INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Anforderungen und Implementierung von dynamischen und verteilten Services einer hochverfügbaren Online-Plattform

Luka Leovac

Aufgabensteller:	Prof. Dr. Martin Wirsing
Betreuer:	Dr. Nora Koch Dr. Andreas Binder (FriendScout24)
Abgabetermin:	21. Dezember 2009

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 21. Dezember 2009

. . . . . .
(Unterschrift des Kandidaten)

Vorwort

Bei Herrn Prof. Dr. Martin Wirsing bedanke ich mich für die Vergabe und Betreuung der Diplomarbeit. Besonderen Dank schulde ich Frau Dr. Nora Koch, die mich durch ihre engagierte Betreuung und stete Diskussionsbereitschaft mit vielseitigen Denkanstößen bereicherte und bei der Erstellung dieser Arbeit unterstützt hat.

Vielen Dank dem gesamten FriendScout24 GmbH Team, das mir die Erstellung dieser Arbeit ermöglicht hat. Mein besonderer Dank gilt auch für die hilfreichen Anregungen und tatkräftige Unterstützung an Dr. Andreas Binder.

Abstrakt

Große Internetportale, wie FriendScout24 (FRS24), haben immer mehr Mitglieder und bieten laufend neue Funktionalitäten an. Um neue Mitglieder zu gewinnen und diese auch auf der Plattform zu behalten ist ein System erforderlich, welches nicht nur 24 Stunden / 7 Tage die Woche verfügbar ist (99,99%), sondern auch sehr effizient und mit minimalem Aufwand neue Funktionen als Services bereitstellt. Kunden sollen zusätzlich die bereitgestellten Services nicht nur im Browser oder am Rechner, sondern auch im mobilen Bereich nutzen können.

Die Installation von neuen Services ist in vielen Fällen mit einem Systemneustart verbunden. Hiermit ist die Erreichbarkeit der Services nicht in vollem Umfang gewährleistet. Außerdem beinträchtigt das Starten und Stoppen von einzelnen Services die Verfügbarkeit einzelner Funktionen. Hinzu kommen Abhängigkeits- und oder Versionierungsprobleme der einzelnen Services, welche dessen Weiterentwicklung und Verwaltung zusätzlich erschweren können.

Ziel dieser Diplomarbeit ist es die Machbarkeit eines Softwaresystems zu untersuchen, das verteilte Services dynamisch bereitstellt, versioniert und verwaltet ohne dessen Verfügbarkeit zu beeinträchtigen. Hierzu werden im ersten Teil der Arbeit verschiedene Frameworks untersucht und anhand eines Kriterienkatalogs verglichen. Dann werden verschiedene Implementiersmöglichkeiten der Frameworks vorgestellt und analysiert. Anhand der Ergebnisse des Kriterien- und Implementierungsvergleichs wird ein Framework für die Umsetzung des FRS24 Portals vorgeschlagen.

Inhaltsverzeichnis

1 Einleitung

1.1 Einführung

Die Internetnutzung ist ein Teil unseres Alltags geworden. Internet wird genutzt, um nach Informationen zu suchen und sich über die neuesten Nachrichten zu informieren. Internet ist für uns außerdem ein Unterhaltungsmedium, mithilfe dessen man z.B. mit oder gegen andere Benutzer verschiedene Online-Spiele spielen kann. Daten werden immer mehr über das Internet ausgetauscht. Das Internet bietet Einkaufsmöglichkeiten und wird auch als Kommunikationsmedium genutzt.

Große Portale stellen den Menschen alle diese Optionen zu Verfügung, von Nachrichtenportalen, Suchmaschinen, Einkaufsportalen bis hin zu verschiedenen Kommunikationsportalen. Kommunikationsportale sind für die Menschen immer wichtiger geworden. Dort kann man mit Freunden oder Bekannten Nachrichten, Bilder, Videos, und vieles mehr austauschen. Die Kommunikationsportale bilden verschiedene Sozial-, Business- und auch Partnernetzwerke. Die Nutzung solcher Portale hat sich in der nahen Vergangenheit mehrmals vervielfacht. Somit steigen auch die Anforderungen an die Systeme, die diese Kommunikationsplattformen für die Nutzer bereitstellen. Es steigt nicht nur die Anzahl der Nutzer und Funktionen, die den Benutzern zur Verfügung gestellt werden, sondern auch die Anforderungen an die Internetgeschwindigkeit. Somit steigen auch täglich die Anforderungen an die Plattformen. Sie müssen nicht nur eine Verbindung mit große Bandbreite haben, sondern auch eine Plattform, bieten die sehr vielen Nutzern gleichzeitig einen schnellen Zugriff auf Funktionen (Services) verschafft.

