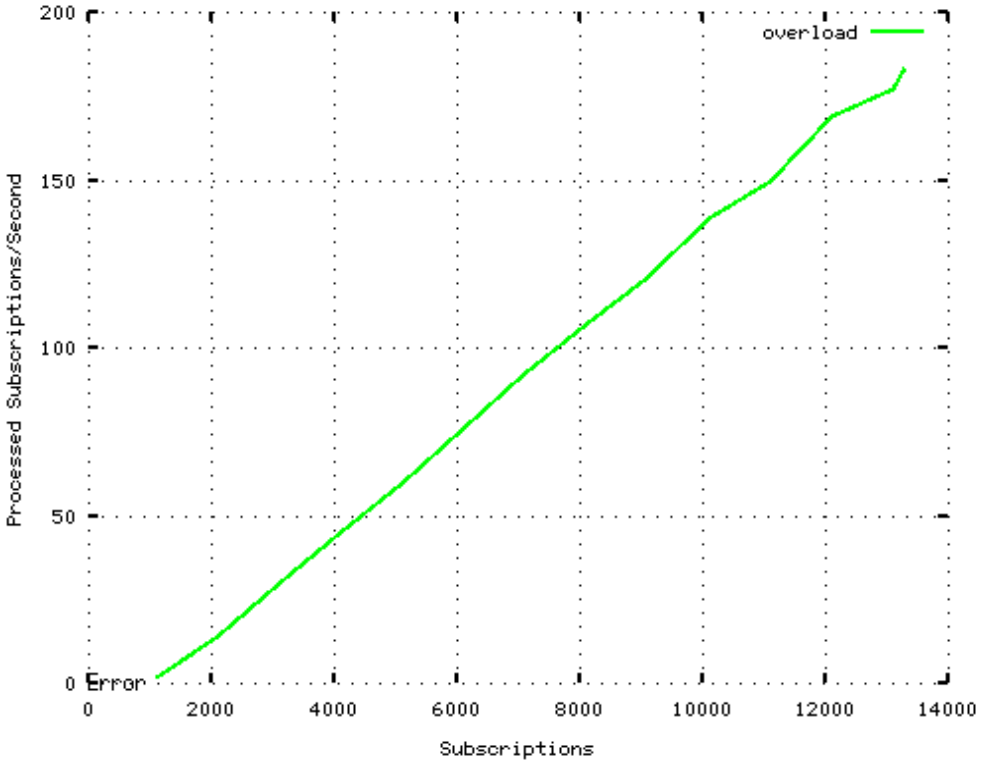

Abbildung 6.23: Auslasten und Skalieren: Anzahl verarbeiteter Abonnements pro Sekunde

