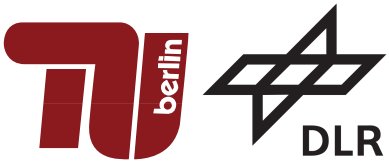

Technische Universität Berlin
in Zusammenarbeit mit dem DLR

Masterarbeit

Analyse und Konzept zur Integration eines Enterprise Service Bus in elastischen Infrastructure as a Service Umgebungen

Eingereicht von:	Younes Yahyaoui yahyaoui@cs.tu-berlin.de
Fachbereich:	Informatik
Fachrichtung:	Kommunikationsbasierte Systeme
Gutachter:	Prof. Dr. Habil. Odej Kao
Zweitgutachter:	Prof. Dr. Hans-Ulrich Heiß
Betreuer (TU Berlin):	Dr. André Hoing, Dr. Matthias Hovestadt
Betreuer (DLR):	Dipl.-Ing. Louis Calvin Touko Tcheumadjeu
Abgabedatum:	11. Dezember 2010

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Masterarbeit selbständig verfasst, noch nicht anderweitig für andere Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Berlin, den 11.12.2010

_____

Younes Yahyaoui

Vorwort und Danksagung

Die vorliegende Masterarbeit ist im Rahmen meiner Tätigkeit bei dem Deutschen Zentrum für Luft- und Raumfahrt in Berlin entstanden.

Ich war in dem Institut für Verkehrssystemtechnik beschäftigt. Diese Arbeit ist Teil meines Studiums der Informatik an der Technischen Universität in Berlin.

Mein erster Dank geht an *Herrn Professor Dr. habil. Odej Kao* für die Themenstellung und die fachliche Betreuung seitens der Universität.

An dieser Stelle möchte ich mich auch beim *Herrn Prof. Dr. Hans-Ulrich Heiß* für die Begutachtung dieser Masterarbeit bedanken.

Mein Dank gilt ebenfalls dem gesamten Team des Fachgebietes Komplexe und Verteilte IT-Systeme der Technischen Universität Berlin, die mir stets das notwendige Equipment für eine effektive Bearbeitung dieser Arbeit bereitgestellt haben.

Ein besonderer Dank richtet sich an *Dr. André Höing* und *Dr. Matthias Hovestadt*, die meine Arbeit betreut haben. Ihre vorbildliche und kontinuierliche Unterstützung hat wesentlich zum Gelingen dieser Masterarbeit beigetragen.

Meinen Dank möchte ich ebenfalls an allen Mitgliedern des DLR-Institutes für Verkehrssystemtechnik, die mich weitestgehend unterstützt und beraten haben.

Besonderer Dank geht an Herrn *Dipl.-Ing. Louis Calvin Touko Tcheumadjeu*, meinem fachlichen Betreuer am Institut. Ich danke ihm für die vielseitige Unterstützung, seine Geduld und sein Engagement.

Zu guter Letzt möchte ich mich bei meiner gesamten Familie für deren volle Unterstützung bedanken.

Zusammenfassung

Ein moderner Einsatz zur Realisierung von flexiblen IT Landschaften sind Service-orientierte Architekturen (SOA). Hierzu wird ein Enterprise Service Bus (ESB) als eine technische Ausprägung betrachtet. Heutzutage wird ein ESB entweder unabhängig bei einem Rechner, oder auch bei Rechnerfarmen eingesetzt, das jedoch nicht elastisch ist. Daher ist eine schnelle und einfache Skalierbarkeit nach oben oder nach unten nicht möglich. Aufgrund dessen erscheint der Gedanke an ein Konzept zur Integration eines ESB in einer elastischen infrastructure-as-a-service Umgebung als notwendig.

Eine mögliche Umsetzung dieses Konzepts setzt eine Clusterbarkeit des ESB voraus. Die Anzahl der ESB-Cluster-Mitglieder, wird in Abhängigkeit zum Ressourcenverbrauch, elastisch hoch und runter skaliert. Damit lassen sich signifikante und optimale Auslastungen der Ressourcen (Server...) erzielen und die Kundenanfragen auf allen ESB-Instanzen gleich gewichtig verteilen. Durch diese Elastizität kann man flexibel auf Lastspitzen reagieren und schnelle Antwortzeiten gewährleisten.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Auf einem sich stark ändernden Markt, müssen Unternehmen zur Sicherung ihrer Existenz konkurrenzfähig sein. Hierzu gehört u.a. eine flexible Reaktion und verantwortungsbewusstes Handeln in Bezug auf Kundenbedürfnisse. So kommt es i.d.R. bei besonderen Ereignissen oder auch bei einer Vergrößerung des Unternehmens z.B. durch eine Fusion mit mehreren Firmen, zu einem drastischen Anstieg der Kundenanfragen. Hierbei kann es zu variablen Lastprofilen mit Lastspitzen, z.B. zu Weihnachten oder abends, wenn alle Feierabend haben, kommen. In solchen Fällen müssen die Unternehmen die *Quality-of-Service* kontinuierlich beibehalten können und die an die IT-Systeme gestellten Anforderungen wie Kosteneffizienz und kurze Antwortzeiten, erfüllen.

Dies stellt eine enorme Herausforderung für Unternehmen dar, so dass auf die Erweiterung der IT-Infrastruktur zurück gegriffen werden muss. In Bezug auf die Hardware müssen beispielsweise die Anzahl der zum Einsatz kommenden Server erhöht werden, die Qualität der Rechner verbessert, das Netzwerk und dessen Bandbreite vervielfältigt und wenn nötig, in Data-Center oder in Supercomputer investiert werden. Bei der Software hingegen, müssen die IT-Systeme über besondere Eigenschaften verfügen. Besonders wesentlich erscheint die Fähigkeit auf mehreren Rechnern gleichzeitig und synchron arbeiten zu können [1].

Um erfolgreich auf solche Szenarien einwirken zu können, ist das Vorhandensein von Eigenschaften wie Lastverteilung und Clusterbarkeit erforderlich. Zudem trägt eine flexible und automatische Verteilung der Software zu Kostenersparnissen bei. Außerdem ermöglichen Überwachungsansätze wie Monitoring und Logging die Beobachtung des Softwareverhaltens und demzufolge die rechtzeitige Entdeckung und

[1] In diesem Zusammenhang sind zahlreiche Aspekte der verteilten Systeme enthalten, die im weiteren Verlauf vorgestellt werden

schnelle Behebung von Störungen.

Für die Realisierung von flexiblen IT Landschaften, ist der Einsatz von Architekturen wie service-oriented-architecture (SOA) empfehlenswert. Als mögliche technische Ausprägung für SOA kommt der Enterprise Service Bus (ESB) in Betracht. Für den Einsatz vom ESB stehen verschiedene Szenarien zur Verfügung. So kann dieser unabhängig bei einem Rechner eingesetzt werden, bei Rechnerfarmen oder auch auf einem physikalischen LAN [2] geclustert werden. Beim letzten Szenarium, laufen mehrere Instanzen vom ESB auf viele Rechnern und arbeiten zusammen, um eine Aufgabe zu erledigen. Allerdings ermöglichen alle diese Einsatzszenarien nicht eine elastische Skalierbarkeit des ESB-Clusters, in der die Anzahl an ESB-Instanzen nach oben oder unten variiert werden.

1.2 Ziele

Ausgehend von dieser Problematik und um dem Ziel -Integration eines ESB in elastischen Infrastructure as a Service Umgebung- näher zu kommen, werden im Rahmen dieser Masterarbeit folgende Aspekte untersucht werden:

1. Es wird untersucht, wie ein Enterprise Service Bus in einer *infrastructure-as-a-service* integriert werden kann. Hierzu sollen zunächst die Anforderungen an dieser Integration festgelegt werden. Danach müssen eine Darbietung und ein Vergleich der möglichen Szenarien erfolgen.
2. Es sind die Voraussetzungen, die eine elastische Skalierbarkeit des Enterprise Service Bus nach oben oder nach unten ermöglichen, zu definieren.
3. Die Erarbeitung eines Konzepts zum automatischen Abfangen potentieller Lastspitzen in einer *infrastrucutr-as-a-service (iaas)* Umgebung ist notwendig.
4. Es erfolgt die Entwicklung eines SOA-mäßigen Prototyps für die FCD-Kette der DLR-Traffic-Data-Platform.
5. Es muss die Prototypische Umsetzung des Konzepts zum automatischen Abfangen der Lastspitzen in einer iaas Umgebung erfolgen.
6. Abschließend ist die Simulation einer Überlast beim FCD-Prozessierungsmodul durchzuführen.

[2] Local Area Network

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen näher erörtert. Zunächst werden die notwendigen Begriffe von service-oriented-architecture erläutert und dann die grundlegenden Konzepte eines Enterprise Service Bus geschildert. Zudem werden die bekanntesten OpenSource Produkte vom ESB vorgestellt. Danach erfolgt die Darstellung der wesentlichen Merkmale vom JBoss ESB wie Clustering und Dienstverwaltung, die für den weiteren Verlauf dieser Arbeit relevant sind. Anschließend wird das Cloud Computing präsentiert und ausführlicher auf die Infrastructure-as-a-Service (IaaS) eingegangen. Zu guter Letzt wird ausgehend von den vorgestellten OpenSource ESBs einige verwandte Arbeiten vorgestellt und analysiert.

Unter Kapitel 3 werden die vorhandenen Möglichkeiten zur Realisierung eines ESB-Clusters analysiert, verglichen, und darauf basierend ein eigenes Konzept erstellt, dass alle an das System gestellten Anforderungen herleitet, um ein entsprechendes Verfahren für die Integration eines ESB in einer elastischen *iaas* Umgebung zu erstellen. Danach wird die Strategie zum automatischen Abfangen von Lastspitzen einer SOA-mäßigen Anwendung in einer *iaas* erarbeitet.

Unter Kapitel 4 erfolgt die technische Umsetzung des unter Kapitel 3 entwickelten Konzepts. Zunächst werden die notwendigen Konfigurationen für das Deployment eines JBoss ESB-Clusters erläutert. Anschließend folgen die technischen Details zur Integration eines ESBs in einer elastischen *iaas* und deren Verknüpfung mit einem Cluster-ESB. Zudem wird die Implementierung von einem Lastüberwachung-Framework verdeutlicht.

Im Rahmen des Kapitels 5, erfolgt zu Beginn eine kurze Vorstellung des DLRs und des Instituts für Verkehrssystemtechnik. Anschließend werden die Traffic-Data-Platform und das Floating-Car-Data-Prozessierungsmodul (FCD) erörtert. Daraufhin folgt die Gestaltung und Umsetzung eines auf SOA basierenden Systems des FCD-Prozessierungsmoduls. Anschließend wird eine Überwachung der FCD-Kette durchgeführt, beobachtet sowie analysiert und gegebenenfalls bei vorhandener Überlast Lastspitzen automatisch in einer *iaas* abgefangen.

Abschließend werden unter Kapitel 6 die erzielten Ergebnisse zusammengefasst sowie ein Ausblick auf weiterführende Entwicklungsmöglichkeiten gegeben.

Kapitel 2

Grundlagen

Im vorliegenden Kapitel werden die Grundlagen näher erörtert. Zunächst werden die notwendigen Begriffe von service-oriented-architecture erläutert und dann die grundlegenden Konzepte eines Enterprise Service Bus geschildert. Zudem werden die verschiedenen OpenSource ESB Produkte, JBoss-, Mule- und Sun Open ESB vorgestellt, von denen JBoss ESB als Kandidat für den weiteren Verlauf der Analyse gewählt und genauer unter die Lupe genommen wird. Anschließend wird der Begriff Cloud Computing eingeführt, definiert und Vor- und Nachteile erörtert. Danach wird ausführlicher auf die Infrastructure-as-a-Service (IaaS) eingegangen. Abschließend werden verwandte Arbeiten betrachtet.

2.1 Service Orientierte Architekturen

2.1.1 Definition und Prinzipien

Der Begriff SOA hat in den letzten Jahren sehr viel Aufmerksamkeit im Bereich des Softwaredesigns auf sich gezogen. Eine Serviceorientierte Architektur (engl. *service-oriented-architecture*) wird definiert als

> "*eine Anwendungsarchitektur, in der alle Funktionen als unabhängige Services mit wohldefinierten, aufrufbaren Schnittstellen vorliegen, so dass eine Auswahl -in einer sinnvollen Reihenfolge aufgerufen- einen Geschäftsprozess abdecken*" [RB07a].

Das SOA-Referenzmodell des Standardisierungsgremiums OASIS [1] definiert SOA als

> "*ein Paradigma für die Organisation und Verwendung verteilter Fähigkeiten,*

[1] OASIS: Organization for the Advancement of Structured Information Standards

die unter der Kontrolle verschiedener Besitzerdomänen stehen können"
[ELMT09].

Durch die Separation von datenorientierten Geschäftsdiensten, -prozessen, -regeln und Integrationslogik werden die Flexibilität und die Wiederverwendbarkeit von Diensten ermöglicht. Außerdem wird durch die asynchrone Kommunikation die lose Kopplung (Entkopplung) ausgelöst, welches ein wesentliches Merkmal der SOA darstellt [Fin]. Zudem bestehen weitere zahlreiche Prinzipien und Aspekte, wie Erweiterbarkeit, Skalierbarkeit, Verteilbarkeit, Komponierbarkeit, Wartung sowie das Anbieten, Suchen und Nutzen von Diensten, die das SOA-Profil ausmachen.

2.1.2 Dienst und SOA-Referenzarchitektur

Ein Dienst stellt die Kernkomponente einer SOA dar. Es kapselt eine Funktion ab und besitzt eine wohldefinierte Schnittstelle. Ein Dient wird folgendermaßen definiert:

"*In der Serviceorientierten Architektur (SOA) wird ein Dienst bzw. Service als eine Software-Komponente bezeichnet, die eine wohl definierte Funktionalität über eine standardisierte Schnittstelle anderen Services oder Anwendungen zur Verfügung stellt*"

[RB07a]. Die SOA stellt eine Vielzahl voneinander unabhängiger, lose gekoppelter Dienste dar. Die SOA-Referenzarchitektur baut auf drei Säulen auf und kann als Dreieck dargestellt werden (vgl Abbildung 2.1).

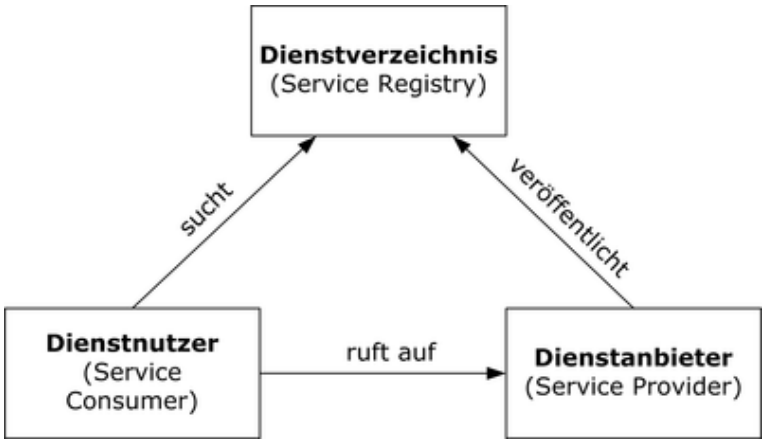

Abbildung 2.1: Dreieck SOA-Referenzarchitektur
[Góm10]

Dienste werden von Dienstanbieter (engl. *service provider*) in einem Dienstverzeichnis (engl. Service Registry) veröffentlicht. Ein Dienstnutzer (engl. *service consu-*

mer) fragt stetig beim Dienstverzeichnis nach einem Service und bekommt dementsprechend die notwendigen Informationen (Adresse, Policy...). Daraufhin stellt der Client eine direkte Anfrage (engl. service request) an diesem Dienst und bekommt die entsprechende Antwort (service response) vom Anbieter. Diese Operationen werden als *Publish-Find-Bind-Execute* Modell bezeichnet [Wik10a].

2.1.3 Komposition von Diensten

Eine der hervorragendsten Fähigkeiten von SOA stellt die Service-Komposition dar. Dank der losen Kopplung, standardisierten und wohldefinierten Service-Schnittstellen, können Dienste in beliebigen Geschäftsprozessen integriert werden. Diese können entweder atomare oder aber auch komplexe Dienste sein. Somit bilden sich neue komplexe Dienste mit standardisierten Schnittstellen. Die Service-Komposition wird häufig in EAI [2] und B2B-Integration [3] Szenarien eingesetzt. Defaultmäßig sind zwei Kategorien der Service-Komposition vorhanden [4]:

- Orchestrierung (engl. Orchestration): Bei dieser Kategorie wird diese Komposition durch einen Koordinator (engl. Orchestrator) gesteuert. Mehrere Services werden in einem Geschäftsprozess integriert. Die Reihenfolge der Ausführung wird vom *Orchestrator* bestimmt [RB07c].
- Choreographie (engl. Choreography): In diesem Szenarium werden die Sequenz und die Bedingung definiert, unter denen mehrere kooperierende unabhängige Dienste Nachrichten austauschen, um eine Aufgabe auszuführen und dadurch ein Ziel zu erreichen [IMC05].

2.1.4 Technische Umsetzung

Es können unterschiedliche Technologien bei der Umsetzung von SOA zum Einsatz kommen. In Betracht kommen u.a. *.Net, CORBA* [5]*, XML-RPC, Web Services* [RB07b]. I.d.R wird SOA überwiegend mit Web-Services verbunden. Da SOA in den meisten Fällen in heterogenen, verteilten Umgebungen eingesetzt wird, ist die Nutzung einer *Integrationslösung* von großer Bedeutung. Allgemein spricht man von einem *Software-Bus*, der die Integration zwischen verteilten, nicht kompatiblen Anwendungen anstrebt. Ein Beispiel hierfür ist der *Enterprise Service Bus (ESB)*, der die technische Infrastruktur einer SOA auszeichnet.

[2] Enterprise Application Integration
[3] Business-to-business
[4] In der Realität gibt es auch eine andere Kategorie: Konversation, die aber selten verwendet wird
[5] CORBA ist eine Middleware zur Kommunikation zwischen Anwendungen, unabhängig von den verwendeten Programmiersprachen, Hardware sowie Software und Netzwerken.

2.2 Enterprise Service Bus

2.2.1 Definition und Aufgaben

Ähnlich wie bei einem Hardware-Bus, der die Integration der Hardware von verschiedenen Herstellern ermöglicht [KBS07], strebt ein ESB nach standardisierten Vorgehensweisen, Softwarekomponenten lose miteinander zu koppeln [BHR07]. Ein ESB wird auch als eine Kombination von traditionellen Middleware Technologien, XML und Webservices betrachtet [WQH+08].

In einer Vergleichstudie der AncudIT [6] wird ein ESB folgendermaßen definiert:

> "*Unter einem Enterprise Service Bus (ESB) versteht man ein Softwareprodukt zur Unterstützung der Integration verteilter Dienste in der heterogenen Anwendungslandschaft eines Unternehmens(...) Der ESB erlaubt, einmal erstellte Funktionalitäten von Diensten für andere Aufgaben wieder zu verwenden. Dadurch verringert sich sukzessive der Entwicklungsaufwand bei der Erstellung weiterer Dienste im Sinne einer Serviceorientierten Architektur (SOA). Ein ESB fungiert also eine Art Dolmetscher zwischen Diensten verschiedener Hersteller, die ggf. unter Verwendung verschiedenster Technologien realisiert wurden. Der ESB sorgt für die reibungsfreie Kommunikation zwischen den Diensten, idealerweise sollten hierfür keine Änderungen an den Diensten der verschiedenen Anwendungen selbst nötig sein*"

[Scm10]

Zu den wesentlichsten Aufgaben eines ESB gehören das (intelligente) *Routing* von Nachrichten unter Verwendung eines generischen Kommunikationsbusses über alle Anwendungen- und Herstellergrenzen hinweg, die Transformation von Nachrichten in unterschiedlichen Formate, sowie das Bereitstellen von verschiedenen Nachrichtenprotokollen und Routing-Mechanismen [Scm10]. Zudem soll ein ESB auf die Verteilung und die Skalierbarkeit ausgelegt sein, um sich einer ständig wachsenden Anzahl von IT-Systemen und Anwendungen anpassen zu können [BHR07].

2.2.2 ESB Architekturen

Bei einem ESB können unterschiedliche Architekturen zum Einsatz kommen. Im Allgemeinen wird zwischen den zwei folgenden Architekturen differenziert:

- *Standard Architektur*: Bei dieser wird die *Java Business Integration(JBI)* als

[6] Ancud IT-Beratung GmbH: `http://www.ancud.de/`

Referenz-Spezifikation angenommen. Zu dieser Gruppe gehören u.a. Produkte wie *Sun Open ESB* [7], *Apache ServiceMix*, *Petals Service Platform*.

- *Proprietäre Architektur*: Wie der Begriff *Proprietär* schon hindeutet, besitzt jedes ESB-Produkt seine eigene Architektur. Zu dieser Kategorie gehören u.a. *JBossESB*, *WSO2 ESB*.

2.2.2.1 Standard Architektur

Der *Enterprise Service Bus* folgt der *Java Business Integration(JBI)*. JBI ist eine Spezifikation, die unter *Java Communtiy Process (JCP)* für ein Konzept zur Umsetzung einer Serviceorientierten Architektur entwickelt wurde. Das *JCP JSR 208 ist Referenz für JBI 1.0* und *JSR 312 für JBI 2.0* [Wik10b]. Diese Spezifikation definiert einen Standard für eine Integrationsplattform und basiert auf einem lose gekoppelten Integrationsmodell, das den Aufbau einer Integrationsplattform erlaubt [Tro05].

Jeder JBI-basierter ESB besteht allgemein aus folgenden Komponenten [THW05]:

- `Normalized Message Router (NMR)`: Dieser fungiert als Brücke zwischen den anderen JBI Komponenten, den Binding-Components (BC) und den Service-Engines (SE). Der NMR ist zuständig für das Vermitteln (engl. Routing) der zu bearbeitenden Nachrichten zur Zielkomponente.
- (eine oder mehrere) `Binding-Components (BC)`: Diese dienen als Adapter, damit die Inkonsistenz zwischen den Partnern (In-und Out ESB) auf Kommunikationsebene beseitigt wird und streben nach einer einheitlichen Nachrichten-Schnittstelle für den NMR.
- (ein oder mehrere) `Service Engine (SE)`: Ein Service-Engine ist *the business logic driver* eines JBI-Sytsems [THW05]. Es kann einfache Dienste wie XSLT-Transformation bei XSLT-Service Engine anbieten oder komplexe Aufgaben wie Service Komposition bei einem BPEL-Service Engine übernehmen.
- `Management Modul (MM)`: Für die Administration der JBI Node und deren Komponente (BCs und SEs) werden verschiedene Typen von Management-Beans (MBean) definiert. Das *Management Modul* stellt Schnittstellen zur Installation von SEs und BCs bereit. Zudem kümmert sich dieser um die *Life-Cycle-Management* der Komponenten (Start/Stop). Außerdem ermöglicht dieser das Deployment von *Componenten-Artifacts* in vorhandenen SE und BC. Ein Beispiel hierfür ist die Installation neuer XSLT [8] Style Sheets in einer XSLT-Service Engine [WQH+08].

[7] derzeit auch Glassfish ESB genannt
[8] Extensible Stylesheet Language Transformation

- External *JMX-based Admin Tools*: Ein *Remote JMX-based Client* ist für die Kommunikation mit dem Management Modul zuständig, der somit einen ortsunabhängigen Zugriff auf dem MM ermöglicht.

Abbildung 2.2 gibt einen zusammenfassenden Überblick über alle Komponenten eines JBI Systems in einem *Top-Level View*.

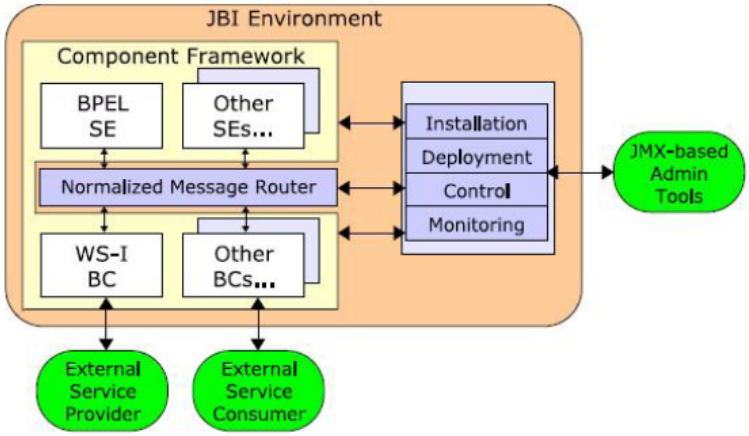

Abbildung 2.2: Ansicht einer JBI Architektur
[THW05]

2.2.2.2 Proprietäre Architektur

In diesem Kontext hat jeder *ESB* seine eigene Architektur. Allerdings existieren Komponente, die denen der Standard-Architektur ähneln. Hierbei befindet sich bei jedem Produkt eine *Messaging-Router* Komponente, die dieselben Aufgaben eines *NMR* erfüllt. Es sind sowohl *Engines* zur Nachrichten-Transformation als auch *Adapter* zur Unterstützung unterschiedlicher Transportprotokolle vorhanden.

2.3 Darbietung von bekannten OpenSource ESB Produkten

Die Auswahl an verfügbaren OpenSource Enterprise Service Bus Systemen ist sehr groß, weshalb sich die Darbietung auf die folgenden drei bekannten ESB Produkte beschränkt:

- JBoss ESB
- Mule ESB

- Sun Open ESB

2.3.1 JBoss ESB

JBoss ESB JBoss ESB ist ein Produkt der Firma JBoss, die von Redhat im Jahre 2006 übernommen wurde. Zu Beginn wurde es als Rosetta ESB von Aviva Canada auf den Markt gebracht [Rüc08]. JBoss ESB unterstützt zahlreiche Transportprotokolle wie HTTP, JMS, Socket... Ab dem *Release 4.2* ist eine Distribution von JBoss ESB-Instanzen über mehrere (physikalische oder virtuelle) Knoten möglich (Clustering-Aspekt von ESB). Die Architektur vom JBoss ESB wird durch die Abbildung 2.3 verdeutlicht.

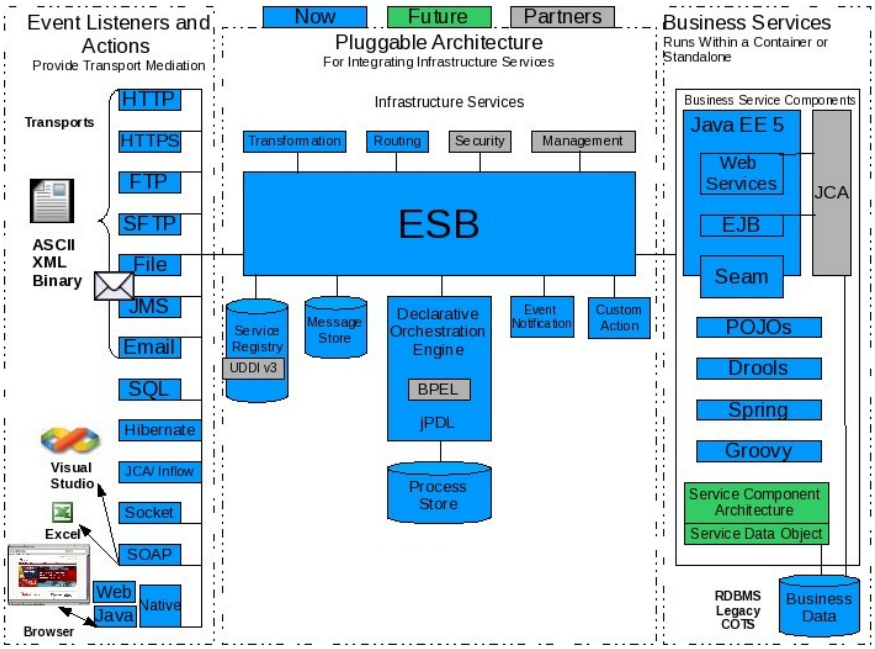

Abbildung 2.3: Architektur von JBoss ESB
[Rüc08]

2.3.2 Mule ESB

Mule ESB ist ein Produkt der Firma MuleSoft. Nach eigenen Angaben, bezeichnet sich, als der weltweit meist eingesetzte OpenSource ESB mit mehr als 1.5 Millionen Downloads [Mul10b]. Mule ESB wird bei vielen bekannten Firmen eingesetzt wie z.B *Siemens, HP, Credit Suisse [BHR07]*. Abbildung 2.4 gibt einen Überblick über Mule ESB.