Zu einer dieser Kommunikationsplatformen gehört auch FriendScout24 (im Folgenden nur FRS24). FRS24 ist Deutschlands Partnerbörse Nr. 1 mit mehr als 10 Millionen Mitgliedern. FRS24 ist nicht nur in Deutschland vertreten, sondern auch in anderen europäischen Ländern wie Spanien, Italien, Österreich, Schweiz, Frankreich und den Niederlanden. Die Zahl der Mitglieder (Synonyme: User, Benutzer), die parallel auf der FRS24 Plattform eingeloggt sind, steigt zu Hochzeiten in den mittleren fünfstelligen Bereich. Die Zahl der Mitglieder steigt stetig weiter und es werden immer neue Funktionen für die Mitglieder hinzugefügt.

1.2 Motivation

Die Anforderungen an das Portal sind sehr groß, angefangen mit der Bereitstellung der Funktionen für die große Anzahl von Benutzer im fünfstelligen Bereich, die parallel online sind, über neue Funktionen bis hin zu der Hochverfügbarkeit der FRS24 Plattform. Alle diese Eigenschaften unter einen Hut zu bringen, ist sehr anspruchsvoll, da bei diesen Anforderungen Widersprüche entstehen. Wenn z.B. den Benutzern neue Funktionen bereitgestellt werden, muss sichergestellt werden, dass die neuen Funktionen die alten nicht beeinträchtigen und somit die Stabilität der Plattform gefährden. Wenn ein Service, der eine Funktion bereitstellt, nicht mehr verfügbar ist, soll die Plattform einen anderen entsprechenden Service

bereitstellen.

Um neue Mitglieder zu gewinnen und diese auch auf der FRS24 Plattform zu behalten, ist ein System für die Bereitstellung der Funktionen erforderlich, welches nicht nur 24 Stunden / 7 Tage die Woche verfügbar ist (99,99%), sondern auch sehr effizient und mit minimalem Aufwand neue Funktionen als Services bereitstellt. Ziel dieser Diplomarbeit ist es, die Machbarkeit eines Softwaresystems zu untersuchen, das verteilte Services dynamisch bereitstellt, versioniert und verwaltet, ohne dessen Verfügbarkeit zu beeinträchtigen. Außerdem soll dieses Softwaresystem die Entwicklung, Betrieb und Wartung der Services so erleichtern, indem es auf standardisierte und weit verbreitete Komponenten setzt.

1.2.1 Lösungsansatz

Um alle Anforderungen des FRS24 Portals so weit wie möglich zu erfüllen, soll ein Framework für die Entwicklung, Betrieb und Wartung der Services gefunden werden. Die Entscheidung, welches Framework auch genommen werden soll, ist nicht einfach, deshalb wird ein Kriterienkatalog erstellt, anhand dessen die Frameworks evaluiert und verglichen werden. Dieser Katalog wird Anforderungen, wie die Verteilung, Versionierung, Wartung, Standard Komponenten und weitere, widerspiegeln. Für die Evaluierung wurden Frameworks ausgesucht, die den OSGi [osg09a] Standard realisieren, da dieser Standard viele Lösungen der FRS24 Anforderungen mit sich bringt. Es werden folgende Frameworks evaluiert: Newton [new09], Paremus Service Fabric [par09] und Apache CXF DOSGi [dos09].

1.2.2 Ergebnisse

Anhand der Evaluierung und dem Vergleich der Frameworks wird für die Umsetzung der FRS24 Plattform das Apache CXF DOSGi Framework empfohlen. Der Vergleich (Kapitel 4.6) zeigt, dass das Paremus Service Fabric und Apache CXF DOSGi ebenbürtig sind. Der große Unterschied liegt in der Bereitstellung des Zugriffs auf die Services in den Containern. Service Fabric baut auf einer Eigenentwicklung auf, die über RMI realisiert wird und CXF DOSGi setzt auf Webservices. Apache CXF bietet viel mehr Möglichkeiten für Zugriffe von anderen Plattformen, auf die Services aus, da Webservicezugriffe weit verbreitet sind und von allen Plattformen aus aufgerufen werden können.