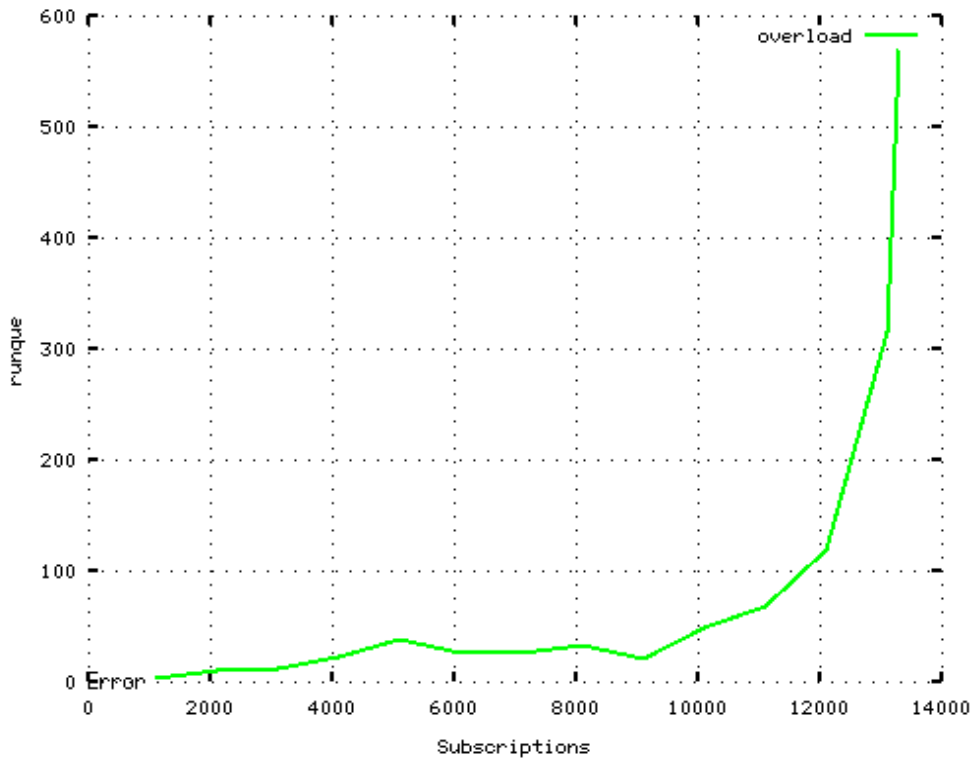

Abbildung 6.24: Auslasten und Skalieren: Anzahl der Prozesse, deren Message-Queue mindestens eine Nachricht enthält

6.4 Interpretation der Testergebnisse

Dadurch, dass in den Testreihen zur Effizienz des Aggregators einzelne Parameter der kontrollierten Nachrichtenströme variiert werden, gibt es einzelne Tests, die in mehreren Testreihen identisch sind. Die "Standardkonfiguration" ist ein Nachrichtenstrom, der 15 Nachrichten ausliefert, niemals neue Nachrichten erzeugt und bei dem jedes Abonnement den gleichen physikalischen Nachrichtenstrom abruft. Sie ist in den folgenden Testreihen zu finden:

[Unterschiedliche Hardware] Hier ist sie der besser ausgestattete Aggregator.

[Anzahl der Nachrichten] Die Konfiguration ist der mittlere Test, bei dem 15 Nachrichten versendet werden.

[Anzahl der Nachrichtenströme] Der Test, bei dem alle Abonnements den gleichen Physischen Datenstrom verwenden.

[Hohes Nachrichtenaufkommen] Hier entspricht sie dem Test, der keine neuen Nachrichten generiert.

[Verzögerung der Nachrichtenströme] Da in der Testreihe zur Verzögerung der Nachrichtenströme alle Tests 1000 unterschiedliche Nachrichtenströme benutzen [1], gibt es hier keine Überschneidung mit der Standardkonfiguration. Eine Überschneidung gibt es in dieser Testreihe zwischen dem Test, der keine Verzögerung hat und dem Test, der 1000 Nachrichtenströme benutzt aus der Testreihe zur Anzahl der Nachrichtenströme.

Die Effizienz des Aggregators kann aus den Kurven, die die verarbeiteten Abonnements pro Sekunde über den im Aggregator befindlichen Abonnements darstellen, abgelesen werden. Diese Kurven steigen in einer Geraden an, nach erreichen eines Maximums fallen sie ab, um anschließend konstant zu verlaufen. Die Effizienz, die in einem Test erreicht wird, ist das Maximum dieser Kurve. Am höchsten ist die Effizienz des Aggregators in der Testreihe zur Anzahl der Nachrichten. Der Test, in dem nur eine Nachricht von jedem Nachrichtenstrom versendet wird, erreicht eine Effizienz von ca. 1.100 abgerufenen Nachrichtenströmen pro Sekunde.

Die niedrigste Effizienz wurde ebenfalls in der Testreihe zur Anzahl der Nachrichten erreicht. In dem Test, in dem der Nachrichtenstrom 100 Nachrichten ausliefert, liegt die maximale Effizienz bei ca. 45 abgerufenen Nachrichtenströmen pro Sekunde. Da die Parameter in den anderen Testreihen stärker variiert wurden, als dies in der Realität zu erwarten wäre, kann davon ausgegangen werden, dass die Länge des Nachrichtenstroms den größten Einfluss auf die Effizienz des Aggregators hat.