Bei stetig wachsender Anzahl an Services und damit verbundener Auslastungen verhält sich Mule ESB flexibel und stabil. Tatsächlich wird dieser ESB nach dem Modell *SEDA (staged event-driven Architecture)* aufgebaut [BHR07]. Diese Architektur ermöglicht es, eine große Anzahl an gleichzeitigen Verbindungen zu bewältigen, indem der ESB in einzelnen Teilen (sog. *Stages*) aufgeteilt wird. Diese *Stages* werden durch Message-Queues miteinander verbunden [BHR07]. Somit ist eine hohe Entkopplung und effiziente Skalierbarkeit der Mule-ESB-Komponenten gewährleistet. Mule ESB unterstützt viele Protokolle und kann in verschiedenen Szenarien ein-

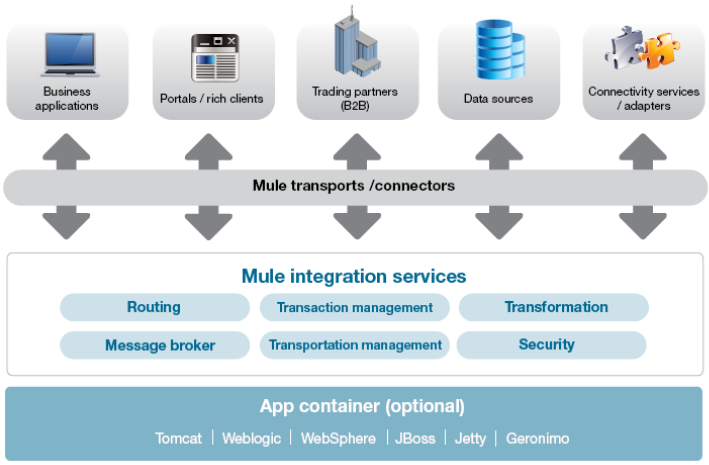

Abbildung 2.4: Mule ESB: Überblick
[Mul10b]

gesetzt werden. Außerdem stehen zahlreiche Adapter für Standard-Softwares zur Verfügung, die jedoch mit einer *kostenpflichtigen-Version* verbunden sind. Allerdings ist eine eigene Entwicklung von Adaptern anhand des *Mule-IDE-Plugins* machbar [BHR07]. Abbildung 2.5 stellt u.a. die unterstützten Transportprotokolle, Frameworks und Webservice-Standards dar.

2.3.3 Sun Open ESB (GlassFish ESB)

Open ESB ist ein Produkt der Firma Sun Microsystems (wurde von Oracle aufgekauft). Es implementiert die *Java Business Integration (JBI)* Spezifikation. JBI ist ein Standard, um einen Enterprise Service Bus mit Hilfe von Java aufzubauen. Das Konzept sieht einen Container vor, der über definierte Plugin-Schnittstellen erweitert werden kann [BHR07].

Open ESB ist eng mit dem *Sun GlassFish Applikation Server* und die *NetBeans IDE* gebunden. Somit ist eine ausführliche und umfangreiche Plattform für die Entwicklung SOA-fähiger Anwendungen bereitgestellt.

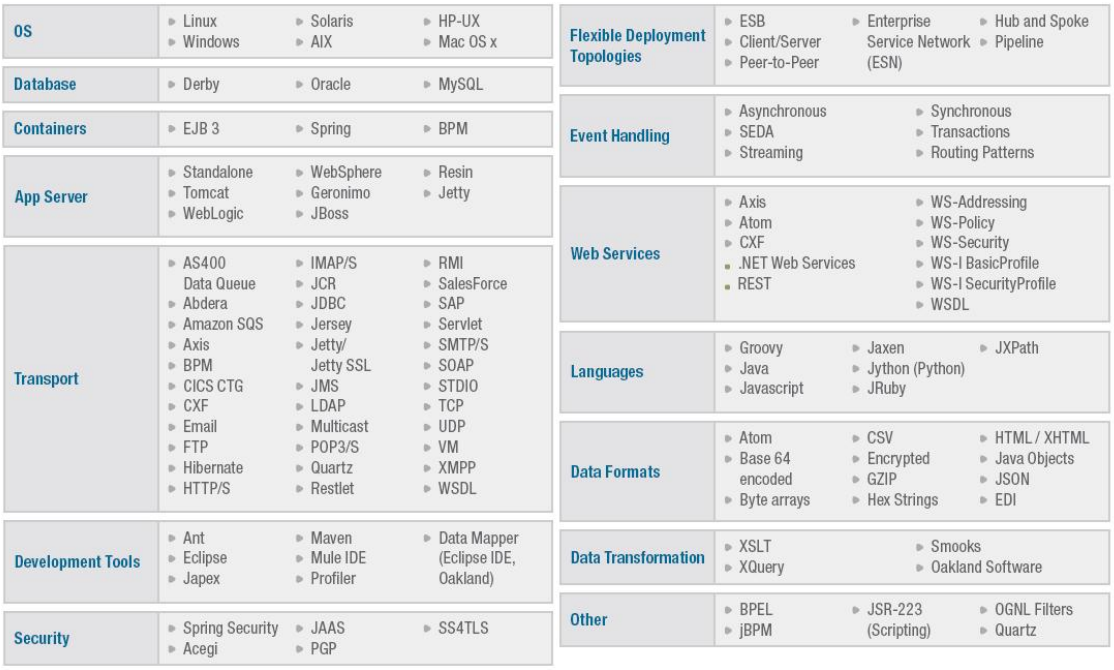

Abbildung 2.5: Technische Spezifikation von Mule ESB
[Mul10b]

Nachfolgend erfolgt eine Aufzählung der Komponenten, die Open ESB verwenden kann:

1. *Binding Components*: U.a. werden folgende *Binding Components* unterstützt:
   - e-Mail BC: Anbindung für das Senden und das Empfangen von E-mails
   - File BC: Anbindung an das Dateisystem
   - HTTP BC: JBI Anbindung für das Senden und das Empfangen von Nachrichten über HTTP Protokolle
   - Database BC: Anbindung für das Lesen (bzw. das Schreiben) von Nachrichten aus (bzw. in) einer Datenbank anhand JDBC
   - SAP BC: Anbindung für SAP
2. *Service Engine*: U.a. werden folgende *Service Engines* unterstützt:
   - BPEL SE: WS-BPEL 2.0 fähige Engine für *Business Process Orchestration*
   - Scripting SE: erlaubt das Scripting und Deployment in ESB
   - Notfication SE: Unterstützung von WS-Notification
   - XSLT SE: XSLT Transformation Engine

- Data Mashup: Aufbau eines Data Mashup Systems

3. *Shared Libraries*: Folgende *Shared Libraries* sind u.a. derzeit verfügbar:
   - sun-encoder-library
   - sun-shared-util-library
   - sun-transform-library
   - sun-wsdl-library

2.4 JBoss ESB Clustering und Dienstverwaltung

2.4.1 Dienstverwaltung in JBoss ESB

In der Welt von JBoss ESB werden lediglich Nachrichten und Dienste definiert. Tatsächlich folgt JBoss ESB eine *Message-driven-Pattern* für die Verwaltung der Interaktion zwischen Kunden und Diensten [mas10]. Diese Interaktion wird als Folge von Anfragen und Antworten durchlaufen. Jede registrierte Service wird benachrichtigt, wenn eine neue Nachricht für sie eintrifft.

Prinzipiell erkennt JBoss ESB zwei Kategorien von Nachrichten an:

- *Externe Nachrichten*: Diese sollen von Systemen, die sich außerhalb vom ESB befinden, geschickt werden. Ein Beispiel hierfür sind die Kundenanfragen.
- *Interne Nachrichten* (auch als *ESB-Aware-Message* bezeichnet): JBoss ESB definiert seinen eigenen Nachrichtenformat für die interne Bearbeitung. Eine *ESB-Aware* Nachricht besteht aus einem `Header`, einem `context`, einem `Body` und aus einem `Attachment`. Der `Body` Teil beinhaltet die nützlichen Daten. Diese werden als `(<keys>-<values>)` Mengen dargestellt. Abbildung 2.6 gibt einen allgemeinen Überblick über den Aufbau einer *ESB-Aware* Nachricht.

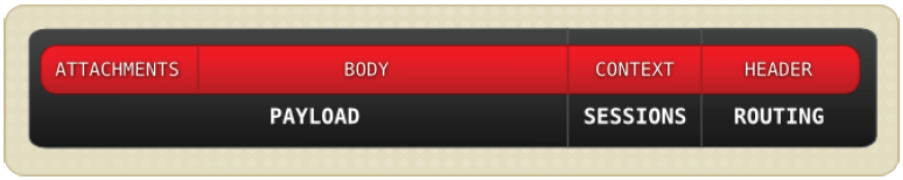

Abbildung 2.6: Überblick über den Aufbau einer ESB-Aware-Message [JBo10a]

Intern besteht ein Dienst aus einer Kette von Aktionen, die die ankommenden Nachrichten nacheinander bearbeitet. Die Kette wird standardgemäß als `Action-Pipeline` bezeichnet [mas10]. Die Logik einer Aktion hängt von der Implementierung ab. So

kann zum Beispiel eine Aktion für die Transformation des Nachrichtenformats von CSV zu XML zuständig sein, während eine andere für die Aktualisierung einer Datenbank, beispielsweise durch das Hinzufügen neuer Daten, verantwortlich ist. Abbildung 2.7 zeigt ein solches Beispiel für eine `Action-Pipeline`.

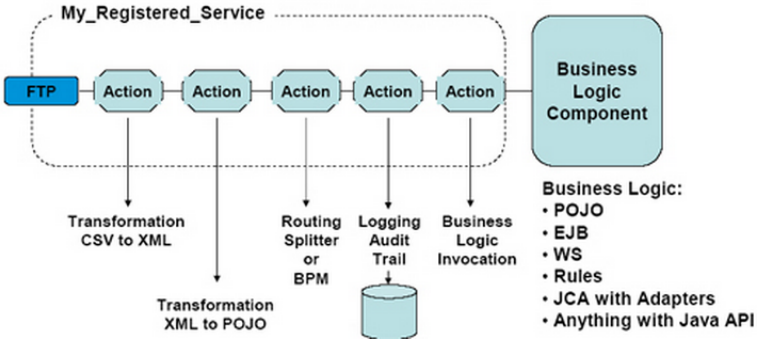

Abbildung 2.7: Beispiel einer Actions-Pipeline
[mas10]

Jedem Dienst steht eine Nachritenwarteschlange (engl. message-queue) zur Verfügung. Diese werden vom *Messaging-System* des JBoss Application Servers (JBoss AS) zur Nutzung bereitgestellt. In diesen werden die eintreffenden Nachrichten erstmals gespeichert, bevor sie bearbeitet werden, um die Überlastung von Diensten -soweit wie möglich- zu verhindern. Sollten freie Ressourcen ausreichend zur Verfügung stehen, dann wird die Nachricht bearbeitet.

Ein Dienst kann beliebig viele `Listeners` definieren. Diese stehen als *Inbound-Router* für den Dienst zur Verfügung [JBo10a] und kümmern sich um das Routen von ankommenden Nachrichten an die `Action-Pipeline`. Diese `Listeners` werden -einmal der Service in ESB deployed - als *Endpoints* in der vom JBoss ESB bereitgestellten *Registry* gespeichert.

JBoss ESB unterscheidet zwischen zwei Kategorien von `Listeners` [JBo10a]:

- `Gateways Listeners`: Diese stellen `Gateway Endpoints` bereit und sind zuständig für die Normalisierung von *externen Nachrichten* durch deren Transformation (engl. Wrapping) zu einer *ESB-Aware* Nachricht.
- `ESB-Aware Listeners`: Diese stellen `ESB-Aware Endpoints` bereit, die dem Austausch von ESB-Aware Nachrichten zwischen den ESB-Aware Komponenten dienen.

Da Dienste nur *ESB-Aware*-Nachrichten bearbeiten können, benötigen diese für die Bearbeitung von Nachrichten, die Außenseiters des JBoss ESBs ankommen, `Gateways`. Abbildung 2.8 stellt einen Beispiel von Interaktion zwischen einen *ESB-Service* und eienem *JMS*-Klient.

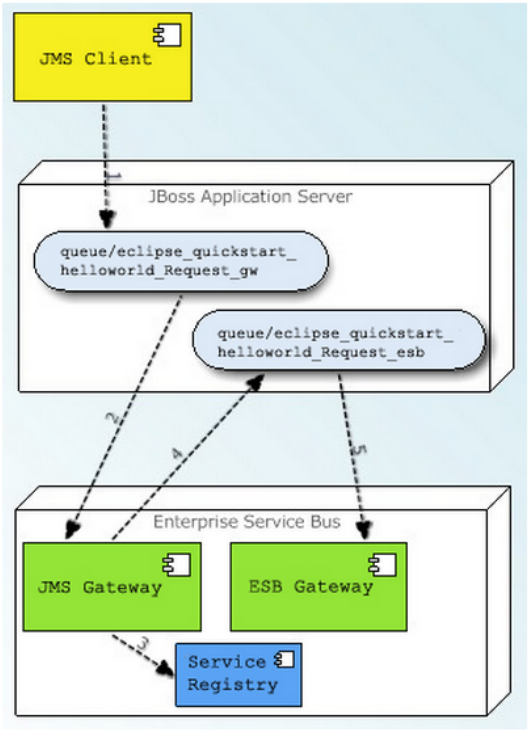

Abbildung 2.8: Interaktion zwischen einen ESB-Dienst und einen JMS-Klient [mas10]

2.4.2 JGroups und Clustering in JBoss AS

Anforderungen wie Ausfallsicherheit, Lastverteilung und Skalierbarkeit können durch *Cluserting* entsprochen werden. Im Fall des JBoss Application Servers (JBoss AS), besteht ein *Cluster* aus einer Menge von untereinander vernetzten JBoss AS, die als ein Computer angesehen werden können. Jeder Cluster-Instanz kennt den Zustand von anderen Mitgliedern. Wenn ein Cluster-Mitglied ausfällt, können andere Mitglieder die noch offenen Aufgaben übernehmen [Möl10]. Ein Cluster ist meist für den Klient transparent, d.h. er weiß nicht ob die eingesetzte Anwendung geclustert oder ungeclustert ist [Möl10].

Zu den Nachteilen eines Clusters gehört eine nicht lineare Skalierbarkeit. Hierbei wird mit zunehmender Anzahl der Knoten auch der Verwaltungsaufwand höher, was sich negativ auf die Performance der geclusterten Anwendung auswirkt [Möl10].

Prinzipiell sind drei Kategorien des Clusterings vorhanden [Möl10]:

- *Vertikaler Cluster*: Bei diesem Typ laufen alle JBoss AS-Instanzen auf derselben Maschine, welche physischer oder virtueller Natur sein kann.
- *Horizontaler Cluster*: Bei diesem Clustertyp werden die Cluster-Mitglieder auf unterschiedliche Maschinen verteilt.

- *Misch-Cluster*: Dieser stellt eine Kombination der beiden oben genannten Clustertypen.

Die Cluster-Kommunikation soll mittels JavaGroups (auch JGroups genannt) Framework erfolgen, welche eine zuverlässige *Multicast-* (bzw. Unicast) Kommunikation und *Auto-Discovery* anbietet [Möl10]. JavaGroups ist ein *Java-basiertes-Toolkit* für zuverlässige *Gruppenkommunikation*. Dieses Toolkit ermöglicht das Senden und Empfangen von Nachrichten von und zu allen Gruppenmitglieder. Zudem sichert dieser, dass alle Knoten die gleichen Nachrichten-Sequenzen in der gleichen Reihenfolge erhalten [Ban10a].

JGroups definiert zwei Hauptkomponenten und zwar:

- *Channels*: Diese ähneln den *BSD Sockets*. Beim Verbinden mit einem *Channel* gibt jedes Mitglied den Namen der Gruppe an, an die er sich anschließen will, um Nachrichten senden und empfangen zu können. Zudem wird diese über den Status aller Clustermitglieder benachrichtigt, das bedeutet, diese hat einen Überblick darüber wer gerade an der Gruppe angeschlossen ist und wer nicht [Ban10a].
- *Protocol Stack*: Dieser besteht aus mehreren Schichten, von denen jede ein Protokoll abbildet, welches nicht zwangsläufig auch ein Netzwerktransportprotokoll sein muss [Möl10]. Wenn eine Nachricht versendet wird, wandert diese beim Sender den *Protocol-Stack* runter und beim Empfänger den *Stack* wieder rauf [Möl10].

Abbildung 2.9 zeigt die Architektur der JGroups.

2.5 Cloud Computing und IaaS

2.5.1 Cloud Computing: Definition, Eigenschaften und Einsatzszenarien

Werden zehn unterschiedliche IT-Experten nach der Definition von Cloud Computing gefragt, bekommt man zehn unterschiedliche Antworten. Das liegt daran, dass Cloud Computing in unterschiedlichen Einsatzszenarien verwendet werden kann. Für den Begriff Cloud Computing gibt es derzeit noch keine eindeutige Definition, sondern viele zahlreiche Erklärungsmodelle. Einige vertreten die Ansicht, dass Cloud Computing keine neue Erfindung der IT-Welt ist, sonder viel eher ein Sammelbegriff für verschiedene Dienstleistungen, die über das Internet beansprucht werden können [htt10]. Andere wiederum betrachten, dass Cloud Computing als eine Weiterentwicklung bekannter Computing-Modelle: (Grid Computing - Utility Computing - Application Serv. Providing (ASP) - Cloud Computing) [Emb09]. Das *NIST*

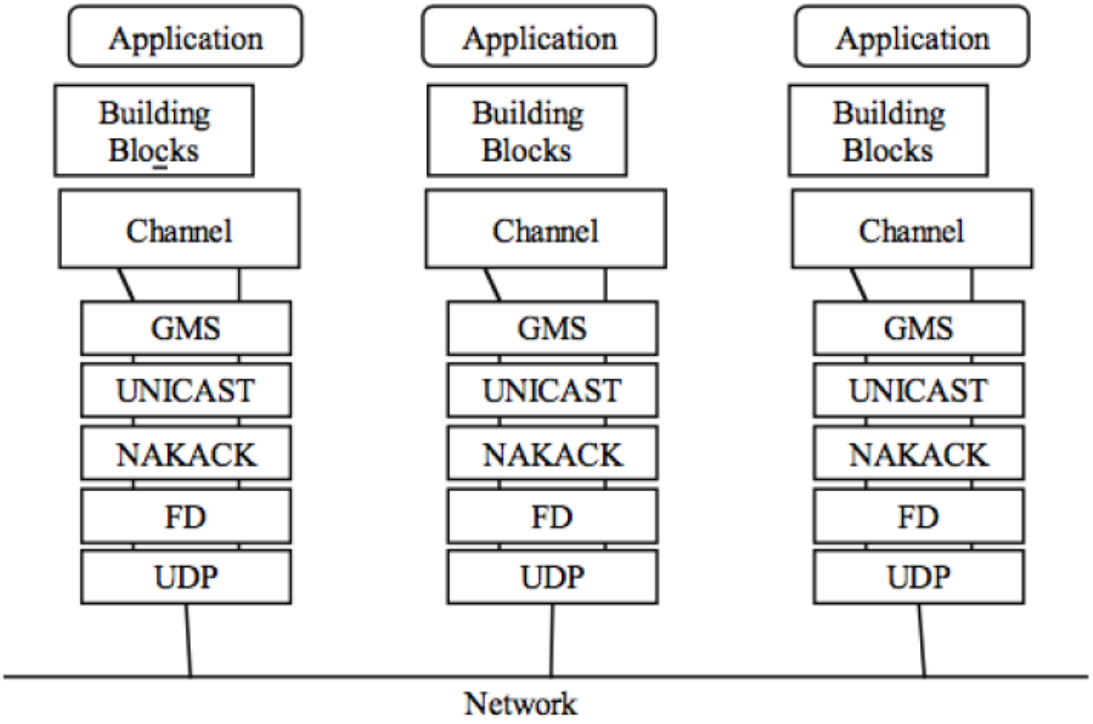

Abbildung 2.9: JGroups-Architektur
[BGPS]

[9]*-Information Technology Laboratory* stellt die folgende allgemeine standardisierte Definition [10] von Cloud Computing bereit:

> "*Cloud Computing is a pay-per-use model for enabling available, convenient, on-demand network access to a shared pool of configurable computing ressources (e.g.,networks, servers, storage, applications,services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [S.L09].

Diese Definition enthält die fünf folgenden Schlüsseleigenschaften [S.L09]:

1. Auf Wunsch Bedienung (engl. *On-demand self-service*): Diese Eigenschaft entspricht der Fähigkeit der Nutzung von Rechnerressourcen ohne menschliche Interaktion mit dem Serviceprovider.

2. Allgegenwärtiger Netwerkzugang (engl. *Ubiquitous network access*) : Diese

[9]NIST: The National Institute of Standards and Technology

[10]Eine andere Definition laut IBM ist: *Cloud Computing ist eine Form der IT-Nutzung, bei der Endbenutzer Applikationen, Daten und IT-Infrastrukturkomponenten in Form von Services über ein Web nutzen und eine große Anzahl virtualisierter IT-Ressourcen so steuern können, dass sie wie eine einzige IT-Umgebung erscheinen.* [Emb09].

Komponente ermöglicht Es ist mit Hilfe von standardisierten Mechanismen möglich von überall auf Ressourcen zuzugreifen. Der Zugriff auf Ressourcen ist von überall anhand standardisierte Mechanismen möglich.

3. Aufenthaltsunabhängige Nutzung der Ressourcen (engl. *Location-independant ressource pooling*): Der Aufenthaltsort der Cloudressourcen ist für den Konsumenten transparent.

4. Schnelle Elastizität (engl. *rapid elasticity*): Diese Eigenschaft ermöglicht die schnelle Bereitstellung der Ressourcen sowie die elastische Skalierung nach oben (engl. scale up) oder nach unten (engl. scale down).

5. Die verbrauchorientierte Bezahlung (engl. *Pay-per-use*): Vorliegend wird die Benutzung von Ressourcen mit Hilfe von metered Services durchgeführt.

Cloud Computing wird häufig u.a. in Szenarien eingesetzt, die eine schnelle Reaktion auf möglichen Änderungen an den Geschäftsprozess, benötigen. Hierzu bildet die IT-Infrastruktur meistens, im Fall von z.B. einer Vergrößerung des Unternehmens durch eine Fusion, eine *Singel-Point-Of-Latency* [11]. Während die Wartung und die Erweiterung der IT-Infrastruktur und der Enterprise-Infrastruktur immer schwer und manchmal unmöglich sind, ändern sich die *Geschäftsprozesse* kontinuierlich, weil sie von den *Upturns and dowturns* der Wirtschaft beeinflusst werden [S.L09]. In diesem Fall bietet Cloud Computing eine hervorragende Lösung durch den Einsatz von *On-Demand* Ressourcen auf allen möglichen Ebenen (IT-Infrastruktur, Software, Prozesse...) an.

2.5.2 Cloud Computing: Vor- und Nachteile

Zu den Vorteilen von Cloud Computing zählen das Reduzieren von Kosten, da lokal der Einsatz von Hardware und Software geringer ausfällt. Andere Firmen, die häufig Cloud Providers genannt werden, sorgen für die Bereitstellung von Rechenzentren, *Storage Devices* , Netzwerk Ressourcen und kaufen Lizenzen für die notwendigen Softwares ... Pflichtgebundene Verträge sind nicht mehr notwendig um Dienstleistungen von Rechenzentren in Anspruch nehmen zu können. Stattdessen werden neue Abrechnungsmodelle, wie Bezahlung nach Nutzung von Services (*Pay-as-you-go*) eingeführt: bezahlt wird nur was tatsächlich beansprucht wird. IT-Abteilungen können sich hierdurch eher auf ihre strategischen Projekte fokussieren, anstatt deren Rechenzentren laufen zu lassen um eingebundene Probleme zu beseitigen [TJE09]. Zudem werden durch Cloud Computing neue Arbeitsformen wie Heimarbeit und Telearbeit ermöglicht [htt10].

[11] Singel-Point-of-Latency: die Unfähigkeit einer IT-Infrastruktur auf einer schnelle Reaktion auf die Änderungen des Businessprozess, wobei alle anderen Abteilungen z.B *sales executives, HumanRessourcen ...* auf solche Änderungen flexible sind. Siehe [S.L09]

Nichtsdestotrotz erschweren viele Nachteile die Akzeptanz von Cloud Computing. Hierzu gehören die Sicherheitsprobleme, die durch das Befinden von unternehmenskritischen Daten außerhalb der eigenen Verfügungsmacht entstehen. Außerdem stellt der Zugriff auf Cloud Computing Ressourcen einen *Singel-Point-of-Failure* Probleme im Fall eines Abbruchs von der Internet Verbindung dar, da dieser Zugriff nur über das Web möglich ist. Darüber hinaus können Unternehmensdaten in Gefahr geraten, wenn die Leistungen des Cloud-Computing Dienstanbietres gestoppt [12] werden, wie es beispielsweise bei einer Insolvenz der Fall ist [htt10].

2.5.3 Cloud Computing BigPlayers

Zahlreiche Firmen bieten Heutzutage sowohl Cloud-Dienste als auch Cloud-Ressourcen. Allerdings gelingt der Durchbruch als Titan nur wenigen Firmen bzw. als *First Movers in the Cloud*. Zu denen mit den größten Marktanteilen gehören Amazon, Google und Microsoft. Nachfolgend werden die sogenannten drei *Big Players* und deren Produkte kurz beschrieben:

1. *Amazon* ist eine der ersten Firmen, die Cloud Dienste für öffentliche Nutzungen bereitgestellt hat, zu denen u.a. folgende Produkte zählen [TJE09]:

   - `Elastic Compute Cloud (EC2)`, welches virtuelle Maschinen für die Benutzer anbietet.
   - `Simple Storage Service (S3)`, welches das Speichern von Daten bis 5GB im *Amazon's virtual Storage service* ermöglicht.
   - `Simple Queue Service (SQS)`, das eine zuverlässige, hochskalierbare, gehostete Warteschlange zum Speichern von Mitteilungen bietet, während diese zwischen den Computern weitergeleitet werden [Web10].

   Figure 2.10 stellt alle Produkte von *Amazon Webservices* dar.

2. *Google*: ermöglicht mit seinem Produkt *Google's App Engine* dem Entwickler die Erstellung und das *Deplyoment* ihrer Webanwendungen in der gleichen Infrastruktur, in denen Google ihre eigene Anwendungen deploy und somit die Integration mit anderen Google Services vereinfacht [TJE09].

3. *Microsoft*: bietet eine Cloud Computing Lösung in Form der *Azure Services Plattform* [13]. Diese Plattform besteht aus mehreren Diensten, die zahlreiche Leistungen anbieten, wie das *User Identity Management*, die Synchronisation von Daten und das *Workflow-Management*.

[12] Ein gutes Beispiel hierfür ist der Fall vom `WikiLeaks.org` bei Amazon.
[13] `http://www.microsoft.com/windowsazure/`

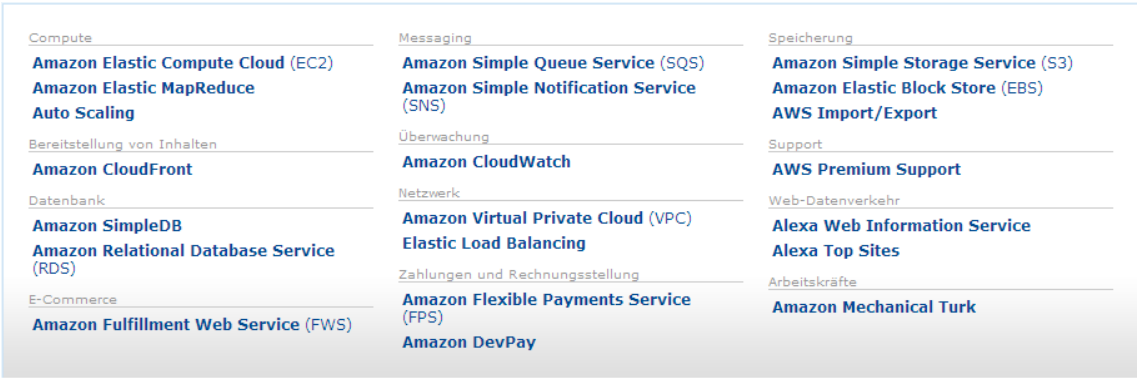

Abbildung 2.10: Amazon Webserivces Produkte
[Web10]

2.5.4 Infrastructure-as-a-Service(IaaS)

Infrastructure-as-a-Service wird auch als *Datacenter-as-a-service* bezeichnet. Bei dieser Cloud-Kategorie stehen alle Rechenressourcen *remotely* zur Verfügung [S.L09]. Diese Kapazitäten werden mit Hilfe von *metered Services* angeboten. Diese Rechenressourcen sind virtuelle Ressourcen und werden in Form von Rechenleistungen, Speicherkapazitäten und Netzwerkressourcen zur Nutzung bereitgestellt [Hol10].

Infrastructure-as-a-service basiert auf Virtualisierungstechnologien. Dank Lösungen wie *XEN* [14] und *VMware* [15] ist es möglich, eine große Anzahl an virtuellen Maschinen auch bei einer beschränkten Menge an physikalischen Servern zu betreiben. Hierzu bilden diese virtuellen Maschinen eine Abstraktionsebene auf die darunter liegenden Server. Diese virtuellen Instanzen stehen mit zugesicherten Mengen an Arbeitsspeicher dem Kunden zur Verfügung [Hol10]. Typischerweise werden unterschiedliche Kategorien von virtuellen Maschinen bezüglich der Performanz (z.B CPU und Speicherkapazität) bereitgestellt.

Im Vergleich zu anderen Cloud Computing Kategorien wie *storage-as-a-service* oder *database-as-a-service*, in denen nur beschränkte Ressourcen zugänglich sind, ermöglicht IaaS einen vollständigen Zugriff auf die geliehenen Maschinen. Zudem steht jedem Benutzer ein *Root-Recht* auf alle Kapazitäten zu. *Infrastructure-as-a-service* bietet zahlreiche anderen Kategorien des Cloud Computings an. Hierzu gehören u.a. *Database-, storage-, goverance-, und security-as-a-service* [S.L09].