1.3 Vorgehensweise

Diese Arbeit beschreibt in Kapitel 2 die Ausgangssituation des bestehenden FRS24 Systems. Auf diesen Schritt war besondere Sorgfalt zu legen, damit die Anforderungen in Kapitel 3 anhand eines Kriterienkatalogs beschrieben werden können. In Kapitel 4 wird kurz die zu verwendende Technologie vorgestellt und Frameworks ausgesucht, die anhand des Kriterienkatalogs detailliert evaluiert werden. Kapitel 5 stellt die verschiedenen Implementierungen, welches Framework unter welchen Bedingungen geeigneter ist. Anhand des 4. und 5. Kapitels wird in Abschnitt 5.6 eine Empfehlung ausgesprochen, welches Framework verwendet werden sollte. Kapitel 6 gibt eine Zusammenfassung der gesamten Arbeit wieder und beschreibt anschließend Möglichkeiten zum weiteren Vorgehen.

2 Ausgangssituation

In diesem Kapitel wird zuerst das bestehende System der FRS24 Plattform beschrieben. Bei der Beschreibung wird auf die Architektur der einzelnen Komponenten und das Zusammenspiel der Services eingegangen. Danach werden die auftretenden Probleme analysiert und Verbesserungsmöglichkeiten genannt.