[1] Dies ist notwendig, da zehn Anfragen parallel auf einen Nachrichtenstrom gemacht werden können und dies mit langen Verzögerungen den Aggregator ausbremsen würde.

Um einen Eindruck von der Effizienz des Aggregators auf realistischen Nachrichtenströmen zu erhalten, muss der Test "Auslasten und Skalieren" betrachtet werden. Hier erreicht der Aggregator eine Effizienz von ca. 200 abgerufenen Nachrichtenströmen pro Sekunde. An diesem Punkt hat die Kurve jedoch noch nicht ihr Maximum überschritten - der Aggregator sendet die Überlastungsnachricht etwas zu früh an den Coordinator. Der Aggregator verarbeitet hier ca. 13.000 Abonnements. Aus vorhergehenden Tests, deren Ergebnisse hier nicht abgebildet sind, war zu erkennen, dass die Kurve zu den verarbeiteten Abonnements pro Sekunde ihr Maximum bei ca. 20.000 Abonnements hat. Der Zeitpunkt, an dem der Aggregator die Überlastungsnachricht an den Coordinator schickt, wird dadurch bestimmt, dass die Runqueue zuvor 20 Sekunden lang über 500 Einträge enthielt. Die Konfiguration dieser Parameter kann und sollte an die Hardware angepasst werden, auf der der Aggregator läuft, um die Überlastungsnachricht zum optimalen Zeitpunkt zu versenden.

Der Aggregatortester fügt dem Aggregator neue Abonnements in regelmäßigen Abständen hinzu. Da die Menge der Abonnements in jedem Zeitschritt gleich ist, sollte die Kurve, die den Zeitpunkt beschreibt, an dem eine Anzahl an Abonnements im Aggregator verarbeitet werden, eine Gerade sein. Wenn der Aggregator ausgelastet ist, werden neue Abonnements nicht so schnell angenommen, wie sie vom Aggregatortester versendet werden. Das Ergebnis davon kann in Abbildung 6.4 betrachtet werden. Der Test, in dem zehn neue Nachrichten pro Minute hinzugefügt werden, lief nach dem Auslasten noch länger weiter. Die Kurve "Zeit" beschreibt nach dem Auslasten (ab ca. 27.000 Abonnements) keine Gerade mehr, sondern beginnt, eine Parabel zu formen. Dies kommt daher, dass der Aggregator die neuen Abonnements nicht mehr so schnell hinzufügen kann, wie er sie vom Aggregatortester übergeben bekommt. Der Aggregator reagiert damit auf die Nachrichten vom Coordinator mit einer Verzögerung, deren Länge davon abhängig ist, wie stark der Aggregator ausgelastet bzw. überlastet ist. Um die Nachrichten zum Migrieren von Abonnements noch verarbeiten zu können, muss der Aggregator seine Überlastungsnachricht an den Coordinator senden, bevor diese Kurve zu steil wird. Irgendwann beginnt der Ejabberd nicht-verarbeitete Nachrichten zu verwerfen und wenn unter den Verworfenen die Migrationsnachrichten sind, funktioniert das Skalieren des Aggregators nicht mehr.

In Abbildung 6.7 kann man gut erkennen, was die Verlängerung der Antwortzeit der einzelnen Nachrichtenströme für einen Einfluss auf die Effizienz des Aggregators hat. Die Abrufzeit des Aggregators verlängert sich um die Verzögerung und die Kurve der verarbeiteten Abonnements pro Sekunde verläuft entsprechend flacher. In dieser Testreihe produzieren die Abonnements ihren ersten Fehler genau an der Stelle der Kurve, an der die Effizienz maximal ist.