Dank IaaS wird die Nutzung teurer, hochperformanter Rechner zu angemessenen Preisen ermöglicht. Zudem übernehmen andere Firmen die Bereitstellung und die Wartung von Rechenzentren [S.L09]. Im Weiteren wird anhand IaaS eine bessere Auslastung der physikalischen Server erzielt: Jedem Kunden wird eine virtuelle

[14] `http://www.xen.org/`
[15] `http://www.vmware.com/`

Maschine zugeteilt, ohne ihm gleichzeitig einen physikalischen Server zuzuordnen [Hol10]. Allerdings bietet IaaS im Vergleich zu *storage-, database-as-a-service* weniger Granularität für die *On-Demand* Erweiterbarkeit. Hierbei wird häufig von den IaaS-Providern verlangt, dass der Benutzer eine gesamte virtuelle Maschine für eine vordefinierte Zeitperiode in Anspruch nimmt [S.L09].

Die Titanen in diesem Bereich, sind neben Amazon mit den Amazon WebServices, GoGrid, Rackspace und FlexiScale [Hol10].

2.6 Verwandte Arbeiten

2.6.1 Unterstützung von ESB-Clustering

Hierbei wird die Skalierbarkeit eines ESBs analysiert. Darunter versteht man die Fähigkeit eines ESBs in einer Cluster-Topologie eingesetzt zu werden.

- JBoss ESB: Ein Clustering Deployment von JBossESB ist ab der Version 4.2 möglich. Dieses basiert auf das Clustering vom eingesetzten *Messaging System*, wie beispielsweise JBoss Messaging. Zudem bietet JBoss ESB auch eine web-basierte Console für das Monitoren des Clusters. Im Weiteren ist ein ortsunabhängiger (engl. remote) Zugriff aus der Ferne auf die Console möglich, welcher dank der Unterstützung vom JAAS [16] gesichert ist. Abbildung 2.11 stellt eine mögliche Architektur für ein JBoss ESB-Cluster dar.

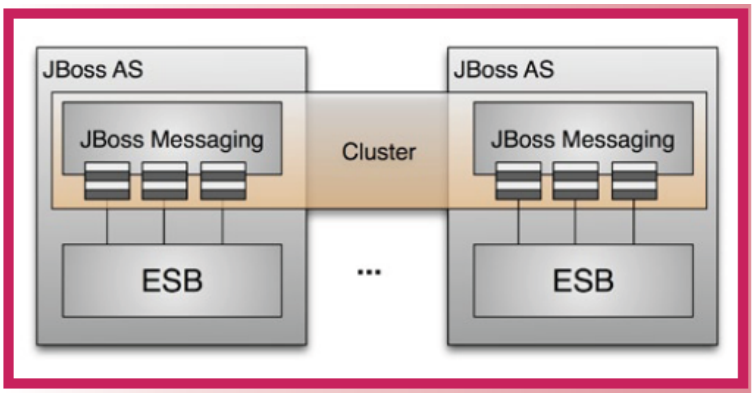

Abbildung 2.11: Mögliche Architektur für einen JBoss-ESB-Cluster [Fin09]

- Mule ESB: Die Inanspruchnahme eines Cluster-basierten Deployments vom Mule ESB ist lediglich durch den Erwerb der Lizenzen möglich [Mul10a]. Für

[16] Java Authentication and Authorization Service

die *Community Version (kostenlose Version)* scheint keine Unterstützung dieser Features vorhanden zu sein. Allerdings kann das Clustering durch den Einsatz von zusätzlichen Lösungen wie *JMS Clustering* oder *Terracotta* [17] ermöglicht werden [Mas08]. Abbildung 2.12 und 2.13 stellen die beiden Möglichkeiten bereit.

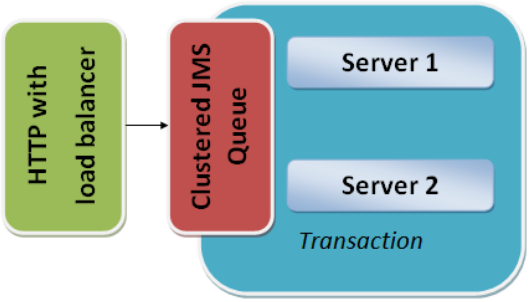

Abbildung 2.12: HTTP + Clustered JMS Queues
[Mas08]

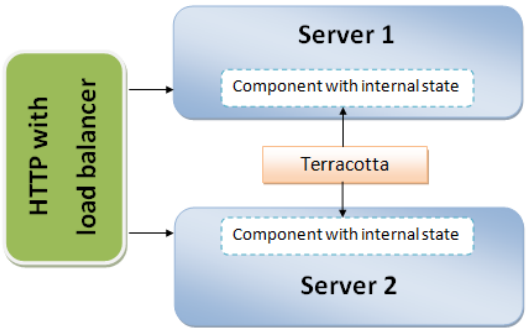

Abbildung 2.13: HTTP + Terracotta
[Mas08]

- Sun Open ESB: Dieses Produkt könnte in einer Cluster Umgebung installiert werden. Umfangreiche und hochqualifizierte Dokumente werden zur Verfügung gestellt [mic09], sodass der Clusteraufbau einfach und problemlos durchgeführt werden kann. Außerdem verfügt Sun Open ESB über eine *web-managed-console*, welche die Konfiguration und das Monitoren des Clusteres übernimmt.

[17] Terracotta ist ein opensource JVM Clustering Technologie, welche die Replikation von Zustanden auf mehrere JVMs ermöglichen kann. Vgl. `http://www.terracotta.org/`

2.6.2 Unterstützung von Cloud Computing

Die Integration eines ESBs in einer Cloud-Computing-Umgebung variiert stark zwischen den vorgestellten ESB Produkten. Hierbei wird analysiert ob Konzepte, Werkzeuge oder sogar virtuelle Maschinen für das jeweilige ESB-Produkt vorhanden sind.

- JBoss ESB: Eine Integration vom JBoss ESB in der Cloud liegt nicht vor. Allerdings bietet die Firma *Redhat/JBoss* viele Werkzeuge (bzw. OpenSource Produkte), die solche eine Integration vereinfachen. Zu diesen Produkten gehört BoxGrinder, welches ein Tool für das Erzeugen von unterschiedlichen Arten von virtuellen Maschinen (z.B. AMI und VMware) darstellt. Zudem verwirklicht der Projekt CirrAS ein automatisches Deployement vom clustered JBoss Application Server im Cloud.
- Mule ESB: MuleSource hat bereits ein Konzept zur Integration von *Mule ESB* im Cloud abgewickelt und sogar ein Produkt im Rahmen von *Integration-as-a-service* entwickelt. Dieses Produkt ist das *MuleOnDemand*, welches von der Firma *OpSource* [18] verwaltet wird. Laut eigenen Angaben [19], bietet *MuleSource* aufgrund der geringen Nachfrage zurzeit keine Integration vom Mule ESB in Amazon-Cloud.
- Sun Open ESB: Eine Integration vom Sun Open ESB in der Cloud besteht. Seit August 2009 wird eine *Amazon Maschine Image (AMI)* mit GlassFish ESB [20] angeboten. Diese Maschine soll auf *OpenSolaris 32-bit* laufen [ESB09].

Wie bereits gezeigt wurde, unterscheiden sich die Firmen stark bezüglich der Integration ihrer Produkte im Cloud Computing. Zudem bietet keiner von denen eine elastische Clusterbarkeit vom ESB an, weshalb im Rahmen dieser Masterarbeit ein neuer Beitrag hierzu erarbeitet und dargelegt werden soll.

[18] `http://www.opsource.net/`
[19] Dies war im Rahmen einer Erkundigungsemail zwischen mich und die *MuleSource*
[20] GlassFish ESB ist der neue Name vom Sun Open ESB

Kapitel 3

Analyse und Konzept

Die Realisierung des im Abschnitt 1.2 verkündigten Zieles wird im Rahmen eines *Clustering* erfolgen. Hierbei werden mehrere Instanzen vom JBoss ESB auf physikalischen und virtuellen Maschinen deployed. Das Clustering wird für externe Systemen (z.B Klienten) transparent sein, so dass der Cluster von draußen als ein einziger einheitlicher ESB wahrgenommen wird. Die Größe des ESB-Clusters soll elastisch sein. Demzufolge wird die Anzahl der beteiligten Rechnern nach oben oder nach unten skalieren.

Unter Kapitel 3 werden zunächst die vorhandenen Möglichkeiten zum Aufbau eines ESB-Clusters (auch Distributed Enterprise Service Bus (DESB) oder Distributed Service Bus (DSB) genannt) erwähnt . Hierbei wird näher auf das *Central-based Management DSB* und dem *Peer-To-Peer basierten DSB*, analysiert und verglichen. Anschließend wird auf Basis der Ergebnisse ein eigenes Konzept erstellt, dass alle an das System gestellte Anforderungen berücksichtigt, um ein entsprechendes Verfahren für die Integration eines ESB in einer elastischen Iaas Umgebung anzufertigen. Abschließend wird die Strategie zum automatischen Abfangen von Lastspitzen einer SOA-mäßigen Anwendung in einer IaaS entworfen.

3.1 Distributed Service Bus (DSB)

Die Literatur [1] gibt zwei Vorgehensweisen für den Aufbau eines *Distributed Enterprise Service Bus(DESB)* an: den Central-based-Management DESB und den Peer-to-Peer-based DESB. Im Weiteren Verlauf werden die beiden Konzepte vorgestellt, erörtert und kritisiert. Zudem werden einige Anmerkungen gemacht als auch Verbesserungsvorschläge angeboten.

[1] Vgl. [WQH+08] und [FIF+10]

3.1.1 Central-based-Management Distributed Service Bus

3.1.1.1 Allgemeine Architektur

Bei einer allgemeinen Architektur eines *Central-based-Management DSBs*, ist zwischen zwei Kategorien an Komponenten zu unterscheiden [WQH+08], und zwar zwischen den mehreren ESB-Knoten, die unabhängig voneinander auf eigene Hosten laufen und dem *Control Center Node(CCN)*. Abbildung 3.1 stellt eine allgemeine Architektur eines *Central-based-Management DSBs* vor.

Dieser *Central-based DSB* gleicht dem *Master-Slave Modell*, in dem die ESB-Knoten die *Slaves* Komponenten und der CCN der *Master* Komponente entspricht. In diesem Modell spielt der CCN die Rolle des Koordinators zwischen den ESB-Knoten. Zudem übernimmt der CCN die Steuerung der Interkommunikation zwischen den ESB-Knoten [WQH+08], welche anhand eines JMS-Queue-Systems stattfindet. Das Monitoring und die Überwachung des Queue-System-Zustands lassen einen konsistenten Schnappschuss des DSBs abbilden [WQH+08].

Der CCN soll folgende Aufgaben übernehmen [WQH+08]:

- Server Management: Diese Funktion übernimmt das Konfigurieren der CCN-Host und führt die Initial Konfiguration vom DSB aus.
- Business-Process-Management: Diese Funktion ist besonders bei Ausfall eines ESB-Knoten, auf dem gerade ein Geschäftsprozess läuft, von Bedeutung. In diesem Fall wird dieser Prozess auf einem anderen Knoten (Backup Node) übertragen und umgesetzt.
- Logging und Protokollierung: Dies bezieht sich insbesondere auf das Logging und Protokollieren von Ereignissen
- JMX Management: Dadurch erfolgt die Konfiguration des gesamten DSBs.
- Service Management: Dies kann u.a. z.B. durch die Registrierung neuer Services erfolgen.
- Management des JMS-Queue-Systems: Darunter versteht man das Monitoring der Warteschlangen (engl.Queues).

Abbildung 3.2 stellt eine mögliche Architektur eines *Control Service Nodes* bereit. Die des CNN genannten Funktionalitäten werden als Module in seiner Architektur dargestellt. Zudem wird dieser durch ein *Laod-Balancer* erweitert [WQH+08] der den DSB vor möglichen Probleme bei Lastszenarien schützen soll. Hierfür werden die eintreffenden Anfragen auf den vorhandenen ESB Knoten verteilt. Zur Veröffentlichung neuer Dienste wird den Service Providern eine *Service-Publishing-Platform(SPP)* zur Verfügung gestellt. Eine mögliche Umsetzung der SPP könnte durch *UDDI* [2] erfol-

[2]Universal Description, Discovery and Integration

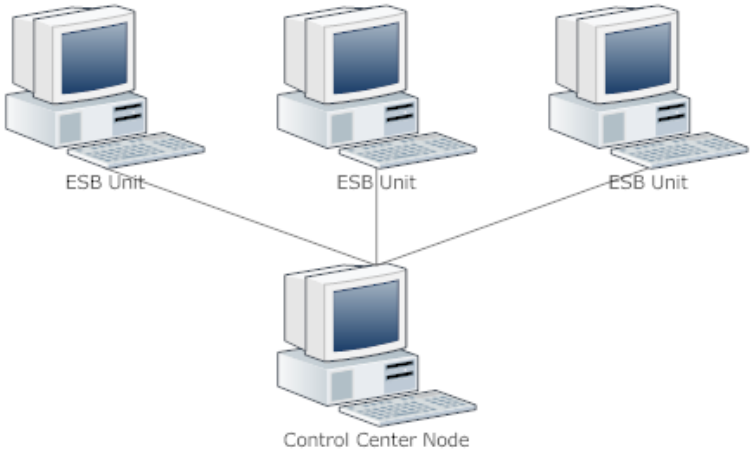

Abbildung 3.1: Allgemeine Architektur eines Central-based-Management DSBs

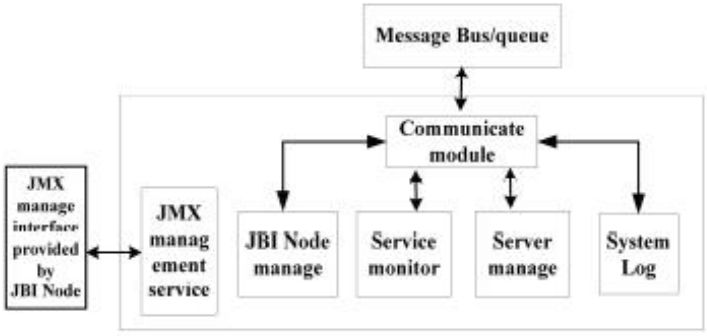

Abbildung 3.2: Architektur eines Control-Center-Nodes [WQH+08]

gen [WQH+08]. Der CCN, der JMS-Queue System und der Laod-Balancer sollen zur Transparenz gegenüber den Kunden und den Dienst-providern beitragen [WQH+08], was eine Realisierung einer SOA Merkmale darstellt.

Abbildung 3.3 fasst die gesamte Architektur zusammen.

3.1.1.2 Interne Workflow

Wie unter 3.1.1.1 erwähnt wird, findet die Kommunikation zwischen den ESB-Knoten und dem CCN durch einen JMS-Queue-System statt. Wird ein neuer Service bei der SPP registriert, teilt dieser dem CCN dies mit. Der CCN sorgt dann für das Deployment der neuen Service in allen ESB-Knoten [WQH+08]. Hierdurch wird ein konsistenter Überblick auf alle Services gewährleistet.

Der Load-Balancer stellt sich als ein Front-End für die eintreffenden Kundenan-

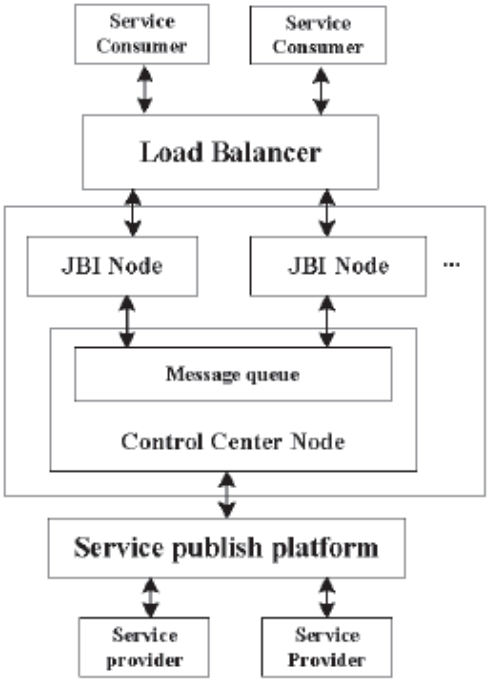

Abbildung 3.3: Erweiterte Architektur des DSBs
[WQH+08]

fragen dar, die dann zu dem nächsten zur Verfügung stehenden ESB-Knoten weitergeleitet werden. Dieser wiederum ruft sogleich den entsprechenden Dienst auf. Dieser Load-Balancer setzt ein DNS-basiertes Verfahren für die Auswahl des ESB-Knotens ein [WQH+08].

3.1.2 Peer-To-Peer basierter DSB

3.1.2.1 Vorstellung

Bei einem *Peer-to-Peer basierten DSB* wird auf ein Management-Knoten verzichtet. Die Kommunikation und Synchronisation zwischen den unterschiedlichen ESB-Knoten sollen nach P2P-Prinzipien erfolgen. Wenn ein neuer ESB-Knoten gestartet wird, schließt dieser sich automatisch am DSB an. Diese Art von ESB-Föderation setzt zwei Bedingungen voraus und zwar [FIF+10]:

1. Definition einer *Technical-Registry*: Diese ist zuständig für das Speichern nicht funktionaler Artefakte (engl. non functional artifacts), die alle notwendigen *Metadaten* für das Routing von Nachrichten bei den Zieldiensten bereitstellen. Als Beispiel von *Metadaten* kommen die Adressen von ESB-Knoten oder die

Endpoint-References (EPRs) von Diensten in Betracht.

2. Erweiterung vom *Message-Transporter*: Der *Message-Borker*, der jedem ESB-Knoten zugeordnet wird, soll erweitert werden, so dass alle DSB-Mitglieder eine einheitliche und konsistente Einsicht in das Nachrichten-Routing haben.

Die *Technical-Registry* kann entweder auf alle ESB-Knoten vervielfältigt (engl. replicated), oder aber verteilt (engl. distributed) werden. Dies erfolgt durch die Verwendung eines *distributed-Hash-Tables* [FIF+10]. Im Falle einer Verzeichnisreplikation, werden alle Registries beim Hinzufügen neuer Eingaben benachrichtigt, u.a. z.B durch *Broadcasting- Algorithmen*. Somit wird ein konsistenter Schnappschuss vom DSB bewahrt.

3.1.2.2 Umsetzungsmöglichkeiten

Da kein zentraler Management-Knoten existiert, wird ein beliebiger ESB-Knoten aus allen Mitgliedern ausgewählt und als virtueller Manager bezeichnet [3]. Dieser bietet sich als *Front-End* (bzw. Load-Balancer) für die eintreffenden Kundenanfragen, und leitet diese weiter zum ESB-Knoten, der den gezielten Dienst mit der wenigsten Last hostet.

Jeder ESB-Knoten definiert lokal einen Lastmessungsmodul, das den Lastzustand aller gehosteten Services misst und die Ergebnisse periodisch zum ESB-Manager sendet.

Für die Auswahl des ESB-Managers, bietet sich als Lösung der Einsatz von verteilten- Algorithmen an. Der *Auswahlalgorithmus* würde die Auswahl eines virtuellen Manager erleichtern. Allerdings stellt diese Vorgehensweise ein potentielles *Singel-Point-Of-Failure* Problem, wenn der ESB-Manager ausfällt. Als Lösung Hierfür, bietet sich die Möglichkeit, den ESB-Manager periodisch *Heart-Beats* an allen anderen ESB-Knoten zu senden. Beim Timeout [4] wird der Auswahlalgorithmus nochmals ausgeführt und ein neuer ESB-Manager ausgesucht.

Da der ESB-Manager ausgelastet werden kann, soll der *Auswahlalgorithmus* auf dem Lastzustand der ESB-Mitglideren für die Auswahl des Managers basieren. So würde die Wahrscheinlichkeit einer Überlastung gering gehalten, jedoch nicht vollständig ausgeschlossen.

Abbildung 3.4 zeigt den Aufbau einer allgemeinen Architektur.

[3] Diese sind eigene Gedanken für das Verwirklichen eines P2P-basierten DSBs
[4] Timeout wird erkannt wenn innerhalb einer vordefinierten Periode keine Heartbeats eintreffen

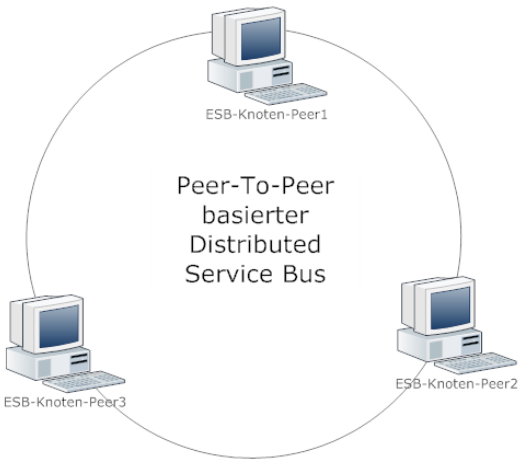

Abbildung 3.4: Allgemeine Architektur eines P2P-basierten DSBs

3.1.3 Auswertung und Vergleich

Zur Bewertung der beiden in 3.1.1 und 3.1.2 vorgestellten DSB-Architekturen, sind golgende Kritiken zusammenfassend zu benennen:

- *Central-based Management DSB*: Der *Central-Center-Node* stellt einen *Singel-Point-Of-Failure* dar. Hierbei wird im Falle eines Absturzes des CCNs der gesamte DSB lahmgelegt. Daher wird auf dieses Modell nicht zurück gegriffen.
- *Peer-to-Peer basierter DSB*: Aufgrund des hohen Komplexitätsgrades für den Einsatz des Auswahlalgorithmus, wird auch aus diesem Konzept kein Gebrauch gemacht.

Für die Realisierung dieser Arbeit, wird eine Kombination der beiden Konzepte angewendet. Dieses eigene erstellte Konzept wird im folgenden Abschnitt eingeführt.

3.1.4 Eigenes Konzept

Es soll ein Kompromiss zwischen den oben eingeführten Verfahren gefunden werden, auf dessen Basis neues Konzept erarbeitet wird. Dieses Konzept soll dem Clustering des JBoss ESBs entsprechen. Bei der Umsetzung sind zwei Schichten zu definieren:

- Controlling Schicht: Hierbei werden ausschließlich die neuen angeschlossenen Mitglieder und den Zustand des ESB-Clusteres berücksichtigt. Ein Control-Modul soll die vorhandenen Mitglieder über den Status des Clusters informieren, also über den Zugang und Abgang neuer Instanzen zum und vom Cluster. Dieses Modul soll als Server laufen. Alle ESB-Konten starten lokal einen ent-

sprechenden Klient, welcher auf Neuzugänge achtet. Diese Controlling-Schicht entspricht dem Verfahren des Central-based Managements, jedoch mit beschränkten Funktionen. Es ist davon auszugehen, dass sich das Control-Modul in die Interkommunikation zwischen den ESB-Knoten nicht einmischen wird.

- Interkommunikation Schicht: Die Kommunikation zwischen den ESB-Cluster-Mitgliedern erfolgt nach dem *Peer-to-Peer* Verfahren. Hierzu verfügt jeder ESB-Knoten über eine konsistente Ansicht des Clusters und übernimmt dementsprechend selber die Kommunikation mit anderen Knoten. Es wird kein *Auswahlalgorithmus* eingesetzt, da die Controlling-Schicht die Aufgaben des virtuellen-Managers übernimmt.

Abbildung 3.5 stellt das kombinierte Modell vor.

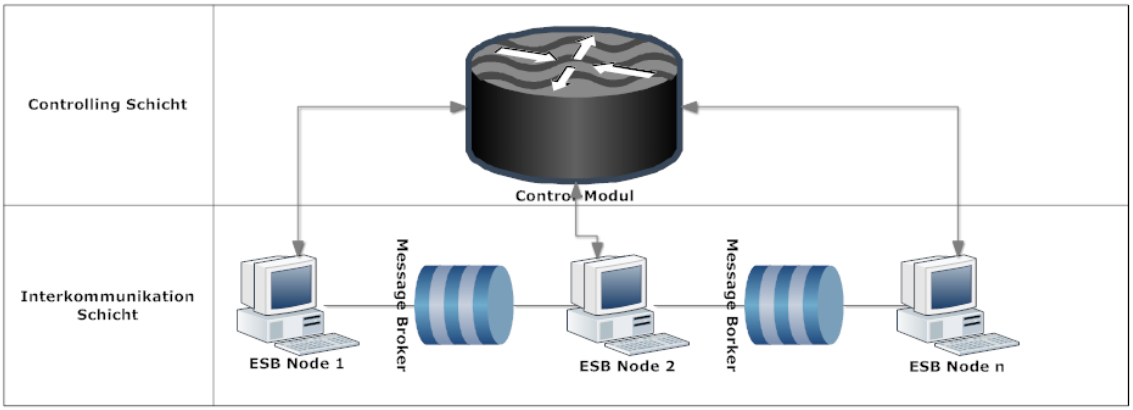

Abbildung 3.5: Eigenes Konzept

3.2 Integration eines ESBs in einer elastischen Umgebung (IaaS)

3.2.1 Anforderungen

Die Anforderungen definieren die Rahmenbedingungen des umsetzbaren Konzepts zur Integration eines ESBs in einer IaaS. Diese werden an Amazon AWS als Cloud-Produk und JBoss ESB als ESB-Produkt unterzogen.

3.2.1.1 Anforderungen an das Cloud Computing

Primär werden diese Anforderungen unter dem Aspekt von Cloud Computing (bzw. IaaS) betrachtet. Grundsätzlich lassen sich diese Anforderungen nach folgendem

Schema aufteilen:

1. **maximale Elastizität**: Alle eingesetzten Ressourcen sollen möglichst elastisch sein. Es wird z.B. ein Datenbank-Server für das JBoss ESB-Clustering benötigt, dann wird der Dienst von *Amazon-RDS* [5] in Anspruch genommen. Folglich wird die Flexibilität erhöht und des *On-Demand* Verfahrens ermöglicht.

2. **Kostenfaktor**: Das Management elastischer Ressourcen, muss die Kosten beachten. Es erfolgt lediglich der Einsatz elastischer Ressourcen, die unabdingbar für eine optimale Realisierung der Arbeit sind. Es soll beispielsweise analysiert werden, welche Art von *Amazon EC2* Maschinen für die jeweilige Aufgabe geeignet ist: *Small, Large oder Extra-Large*. Zudem soll, wenn nicht relevant, auf statische IP-Adressen *(Amazon Elastic IP)* verzichtet werden.

Diese Anforderungen erscheinen in Widerspruch zueinander zu stehen. Daher soll ein Kompromiss zwischen diesen eingegangen werden.

3.2.1.2 ESB Anforderungen

Da die Services auf verteilten Systemen und Rechnern deployed werden, soll der ESB, möglichst, eine *volle Clusterbarkeit* erzielen. Alle ESB-Knoten werden miteinander durch einen *Message-Queue-System* verbunden.

3.2.1.3 Funktionale Anforderungen

1. **Hohe Verfügbarkeit und Fail-Over**: Die Services sollen immer erreichbar sein, weshalb Mechanismen, wie *Service Replikation* eingesetzt werden müssen. Sollte eine Serviceinstanz versagen, dann erledigt ein anderer Dienst die Aufgaben. Außerdem sollen die Kundenanfragen im Falle einer Überlast optimal auf allen verfügbaren Serviceinstanzen verteilt werden. Daher müssen Mechanismen wie Load-Profiling and Load-Balancing eingesetzt werden.

2. **Machbarkeit und Akzeptanz**: Der Konzeptkandidat muss einfach bis mitteleinfach umsetzbar sein. Soll die Umsetzung unrealistisch sein, so erschwert dies die Akzeptanz des Konzepts.

[5] Amazon Relational Database Service

3.2.2 Aufbaumöglichkeiten

3.2.2.1 Vorstellung

Aufgrund des Einsatzes vom JBoss ESB in einer verteilten Umgebung und der Vielzahl an EPRs [6] einer Service [7], sind die beiden Komponenten *Load-Balancer* und das *Message-Queue-System* im Konzeptsaufbau von großer Bedeutung. Erstere strebt nach einer optimalen Verteilung der Klientanfragen, die zweite übernimmt die Rolle eines Brokers zwischen den unterschiedlichen ESB-Knoten.

1. **Load Balancer**:Folgende Szenarien sind zu beachten:
   - Der *ServiceInvoker* vom JBoss ESB übernimmt die Rolle des *Load-Balancers* zwischen den unterschiedlichen EPRs eines Services. Das ist das typische Einsatzszenarium bei JBoss ESB. Drei *Policies* stehen derzeit zu Verfügung: *First-Available, Round-Robin und Random-Robin* [JBo10b]. In der Praxis wird typischerweise die *Roud-Robin Policy* eingesetzt. Allerdings sind eigene Definition und Implementierungen der *Policy* möglich.
   - Bei dem Einsatz von *Amazon Elastic Load Balancing* ist es fraglich, ob diese Lösung auch physikalische ESB-Knoten mit berücksichtigt oder nicht.
   - Es ist die Entwicklung eines eigenen Laod-Balancers, wie z.B den *Membrane Load-Balancer* [8], in Betracht zu ziehen.
2. **Message-Queue-System**: Hierbei werden zwei Vorgehensweisen betrachtet:
   - Der Einsatz eines vorinstallierten *clustered Message-Queue-System* wie JBoss Messaging ist zu betrachten.
   - Auch der Einsatz von einem elastischen *Message-Queue-System* wie *Amazon SQS* [9] ist zu berücksichtigen. Dadurch wird das JBoss ESB-Clustering *On-Demand* realisiert. Abbildung 3.6 zeigt ein Muster dieser Vorgehensweise.