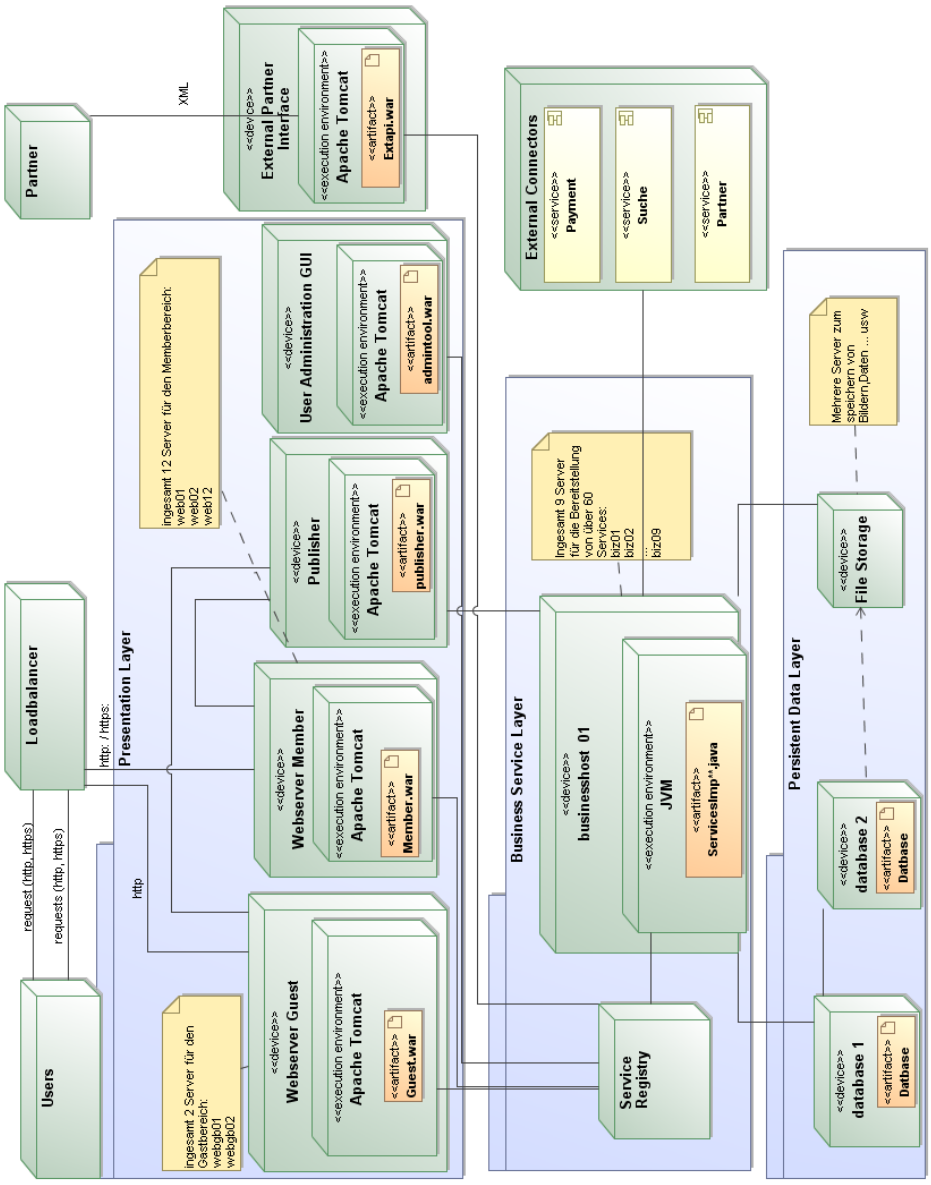

Abbildung 2.1: UML Deploymentdiagramm der aktuellen Systemarchitektur

2.1 Bestehendes System

Das bestehende FRS24 System ist nach dem dreischichtigen (3 Layer) Architekturkonzept [DEF+08, Seite 42] aufgebaut: Presentation Layer, Business Service Layer und Persistent Data Layer (siehe Abbildung 2.1). Diese drei Schichten spiegeln die grundsätzlichen Aufgaben des FRS24 Softwaresystems wieder: Eine Benutzeroberfläche und Services, die Geschäftsprozesse realisieren sowie Services für die persistente Verwaltung der Benutzerdaten.

2.1.1 Presentation Layer

Die Präsentationsschicht dient dem Benutzter zur Interaktion mit dem Dienst. Dabei können zwei Aspekte unterschieden werden: Die Darstellung der Weboberfläche und die Dialogkontrolle.

Wie in Abbildung 2.1 wird das FRS24 Portal in den Webservern Guest und Webserver Member gehostet. Wenn ein Benutzer die URL von FRS24 aufruft wird sein Request durch den Loadbalancer auf einen der Guest Webserver weitergeleitet. Dort wird die Weboberfläche an den Benutzer ausgeliefert. Ein Benutzter, der sich hier einloggt oder registriert, wird auf einen Webserver im Memberbereich (für registrierter Benutzter) weitergeleitet. Die Weboberfläche wird den Benutzern mit Hilfe folgender Technologien umgesetzt: Java Server Pages (JSP) [jsp09], Javascript [jav09] und CSS [CSS09]. Die Weboberfläche wird mit Hilfe von Apache Tomcat [tom09] Webservern über ein HTTP Protokoll zur Verfügung gestellt.

Desweiteren übernimmt die Präsentationsschicht die Dialogkontrolle. Sie sendet die vom Benutzer eingegebenen Daten an die Business Service Schicht und bereitet sie für die Darstellung der Folgefenster auf. Für die Dialogkontrolle und die Darstellung der Daten des Business Service Layers wird ein selbst erweitertes Apache Struts Framework [apa09] verwendet.

Neben den Portal existiert in der Persistenzschicht auch die User Administration GUI (auch Admintool genannt) und der Publisher (FRS24 Content Management System - CMS). Das Admintool dient denm Kundenservice dazu Benutzerdaten zu verwalten. Durch das Admintool können die Texte und Bilder der Benutzer freigeschaltet werden, d.h den anderen Benutztern sichtbar gemacht werden. Der Publisher dient als Redaktionssystem für Texte und Bilder und für die Verwaltung der Übersetzungen in andere Sprachen. Der Publisher ist eine Eigenentwicklung von FRS24 und bietet eine Benutzeroberfläche für die Redaktion an Texte und Bilder des Portals zu ändern ohne dabei das System negativ zu beeinträchtigen.

2.1.2 Business Service Layer

Hier werden die FRS24 Portal Geschäftsprozesse realisiert. Der Business Service Layer besteht aus 5 Komponenten:

- Services - Die Services stellen die eigentlichen Geschäftsprozesse zur Verfügung und reagieren auf Anfragen, die von der Präsentationsschicht kommen. Die Services benutzen die Persistent Service Layer um Benutzerdaten abzufragen und zu speichern.