Die Menge an Arbeitsspeicher, die der Aggregator zu einem Zeitpunkt verbraucht, lässt sich leider nicht genau bestimmen. Der Grund dafür ist, dass Erlang Garbage-Collection auf Prozess-Ebene macht. Es gibt keine Stelle im Laufzeitsystem, an der bekannt ist, wieviel Speicher alle Prozesse zusammen verbrauchen. Jeder einzelne Prozess müsste danach

gefragt werden, wieviel Speicher er verbraucht, wodurch jedoch ein nicht unerheblicher Aufwand getrieben würde (denn dies müsste zehn mal pro Sekunde wiederholt werden), was wiederum das Ergebnis der Messung beeinflussen würde.

Der hier gemessene Verbrauch an Arbeitsspeicher wird durch Aufruf des Unix-Kommandos ps ermittelt und kann als obere Grenze des tatsächlichen Verbrauchs interpretiert werden [erlfaq:memory]. Leider schwankt der Verbrauch über die Zeit sehr stark, weshalb die entsprechenden Diagramme in den Testergebnissen nicht aufgeführt wurden. In Diagramm 6.21 ist die Speicherauslastung aus den Rohdaten des Testlaufs "Auslasten und Skalieren" einzeln abgebildet.

Aus den Ergebnissen der Testreihe zu unterschiedlicher Hardware kann abgelesen werden, dass die Effizienz des Aggregators durch Verdoppeln der Menge des Arbeitsspeichers sowie der Anzahl an CPU-Kernen, sich ebenfalls verdoppelt. In dem Beispiel konnten die verarbeiteten Abonnements pro Sekunde von ca. 215 auf ca. 390 mehr als verdoppelt werden.

In den Rohdaten zum Test "Auslasten und Skalieren", abgebildet in 6.19 und 6.20, kann der korrekte Ablauf des Tests nachvollzogen werden. Die Menge an Abonnements, die vom Haupt-Aggregator migriert werden, veranlassen den Fallback-Aggregator dazu, ab Zeitstempel 15.000 aktiv zu werden.

Nach Erreichen der maximal verarbeiteten Abonnements pro Sekunde arbeitet der Aggregator noch lange weiter, auch wenn sich die Zeit, die zwischen zwei Abfragen an den gleichen Datenstrom vergeht, verlängert. Diese Eigenschaft haben auch andere in Erlang geschriebenen Programme. Für den Ejabberd ist diese Eigenschaft in [xmpp-scalability] beschrieben. Im Überlastungsfall stürzen die anderen, am Vergleich beteiligten XMPP-Server, ab. Da ein XMPP-Server im Aggregator enthalten ist, kann man davon ausgehen, dass die hier entstandene Implementierung des Aggregators, im Vergleich zu anderen, eine höhere Ausfallsicherheit bietet.

Im Test "Auslasten und Skalieren" kann beobachtet werden, dass an dem Zeitpunkt, an dem die Effizienz des Aggregators ihr Maximum erreicht hat, weder die CPU noch der Arbeitsspeicher ausgelastet sind. Ähnliche Beobachtungen können in den anderen Testreihen gemacht werden. Den Grund dafür festzustellen war uns leider nicht möglich. Wir vermuten, dass die HTTP-Client-Bibliothek einen Flaschenhals enthält, der den Aggregator davon abhält seine maximale Effizienz zu erreichen [2].

Im Folgenden sind noch einmal die wichtigsten Beobachtungen aufgelistet:

[Effizienz] Die Anzahl Abonnements, die der Aggregator zu verarbeiten in der Lage ist, hängt von dem Verhalten der Nachrichtenströme ab. Den größten Einfluss hat die Menge bzw. Länge der Nachrichten, die bei jedem Abrufen des Nachrichtenstroms übertragen werden. Die in den Testreihen ermittelte Effizienz liegt zwischen 45 und

[2] Dies trifft nicht zu, wie in der Testreihe zur Menge der ausgelieferten Nachrichten gesehen, wenn der Aufwand zum Interpretieren der Nachrichtenströme überwiegt.

1.100 verarbeiteten Nachrichtenströmen pro Sekunde. Auf realistischen Datenströmen wurden ca. 200 Nachrichtenströme pro Sekunde verarbeitet.