Bei einer Kombination der vorgestellten Szenarien kommen sechs Alternativkonzepte in Betracht. Tabelle 3.1 fasst alle möglichen Konzepte und deren entsprechenden *Elasticity-Level (**EL**)* zusammen.

Abbildung 3.7 stellt den *Elastizitätsgrad* jedes Konzepts dar.

[6] End-Point-References
[7] definierte Anforderung: hohe Verfügbarkeit
[8] Vgl. `http://www.membrane-soa.org/soap-loadbalancing.htm`
[9] Amazon Simple Queue Service

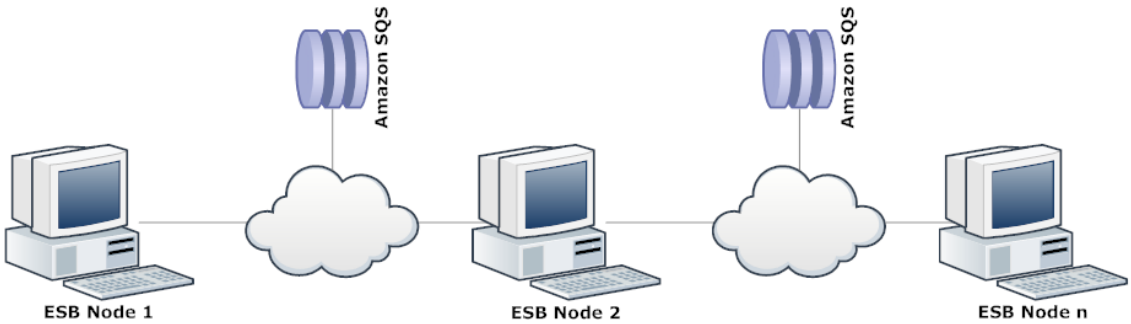

Abbildung 3.6: Elastisches Messaging-Queue-System

LoadBalancer/Messaging-System	Clustered JBoss Messaging	Elastische Messaging
ServiceInvoker	**1**:EL:(0)	**2**:EL:(+)
Eigener Load-Balancer	**3** :EL:(0)	**4**: EL:(+)
Amazon Elastic Load Balancing:	**5** :EL:(+)	**6**:EL:(++)

Tabelle 3.1: **Systemaufbau Möglichkeiten**

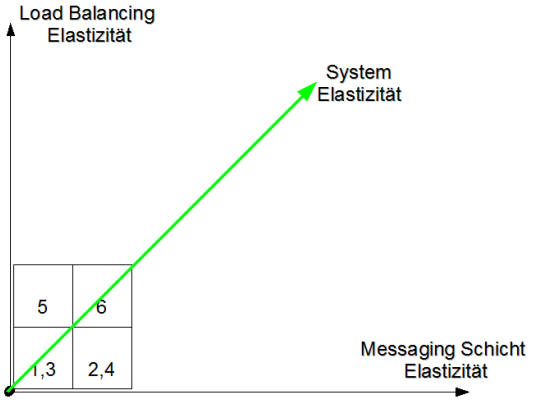

Abbildung 3.7: Elastizitätsgrad der Konzepte

3.2.2.2 Bewertung

- **Kandidaten 2, 4, 5 und 6**: Trotz der hohen erzielten Elastizität, fällt die Akzeptanz, aufgrund der hohen Kosten bei Einsatz von *Elastic Load Balancing* und *Amazon SQS*, gering aus.
- **Kandidat 3**: Aufgrund des hohen Komplexitätgrads, bezüglich der Entwicklung eines eigenen Load-Balancers, ist die Verwirklichen dieses Konzepts nicht möglich.
- **Kandidat 1**: Es handelt sich hierbei um ein typisches Szenarium beim JBoss ESB. Alle ESB-Knoten greifen auf ein einziges Message-Queue-System zurück.

Ein deployed Dienst horcht an einer verteilten Warteschlange (engl. clustered queue). Es wird eine Vielzahl an *Message-Queue-Systeme* angeboten, wie u.a. JBoss Messaging, HornetQ, Oracle AQ [10].

Für das ESB-Clustering kommen zwei Architekturen vor:

1. *Volle Clusterbarkeit (engl. hard clustering)*: Hierbei handelt es sich um einen einzigen ESB, der auf mehreren Rechnern deployed wird. Die ESB-Module werden entsprechend folgendem Schema zu Verfügung gestellt:
   - *1 Message Broker/Cluster*
   - *1 ESB-Module/Cluster*

   Abbildung 3.8 erläutert dieser Betrachtung. Obwohl dieses Konzept ei-

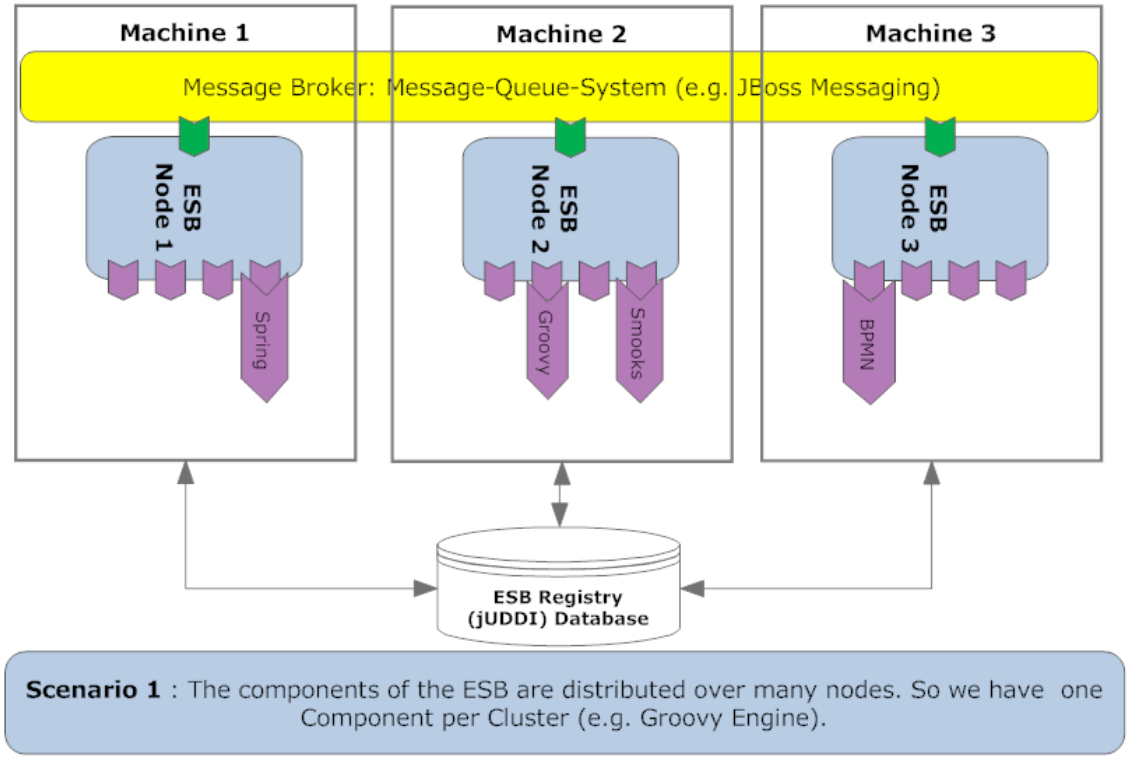

Abbildung 3.8: Volle Clusterbarkeit eines ESBs

ne volle Clusterbarkeit eines ESB darstellt, sind folgende Nachteile zu erkennen:

(a) *Singel-Point-of-Failure*: Es gibt pro Cluster ausschließlich eine einzige ESB-Komponente. Sollte diese ausfallen, würde folglich das gesamte Cluster entweder beschränkt nutzungsfähig sein, da ein Mangel an

[10] Oracle Advanced Queueing

Funktionalitäten besteht (z.B XSLT-Transformation-Engine [11]), oder total lahm gelegt, falls die jUDDI Registry versagt [12].

(b) *Große Netzwerksüberlast und lange Antwortzeiten*: Die Interaktion zwischen den ESB-Komponente würden ewig dauern, weil sie sich auf unterschiedliche Rechner befinden. Im normalen Fall laufen alle ESB-Komponenten auf der gleichen JVM [13].

2. *Beschränkte Clusterbarkeit*: Hierbei wird ein kombinierter Ansatz betrachtet und zwar die Replikation und das Clustering. Mehrere ESB-Knoten laufen auf unterschiedlichen Maschinen. Es entsteht folgende Architektur:

   – *1 Message Broker/Cluster*

   – *n ESB-Komponente/Cluster*

   Abbildung 3.9 stellt diese Architektur vor.

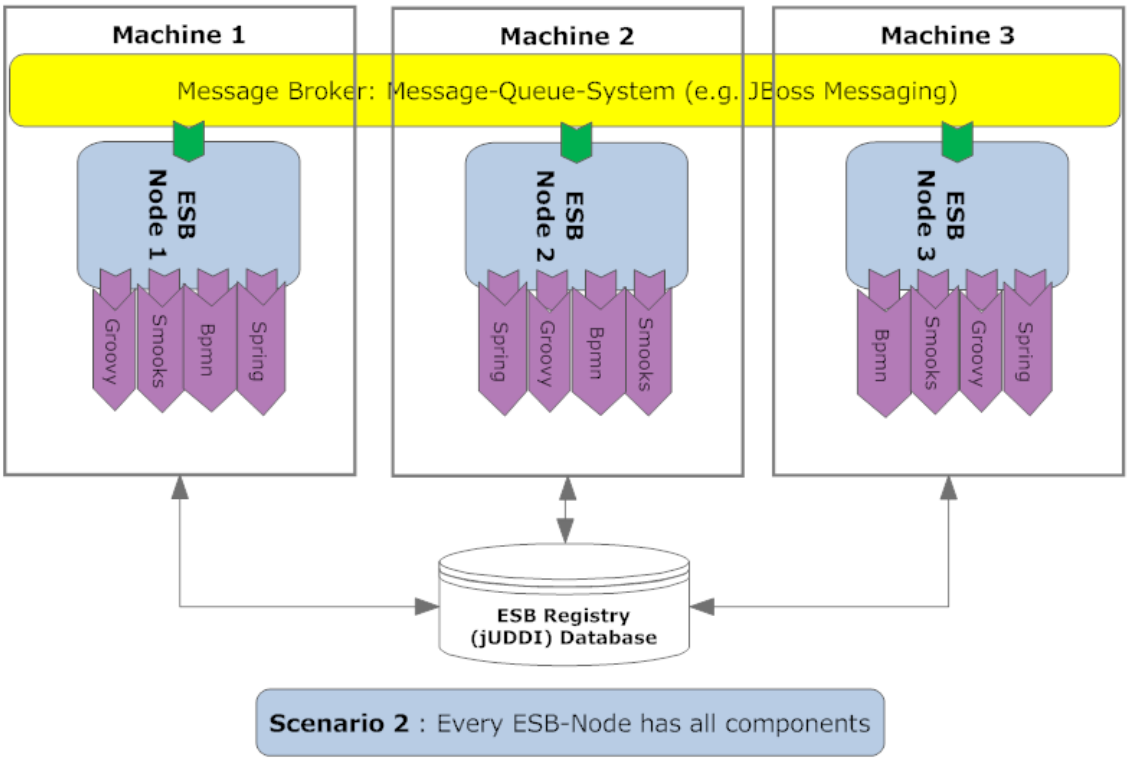

Abbildung 3.9: Beschränkte Clusterbarkeit eines ESBs

[11] Wenn dieses Modul ausfällt dann wird die Transformation von XML Nachrichten nicht mehr möglich

[12] Der Aufruf von Diensten wird nicht mehr möglich

[13] Java Virtual Machine

Wegen der genannten Probleme, wird auf eine volle Clusterbarkeit verzichtet. Stattdessen wird die beschränkte Clusterbarkeit angenommen.

Eine akzeptable Elastizität wird hierbei durch den Einsatz von Amazon EC2 Ressourcen erzielt. Zudem sind die notwendigen Kosten tragbar, da virtuelle Maschinen lediglich im Falle einer Überlast gemietet werden. Die Hochverfügbarkeit und das Failover von Diensten werden durch den Einsatz von mehreren gleichzeitig laufenden Maschinen (physikalisch und virtuelle) verwirklicht. Schließlich liegen die Akzeptanz und die Umsetzungsmöglichkeit dieses Modells im machbaren Bereich, weshalb das erste Konzept für den weiteren Verlauf übernommen wird.

3.3 Automatisches Abfangen der Anfrage-Lastspitzen in SOA anhand von Cloud Computing

3.3.1 Vorstellung

Im vorliegenden Abschnitt ist ein Konzept zu realisieren, das die Vorgehensweisen zum automatischen Abfangen der Anfrage-Lastspitzen in einer IaaS erlaubt. Dies könnte sowohl durch den Verbrauch der konsumierten Ressourcen durch den ESB-Server als auch durch die Antwortzeit der gehosteten Services gemessen werden. Die beiden sind stark miteinander verwandt. In der Realität wird es aufgrund des Mangels an Ressourcen verlängerte Antwortzeiten geben. Selbstverständlich muss es jedoch für längere Antwortzeiten einen Grund geben.

Prinzipiell sind hierfür zwei Verfahren zu wie folgt zu unterscheiden:

- Statisches Verfahren: Zunächst werden die genutzten Ressourcen gemessen und die Ergebnisse beispielsweise in einer Datenbank gespeichert. Danach folgt eine Evaluierung des Systemzustands, auf deren Grundlage eine angemessene Entscheidung getroffen. Das Messen von genutzten Ressourcen erfolgt periodisch im Rahmen vordefinierter Zeitintervalle. Es handelt sich hierbei um einen Offline-Verfahren.
- Dynamisches Verfahren: Hierbei werden die Ressourcen kontinuierlich überwacht und die Werte werden periodisch erhoben. Da die Evaluierung der Werte rechtzeitig geschieht, kann rechtzeitig darauf reagiert werden. Man spricht hier von einem Online-Verfahren.

Im Rahmen dieser Arbeit wird das dynamische Verfahren implementiert, wobei drei Module zu unterscheiden sind:

- LoadMonitor Modul: Dieser ist für das Erheben von Messwerten zuständig.

- ESBClusterManager Modul: Dieser ist für die Bewertung der gemessenen Werte zuständig und skaliert je nach dem die Anzahl der DSB-Mitglieder nach oben oder nach unten.
- Charting Modul: Dieser stellt die gemessenen Werte rechtzeitig graphisch dar.

Die drei genannten Module bilden den sogenannten LoadMonitoringFramework. Dieser Framework soll parallel zur produktiven Anwendung laufen. Abbildung 3.10 bildet die allgemeine Architektur des LoadMonitoringFrameworks ab.

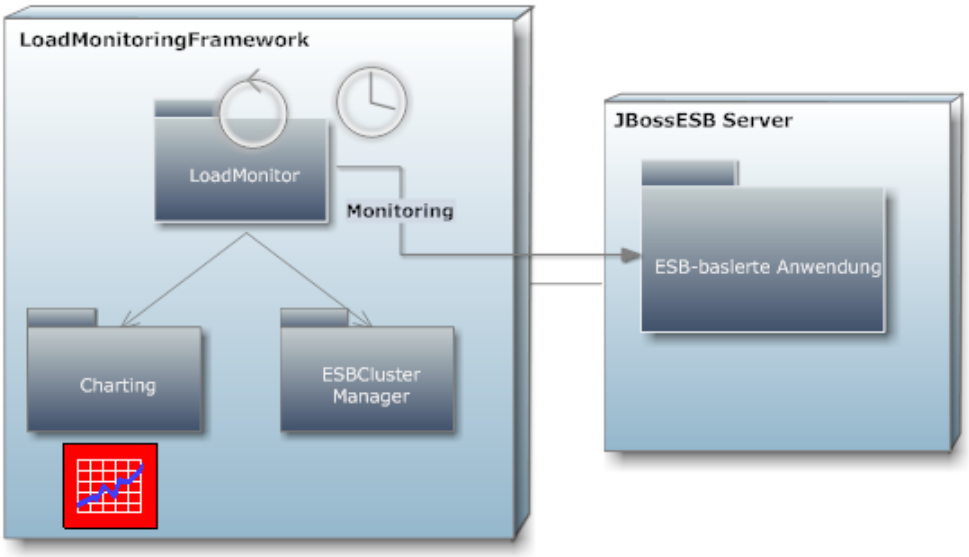

Abbildung 3.10: Architektur des LoadMonitoringFrameworks

3.3.2 Arbeitsweise des LoadMonitoringFrameworks (LMF)

Die produktive Anwendung [14] soll zunächst auf einen einzigen Server laufen. Parallel dazu erfolgen die Monitoring-Aktivitäten. Die drei Module der LMF werden im selben Zeitpunkt gestartet. Der LoadMonitor erhebt die Lastwerte (Antwortzeiten der Services oder Werte der genutzten Ressourcen) periodisch (z.B. jede Minute) in einer endlosen Schleife.

Der ESBClusterManager befindet sich solange in einer *Wait-Phase*, bis die erhobenen Werte vom LoadMonitor bei diesem eintreffen. Anschließend folgt eine *Evaluation-Phase*, in der er die Werte mit der bei ihm vordefinierten Schwelle vergleicht. Sobald der ESBClusterManager merkt, dass die Werte unter der Schwelle liegen, wechselt er zu einer *Launching-Phase*. Bei dieser Phase wird eine vordefinierte Anzahl an *Amazon EC2* virtuellen Maschinen gestartet. In diesem Zeitpunkt

[14] Traffic Data Platform- FCD Prozessierungsmodul des DLRs

wird der ESBClusterManager in einer *Sleeping-Phase* versetzt, in die er für eine gewisse Zeitperiode nicht mehr reagiert. Diese Periode entspricht der benötigten Zeit für das Starten vom JBoss ESB auf einer EC2-Maschine sowie die Bereitstellung und Erweiterung vom ESB-Cluster. Danach kehrt dieser zur *Waiting-Phase* zurück und wartet auf neue Werte. Sollte die Werte, destotrotz, unter der Schwelle bleiben, skaliert der ESBClusterManager die Anzahl der EC2-Maschinen nach oben. Wenn die erhobenen Werte im normalen Bereich liegen, tritt der Manager abschließend der *CleaningUp-Phase* bei, in der dieser die Anzahl der EC2-Maschinen nach unten skaliert und die notwendigen Aktualisierungen durchführt.

Abbildung 3.11 beschreibt das ESB-Erweiterungsverfahren im Falle einer Überlast.

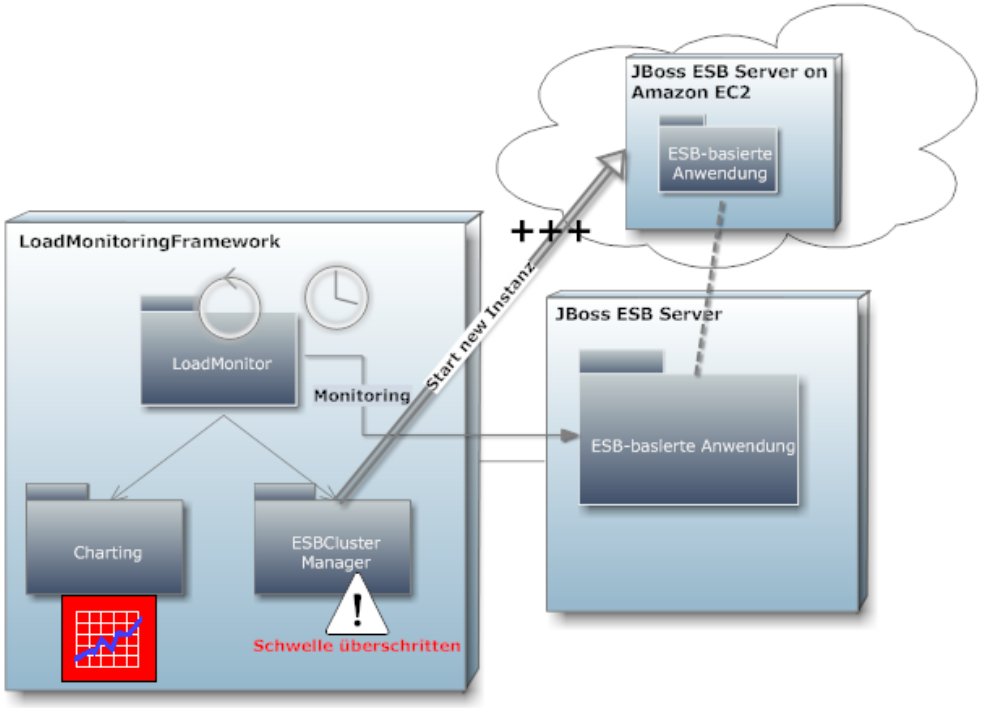

Abbildung 3.11: LoadMonitoringFramework: Automatisches Lastabfangen

Kapitel 4

Implementierung

Im folgenden Kapitel erfolgt die technische Umsetzung des unter Kapitel 3 entwickelten Konzepts. Zunächst werden die notwendigen Konfigurationen und technischen Details für das Deployment eines JBoss ESB-Cluster erläutert, sowie dessen Integration in einer elastischen *infrastructure-as-a-service* auf Grundlage der Ressourcenlastüberwachung. Hierfür werden Amazon EC2 und Amazon VPC als Produkte für IaaS ausgewählt und eingesetzt. Darauf aufbauend wird die Erzeugung von JBoss ESB-Amazon Machine Image (AMI) anhand JBoss Boxgrinder vorgestellt. Abschließen erfolgt die Umsetzung des `Load Monitoring Frameworks` und eine Erörterung der bestehenden Schwierigkeiten bei der Umsetzung.

4.1 Realisierung vom Clustered JBoss ESB

JBoss ESB kann in den folgenden drei Formen eingesetzt werden:

- als alleinstehend (engl. standalone): Dieser kann allein einbezogen werden.
- als *jbossesb-server binary distribution*: Diese Distribution basiert sich JBoss Microkernel [1] Architektur [mas10], die mit leichtgewichteten Funktionen ausgestattet ist. Der Einsatz dieser Distribution eignet sich insbesondere für Entwicklungsaufgaben und für ein schnelles Testen der Anwendungen. Zudem unterstützt diese kein Clustering und wird auch bei produktiven Szenarien empfohlen.
- *deployed in JBoss Application Server (JBoss AS)*: Diese Distribution kommt mit vollständigen Funktionen vom JBoss AS. Zudem wird die Clustering-Fähigkeiten mitgeliefert, so dass sich diese besonders bei produktiven Einsätzen eignet.

[1] `http://community.jboss.org/wiki/JBossMicrokernel`

Daher wird für den weiteren Verlauf, auf die dritte Form, die der deployed in JBoss Application Server (JBoss AS) zurück gegriffen.

Das Clustering vom JBoss ESB basiert auf das Clustering vom genutzten *Messaging-System*. Es kommen unterschiedliche *Message-Queue-Systems* mit der Nutzung des JBoss ASs in Betracht, wie beispielsweise JBoss Messaging, JBoss MQ, HornetQ, etc. Vorliegend wird JBoss Messaging als default Messaging-System eingesetzt. Es ist ein *JMS-basiertes Enterprise-Messaging-System*, welches eine hervorragende Performanz, Zuverlässigkeit sowie Skalierbarkeit mit hohem Durchsatz und niedriger Latenz anbietet. Außerdem eignet es sich hervorragend für SOA Anwendungen und wird häufig als *Message-Broker* in verschiedenen ESB-Implementierungen eingesetzt.

JBoss Messaging stellt unterschiedliche Service-Kategorien zur Verfügung. Die Zusammenarbeit zwischen diesen Services bietet ein *JMS-API-Level* Dienst zu den Klientanwendungen an [JBo07].

Für die Konfiguration von JBoss Messaging sind folgende Dateien von großer Bedeutung: `messaging-service.xml, XXXX-persistence-service.xml und jms-ds.xml`. `XXXX` werden durch den Namen des verwendeten Datenbanksystems ersetzt. Die erste Datei ist für die Einstellung der `sever-peer` Komponente zuständig. Diese ist das Herz vom JBoss Messaging, indem alle Dienste eingestellt sind. Ein Cluster vom JBoss Messaging sieht eine Menge an `server-peer` Komponenten vor, die die gleiche Datenbank verwenden.

Das Clustering vom JBoss Messaging setzt folgende Merkmalen voraus [JBo07]:

1. Zuerst muss jedes deployed `server-peer` in einer Cluster-Umgebung eine eindeutige Identifikation besitzen. Die Konfiguration soll in `messaging-service.xml` erfolgen. Listing 4.1 stellt ein Konfigurationsbeispiel bereit.

```
<attribute name="ServerPeerID">1</attribute>
```

Listing 4.1: Einstellung vom `server-peer`

2. Außerdem sollen deployed Nachrichten-Warteschlangen (engl. message queues) als *distributed* dargestellt werden. Dies ermöglicht allen deployed Instanzen eines Dienstes auf diese Warteschlange zu horchen. Infolgedessen werden die ankommenden Nachrichten in unterschiedlichen Knoten (Mitglieder der Cluster) konsumiert. Für die Zuweisung von Nachrichten zu den Knoten, wird ein internes *Load-Balancing* Verfahren eingesetzt. Hierzu wird die Attribute `clustered` in der *deployment discriptor* Datei als *true* gesetzt. Für jede Warteschlange steht eine *MBean*-Komponente zur Verfügung, welche deren Konfiguration übernimmt. Listing 4.2 veranschaulicht ein Beispiel für die Konfiguration einer *clustered*-Warteschlange.

```
  <mbean code=...
```

```
<attribute name="Clustered">true</attribute>
</mbean>
```

Listing 4.2: Clustered Message-Queue

3. Im Weiteren soll eine gemeinsame Datenbank für alle Cluster-Mitgliedern bereitgestellt werden. Diese speichert die *Metadaten* u.a. von der *deployed message-queue* und von den ankommenden Nachrichten. JBoss AS verwendet *HSQLDB* [2] für seine Datenbanken, die *In-Memory* Datenbank anbietet. Diese soll nur für Entwicklungszwecke genutzt werden. Bei technischen Dokumentationen, wird von dem Einsatz der *HSQLDB* für produktive Zwecke oder bei großen Datenmengen abgeraten [JBo07].

   Die Konfiguration der Datenbank befindet sich üblicherweise unter `<JBOSS_HOME>/server/<profile>/deploy/hsqldb-ds.xml`. *JBOSS_HOME* verweist auf den *Pfad* vom JBoss AS und die *profile* für die verwendete Servereinstellung. Für diese Arbeit wird das clusteresb [3]-*profile* eingesetzt, welche die Clustering-Fähigkeit besitzt. `hsqldb-ds.xml` enthält die Einstellung von der *Datasource*, die von allen JBoss AS-Datenbanken genutzt wird. Standardgemäß wird `DefaultDS` als *default-datasource* für alle Datenbanken verwendet. Für den Clustering-Zweck wird JBoss Messaging eine eigene *Datasource* (bzw. Datenbank) zugewiesen, deren Einstellung unter `<JBOSS_HOME>/server/<profile>/deploy/jboss-messaging.sar/jms-ds.xml` vorhanden ist.

   Bei der Umsetzung sollen sich die Cluster-Mitglieder in unterschiedlichen Netzwerken (Heimatnetz und Amazon-Netz) befinden. Damit alle ESB-Knoten miteinander kommunizieren können und synchronisiert werden, müssen diesie auf derselben JBoss-Messaging-Datenbank zugreifen. Aus diesem Anlass braucht der Datenbankserver einen festen *Hostname*. Wegen der Sicherheitsrichtlinien des*DLR*- und *TUBIT-Netzes*, kann diese Anforderung schwer erfüllt werden.