- Service Registry. Hier registrieren sich die Services, die die Geschäftsprozesse darstellen. Die registrierten Services stehen dann den Clients (Guest, Member, ExtApi...) zum Auffinden von Serviceschnittstellen zur Verfügung. Diese beschreiben dann wiederum welche Fuktionen (Methoden) die Services zu Verfügung im Detail bereitstellen.

- Business Hosts - Server, in den die eigentlichen Implementierungen der Services laufen. Insgesamt laufen über 50 Services im Live-Betrieb, die auf 9 Business Server verteilt sind.
- External Connectors - Payment, Suche, Partner, ... sind Konnektoren, die Verbindungen zur externen Partnern herstellen. Über einen externen Dienstleister wird die Suche der FRS24 Plattform ausgeführt. Zudem existieren Konnektoren zu Partnern, die Bezahldienste via Kreditkarte oder andere Methoden anbieten. Außerdem werden den Benutzern wissenschaftliche Persönlichkeitstests über Partner-Seiten angeboten. An die Partner werden statistische Informationen über das Benutzerverhalten gesendet, um das Portal anhand der Auswertungen der Daten immer wieder verbessern zu können.
- External Partner Interface - repräsentiert eine externe Schnittstelle, über die Systemfunktionen über HTTP XML-Requests aufgerufen werden können. Über die Schnittstelle ist es möglich neue Benutzer zu registrieren, Benutzer einzuloggen, nach Partnern zu suchen, die Anzahl der ungelesenen Nachrichten und Chats abzufragen, usw. Es gibt die Möglichkeit für jeden Partner den Zugriff auf bestimmte Funktionalitäten über Konfigurationsdateien zu definieren. Diese Schnittstelle parst die eingehenden Daten und ruft bestimmte Services im System aus. Somit können Benutzer über die WAP Version, die von externen Partner enwickelt wird, FRS24 Services benutzen. Über Partnerseiten können Suchen im FRS24 System ausgeführt werden und die Ergebnisse dann beim Partnern angezeigt werden.

Der Business Service Layer ist lose von den anderen beiden Schichten gekoppelt. Somit weiß der Business Service Layer nichts über die Webdarstellung, die Dialogkontrolle und darüber wie die Daten in der Persistenzschicht verwaltet werden.

2.1.3 Persistent Data Layer

Die Persistenzschicht sorgt dafür, dass Daten dauerhaft gespeichert und auch geladen werden können. Bei FRS24 gibt es 3 verschiedene Komponenten für die Haltung der Persistenten Daten:

- Datenbank speichert die Benutzerdaten in einer relationalen Datenbank. Um die Daten ausfallsicher zu machen existieren zwei Datenbanken, die synchronisiert sind. Die Datenstruktur hat ca. 50 verschiedene Tabellen. Dort stehen Daten mit allen Informationen über den Benutzer, z.B. Texte, gespeicherte Suchen, Favoriten und Links zu den Bildern.
- File Storage - Bilder und Messaging Daten werden auf einem SAN (Storage Area Network) Plattenarray gespeichert. Weitere Informationen über SAN kann in [SAN09] gefunden werden
- Suchindex. Die Suche wird über einen externen Dienstleister realisiert. Dieser erstellt mit Hilfe der Datenbank einen Suchindex aus dem die Suche ihre Ergebnisse liefert. Der Suchindex wird auf der SAN gespeichert und bei Änderungen aktualisiert.

  Zusammenspiel der Komponenten - Bespiel: Wenn ein Benutzter ein Photo hochlädt wird es auf den File Storage Server gespeichert und in der Datenbank wird ein Eintrag mit den Eigenschaften (Benutzer, ID, Reihenfolge, Art) des Photos hinterlegt.

2.1.4 Common Object Request Broker Architecture- CORBA

Das System benutzt CORBA um einen Informationsaustausch zwischen den Webservern und Services zu ermöglichen. Auf einen CORBA Kern wird ein von FRS24 erstelltes Framework (Vincent) benutzt um die Kommunikation zwischen verschieden Servern zu ermöglichen.