[Robustheit] Im Falle einer Überlastung stagniert die Effizienz, bevor der Aggregator abstürzt. Diese Eigenschaft erbt der Aggregator von dem Programmiermodell, für das Erlang entwickelt wurde.

7 Fazit und Ausblick

Die Implementierung des Aggregators wird an den Punkten, die im Kapitel Anforderungsanalyse aufgestellt wurden, gemessen.

[Effizienz] Die Effizienz des Aggregators (Anzahl der abgerufenen Nachrichtenströme pro Sekunde) übertrifft die Anforderungen. In dem Test "Auslasten und Skalieren" werden Nachrichtenströme abgerufen, die durch Web-Crawlen gefunden wurden. Viele dieser Nachrichtenströme verhalten sich schlechter als die Nachrichtenströme, die im echten Einsatz des Aggregators Verwendung finden [1]. Selbst von diesen, vermutlich "schlechten" Nachrichtenströmen, kann der Aggregator ca. 14.000 Abonnements verwalten. Dieser Wert kann durch ein längeres Pollingintervall vergrößert werden. Eine obere Schranke stellt hier die Menge an Arbeitsspeicher dar, denn der aggregator_connector, der sich um das jeweilige Abonnement kümmert, braucht Arbeitsspeicher. Die genaue Menge an Arbeitsspeicher, die ein Prozess benötigt, hängt vom Nachrichtenstrom ab. Entscheidend ist hier die Größe der Schlüssel, anhand derer bereits bekannte Nachrichten aussortiert werden. Die Größe der Schlüssel hängt nur vom Nachrichtenstrom ab.

[Integration] Der hier implementierte Aggregator wurde nicht mit anderen Komponenten des PRISMA-Systems getestet. Daher ist es nur schwer möglich, Aussagen darüber zu machen, wie gut der Aggregator sich integriert.

Es lagen zwei Mitschnitte von XML-Nachrichten aus dem echten PRISMA-System vor, eine Subscription und eine Message. Diese beiden Nachrichtentypen sollten konform zu anderen Komponenten des PRISMA-Systems sein. Sie werden zwischen dem Aggregator und dem Aggregatortester während der Testläufe ausgetauscht.

Andere Nachrichtentypen, etwa zum Migrieren eines Abonnements, wurden hier definiert. Daher muss noch Arbeit in deren Integration gesteckt werden, wenn der Aggregator im echten PRISMA-System arbeiten soll.

Die Filterspezifikation wird von dem Aggregator nicht umgesetzt. Grund dafür sind zum Einen die fehlenden Beispieldaten bzw. eine Implementierung des Coordinators aus dem PRISMA-System, zum Anderen das Design der Filter. Es zielt auf eine Programmiersprache mit Objektorientierung ab und entspricht der Abbildung einer Objektstruktur in Json. Die Umsetzung in Erlang, das kein Objektsystem hat,

[1] Die "echten" Nachrichtenströme sollten existieren, valides XML ausgeben und innerhalb einer kurzen Zeitspanne antworten - ansonsten würden sie keine Leser finden.

ist daher aufwendig und undogmatisch. Eine andere Filterspezifikation, etwa mit Prädikaten und logischen Operatoren, wäre von der Objektstruktur her einfacher und in einer funktionalen Sprache wie Erlang einfacher umzusetzen.

[Heterogene Nachrichtenströme] Nachrichtenströme der Typen RSS und ATOM können vom Aggregator auf neue Nachrichten überprüft werden. Da die Datenströme, die hier zu Testzwecken verwendet wurden, in dem Grad der Erfüllung ihrer jeweiligen Spezifikation stark variieren, kann die Implementierung des Aggregators vermutlich nicht alle RSS- und ATOM-Datenströme korrekt interpretieren.

Die Umsetzung einer Aggregator-Komponente für XMPP-Datenströme ist nicht erfolgt. Es gibt nach unseren Recherchen kein verbreitetes XMPP-Nachrichtenstrom-Format und der Overhead, gegenüber dem Empfangen einfacher XMPP-Nachrichten, ist gering. Ein Vergleich unterschiedlicher XMPP-Server, auch des Ejabberd, ist in [xmpp-scalability] zu finden.