   Als eine Alternativlösung wird ein externer Datenbankserver mit definiertem Hostname verwendet [4]. Die notwendige *shared*-Datenbank wird für JBoss Messaging auf diesem Server bereitgestellt. Listing 4.3 zeigt einen Part dieser Konfiguration.

```
<datasources>
<local-tx-datasource>
<jndi-name>JmsDS</jndi-name>
<connection-url>jdbc:mysql://younesmysql.
dyndns.org:3306/jbmessagingdbfcd</connection-url>
....
```

[2] `http://hsqldb.org/`
[3] Das ist ein selbst eingestellte Profile, das auf `all` Profile vom JBoss AS basiert
[4] Ein eigener Server wurde Zuhause eingerichtet

```
  </local-tx-datasource>
  </datasources>
```

Listing 4.3: Einstellung von der JBoss Messaging-Datasource

4. Zu den angebotenen Diensten vom JBoss Messaging gehört das Post-Office, welches das Routing von Nachrichten zu den Endzielen übernimmt [JBo07]. Dieses bildet die Adressen, an denen die Nachrichten zugeschickt werden sollen, also an die entsprechenden Nachrichtenwarteschlangen [JBo07]. Für ein lauffähiges Clustering des JBoss Messagings (bzw. JBoss ESB), muss das Post-Office geclustert werden. Somit erfolgt das Routing von Nachrichten zwischen allen Cluster-Knoten. Für diesen Zweck, genügt es die `clustered` Attribute vom Post-Office als *true* einzustellen. Diese Konfiguration soll unter `<JBOSS_HOME>/server/<profile>/deploy/jboss-messaging.sar/mysql-persistence-service.xml` stattfinden. Listing 4.4 stellt die notwendigen Einstellungen dar.

```
<mbean code=
"org.jboss.messaging.core.jmx.MessagingPostOfficeService"
 name="jboss.messaging:service=PostOffice"
 xmbean-dd="xmdesc/MessagingPostOffice-xmbean.xml">
...
<attribute name="PostOfficeName">JMS post office
</attribute>
...
<attribute name="Clustered">true</attribute>
...
<mbean/>
```

Listing 4.4: Einstellung vom Post-Office Dienst

5. JBoss ESB verwendet defaultmäßig eine auf *HSQLDB* basierte Datenbank für sein *Service-Registry*. Diese wird *in-memory* eingesetzt und unterstützt kein Clustering. Daher ist es -wie bei JBoss Messaging- notwendig eine gemeinsame (engl. shared) Datenbank bereitzustellen. Diese soll auf einen ausgereiften und stabilen Datenbankserver wie *Oracle, MySQL oder PostgresSQL* zurückgreifen können. Für diese Arbeit wird *MySQL* als DatenbankManagementSystem (DBMS) verwendet. Die Konfiguration von dieser Datenbank basiert auf einer XML-Datei. Diese befindet sich unter `JBOSS_HOME>/server/<profile>/deploy/jbossesb-registry.sar/juddi-ds.xml`. Listing 4.5 stellt die notwendigen Einstellungen bereit.

```
  <datasources>
  <local-tx-datasource>
    <jndi-name>juddiDB</jndi-name>
    <connection-url>jdbc:mysql://younesmysql.dyndns.org
```

```
    :3306/juddidbfcd</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    ....
  </local-tx-datasource>
</datasources>
```

Listing 4.5: Einstellung von der JBoss ESB-Registry-Datasource

4.2 Einstellungen zur Integration vom JBoss ESB in einer IaaS

Standardmäßig wird das Clustering vom JBoss ESB in einem *virtuellen LAN (VLAN)* einer Firma eingesetzt. Ein *VLAN* ist ein logisches Teilnetz innerhalb eines physikalischen Netzwerks [Möl10]. Dafür wird häufig auf den Einsatz von Technologien wie *Mulicasting* und Transportprotokolle wie *UDP* zurückgegriffen. Außerdem bestehen keine Ports- und Adressierungsprobleme.

Bei der Umsetzung soll der ESB-Cluster zwischen zwei unterschiedlichen Netzwerken aufgebaut werden. Diese Netze sind:

- Das Heimatnetzwerk: Dies ist das Netzwerk auf dem die Anwendung ursprünglich laufen wird. Dies entspricht beispielsweise dem *DLR- oder dem TUBIT* Netz.
- Das Amazon-Netzwerk: Auf diesem Netz sollen die virtuellen Maschinen laufen. Diese werden erst im Falle einer Überlast gestartet.

Wegen dieser Architekturtopologie, werden viele Privilegien eines *LANs* nicht mehr anwendbar, wie beispielsweise der Einsatz der UDP-Transportprotokolle. Außerdem werden Adressierungsprobleme entstehen. Zudem sind die meisten Ports wegen Sicherheitsmaßnahmen (zumindest beim DLR oder TUBIT-Netz) ausgeschaltet. Ein weiteres Problem ist, dass Amazon den Einsatz vom *Multicasting* im Cloud, welches ein häufig eingesetztes Verfahren für JBoss ESB-Clusteing ist, verbietet.

Um mit den Ports- und Adressierungsproblemen umgehen zu können, ist der Aufbau eines *Virtual-Private-Networks(VPN)* zwischen den beiden Netzwerken von großer Bedeutung. Hierzu bietet Amazon als Lösung das Amazon (VPC) [aw10] an, welches das Heimatnetz mit den AWS-Ressourcen erweitern kann. Im Rahmen dieser Arbeit wird diese Lösung in Anspruch genommen. Abbildung 4.1 stellt ein Muster der Architektur bereit [ser10].

Da das *UDP-Mutlicasting* in Amazon Cloud nicht gestattet ist, wird das *TCP-Unicasting* verwendet. Das betrifft hauptsächlich die *prtocol-stacks*, die vom JBoss Messaging genutzt werden. Hierfür sind folgenden *Channels* von Bedeutung [JBo07]:

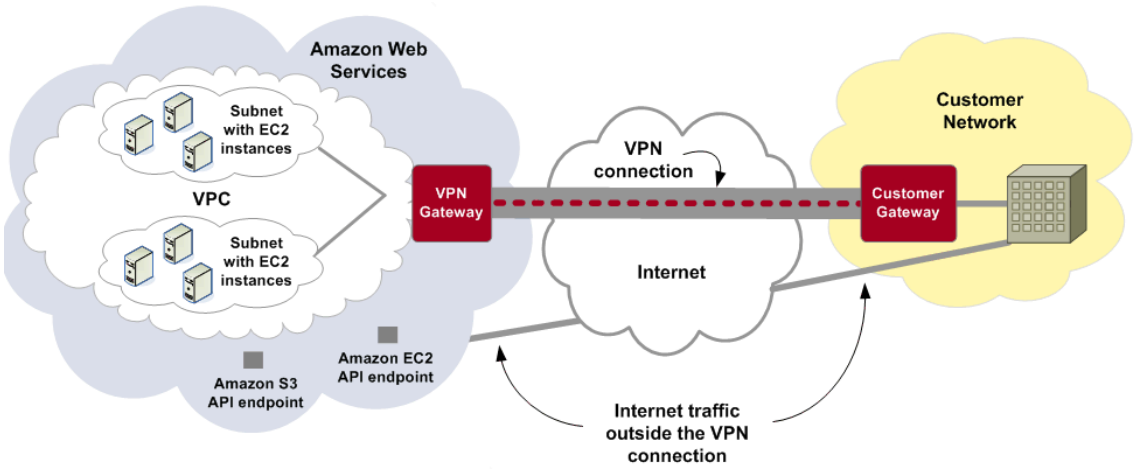

Abbildung 4.1: Aufbau eines VPNs zwischen dem Heimatnetz und dem Amazon-Netz

[ser10]

- `control-channel`: Dieser wird für das Senden von Anfragen und das Empfangen von Antworten zwischen den Cluster-Knoten benötigt.
- `data-channel`: Dieses wird für das Senden und Empfangen von Nachrichten innerhalb des Clusterbereiches eingesetzt.

Die beiden *Channels* werden von der `Post-Office` Komponente verwaltet. Daher erfolgt die Konfiguration unter `mysql-persistence-service.xml`. Listing 4.6 (bzw. 4.7) zeigen die Einstellung der `control-channel` (bzw. `data-channel`). Jeder CHANNEL muss ein Port zugewiesen werden, welches für die Interkommunikation zwischen den Cluster-Mitgliedern verwendet wird. Hierbei wird der Port `8000` (bzw. der Port `7900`) für das `control-channel` (bzw. das `data-channel`) eingeschaltet.

```
<attribute name="ControlChannelConfig">
 <config>
  <TCP
    bind_addr="${jboss.bind.address}" start_port="8000"
...
 </config>
</attribute>
```

Listing 4.6: Konfiguration der `control-channel`

```
<attribute name="DataChannelConfig">
 <config>
  <TCP
    bind_addr="${jboss.bind.address}"
```

```
    start_port="7900"
...
 </config>
</attribute>
```

Listing 4.7: Konfiguration der `data-channel`

Für die Realisierung der in Abschnitt 3.5 definierten Controlling Schicht wird auf die JGroups Kommuniktation-Framework zurückgegriffen. Hierzu werden unterschiedliche Verfahren in Bezug auf *TCP* geliefert, welche die Initialen-Mitglieder eines Clusters feststellen und diesen gegebenenfalls die Anfragen zum Anschließen am Cluster zuschicken [Ban10b]. Prinzipiell werden vier Verfahren angeboten:

- TCPPING: Es verwendet eine vordefinierte statische Liste an Gruppenmitgliedern. Diese Alternative wird lediglich eingesetzt, wenn alle Cluster-Mitglieder von Anfang an festgelegt sind. Das entspricht nicht dem Fall beim Amazon-Cloud.
- PING with GossipRouter: Dies entspricht einem *lookup*-Dienst für die Mitglieder einer Gruppe, der in einem vorbenannten Host mit einem vordefinierten Port gestartet wird. Dieser bietet Schnittstellen an, welche die *Discovery* und die Verwaltung einer Gruppe (bzw. eines Clusters) ermöglichen. Zu denen gehören *GET(group)* und *REGISTER(group, member)*. Die erste Methode ermöglicht die Bereitstellung aller Gruppenmitglieder. Die Zweite fügt ein neuen Knoten zu einer Gruppe hinzu. Jedes Mitglied muss sich periodisch nochmals beim *GossipRouter* anmelden, da dieser sonst von der Mitgliederliste gelöscht wird. `gossip_refresh` stellt die Hearbeats-Periode fest [Ban10b].
- TCPGOSSIP: Das ist gleich wie *Ping*. Allerdings gestattet dieses Verfahren die Definition von mehreren *GossipRouter*.
- S3_PING: Dieses Verfahren ermöglicht die Verwendung eines *shared-storages* für das Speichern von Clusterdaten. Es wird ab der Version *2.6.12* von JGroups angeboten. Als *shared-storage* wird *Amazon S3* verwendet [OM10].

Für diese Arbeit eignet sich am besten das *S3_Ping* Verfahren, da dieses die Verwendung elastischer Ressourcen ermöglicht. Allerdings unterstützt JBoss AS-4.2.3 [5] nicht JGroups-2.6 [6]. Daher wird auf dieses Verfahren verzichtet. Stattdessen wird die zweite Methode eingesetzt. Listing 4.8 stellt die Konfiguration in `mysql-persistence-service.xml` breit.

```
<PING gossip_host="younesmysql.dyndns.org" gossip_port="5555"
gossip_refresh="15000" timeout="2000" .../>
```

Listing 4.8: Controlling-Schicht: Ping with GossipRouter

[5] Version des verwendeten AS
[6] JBoss AS 4.2.3 unterstützt JGroups4

4.3 Erzeugung von der JBoss ESB-AMI

Für die Erzeugung einer Amazon Machine Image (AMI) für JBoss ESB wird das *OpenSource*-Projekt JBoss BoxGrinder eingesetzt. Abbildung 4.2 zeigt das Konzept vom BoxGrinder.

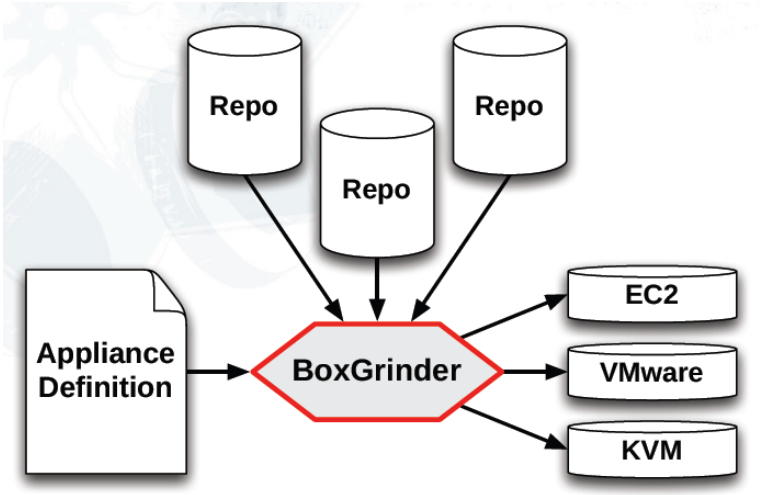

Abbildung 4.2: das Konzept vom JBoss BoxGrinder [McW10]

Boxgrinder basiert auf einer *appliance-definition-file* für den Aufbau einer `appliance`. Diese Datei wird in YAML geschrieben und besteht aus folgenden Komponenten [Gol10]:

- `name`: Hier wird der Name der `appliance` festgelegt, der eindeutig sein muss.
- `summary`: Diese entspricht einer Zusammenfassung der `appliance` sowie u.a. die Ziele.
- `appliances`: Ähnlich wie beim *Objektorientierungsmodell*, kann man hier schon verwendete `appliances` neu definieren und einsetzen. Hierzu kann man eine `appliance`-Template erstellen und dann in anderen `appliances` modifizieren und re-definieren.
- `packages`: Diese enthält alle notwendigen `packages` für die virtuelle Maschine.
- `repos`: Hierzu werden die Namen der *Repositories* erläutert, in denen sich die `packages` befinden. Diese können *lokale* oder *entfernte* (engl. remote) *Repositories* sein.
- `hardware`: In dieser Sektion wird die virtuelle Hardware definiert, wie die Anzahl der verwendeten CPU und die Speicherkapazität.

- `os`: Anbei wird das Betriebssystem spezifiziert, welches auf die `appliance` installiert wird.
- `post`: Hierbei werden die `Shell-Befehle` erläutert, die nach der Erzeugung der `appliance` ausgeführt werden sollen.

Listing 4.9 zeigt den `appliance`-definition-file für JBoss ESB. Diese Datei hat die Endung `.appl`.

```
name: jbossesb
summary: Appliance for JBoss ESB 4.8 with JBossAS-4.2.3.GA
version: 1
release: 1
appliances:
  - f13-basic
packages:
  includes:
    - java-1.6.0-openjdk
    - jbossesb-4.8-1.noarch
repos:
  - name: "local-repo"
    baseurl: "file:///opt/repo/RPMS/noarch"
    ephemeral: true
```

Listing 4.9: appliance-definition-file für JBoss ESB

Diese virtuelle-Maschine-Spezifikation überschreibt (engl. overriden) die `appliance`-Template : `f13-basic.appl`. Hierzu werden die notwendigen Bibliotheken in der Sektion `packages` erläutert. Die sind `jbossesb-4.8-1.noarch` und `java-1.6.0-openjdk`. `java-6` ist notwendig für das Starten von JBoss AS. Auf `jbossesb`-Package wird noch ausführlich eingegangen. Diese Bibliotheken sind `RPM-Packages` und haben die Endung `.rpm`.

Die verwendete *Hardware und Betriebssystem* werden in der originalen `appliance`-Template festgelegt. Hierfür wird das Betriebssystem *Fedora 13* verwendet. Listing 4.10 stellt Teilchen dieser Konfiguration dar.

```
name: f13-basic
...
os:
  name: fedora
hardware:
  memory: 1024
...
```

Listing 4.10: Appliance-Template

Die Package `jbossesb-4.8-1.noarch` basiert auf die `RPM`-Technologie. `RPM` ist ein Werkzeug zum Managen von Software-Paketen auf einem *Linux*-System [Sch00]. Für die Erzeugung von *Package* werden folgende Komponenten verwendet:

- `jbossesb.spec`: Dieses ist die Spezifikations-Datei für die zu erzeugende `package`. Eine `.spec`-Datei ist in einer Syntax geschrieben , welche eine *macro-progrmming-language* mit *shell-commands* verschachtelt. Jede `.spec`-Datei besteht aus mehreren eng verwandten Bereichen : `Header`, `Prep`, `Build`, `Install`, `Files`, `Scripts` und `Changelog` [Lab05]. Listing 4.11 zeigt Abschnitte der verwendeten `.spec`-Datei.

- `jbossesb.init`: Dieses ist ein SysV-init-Skript, welches das erzeugte Package als *standalone-daemon* starten lässt [Lab05]. Hierzu wird das Package automatisch beim System-*Booting* initialisiert und gestartet. Die wesentlichen Merkmale dieses Skripts sind folgende:

  – Wie unter 4.1 bereits erwähnt wurde, müssen alle ESB-Knoten eine eindeutige Identifikation für deren JBoss Messaging-`server-peer` definieren. Da die Maschinen zur Laufzeit im Amazon-Cloud gestartet werden, muss eine dynamische Zuweisung des Identifikators stattfinden. Hierfür wird ein Random-Verfahren verwendet, welches zufallsbedingte den server-peers-Nummeren zuweist. Somit ist die Wahrscheinlichkeit, dass zwei server-peer den gleichen Identifikator haben, minimal. Mit der Hilfe der `RANDOM` *shell-variable* wird dieses Verfahren umgesetzt.

  – Jeder JBoss AS muss beim Starten anhand der *jvm Parameter* -b (bind) zu einer Adresse zugeordnet werden. Im Amazon-Cloud wird die Adresse jeder virtuellen Maschine durch den *Shell-Befehl* `ip addr list eth0 | grep ïnet  cut -d' ' -f6 | cut -d/ -f1` erfasst. Das entspricht der *private-ip* Adresse einer EC2-Maschine.

  Listing 4.12 zeigt die wichtigsten Abschnitte der verwendeten `jbossesb.init`-Datei.

- `jboss-4.2.3.GA.tar.gz`: Diese enthält JBoss ESB-4.8 deployed in einem JBoss AS-4.2.3.GA.

```
Name: jbossesb
Summary: JBossesb-4.8 installed in JBossas-4.2.3.GA!
...
Requires: java-1.6.0-openjdk
Requires: initscripts
Requires: shadow-utils
Requires(post): /sbin/chkconfig
...
%install
rm -rf $RPM_BUILD_ROOT
```

```
install -d -m 755 $RPM_BUILD_ROOT/opt/%{name}-%{version}
cp -R . $RPM_BUILD_ROOT/opt/%{name}-%{version}
...
install -d -m 755 $RPM_BUILD_ROOT/etc/sysconfig
...
%clean
rm -rf $RPM_BUILD_ROOT
...
%pre
/usr/sbin/groupadd -r %{name} 2>/dev/null || :
/usr/sbin/useradd -c "%{name}" -r -s /bin/bash -d
/opt/%{name}-%{version} -g %{name} %{name} 2>/dev/null || :
...
%post
/sbin/chkconfig --add %{name}
/sbin/chkconfig %{name} on
```

Listing 4.11: Abschnitte vom `jbossesb.spec`

```
# Server with JBoss ESB
...
#define ServerPeerID
JBOSS_SERVER_PEER_ID=${RANDOM}
...
#define what IP address for running jboss
#JBOSS_BIND_ADDR=
`ip addr list eth0 | grep "inet" | cut -d' ' -f6 | cut -d/ -f1`
...
#define Cluster name
CLUSTER_NAME=${CLUSTER_NAME:-"ClusterESB"}

#define the script to use to start jboss
JBOSSSH=${JBOSSSH:-"$JBOSS_HOME/bin/run.sh -c $JBOSSCONF
-g $CLUSTER_NAME -Djboss.messaging.ServerPeerID=
$JBOSS_SERVER_PEER_ID"}
...
```

Listing 4.12: `jbossesb.init`

Listing 4.13 stellt den notwendigen *Shell-Befehl* für die Erzeugung der jbossesb-4.8-1.noarch.rpm Package bereit.

```
rpmbuild -ba jbossesb.spec
```

Listing 4.13: Shell-Befehl zur Erzeugung der rpm package

Für die Erstellung der JBoss ESB-AMI, muss der im Listing 4.14 vorgestellte Befehl angegeben wird. Abbildung 4.3 stellt die *Building-Pipeline* vom BoxGrinder bereit.

```
boxgrinder -build -V jbossesb.appl -p ec2 -d ami
```

Listing 4.14: Befehl zur Erzeugung der **JBoss ESB**-AMI

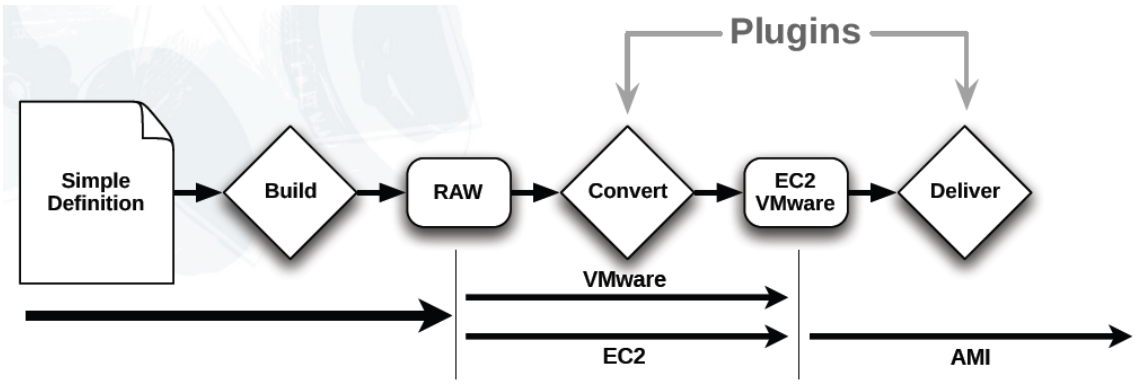

Abbildung 4.3: das *Building-Pipeline* vom BoxGrinder [McW10]

4.4 Umsetzung des LoadMonitoringFrameworks

Für die Implementierung des *Frameworks* wird `java` und `groovy` eingesetzt. Wie in Abschnitt 3.3 angesprochen wurde, besteht das LoadMonitoringFramework aus einem LoadMonitor, einem ESBClusterManager und einem Charting Modul.

Der LoadMonitor überwacht periodisch das FCD-Prozessierungsmodul deployed auf dem *DLR-Server*. Hierzu wird die Klasse `LoadMonitorAgent` implementiert, welche einen `timer` [7] verwendet. Mit Hilfe anderer Klassen erhebt diese im Takt von einer Minute die Daten und benachrichtigt die bei ihr registrierten Module.

Diese Modulen implementieren jeweils einen `Listener` [8], den sog. `LoadReportListener`. Wenn neue Daten beim `LoadReportAgent` eintreten, benachrichtigt dieser dann alle in einer `EventListenerList` angemeldeten Klassen. Hierbei übernimmt die Methode `notifyLoadReport( LoadReportEvent event )` diese Aufgabe. Abbildung 4.4 stellt diese graphisch dar.

Für die Überwachung und Verwaltung vom JBoss ESB steht eine *JMX-Management-*

[7] Vgl `java.util.Timer`
[8] Vgl. `java.util.EventListener`

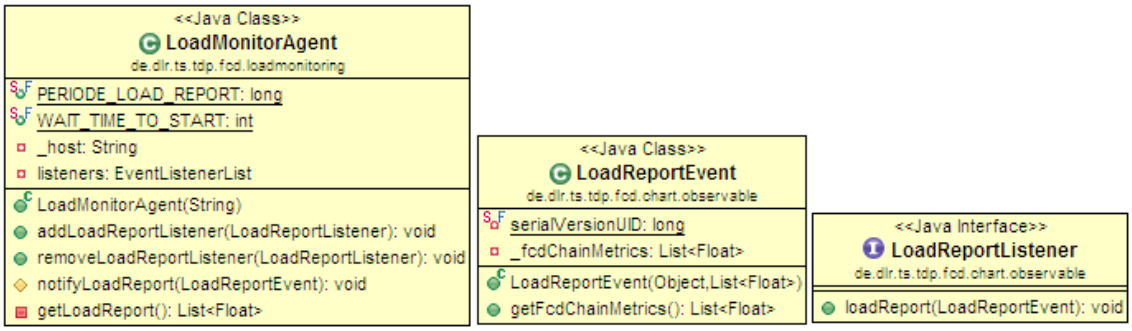

Abbildung 4.4: `LoadReportAgent` und die entsprechenden Listeners

Console [9] zur Verfügung. Diese *Console* kann durch ein *Application Programming Interface (API)* angesprochen werden. Jedes Objekt innerhalb vom JBoss ESB kann anhand eines managed-beans(MBean) verwaltet werden. Hierzu werden zwei Kategorien von Objekten im JBoss ESB überwacht und zwar die `esb-services` und die `jvm`, die vom JBoss ESB) verwendet wird. Für die Dienste werden die Anzahl der bereits bearbeiteten Nachrichten pro Minute gezählt. Zudem wird ermittelt, wie viel Speicherkapazität übrig für den `jvm` bleibt. Hierfür stehen zwei `Beans`-Klassen für die Darstellung der `esb-services` zur Verfügung, die `ServiceMetricsBean` und die `ServiceReporterBean`. Die erste Klasse dient dem Speichern der erhobenen Daten und die Zweite der Darstellung eines Dienstes. Abbildung 4.5 bildet visuell die Struktur der Beiden Klassen ab.

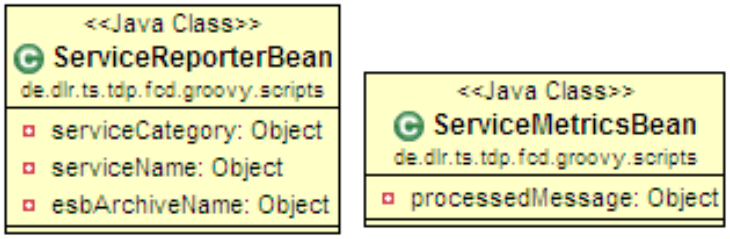

Abbildung 4.5: `ServiceMetricsBean` und `ServiceReporterBean`

Für das Erheben von Daten werden zwei Klassen verwendet und zwar, `LoadUtil` und `LoadReport`. Die erste Klasse bietet Methoden für die Interaktion mit der *JMX-Management-Console* wie `getIntegerAttribute(ObjectName on, String an)` und `getLongAttribute(ObjectName on, String an)`, welche die Werte der beobachteten Objekte in unterschiedliche Formate darstellen können. Die Methoden `getEsbServiceProcessedMessage(serviceReportBean)` und `getJVMMemoryFreeSpace()` dienen zum tatsächlichen Sammeln von Daten. Die zweite Klasse übernimmt das

[9] JMX (Java Management eXtensions) stellt einen standardisierten Weg zur Verfügung, Java Anwendungen und Dienste verwaltbar zu machen [Köh03].

Vereinnahmen der Daten von allen Diensten eines FCD-Processierungsmodules sowie welche von der `jvm`. Hierbei parset die Methode `generateMetrics()` eine Konfigurationsdatei (`ServiceConf.xml`), in der sich alle Dienste einer *FCD-Kette* befinden und verwendet anschließend ein *Multithreading* Verfahren für ein paralleles Erheben der Daten. Hierfür wird jeden angemeldeten ESB-SERVICE ein *Thread* gestartet, welcher die `overall-service-message-count` zurückliefert. Abbildung 4.6 stellt graphisch die beiden Klassen vor.

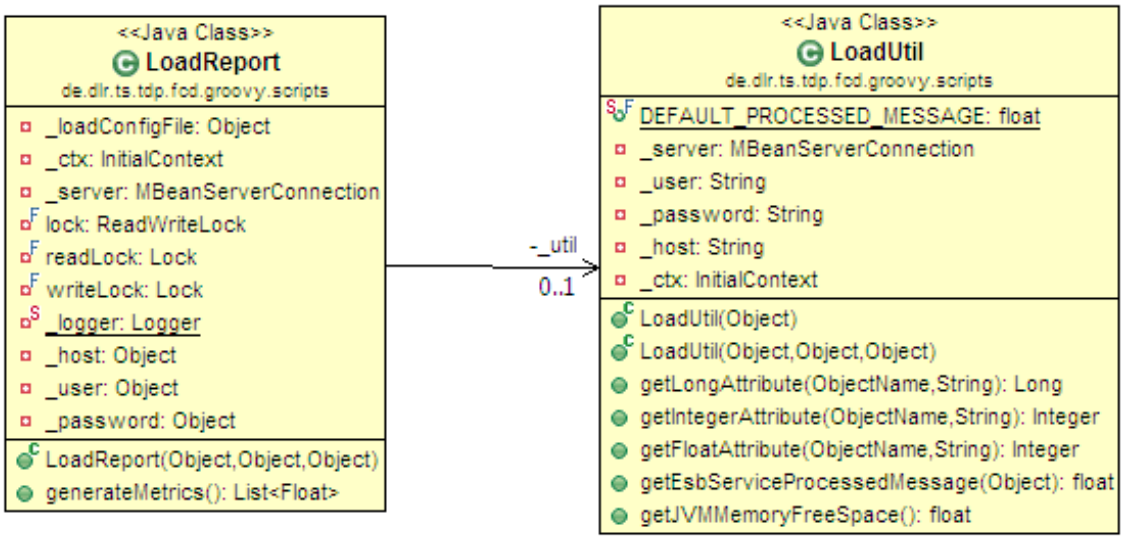

Abbildung 4.6: `LoadReport` und `LoadUtil`

Die Klasse `JMXAttributeFinder` übernimmt die Interaktion mit der *management-console*. Hierfür müssen die Hostadresse des Servers, der `username` und das `password` angegeben werden. Die Methode `query(ObjectName oname, String attname)` wird verwendet, um nach den Wert eines angegebenen Attributes für eines *ESB-Objekt* beim JBoss ESB-Server zu fragen. Falls eine Authentifizierung notwendig ist, dann wird die Methode `login()` eingesetzt, welche auf die Hilfsklasse `LGCallbackHandler` verlässt. Ansonsten werden der `username` und das `password` mit dem Wert `null` belegt. Abbildung 4.7 stellt graphisch die beiden Klassen vor.