Die Common Object Request Broker Architecture ist eine Spezifikation der Object Management Group OMG. Die OMG ist ein Industriekonsortium, das 1989 gegründet wurde und das Ziel hat, durch die Definitionen von Industriestandards den Einsatz Objektorientierter Software zu fordern. Mit CORBA hat sich damit quasi die gesamte Software-Industrie auf die gemeinsame Definition eines verteilten Objektmodells geeinigt. CORBA unterstützt die Kooperation verteilter Anwendungen vom sprachunabhängigen Methodenaufrufen und ist für nahezu alle verfügbaren Plattformen erhältlich. Insbesondere enthält Java seit der Version 2 bereits eine voll funktionstüchtige CORBA -Implementierung. [Haa08]

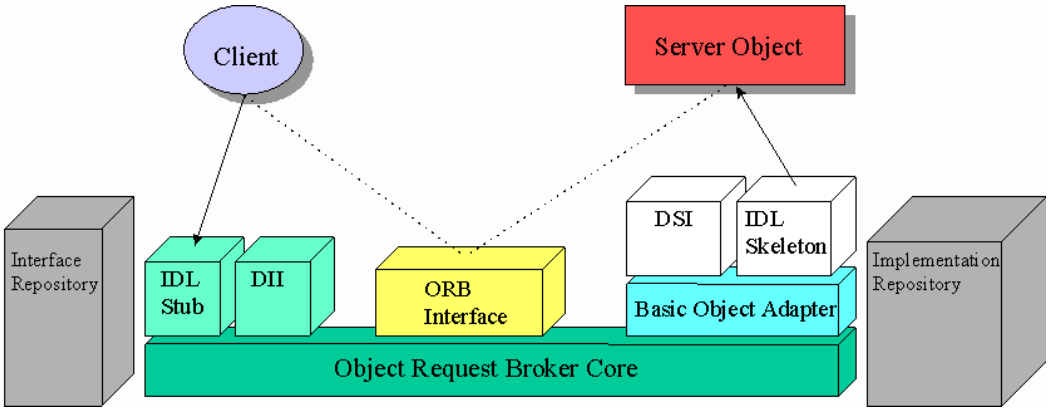

Abbildung 2.2: CORBA 2.0 Architektur Standard [Sta09]

Die **Interface Definition Language (IDL)** beschreibt die Schnittstelle (Interface) des Objektes d.h. im Wesentlichen die Deklarationen der einzelnen Methoden des Objekts. Die eigentliche Implementierung des Objekts bleibt dem Client verborgen. Jede Methode des Interfaces besitzt einen Namen, Eingabeparameter, Ausgabeparameter und einen Rückgabewert.

Wenn ein Client eine Methode am entfernten Server aufruft, wendet er sich in Wirklichkeit an ein lokales Stellvertreterobjekt des Servers, den sogenannten Stub. Der Stub befindet sich beim Client und bietet dieselbe Schnittstelle wie der Server. Der Server verpackt die Methode und die Methodenparameter in ein geeignetes Transportformat und schickt es an den Server. Das Verpacken wird auch Marshalling genannt. Auf der Serverseite wird der Aufruf von Skeleton empfangen und die Parameter der Methode in ihr Ausgangsformat konvertiert. Dieser Vorgang wird Demarchalling genannt. Der Skeleton reicht den Aufruf an die eigentliche Objekt-Implementierung weiter, welche den Aufruf bearbeitet und an den Skeleton zurückgibt. Der Skeleton übernimmt das Marshalling und schickt die Antwort an den Client.

Die Stubs und Skeletons setzen auf dem **ORB**, dem **Objekt Request Broker**, auf. Der ORB ist das Herzstück von CORBA. Er lokalisiert die entfernten Objekte, vermittelt den Methodenaufruf zwischen Client und Server und stellt dazu diverse Synchronisationsmechanismen zur Verfügung. Die ORBs einzelner Systeme unterhalten sich über das **Inter-**

net Inter-ORB-Protocol (IIOP), einer Protokollschicht, die über dem Internet-Protokoll TCP/IP liegt. [HKF03]