[Dokumentformat] Der Aggregator gibt die einzelnen Nachrichten in einem einheitlichen Format an den Accessor weiter. Die Identifikation der Nachrichtenteile, in denen Informationen enthalten sind und deren Format, muss durch andere Komponenten des PRISMA-Systems erfolgen. Grund dafür ist, wie im vohergehenden Punkt, die Varianz im Format der einzelnen Nachrichtenströme.

[Testreihen] Die Ergebnisse der Testreihen sind in dieser Arbeit dokumentiert. Die Software, die zur Durchführung der Testreihen mit anschließender Analyse der Testdaten benötigt wird, ist im Rahmen dieser Arbeit entstanden. Mit ihr kann die hier entstandene Implementierung des Aggregators mit anderen Implementierungen verglichen werden. Vorraussetzung dafür ist, dass der Aggregator seine Laufzeitstatistik wie in Kapitel 4.2.6 beschrieben, in einer Datei dokumentiert.

Durch diese Arbeit bleiben einige Fragen unbeantwortet, die Gegenstand nachfolgender Arbeiten sein könnten:

- Die Grundlage für den Vergleich unterschiedlicher Implementierungen des Aggregators wurde hier geschaffen, doch eine zweite Implementierung fehlt. Ob die Effizienz des Aggregators, der in dieser Arbeit erstellt wurde, gut oder schlecht ist, kann ohne Vergleichsimplementierungen nicht bestimmt werden.
- Die Integration in das PRISMA-System könnte fertiggestellt und die Effizienz des Aggregators im produktiven Einsatz untersucht werden.
- Der in der HTTP-Client-Bibliothek vermutete Flaschenhals könnte näher untersucht werden.

Literaturverzeichnis

[doe97] Joe Armstrong. The development of erlang. Technical report, Computer Science Laboratory, Ericsson Telecom AB, 1997.

[erlang-http-clients] Trapexit Forum. http://www.trapexit.org/forum/viewtopic.php?p=63187, 2010.

[erldoc:mnesia] erlang.org. Erlang/otp documentation. http://www1.erlang.org/documentation/doc-5.0.1/lib/mnesia-3.9.2/doc/html/Mnesia_chap1.html#1, 2011.

[erlfaq:memory] erlang.org. Frequently asked questions about erlang. http://www.erlang.org/faq/how_do_i.html#id49523, 2011. [Online; Stand 4. Juli 2011].

[jabber-draft] J. Miller. The jabber.org project. http://xmpp.org/internet-drafts/attic/draft-jabber-00.html, 2011.

[prisma-architecture] Katz, Lunze, Feldmann, Röhrborn, and Schill. System architecture for handling the information overload in enterprise information aggregation systems. Technical report, Techinische Universität Dresden und Communardo Software GmbH, 2011.

[webserver-comparison] Joe. Nginx vs yaws vs mochiweb : Web server performance deathmatch. http://www.joeandmotorboat.com/2009/01/03/nginx-vs-yaws-vs-mochiweb-web-server-performance-deathmatch-part-2/, 2009.

[wiki:atom] Wikipedia. Atom (format) — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Atom_(Format)&oldid=89021135, 2011. [Online; Stand 25. Mai 2011].

[wiki:rss] Wikipedia. Rss — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=RSS&oldid=88803697, 2011. [Online; Stand 25. Mai 2011].

[wiki:xmpp] Wikipedia. Extensible messaging and presence protocol — wikipedia, die freie enzyklopädie. http://de.wikipedia.org/w/index.php?title=Extensible_Messaging_and_Presence_Protocol&oldid=88490145, 2011. [Online; Stand 30. Mai 2011].

[xmpp-scalability] Peter Frohberg. *Skalierbarkeitsanalyse einer XMPP-Lösung.* PhD thesis, Technische Universität Dresden, 2011.