Für das Modul ESBClusterManager steht die Klasse `EsbClusterManager` zur Verfügung. Diese implementiert einen `LoadReportListener` und wird somit vom `LoadMonitorAgent`, falls neue Daten vorhanden sind, benachrichtigt. Hierbei wird auf Basis der noch zur Verfügung stehenden Speicherkapazität, von der für den JBoss ESB zugewiesenen JVM, eine Entscheidung getroffen und die Größe des *ESB-Clusters* nach oben oder nach unten skaliert. Falls diese Kapazität weniger als einen vordefinierten Wert `FREE_JVM_MEMORY_BARRIER` liegt, wird die Anzahl der Cluster-Mitglieder nach oben skaliert. Im Rahmen dieser Arbeit wird die Schwelle auf `50MB` festgelegt. Der `EsbClusterManager` verwendet einen `AmazonEC2Client` für die Inter-

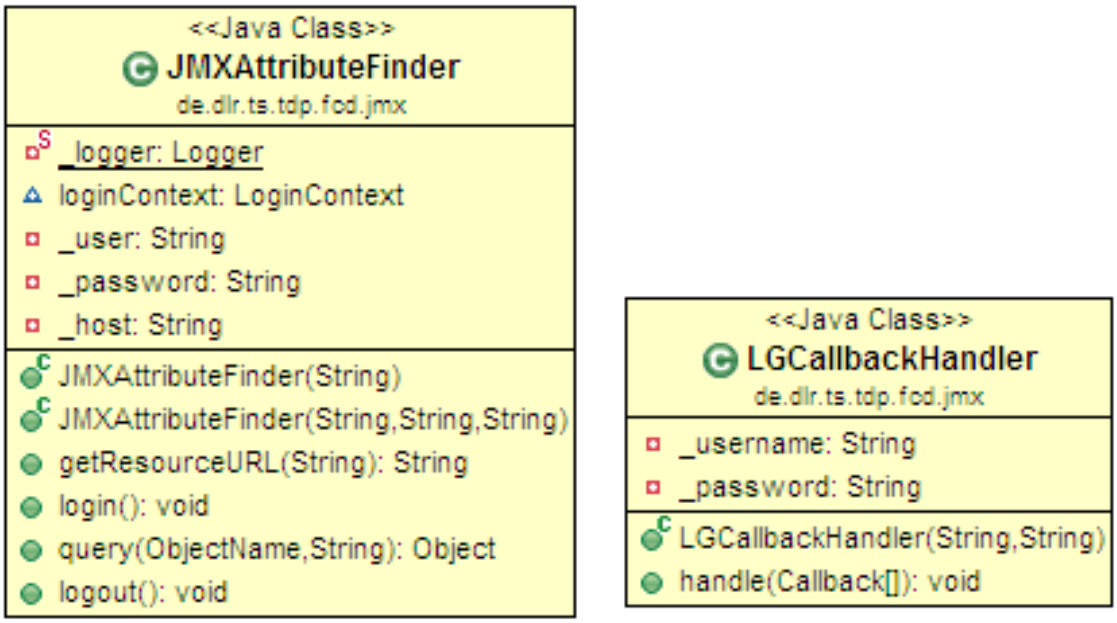

Abbildung 4.7: `JMXAttributeFinder` und `LGCallbackHandler`

aktion mit dem Amazon EC2 Cloud. Die Methode `startNewEC2Machine()` kümmert sich um das nach oben Skalieren des ESB-Clusters durch das Starten von einer neuen virtuellen Maschine, welche automatisch einen JBoss ESB-Server hochfährt und sich am vorhandenen ESB-Cluster anschließt. Nachdem Starten dieser Maschine tritt der `EsbClusterManager` der in 3.3.2 beschriebenen *Sleeping-Pahse* bei. Die Dauer dieser Phase entspricht der approximativen Zeit, welche für die Bereitstellung des Clusterings und der Lastverteilung der Anfragen notwendig ist. In dieser Arbeit wird diese Periode auf `5 min` festgelegt. Zudem reagiert der ESBClusterManager und das Charting Modul innerhalb der *Sleeping-Phase* auf keine eintreffenden Daten. Wenn die 5 Minuten vorbei sind, wird der `EsbClusterManager` wach und reagiert wieder auf die ankommenden Meldungen. Falls die Werte der `jvm` unter der Schwelle liegen, fährt der ESBClusterManager eine *EC2-Maschine*, falls eine vorhanden ist, herunter. Andernfalls startet er immerhin neue virtuelle Maschinen. Abbildung 4.8 stellt den `EsbClusterManager` und seine Beziehungen mit den anderen Klassen graphisch dar. Die Klasse `LoadMonitoringFrameworkAccessPoint` besitzt eine `main`-Methode und stellt sich als den *Accesspoint* für die Ausführung des Load-Monitoring-Frameworks.

Für das Modul CHARTING ist die Klasse `FcdChainMeteredDataChart` zuständig. Diese implementiert auch die Interface `LoadReportListener` und wird somit periodisch mit den neuen eintreffenden Daten benachrichtigt. Die Klasse verwendet die `jreechart`-Bibliothek, welche ein java-basierte Framework für *Charting* ist. `FcdChainMeteredDataChart` stellt die Werte, im Zusammenhang zur Zeit, graphisch

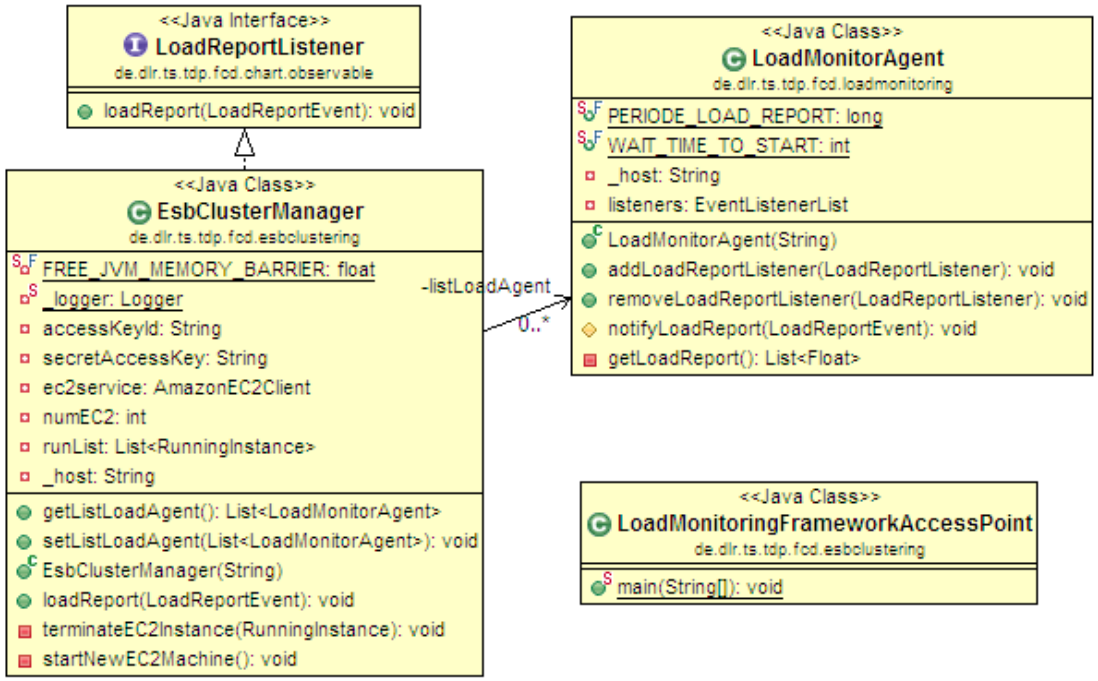

Abbildung 4.8: `EsbClusterManager`

dar. Diese Darstellung erfolgt *quasi-rechtzeitig* [10]. Abbildung 4.9 stellt diese Klasse visuell dar.

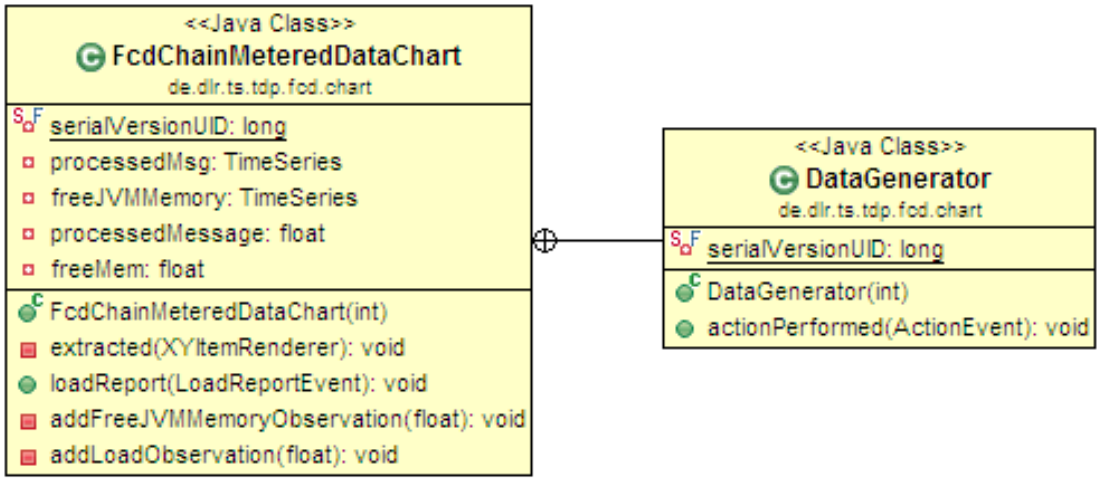

Abbildung 4.9: Die Klasse `FcdChainMeteredDataChart`

[10] Ein Realtime Charting ist nicht möglich bei `jfreechart`. Stattdessen wird auf eine periodische Aktualisierung der Daten verlassen.

Abbildung 6.2 zeigt einen allgemeinen Überblick des KlassenDiagramms vom LoadMonitoringFramework.

4.5 Schwierigkeiten bei der Umsetzung

Die im 4.2 auf Amazon-VPC basierte Lösung für den Aufbau eines JBoss ESB-Clusters konnte trotz des praktikablen Konzeptes nicht realisiert werden, da die Sicherheitsmaßnahmen des DLRs die Konstruktion eines VPNs zwischen dem DLR-Netz und dem Amazon-Netz nahezu unmöglich machen. Um dieses Problem zu umgehen, wurden folgende Maßnahmen ergriffen:

1. Zuerst wurde versucht, den Aufbau des VPNs auf dem TUBIT-Netz zu lagern. Jedoch scheiterte auch dieser Versuch, aus den gleichen Gründen bei dem DLR-Netz, da auch das TUBIT-Netz durch Sicherheitsmaßnahmen unzugänglich ist.

2. Anschließend wurde als Alternative ein externer virtueller Server angemietet, wo die Kosten auch noch im Rahmen des Möglichen liegen, um auf diesen den ESBCluster aufzubauen. Dieser Server wurde jedoch mit einem *Kernel* ausgestattet, das kein *IPSec* und keine *Erzeugung von Sicherheitsschlüssel* Mechanismen unterstützt. Eine Erweiterung des Kernels mit diesen Features ist nicht möglich. Die Kosten eines Servers, der den Aufbau eines VPNs unterstützt, liegt im Rahmen des Unmöglichen für einen Student.

Eine potentielle Lösung, wäre der Aufbau dieses Clusters auf dem UNI-Netz durch die Verwendung des *CIT-Eucalyptus-Clouds*. Es käme noch eine weitere Alternative in Betracht, und zwar, das Deployment des gesamten Clusters innerhalb vom Amazon Netz. Die letzte Vorgehensweise entspricht nicht dem dargebotenen Konzept in dieser Arbeit, würde jedoch die Beobachtung der elastischen Skalierung eines ESB-Clusters innerhalb vom Amazon-Cloud sowie die Wahrnehmung der Lastverteilung und der Erholung von Ressourcen im Falle von abgefangenen Lastspitzen, ermöglichen.

Kapitel 5

Evaluation anhand einer Fallstudie: DLR-Traffic Data Platform/FCD-Prozessierungsmodul

Im Rahmen dieses Kapitels erfolgt zu Beginn eine kurze Vorstellung des *DLRs* und des *Instituts für Verkehrssystemtechnik*. Anschließend werden die *Traffic-Data-Platform* und das *Floating-Car-Data-Prozessierungsmodul (FCD-Kette)* erörtert. Daraufhin folgt die Gestaltung und Umsetzung eines auf SOA basierenden Systems der *FCD-Kette*. Anschließend wird eine Überwachung des *FCD-Prozessierungsmoduls* durchgeführt, analysiert und gegebenenfalls bei vorhandener Überlastung, durch den Einsatz von Cloud Computing, automatisch abgefangen .

5.1 Einführung

5.1.1 Vorstellung des DLRs und des DLR-TSs

Das DLR ist das Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Seine umfangreichen Forschungs- und Entwicklungsarbeiten im Bereich der Luft- und Raumfahrt, Energie und Verkehr und führt nationale und internationale Kooperationen. Über die eigene Forschung hinaus, ist das DLR als Raumfahrtagentur im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig. Mit Hilfe von Satelliten, wird eine weltweite Kommunikation ermöglicht, die wichtige Daten über unsere Umwelt und das Weltraum liefern. Davon können ebenfalls andere Industriezweige für Innovationen aus Luft- und Raumfahrt profitieren, von der Werkstoff-Technologie über neue me-

dizintechnische Verfahren bis zu Software-Entwicklungen.

Die Aufgaben des DLR strecken sich von der Erforschung von Erde und Sonnensystem sowie die Forschung für den Erhalt der Umwelt und die Entwicklung umweltverträglicher Technologien zur Steigerung der Mobilität bis hin zur Kommunikation und Sicherheit und ermöglichen somit die Entwicklung von innovativen Anwendungen und Produkten der Zukunft [DLR10].

Das DLR setzt sich aus mehreren Instituten zusammen, wobei für diese Arbeit das Institut für Verkehrssystemtechnik von Bedeutung ist, das im Jahr 2001 unter der Leitung von Prof. Dr.-Ing. Karsten Lemmer am Braunschweiger Forschungsflughafen gegründet wurde und seitdem stark gewachsen ist und seine Forschungsaktivitäten ausgebaut hat.

Der Stellenwert der Mobilität von Menschen und Transport von Gütern nimmt kontinuierlich zu. Diese sollte möglichst schnell als auch kostengünstig und sicher erfolgen. Folgen der Mobilität sind Umweltbelastungen, Unfälle und Staus, die möglichst zu minimieren sind. Hierzu bedarf es der Kooperation der Wissenschaftler aus unterschiedlichen Fachrichtungen auf nationaler und internationaler Ebene mit Partnern und Kunden, Wissenschaft und Politik [DT10a].

Das Institut gliedert sich in folgende Bereiche:

- Automotive: Basierend auf das untersuchte Fahrverhalten, werden Fahrerassistenzsysteme entwickelt, die die Sicherheit und Effizienz im Straßenverkehr nachhaltig erhöhen und die Anzahl d er Unfälle reduzieren. Die Umsetzung wird in Fahrversuchen in der Simulation und im Realverkehr überprüft.
- Bahnsysteme: Durch den stetig wachsenden Verkehrsaufkommen, das hauptsächlich von der Straße aufgenommen wird, muss der Anreiz zur Nutzung des Schienenverkehrs verstärkt werden. Hierzu bedarf es der betrieblichen, technischen und wirtschaftlichen Optimierung bei Entwicklung und Erarbeitung innovativer Technologien, Methoden und Konzepte für das Bahnsystem.
- Verkehrsmanagement: Ziel ist durch Organisation und Betrieb von Verkehr die Effizienz im Straßenverkehr zu erhöhen. Hierfür werden Informationen über den aktuellen Verkehrszustand erfasst. Die Aufgaben lassen sich so in zwei Bereiche gliedern: die Entwicklung innovativer Methoden zum Monitoring von Verkehr (Verkehrserfassung) und die Entwicklung von Methoden zur Einflussnahme auf Verkehrsabläufe (Verkehrsbeeinflussung).

[DT10b]

5.1.2 Beschreibung der Traffic-Data-Platform

Für den Zugriff und Austausch von Verkehrsdaten (Schleifendaten, Floating Car Data, Videodaten, prozessierte und fusionierte Daten, u.s.w.) sowie die Bereitstellung der Telematikdienste, wie das Multimodale Routing, hat das Institut für Verkehrssystemtechnik des Deutschen Zentrums für LuftundRaumfahrt (DLR-TS) eine Traffic-Data-Platform (TD-Platform) im Entwicklungsstadium.

Die TD-Platform verarbeitet die Verkehrsdaten (Rohdaten) aus unterschiedlichen Quellen wie Floating Car Data, Schleifendaten, Videodaten u.s.w. Für die Verarbeitung kommen unterschiedliche Prozessierungssysteme oder Module zum Einsatz (z.B. Filter, Matching-Algorithmen, Datenaggregierungsmodule). Jede Client-Anwendung, die Rohdaten, prozessierte oder fusionierte Daten benötigt, kann jederzeit und überall über die durch die TD-Platform bereitgestellten Schnittstellen, die gewünschten Daten abfragen oder einige Basis-Dienste nutzen.

Eine Anforderung an die TD-Platform ist, dass sie große Datenmengen verarbeiten und eine große Zahl an Anfragen abarbeiten kann. Es ist zu erwarten, dass mit einem höheren Verwendungsgrad die Prozessierungsauslastung durch die Verarbeitungsmodule und die aufgesetzten Dienste steigen werden und ebenso zeitweise kurzfristig hohe Zugriffsauslastungen von Clients zu erwarten sind. Abbildung 5.1 verdeutlicht die modular Architektur von TDP.

Die *TD-Platform* ist in Schichten aufgebaut. Jede von denen übernimmt bestimmte Aufgaben. Folgende Schichten sind zu differenzieren:

- *System Management Layer* : Diese Schicht ist für das Monitoring und die Überwachung der laufenden Prozesse zuständig.
- *Services Layer* : In dieser Schicht werden zahlreiche Dienste definiert. Diese werden den Klienten zur Verfügung gestellt.
- *Processing Layer* : Hierbei werden die Rohdaten bearbeitet. Zu dieser Schicht gehört das *Floating Car Data Prozessing Modul(FCD)*. Für den Rest der Arbeit, wird ausschließlich auf dieses Modul konzentriert.
- *Map Layer*: Diese Schicht sorgt für die Bereitstellung geographischer Dienste.
- *Content Management Layer*: Dieser stellt sich als die *Back-End*-Schicht der *TD-Platform*. Zahlreiche Datenbanken werden hierzu aufgebaut, welche die *Storage* von *Rohdaten und prozessierten Daten* übernimmt.

Die *TD-Platform* verarbeitet unterschiedliche Datentypen (input data). Diese und die prozessierte Daten werden für die Partner und Kunden mit Hilfe von definierten Schnittstellen bereitgestellt.

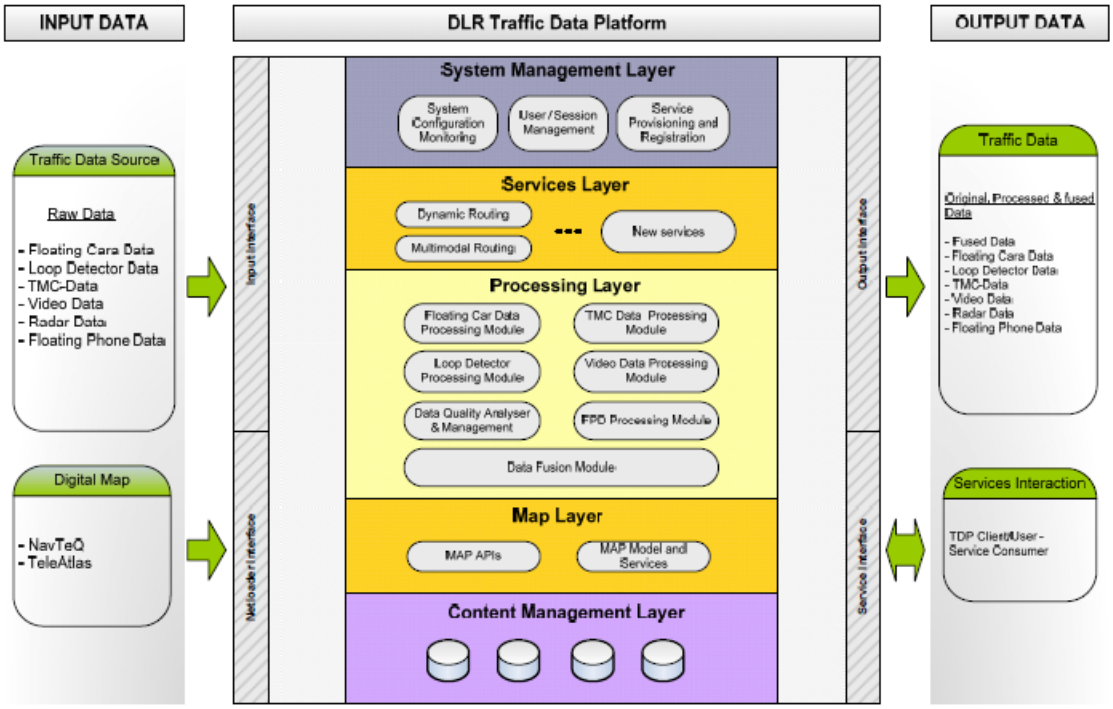

Abbildung 5.1: Modulare Architektur der Traffic Data Platform [TREY10]

5.1.3 Vorstellung vom FCD-Processierungsmodul (oder FCD-Kette)

Die *Floating-Car-Data-Processing-Module(FCD)* oder auch FCD-Kette sind für die *Filterung, die Aggregation und Prozessierung* von FCD-Rohdaten zuständig. Als Ergebnis werden Reisezeiten und linkbasierte Fahrgeschwindigkeiten bereitgestellt. Die FCD-Kette besteht aus vier Hauptkomponenten:

1. *Filter:* Dieser implementiert *Filterungs-Regeln*, die für die Erkennung und Entfernung invalider FCD-Rohdaten eingesetzt werden.
2. *Trajectoriser:* Mit Hilfe von Fahrzeug-IDs eines bestimmten FCD-Systems bildet dieses Fahrzeugtrajektorien aus einfachen GPS-Punkten ab.
3. *Matcher:* Basierend auf speziellen Algorithmen berechnet dieser die *map-matched*-Positionen der Trajektorien von GPS Punkten.
4. *LinkSpeedGenerator:* Dieser liefert, in Echtzeit, relevante Trafficdaten für alle Kanten eines betrachteten Straßennetzes. Somit unterstützt dieser den *Matcher* beim Liefern optimaler Daten.

Außerdem wird für den Import von FCD-Rohdaten einen `DataImporter` festgelegt, welcher die Daten von unterschiedlichen Quellen (`DataProvider`) einbringt. Abbildung 6.3 stellt einen allgemeinen Überblick über das *FCD-Prozessierungsmodul*.

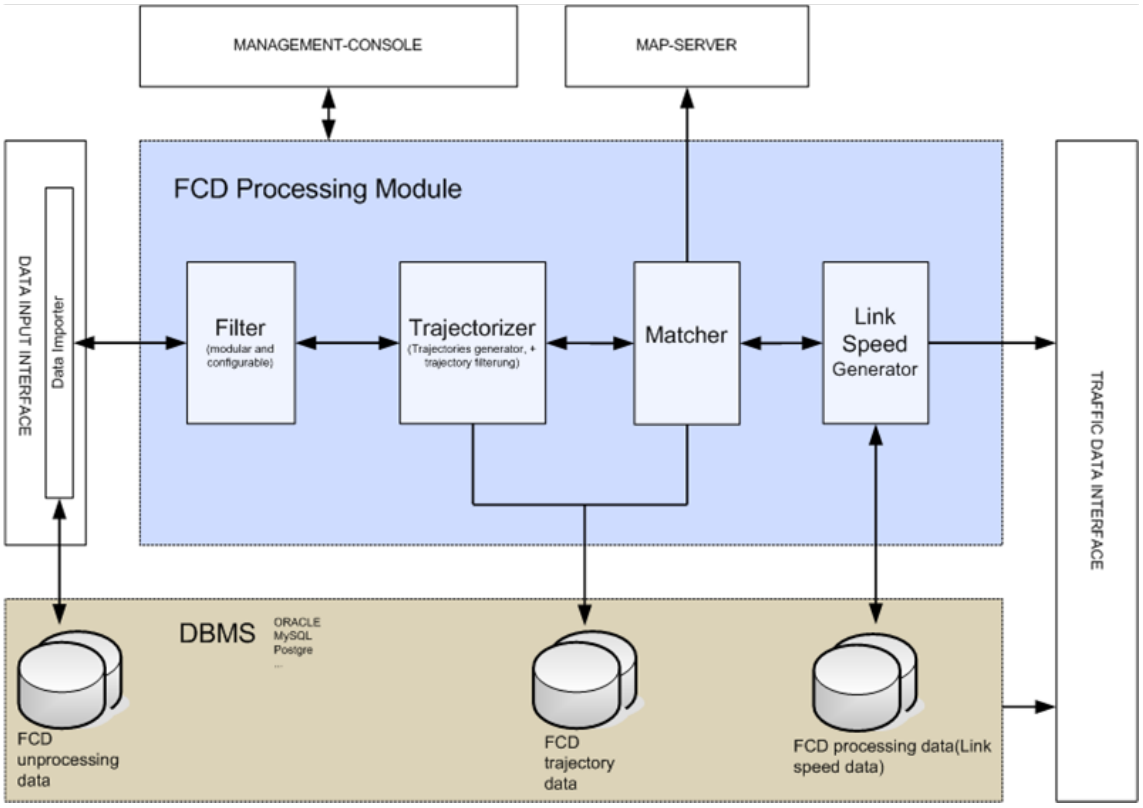

Abbildung 5.2: Floating-Car-Data-Processing-Module (FCD) [TREY10]

5.2 SOA-mäßiger Aufbau der FCD Kette

Bisher wurde die *FCD-Kette* des *DLR-TSs* als einzige *Java-Anwendung* implementiert und auf einem einzigen Server deployed. Alle Komponente wurden auf der gleichen `jvm` ausgeführt. Diese Architektur und Art von *Deployment* ist als akzeptable und anwendbar zu betrachten, wenn Probleme wie Serverabsturz oder ähnliches ausgeschlossen werden können. Jedoch erfüllt diese Lösung nicht die Anforderungen wie *Ausfallsicherheit, Skalierbarkeit und Erweiterbarkeit*, was eine Gefahr für die Beibehaltung der *Quality-Of-Service* darstellt. Zudem wird die *FCD-Kette* ,trotz aller entwickelten Überwachungslösungen wie *Process-State-Monitor(PSM)*, im Fall eines internen Fehlers oder eines Serverabsturzes lahmgelegt. Diese Monitoringlösung

dient nur dem Administrator, indem der Status der FCD-Kette angezeigt wird oder über einen Fehler benachrichtigt wird. Im Weiteren wird für jede Stadt, welche die FCD-Rohdaten besorgt, eine neue `fcd-jvm` zugewiesen. Somit wird z.B. eine Anwendung für *Berlin* gestartet, eine andere für *Hamburg* u.s.w.

In diesem Abschnitt wird ein neues Konzept für das FCD-Prozessierungsmodul auf Basis von SOA vorgeführt.

5.2.1 Konzept

Da im Rahmen dieser Arbeit davon ausgegangen wird, dass auf SOA Verlass ist, werden die Komponenten der *FCD-Kette* als *Dienste* entworfen, implementiert und deployed. Zwei Möglichkeiten liegen für den Entwurf und die Implementierung der Dienste in JBoss ESB vor: entweder als *ESB-services* oder als *web-services*. Hierbei wird die erste Möglichkeit angenommen. Die genannten Komponenten des *FCD-Prozessierungsmodules* sowie der `DataImporter` und der `DataProvider` werden als `ESB-dienste` implementiert. Hierzu werden andere Dienste festgelegt, welche die Verwaltung der *Backend*-Schicht der *FCD-Kette* übernehmen.

Um die Funktionalitäten des *FCD-Prozessierungsmodules* zu verwirklichen, müssen all diese Dienste komponiert werden. So entsteht ein Dienst mit einem komplexen Konstrukt, welches dem externen Kunden und Partner als genau ein Dienst erscheint. Die Komposition von Diensten ist für den Kunden transparent. Für deren Realisierung, wie bereits unter 2.1.3 erwähnt wurde, das Konzept der Orchestrierung und Choreography in Betracht kommen. Da jeder Dienst weißt, mit welchem anderen Dienst er kommunizieren wird, eignet sich hierfür eher den Einsatz von einer *Choreography*.

In der vorliegenden Arbeit wird vom Konzept der *Choreography* ausgegangen. Im Rahmen dessen, entsteht eine Peer-To-Peer Kollaboration zwischen den Diensten.

Insgesamt werden zwölf Dienste für die Umsetzung der *FCD-Kette* in Betracht gezogen:

1. `DataImporter`: Zu dessen Aufgaben gehört sowohl das Importieren von Rohdaten aus dem *DataProvider* als auch die Bereitstellung dieser Daten für andere Komponenten der *FCD-Kette* wie Filter etc.

2. `DataProvider`: Dieser Dienst sorgt für die Bereitstellung der Rohdaten von unterschiedlichen Städten. Bezüglich der vorliegenden Arbeit, liegen *offline*-Daten von sechs Rohdaten-Anbietern in einer Datenbank vor.

3. `Fcd_Current_Speed_DB_Service`: Dieser Service kümmert sich um die Verwaltung der Datenbank `Current_Speed_DB`.

4. `Fcd_Hist_Speed_DB_Service`: Dieser hingegen sorgt für die Verwaltung der Datenbank `Hist_Speed_DB`.
5. `Fcd_Matches_DB_Service`: Dieser Service ist zuständig für die Verwaltung der Datenbank `Matches_DB`.
6. `Fcd_On_Edge_DB_Service`: Dieser Service beschäftigt sich mit der Verwaltung der Datenbank `Fcd_On_Edge_DB`.
7. `Filter`: Dieser Service kapselt die Logik von *Filter-Module*. Er filtert der FCD-Rohdaten.
8. `LinkSpeedGenerator`: Dieser Service übernimmt die Aufgaben der *LinkSpeedGenerator-Module*.
9. `Matcher`: Dieser übernimmt die Aufgaben der *Matcher-Module*
10. `Raw_Data_DB_Service`: Dieser Service speichert die eintreffenden Rohdaten in der Datenbank `Raw_Data_DB`.
11. `Trajectories_Container_DB_Service`: Dieser Service sorgt für das Speichern von Trajektorizerbezogenen-Daten, wie die erzeugten *Trajektorien* in der Datenbank `Trajectories_Container_DB`.
12. `Trajectorizer`: Dieser kapselt die Funktionen der *Trajektorizer-Module*

Der interne Arbeitsablauf des *FCD-Prozessierungsmodules* wird im Anhang in der Abbildung 6.1 als eine *Business Process Modeling Notation(BPMN)* dargestellt.

5.2.2 Umsetzung

Für die Implementierung des umzusetzenden Konzeptes, wird auf die beiden Projekte pi4soa [1] und JBoss Overlord CDL [2] zurück gegriffen.

Pi4soa verwendet die Choreography Description Language Standard [3] und bietet einen *Graphic-Editor* für die Erstellung eines choreography-description-files (*.cdm) an. Dieses Werkzeug wird als *Plugin* in der Entwicklungsumgebung *Eclipse* [4] integriert.

JBoss Overlord ist ein OpenSource Projekt der Firma *Redhat*, welches *Governance* und *Monitoring* von SOA-basierten Systemen anbietet. Zudem arbeitet JBoss Overlord eng mit JBoss ESB und WS-BPEL zusammen und ermöglicht die Erzeugung von *Services-Skeleton* basierend auf einem choreography-description-file. Darüber hinaus

[1] `http://sourceforge.net/apps/trac/pi4soa/wiki`
[2] `http://jboss.org/overlord`
[3] `http://www.w3.org/2002/ws/chor/`
[4] `http://www.eclipse.org/`

bietet Overlord die Möglichkeit, die entwickelten *ESB-services* oder *Web-services* gegen eine *Choreogrphay-Description* zu validieren.

5.2.2.1 Aufbau der Choreogrphy

Die *Choreography Description Language(CDL)* bietet ein Mittel zur Beschreibung eines Prozesses an, welches bei unabhängigen (verteilten) Diensten eingesetzt wird. Diese Beschreibung stellt eine globale Perspektive des (distributed) Systems dar. Für den Aufbau eines choreography-description-files (*.cdm) wird auf die *Web Services Choreography Description Language (WS-CDL)* zurückgegriffen. Hierfür müssen folgende Kernkonstrukte erfüllt sein [IMC05]:

- `package`: In diesem muss mindestens eine Choreographie definiert werden. Dieser bündelt eine Reihe von WS-CDL Typdefinitionen und stellt den Namespace für die Definitionen bereit.
- `roleType`: Ein Rollentyp definiert Rollen für Geschäftsprozesse und zählt die, von außen beobachtbaren, Verhaltensweisen einer Geschäftspartei auf, die in Kooperationen auftreten.
- `relationshipType`: Dieser legt die Beziehungen zwischen jeweils zwei Rollentypen fest, die für eine erfolgreiche Zusammenarbeit unabdingbar sind.
- `participantType`: Dieser Typ identifiziert all die Rollentypen, die logisch zusammen gehören und gemeinsam implementiert werden sollten. Bezüglich des beobachtbaren Verhalten bilden die Rollentypen eine Einheit und können nur von derselben Organisation implementiert werden.
- `channelType`: Dieses Konstrukt verwirklicht die Schnittstelle, über die Informationen bei Zusammenarbeit zwischen Parteien ausgetauscht werden können, regelt den Informationsfluss und ordnet diesen dem zuständigen Rollentyp zu.
- `informationType`: Diese beschreiben die Art der Information, die in einer Choreographie benutzt werden.
- `variable`: Variablen speichern Informationen über sichtbare Objekte in einer Zusammenarbeit. Es wird zwischen drei Typen unterschieden.
- `token`: Tokens sind Bestandteile einer Variable, die zur Nutzung durch eine Choreographie notwendig sind.
- `tokenLocator`: Definiert den Bereich, in dem ein Token sich befindet und bietet somit einen Mechanismus zur Abfrage für diese Informationsausschnitte.
- `choreography`: Diese stellt den Kern der WS-CDL Technik dar. Sie regelt den Nachrichtenaustausch zwischen den interagierenden Parteien sowie den einzelnen zu verrichtenden Aktivitäten.

- `interaction`: Diese bilden die Grundlage für die Zusammenstellung einer Choreographie, die aus mehreren Interaktionen besteht und in verschiedenen Geschäftsumfeldern genutzt werden kann.
- `workunit`: Eine Workunit beschreibt die notwendigen Bedingungen für den Fortschritt in einer Choreographie und ermöglicht die Fortsetzung der aktuellen Arbeit.
- `exceptionBlock`: In diesen werden Fehlersituationen ausgearbeitet, die durch Fehlervariablen in Interaktionen verursacht werden. Eine Exception kann aktiv ausgelöst werden oder sie wird durch Constraints (wie Überschreitung der Antwortzeit) automatisch aktiviert.
- `finalizer`: Wird eine Choreographie erfolgreich abgeschlossen, kann evtl. die Durchführung anderer Aktivitäten, wie rückgängig machen oder ändern notwendig sein, um Ergebnisse der vollendeten Aktivitäten zu bestätigen. Um diese Aktivitäten umsetzen zu können, werden finalizer Blocks definiert.

[IMC05]

Der *Graphic-Editor* ist ein Werkzeug von pi4soa, das eine graphische Modellierung einer Choreography in verschiedenen Ansichten ermöglicht [IMC05]. `Roles`, `Behavior` und `Relationship` werden in Roles and Relationships Ansicht festgelegt. Dabei werden die grundlegenden Verhältnisse, Interaktionen zwischen den Diensten und Rollen des Geschäftsprozesses fixiert. Abbildung 5.3 zeigt die `Roles`, `Behaviors` und `Relationships` zwischen den in 5.2.1 erwähnten Dienste.

Der Ablauf (tatsächliche Workflow) des Geschäftsprozesses wird in der Choreography Flow gestaltet. Hierzu können verschiedene Algorithmenstrukturen verwendet werden. Zu diesen Strukturen gehören *sequentielle-, parallele- und bedingtbasierte* Abfolgen. Außerdem sind Strukturen wie die *while*-Schleife und die *if-then-else* anwendbar so dass auch komplexe Geschäftsprozesse gestaltet werden können. Abbildung 5.4 gibt graphisch Teile des FCD-Kette-Geschäftsprozesses wieder.

Wie Abbildung 6.4 zeigt, besteht der Geschäftsprozess der FCD-Kette aus zwei parallelen *Sub-Prozessen*. Der erste stellt die Aktivitäten und die Interaktionen des `DataImporters` dar. Dieser Dienst fragt periodisch (jede 2 Minuten) den `DataProvider` nach FCD-Rohdaten nach, die in einem Intervall von [t,t+10min] liegen. Treffen diese bei ihm ein, leitet er sie zum `Filter` weiter. Außerdem sendet er diese Daten zum `Raw_Data_DB_Service`, der sie in der `Raw_Data_DB` Datenbank speichert. Anschließend wird der `DataImporter` in einen Ruhezustand mit einer vordefinierten Periode versetzt. Nach dem Ablauf dieser Periode, startet er ein neuen Zyklus.

Der `Filter` wird beim Eintreffen neuer Daten getriggert. Zunächst filtert er diese und leitet sie anschließend zum `Trajektorizer`. Dieser verwendet die gefilterten Daten, um *Trajektorien* zu erzeugen. Anschließend schickt er diese zum `Matcher`, der nach dem *Matchen* der Daten die Ergebnisse zum `Trajektorizer` zurück leitet

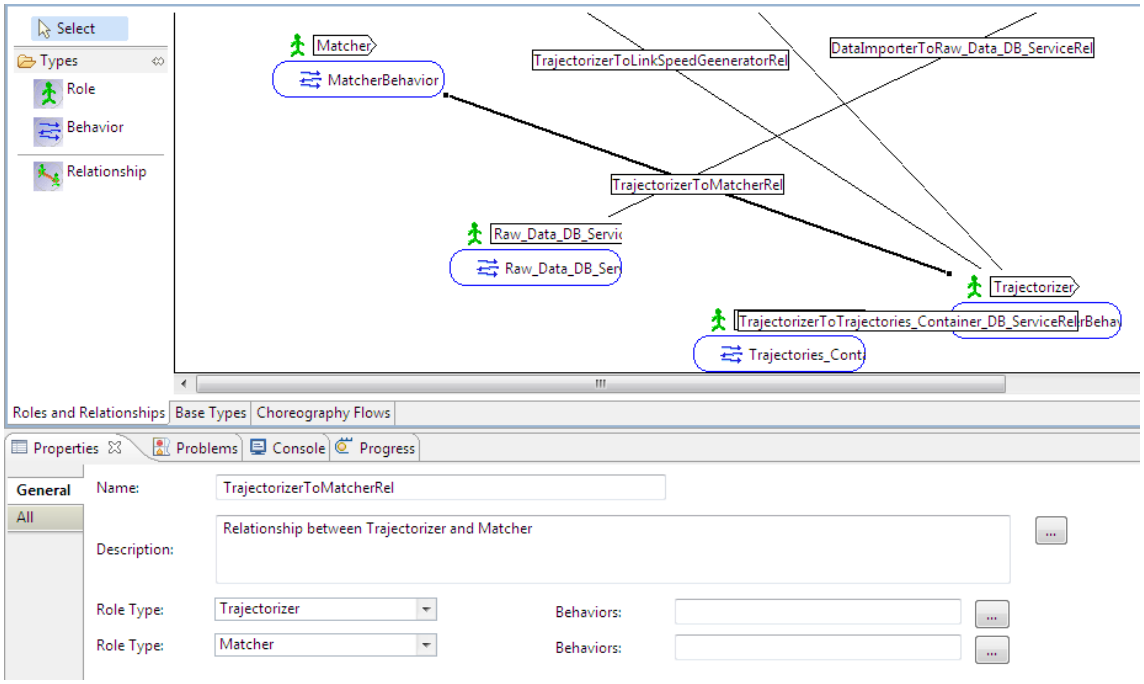

Abbildung 5.3: `Choreography: Role, Behavior und Relationship`

und dieser wiederum nach vorgetroffenen Änderungen an den `LinkSpeedGenerator`. Dieser sorgt für die Optimierung der Daten. Abschließend speichert dieser die Daten in unterschiedlichen Kategorien in den entsprechenden Datenbanken. Abbildung 6.1 zeigt den Ablauf des FCD-Geschäftsprozesses als *BPMN*-Modellierung.

5.2.2.2 Implementierung von ESB-services

JBoss Overlord CDL verwendet das oben vorgestellte `fcd.cdm`, um die Struktur von jboss-esb-services festzulegen. Diese können statefull oder stateless sein. Die statefull Services verwenden die Sessions für die Bearbeitung der Anfragen. Jede Session hat einen eindeutigen Identifikator. Abbildung 5.5 zeigt wie die Erzeugung von Service-Gerüsten durchgeführt wird.

JBoss ESB definiert für jeden *ESB-service* drei *XML-basierte* Konfigurationsdateien, die den Kern eines Dienstes ausmachen:

- `jboss-esb.xml`: Bei dieser Datei wird sowohl der Name des `JMS-Providers` angegeben als auch die notwendigen `Listeners` und die `Action-Pipeline` des Dienstes festgestellt.
- `deployment.xml`: Im Rahmen dieser Datei werden *deployment-spezifische* Daten eingestellt, wie z.B die Namen der Endpointreferenzen eines Services.
- `jbm-queue-service.xml`: In dieser Datei wird die Einstellung der Nachrich-

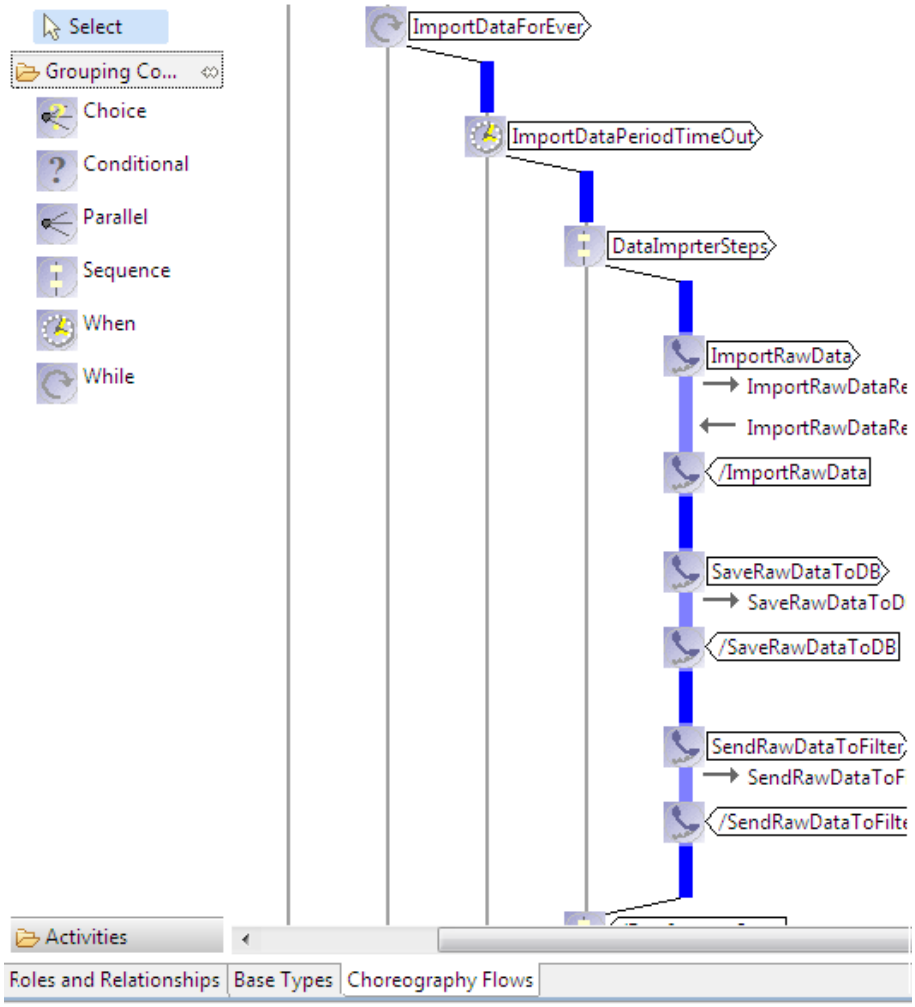

Abbildung 5.4: Teilen des FCD-Kette-Geschäftsprozesses

tenwarteschlangen festgelegt. Wie unter 2.4.1 erwähnt wird, steht jedem Dienst eine oder mehrere *Message-Queue* zur Verfügung. Diese dienen zum vorübergehend Speichern der ankommenden Nachrichten. Bezüglich des ESB-Clustering müssen einige [5] Warteschlangen geclustert werden. Abbildung 5.6 stellt ein Beispiel der Einstellung für einen Dienst aus der *FCD-Kette* dar.

Hierzu werden noch weitere Dateien definiert, die den Logik eines Dienstes verdeutlicht. JBoss ESB gestattet die Definition eigener Aktionen in der `Action-Pipeline`, wofür die erzeugte Aktion die Klasse `AbstractActionLifecycle` erweitern muss. Abbildung 5.7 bildet die internen Komponenten eines Dienstes ab.

[5] Die Queues, die für die interne Kommunikation innerhalb des ESBs zuständig sind, müssen nicht unbedingt geclustert. Im hingegen müssen die Queues, die für die externe Interaktion verantwotlich sind, geclustert werden.

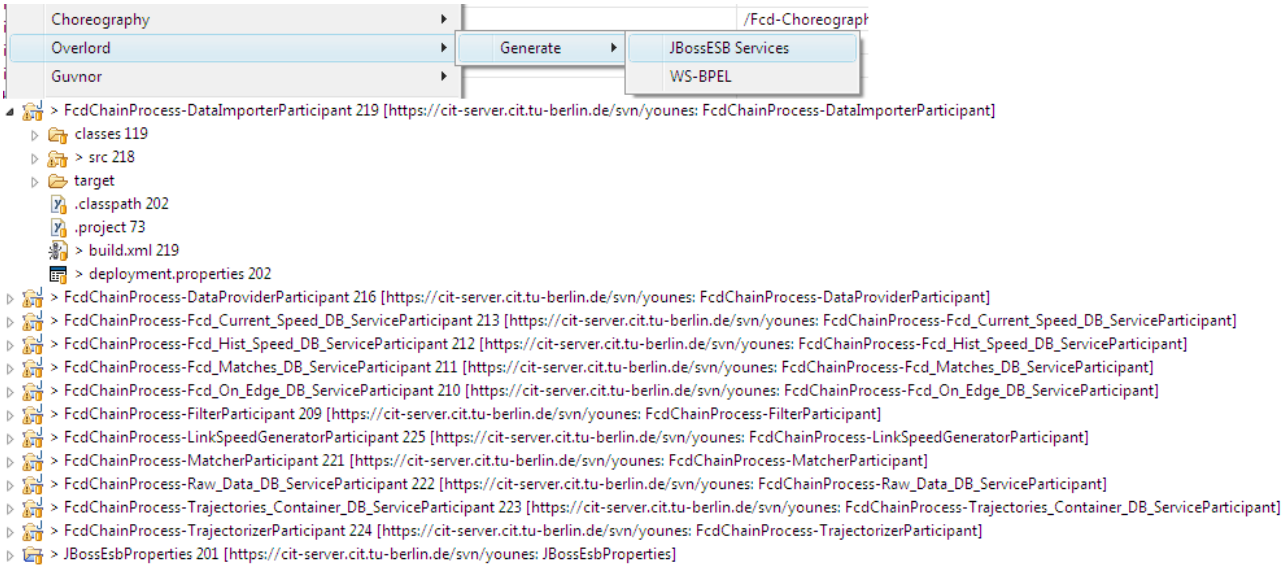

Abbildung 5.5: Erzeugung von Gerüsten der esb-services

```
<server>
    <mbean code="org.jboss.jms.server.destination.QueueService"
           name="org.pi4soa.fcdchain.fcdchain.destination:service=Queue
           xmbean-dd="xmdesc/Queue-xmbean.xml">
        <depends optional-attribute-name="ServerPeer">
            jboss.messaging:service=ServerPeer
        </depends>
        <depends>jboss.messaging:service=PostOffice</depends>
         <attribute name="Clustered">true</attribute>
    </mbean>
```

Abbildung 5.6: Abschnitte aus des `jbm-queue-service.xm`

5.3 Simulation des automatischen Abfangens der Lastspitzen basierend auf Cloud Computing

Um eine Überlast zu simulieren, werden die *FCD-Rohdaten* aus unterschiedlichen FCD-Rohdaten-Quellen gleichzeitig bearbeitet. Die Daten kommen in einer beliebigen Reihenfolge an, können aber nach anderen Distributionen wie *die Poisson Verteilung oder die Normaleverteilung* ankommen. Die Daten werden von sechs Anbietern zur Verfügung gestellt. Diese sind *Berlin, Hamburg, München, München_Gefos, Stuttgart und Wien.*

Eine Überlast wird erst erfasst, wenn die für den JBoss ESB bereitgestellte jvm eine Speicherkapazität besitzt, welche weniger als 50 MB beträgt. Diese Schwelle wird wie bei 4.4 erwähnt wurde in der *static* Variable `FREE_JVM_MEMORY_BARRIER` festgestellt. Zunächst wird der ESBClusterModule eine EC2 virtuelle Maschine (VM)

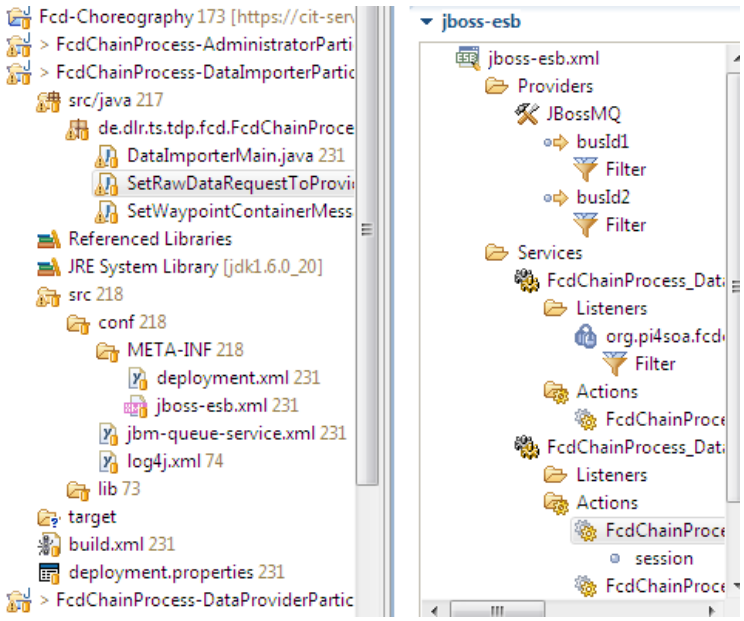

Abbildung 5.7: Die internen Komponenten eines ESB-services

starten. Die VM lädt automatisch einen JBoss ESB-Server hoch, welcher vorkonfiguriert ist, am vordefinierten Cluster anzuschließen. Der Cluster besitzt den Name ClusterESB und wird beim Starten jedes JBoss ESB-Servers anhand die Programm-Parameter `-g` [6] angegeben. Listing 5.1 zeigt die vollständigen jvm-Parametern, die beim Starten des Servers vorhanden sein müssen.

```
.\run.sh -c=clusteresb -b 192.168.178.21 -g ClusterESB
-Djboss.messaging.ServerPeerID=1
-Djboss.default.jgroups.stack=tcp
```

Listing 5.1: Programm-Parameter für das Starten des Servers

Anschließend tritt der ESBClusterManager in einer *Sleeping-Pahse* bei, in der er nicht mehr auf die eintreffenden gemessenen Daten reagiert. Die Dauer dieser Phase wurde in der *static*-Variable `SLEEPING_PHASE_PERIODE` auf 5 min festgestellt. Erst wenn diese Periode vorbei ist, übernimmt der ESBClusterManager wieder sein Aufgaben. Hierzu überprüft er, ob die jetzige zur Verfügung stehende Speicherkapazität über der Schwelle `50 MB` gestiegen ist. Zwei Möglichkeiten kommen hierbei in Betracht vor:

- **Wenn nein**: Der ESBClusterManager startet eine weitere EC2 Maschine.
- **Wenn ja**: Der ESBClusterManager fährt eine EC2 virtuelle Maschine, falls eine vorhanden ist, herunter.

[6] `-g`: festlegt die Gruppe, an der der startende Server sich anschließt

Solch ein Verfahren bewirkt eine flexible Skalierung des JBoss ESB-Cluster nach oben oder nach unten und dementsprechend eine schnelle Reaktion auf potentielle Lastspitzen.

Abbildung 5.8 zeigt den Status des JBoss ESB-Servers im Leerlauf an. Hierzu werden keine Rohdaten bearbeitet. Dies ist an der Anzahl der zu bearbeitenden Nachrichten (grüne Linie) zu bemerken, die bei `null` liegen. Zudem kann festgestellt werden, dass die zur Verfügung stehende freie Speicherkapazität im Bereich [`110 MB`-`170 MB`] liegt.

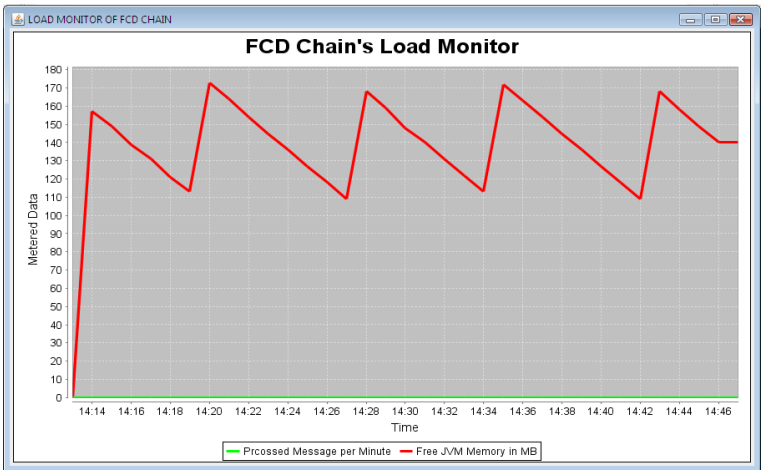

Abbildung 5.8: Monitoring des JBoss ESB-Servers im Leerlauf

Zuerst werden die Daten aus nur einem Data-Provider ankommen lassen. Abbildung 5.9 zeigt die gemessenen Werten. Man merkt schnell, dass die freie Speicherkapazität bis weniger als `70 MB` runter geht. Zudem wird bemerkt, dass die Anzahl der bearbeitenden Nachrichten gestiegen wird. In diesem Szenarium wird noch keine Überlast entdeckt.

Beim Hinzufügen eines zweiten Data-Provider wird erst eine Überlast festgestellt. Hierbei zeigt Abbildung 5.10, dass die freie zur Verfügung stehende Speicherkapazität einen Wert weniger als `50 MB` erreichtet und geht sogar bis zur `20 MB`. Parallel dazu wird die Anzahl der bearbeitenden Nachrichten weiter steigen. Darauf aufbauend startet der ESBClusterManager eine EC2 virtuelle Maschine, welche am ESB-Cluster anschließen soll. Die Informationen über die VM sowie seiner Status werden auf die *AWS Management Console* dargestellt.

Da die Realisierung des ESB-Clusteres aufgrund der im 4.5 bereitgestellten Problemen nicht möglich war, kann sich leider die gestartete virtuelle Maschine nicht am JBoss ESB-Clusters anschließen. Schlussfolgernd können Teile der ankommenden Anfragen nicht an dieser virtuellen Maschine im Rahmen von der Lastverteilung umgeleitet werden. Daher wird keine Erholung der Speicher beim originalen Server stattfinden.

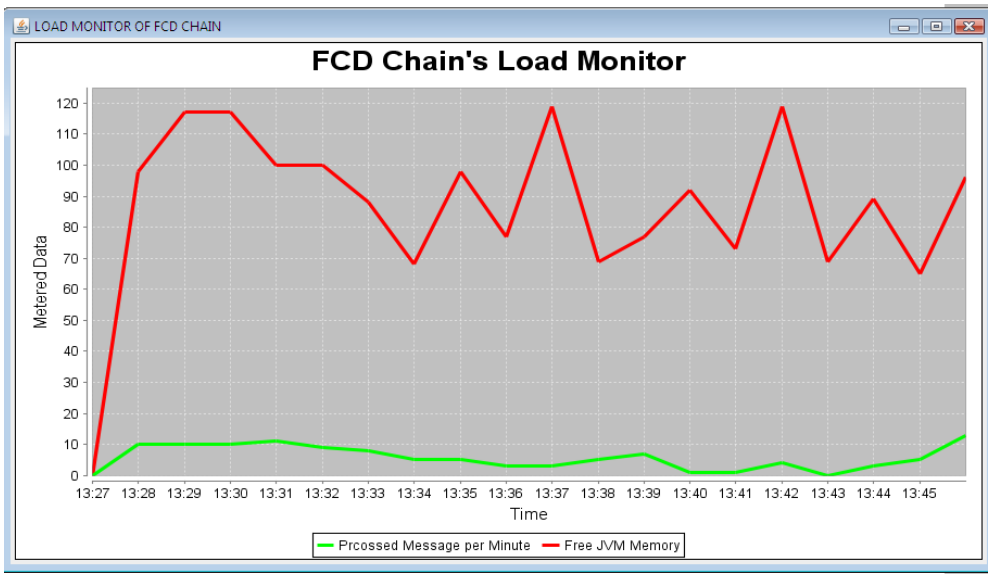

Abbildung 5.9: Monitoring bei einem Data-Provider

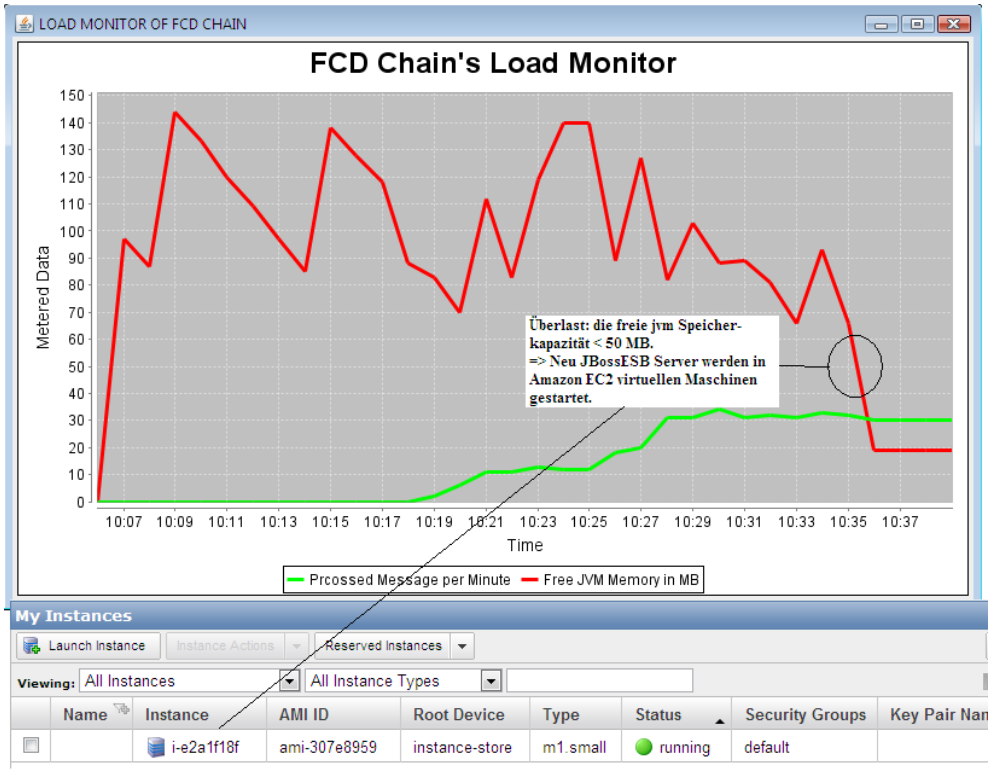

Abbildung 5.10: Automatisches Lastabfangen in Amazon EC2 Cloud

Kapitel 6

Zusammenfassung

Die Kommunikation zwischen verteilten Diensten wird heutzutage durch ein Enterprise Service Bus realisiert. Dabei können diese mit Hilfe einer Workflow-Engine zu höherwertigen Services komponiert werden. Durch unterschiedliche Lastprofile der einzelnen Aktivitäten innerhalb des Workflows können zeitweise einige Dienste überlastet sein und die Gesamtausführung des Systems hinaus zögern.

Sinn und Zweck dieser Arbeit bestand darin, aufbauend auf Analysen ein Konzept zur Integration eines Enterprise Serive Bus in einer elastischen *Infrastructur-as-a-Service (IaaS)* Umgebung zu erstellen. Außerdem sollte eine SOA-basierte Lösung für das *Traffic-Data-Platform/FCD-Prozessierungsmodul* entwickelt werden und darauf basierend ein Ansatz angefertigt werden, das Lastspitzen in SOA-Workflows auf Grundlage des Cloud Computings automatisch abfängt.

Nach der Erörterung grundlegender Konzepte und der Betrachtung weitverbreiteter OpenSource ESB-Produkte, ließ sich feststellen, dass keine ernsthafte Lösung angeboten wird, welche die Integration eines ESBs in einer IaaS gestattet und die Skalierbarkeit elastisch verwaltet.

Voraussetzung für diese Integration ist die Unterstützung der Clusterbarkeit. In diesem Zusammenhang wurden zwei Architekturen für den Aufbau eins ESB-Clusters vorgestellt, untersucht und verglichen. Ausgehend davon ergibt sich die Notwendigkeit, über ein neues Konzept nachzudenken, welches die Schwächen der vorhandenen Modelle eliminiert und eine Kombination anwendet, damit eine verbesserte Architektur entsteht.

Ein wesentlicher Teil dieser Ausarbeitung befasst sich mit der Erörterung der Anforderungen, nach deren Erfüllung mögliche Szenarien zur Integration eines ESBs in einer IaaS verglichen werden. Diese Anforderungen beschaffen eine optimale Elastizität und eine Kosteneffizienz. Zudem wurde ein Konzept für das automatische Abfangen der Anfragen-Lastspitzen in einer IaaS erarbeitet und darauf aufbauend ein Load-Monitoring-Framework entwickelt, das die vom ESB verbrauchten Ressourcen

überwacht sowie rechtzeitig meldet und die Werte in Form eines Charts darstellt. Im Falle einer Überlast reagiert dieses Framework automatisch und kräftigt den ESB-Cluster durch das Hinzufügen von Amazon-EC2 virtuellen Maschinen. Dadurch werden die Kundenanfragen gleichermaßen auf alle ESB-Instanzen verteilt und dementsprechend die überlasteten Ressourcen entlastet.

Zum Verwirklichen eines ESB-Clusters zwischen zwei unterschiedlichen Netzwerken und zwar dem Heimatnetzwerk und dem Amazon Netzwerk, wird das Produkt Amazon Virtual Private Cloud (VPC) verwendet, das die Erweiterung des Netzwerks einer Firma durch AWS Ressourcen gestattet. Aufgrund der Sicherheitsmaßnahmen, die sowohl das DLR-Netz als auch das TUBIT-Netz erheben, erschwert sich der Aufbau des Clusters. Daher wurde für diesen alternativ ein virtueller Server angemietet, um den Versuch neu zu starten. Mangels der notwendigen Funktionen im Kernel des angemieteten Servers, konnte der Aufbau des Clusters nicht realisiert werden. Die Funktionen umfassen Features für den Aufbau eines Virtual Private Networks wie IPSec und Werkzeuge für die Erstellung der Sicherheitsschlüssel. Eine Erweiterung des Kernels ist nicht möglich. Die alternative Lösung hierzu ist der Erwerb eines neuen vpn-fähigen Servers, dessen Kosten über den tragbaren Rahmen eines Studenten liegen.

Insgesamt erfüllt das vorgestellte Load-Monitoring-Framewrok seinen Zweck. Es wurde mit dem SOA-basierten entwickelten FCD-Prozessierungsmodul der DLR-Traffic-Data-Platform getestet. Im Fall einer Überlast werden automatisch neue EC2 virtuelle Maschinen gestartet und am ESB-Cluster angeschlossen. Es ist zu berücksichtigen, dass es sich hierbei um einen Prototypen handelt, der selbstverständlich noch nicht ausgereift ist, jedoch aufbaufähig ist.

Als weiterführende Aufgabe zu dieser Themenstellung, wäre der Aufbau des ESB-Clusters auf dem Eucalyptus Cloud der TU-Berlin von großem Interesse. Zudem wäre es interessant zu untersuchen, wie die Art und Weise des Eintreffens der Anfragen bei den Diensten, den Ressourcenverbrauch sowie die entstehenden Kosten, beeinflusst. All diese Faktoren könnten leichter überblickt und somit kontrolliert werden.

Literaturverzeichnis

[aw10] amazon webservices. Häufig gestellte Fragen zu Amazon VPC. `http://aws.amazon.com/de/vpc/faqs/#1`, November 2010.

[Ban10a] Bela Ban. JavaGroups - A Group Communication Toolkit for Java. `http://www.cs.cornell.edu/home/bba/`, Novemebr 2010.

[Ban10b] Bela Ban. Reliable Multicasting with the JGroups Toolkit. Technical report, Red Hat Inc, Oktober 2010.

[BGPS] B.Stansberry, G.Zamarreno, P.Ferraro, and S.Kittoli. JBoss AS 5.1 Clustering Guide : High Availability Enterprise Services with JBoss Application Server Clusters. Technical report, Red Hat, Inc.

[BHR07] Andreas Buchholz, Rüdiger Hohloch, and Tim Rathgeber. Vergleich von Open Source ESBs. Technical report, Institut für Architektur von Anwendungssystemen-Universität Stuttgart, 2007.

[DLR10] DLR. Das DLR im Überblick. `http://www.dlr.de/desktopdefault.aspx/tabid-636/1065_read-1465/`, November 2010.

[DT10a] DLR-TS. Institut für Verkehrssystemtechnik. `http://www.dlr.de/fs/DesktopDefault.aspx/tabid-1221/1665_read-3070/`, November 2010.

[DT10b] DLR-TS. Institut für Verkehrssystemtechnik: Abteilungen. `http://www.dlr.de/fs/desktopdefault.aspx/tabid-1227/`, November 2010.

[ELMT09] J.A. Estefan, K. Laskey, F.G. McCabe, and D. Thornton. Reference Architecture Foundation for Service Oriented Architecture Version 1.0. Technical report, OASIS, 2009.

[Emb09] Georg Ember. Cloud Services Dynamische Infrastrukturen für serviceorientierte Architekturen. *IBM Deutschland GmbH*, 2009.

[ESB09] Open ESB. GlassFish ESB v2.1 on OpenSolaris 2008.11 32-bit AMI. `https://open-esb.dev.java.net/servlets/NewsItemView?newsItemID=7557`, August 2009.

[FIF+10] F.Baude, I.Filali, F.Huet, V.Legrand, E.Matthias, P.Merle, C.Ruz, R.Krummenacher, and E.Simperl. ESB Federation for Large-Scale SOA. *ACM*, 2010.

[Fin] Torsten Fink. Die soa platform von jboss.

[Fin09] Torsten Fink. Die SOA-Plattform von JBoss. *JavaSPEKTRUM*, Januar 2009.

[Góm10] Jorge Carlos Marx Gómez. Serviceorientierte Architektur. `http://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Serviceorientierte-Architektur`, Novemebr 2010.

[Gol10] Marek Goldmann. BoxGrinder Build: Appliance Definition File. `http://community.jboss.org/wiki/BoxGrinderBuildApplianceDefinitionFile`, November 2010.

[Hol10] DR. Bernhard Holtkamp. Cloud Computing für den Mittelstand am Beispiel der Logistikbranche. Technical report, Fraunhofer-Institut für Software- und Systemtechnik ISST, Juli 2010.

[htt10] http://www.expertenthema.de. Cloud Computing - was ist das? Definition Cloud Computing. `http://www.expertenthema.de/text/cloud-computing`, November 2010.

[IMC05] I.Hildebrandt, M.Wagner, and C.Stützner. Evaluierung verschiedener Szenarien zur Anwendung von BPEL und WS-CDL. Technical report, Universität Stuttgart-Institut für Architektur von Anwendungssystemen (IAAS), Oktober 2005.

[JBo07] JBoss. JBoss Messaging 1.4.0.GA User Guide. Technical report, Red Hat, Inc, Oktober 2007.

[JBo10a] JBoss. JBoss Enterprise SOA Platform 5.0.1 ESB Programmers Guide. Technical report, Red Hat, Inc, 2010.

[JBo10b] JBoss/Redhat. JBoss Enterprise SOA Platform 5.0.1 ESB Programmers Guide. Technical report, Red Hat, Inc, 2010.

[KBS07] Dirk Krafzig, Karl Banke, and Dirk Slama. *Entreprise SOA*. mitp, 2007.

[Köh03] Kristian Köhler. JMX - Management ohne Wasserkopf. `http://www.oio.de/public/java/jmx/jmx.htm`, Februar 2003.

[Lab05] Guru Labs. CREATING RPMS (Student Version) v 1.0. Technical report, Guru Labs, 2005.

[Mas08] Ross Mason. Mule Clustering. `http://www.mulesoft.org/display/`

MULEUSER/Clustering, Januar 2008.

[mas10] mastertheboss.com. JBoss ESB with JBoss Tools plugin. http://www.mastertheboss.com/soa-a-esb/78-jboss-esb.html, Novemebr 2010.

[McW10] Bob McWhirter. BoxGrinder. *JUDCon*, 2010.

[mic09] Sun microsystems. Configuring glassfish esb for clustering. Technical report, Sun microsystems, 2009.

[Möl10] Sascha Möllering. JBoss EAP Clustering. *Javamagazin*, November 2010.

[Mul10a] MuleSoft. High Availability for Mule ESB. http://www.mulesoft.com/downloads/high-availability-for-mule-esb.pdf, April 2010.

[Mul10b] MuleSoft. Mule ESB: The Leading Open Source Enterprise Service Bus. http://www.mulesoft.com/downloads/mule-esb.pdf, April 2010.

[OM10] Noel O'Connor and E. Meuwese. Running Infinispan on Amazon Web Services. http://community.jboss.org/wiki/RunningInfinispanonAmazonWebServices, Dezember 2010.

[RB07a] U. Rerrer-Brusch. Serviceorientierte Architekturen, Kap.1: Einführung. *Vorlesung SOA, TU-Berlin*, 2007.

[RB07b] U. Rerrer-Brusch. Serviceorientierte Architekturen, Kap.2: Technologien. *Vorlesung SOA, TU-Berlin*, 2007.

[RB07c] U. Rerrer-Brusch. Serviceorientierte Architekturen, Kap.7: Komposition von Web Services. *Vorlesung SOA, TU-Berlin*, 2007.

[Rüc08] Bernd Rücker. SOA mit dem JBoss ESB. Technical report, camunda services GmbH, 2008.

[Sch00] Thomas Schletter. Das RPM Buch. http://web.archive.org/web/20070314130933/www.xinux.de/docs/linux/rpm/node3.html, Februar 2000.

[Scm10] Dr. Holger Scmidt. Open Source Entreprise Service Bus(ESB): Ein Vergleich aktueller Produkte. Technical report, Ancud IT-Beratung GmbH, 2010.

[ser10] Amazon Web services. Get Started with Amazon VPC. http://docs.amazonwebservices.com/AmazonVPC/latest/GettingStartedGuide/, Dezember 2010.

[S.L09] David S.Linthicum. *Cloud Computing and SOA Convergance in Your Enterprise A Step-by-Step Guide*. Addison-Wesley, September 2009.

[THW05] Ron Ten-Hove and Peter Walker. Java Business Integration. Technical

report, Sun Microsystems, 2005.

[TJE09] Anthony T.Velte, Toby J.Velte, and Robert Elsenpeter. *Cloud Computing A Practical Approach*. Tata Mcgraw Hill Education Private Limited, 2009.

[TREY10] L.C Touko Tcheumadjeu, R.Sten, E.Brockfeld, and Y.Yahyaoui. Traffic Data Platform based on the service oriented architecture (SOA). Technical report, German Aerospace Center (DLR) - Institute of Transportation Systems, Juli 2010.

[Tro05] Bernd Trops. Java Business Integration. *JavaSPEKTRUM*, Mai 2005.

[Web10] Amazon Webservices. Amazon Webservices Produkte. `http://aws.amazon.com/de/`, November 2010.

[Wik10a] Wikipedia. Dienstorientierte architektur. `http://de.wikipedia.org/wiki/Dienstorientierte_Architektur`, Oktober 2010.

[Wik10b] Wikipidea. Java Business Integration. `http://en.wikipedia.org/wiki/Java_Business_Integration`, Oktober 2010.

[WQH+08] W.Xue, Q.Wang, H.Xu, X.Luo, and Y.Li. A Based Distibuted Service Bus Platform Design. *ACM*, 2008.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Anhang

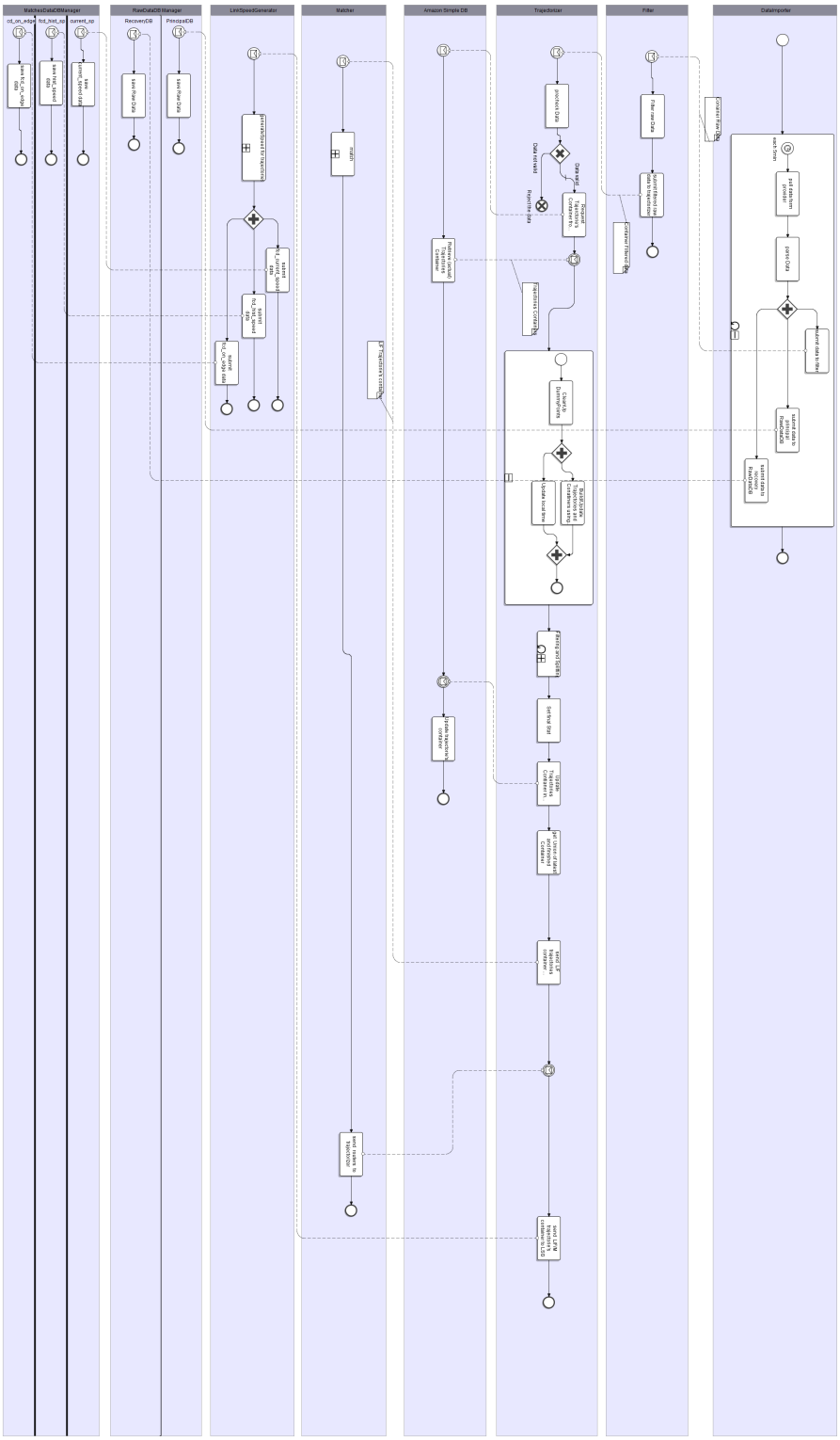

Abbildung 6.1: BPMN Modellierung der Flow in der FCD-Kette

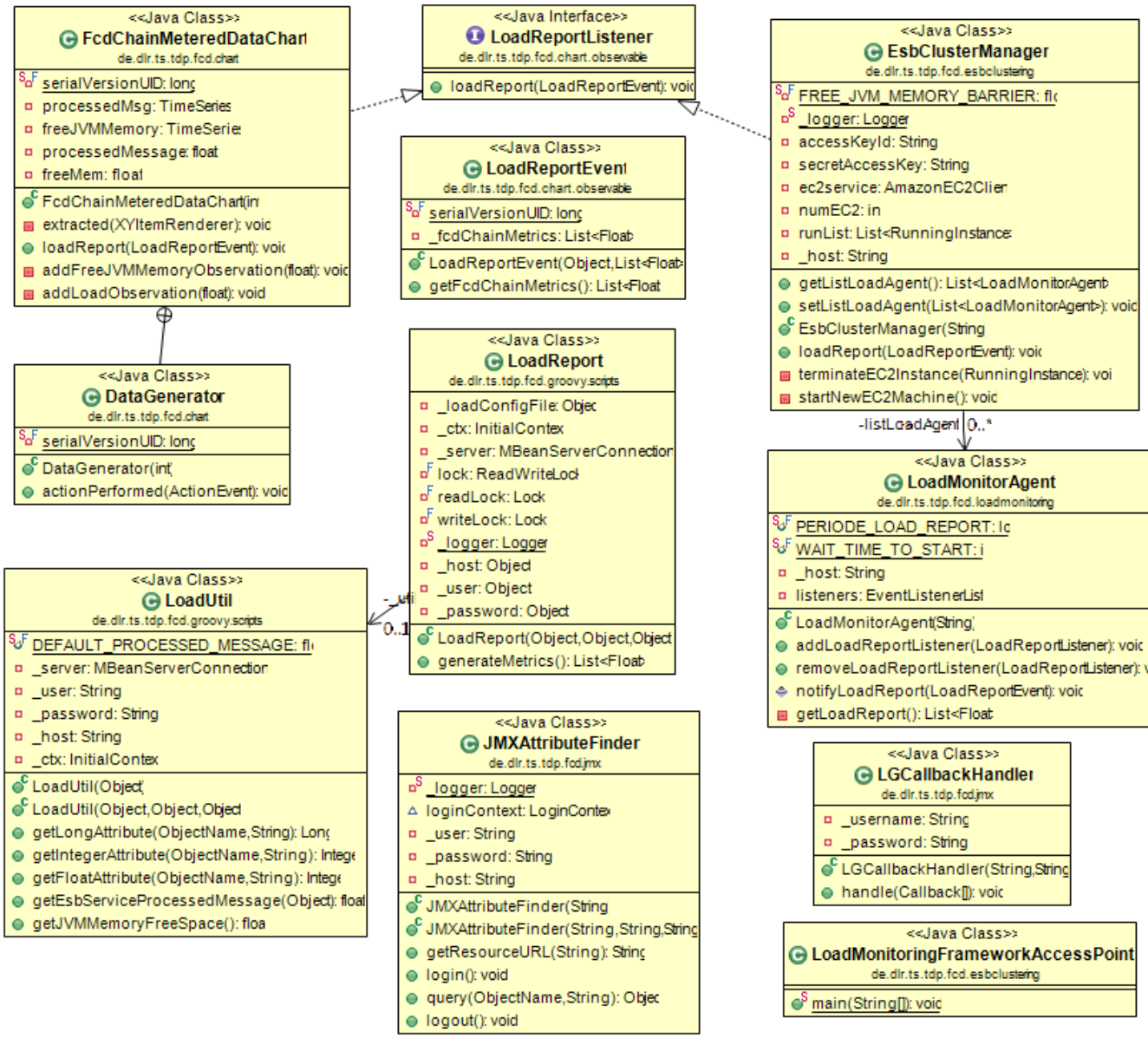

Abbildung 6.2: Klassendiagramm des LoadMonitoringFramewroks

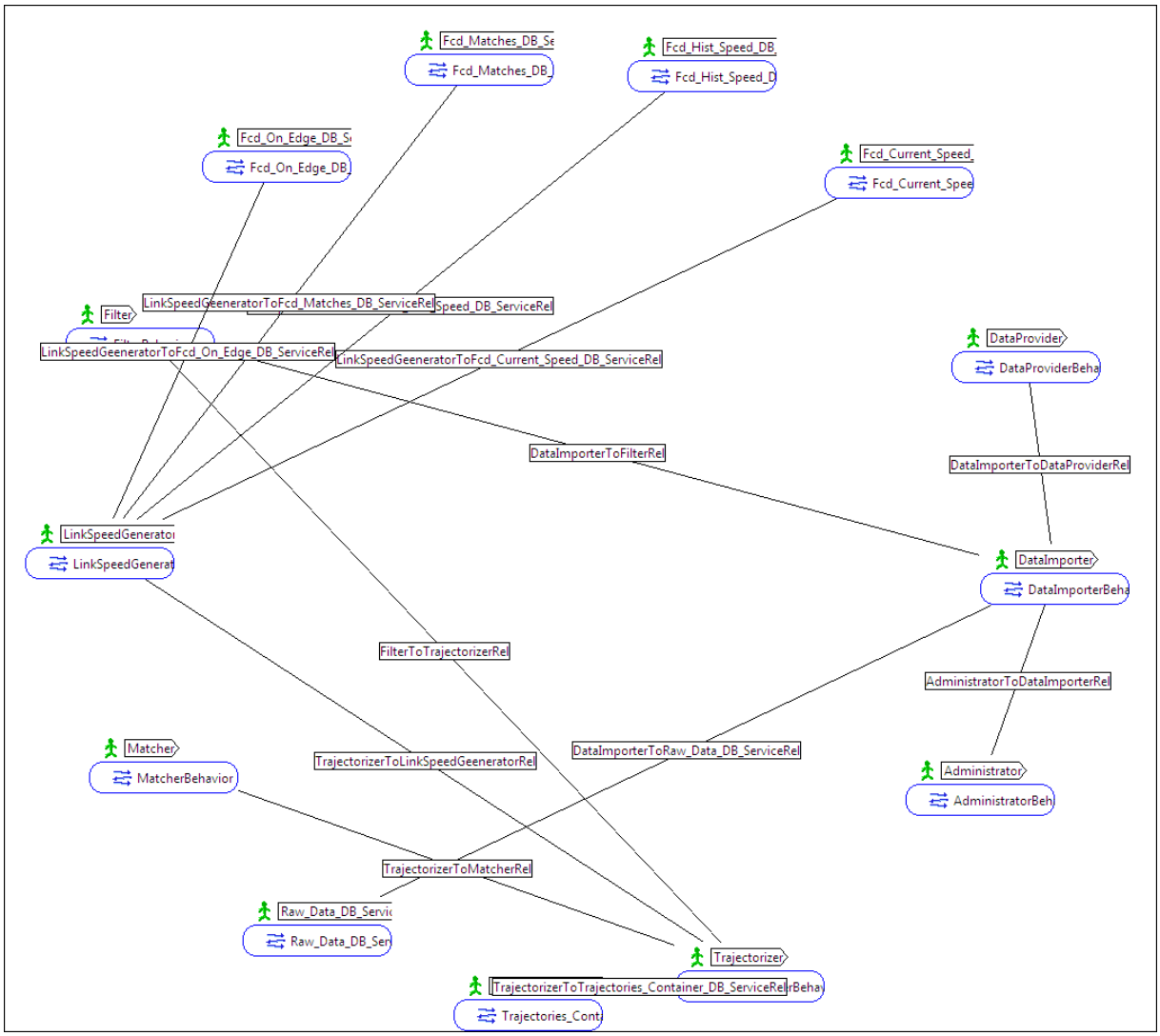

Abbildung 6.3: Graphische Darstellung der Relationship zwischen den Diensten

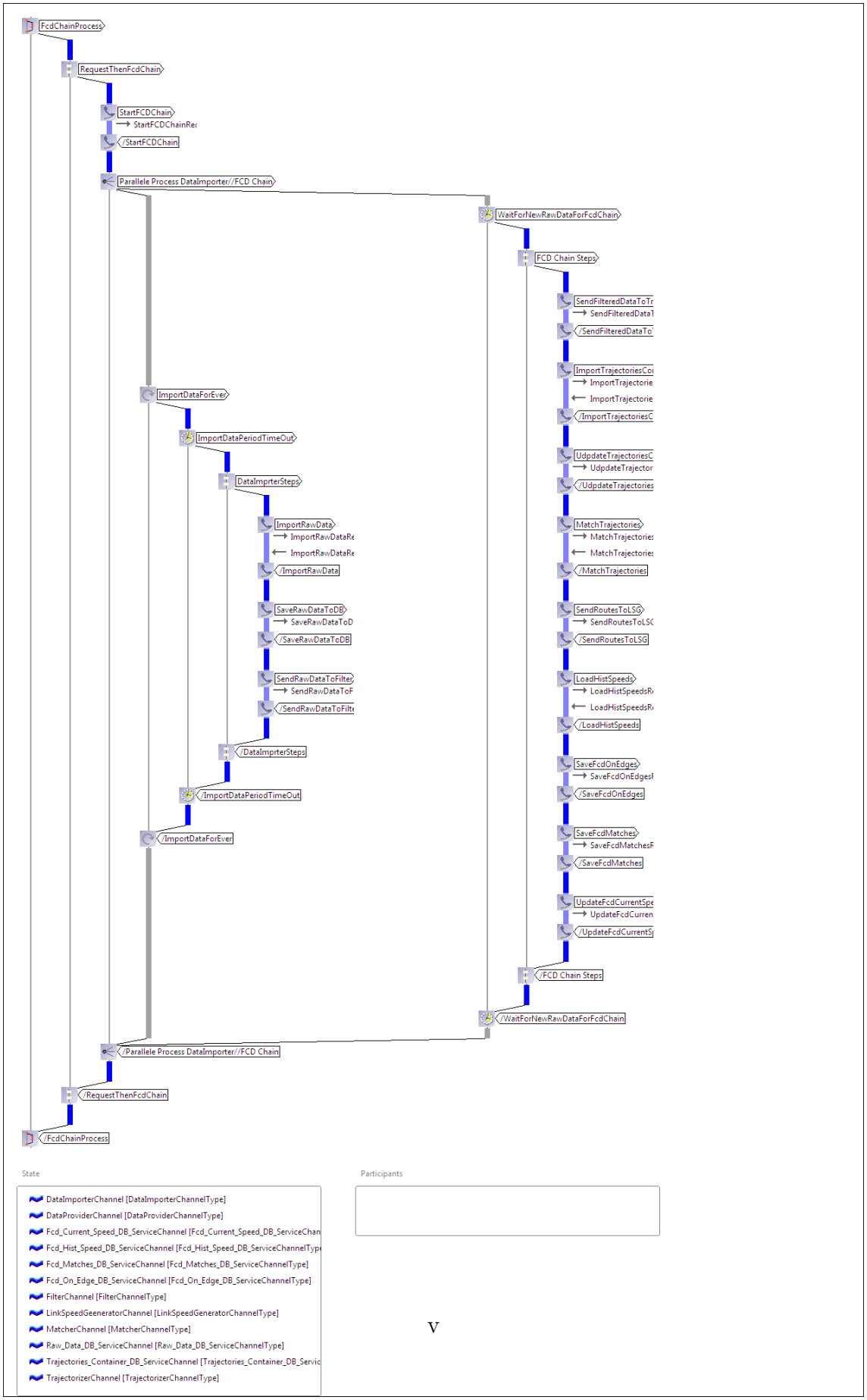

Abbildung 6.4: Graphische Darstellung des FCD-Geschäftsprozesses