Abbildung 2.2 bildet den heutigen CORBA 2.0 Standard ab. Im bisher beschriebenen CORBA-Modell müssen Clients bereits zur Kompilierzeit Kenntnis über die von ihnen aufgerufenen CORBA-Objekte besitzen. Gerade in hochdynamisch konfigurierbaren und verteilten Systemen ist eine solche Einschränkung nicht akzeptabel. Daher bietet der Standard ein sogenanntes Dynamic Invocation Interface (DII) und ein Interface Repository. Letzteres enthält IDL-Beschreibungen über die im System registrierten CORBA-Schnittstellen, während das DII unter Zuhilfenahme dieser Typinformationen ein dynamisches Kreieren und Absetzen von Methodenaufrufen ermöglicht. Dem aufgerufenen CORBA-Server bleibt verborgen, ob der Client ihn über ein statisches Stub oder dynamisch über das DII aufgerufen hat. Auf der Serverseite steht als Pendant zum DII das sogenannte Dynamic Skeleton Interface (DSI) zur Verfügung. Dieses erlaubt Servern das Bereitstellen von CORBA-Objekten, ohne bereits zur Laufzeit deren Schnittstellen kennen zu müssen.

CORBA-Server melden sich über den sogenannten Basic Object Adapter beim CORBA-System an. Daraufhin erfolgt ein Eintrag in das Implementation Repository, der unter anderem die physikalische Position des Servers enthält, wie z.B. den Verzeichnispfad des ausführbaren Serverprogramms. Der Basic Object Adapter fungiert dabei als zentrale Schnittstelle zwischen Server und Broker. Er ist beispielsweise für die Aktivierung noch nicht gestarteter Server und die Generierung von Objektreferenzen zuständig [Sta09].

2.1.5 Service Registry und Repository

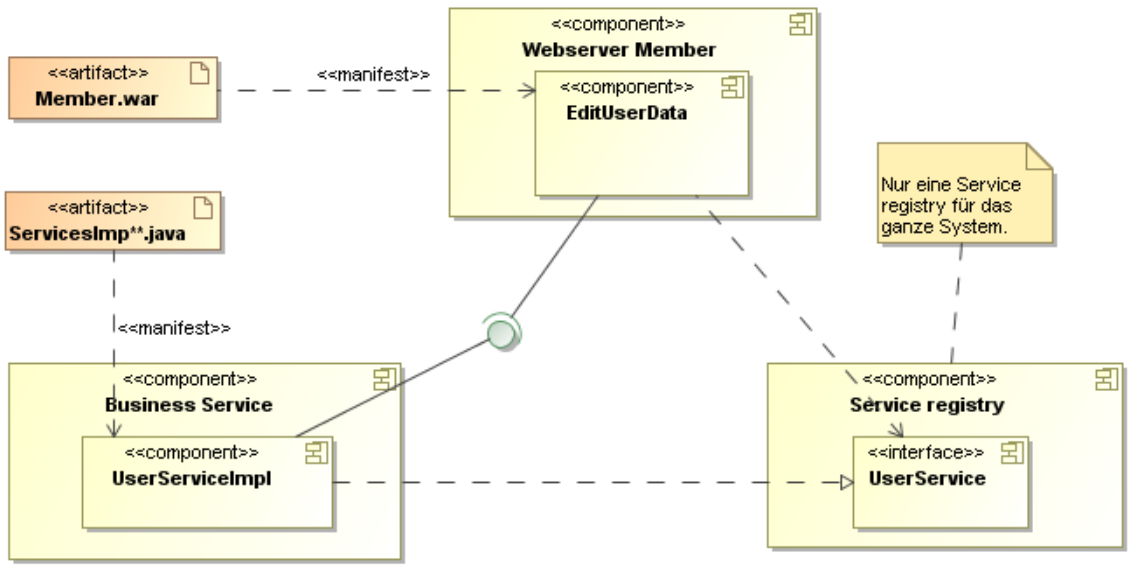

Abbildung 2.3: UML Komponentendiagrammm der bestehenden Service Registry

Um die Kommunikation zwischen Services und Webservern sowie zwischen Services untereinander zu gewährleisten wird eine Service Registry und Repository benötigt.

Die Service Registry dient der Speicherung von Daten, die für die Verwaltung der Services zur Systemlaufzeit erforderlich sind: so liegen hier alle Angaben zur technischen Ansteuerung

der Services, inklusive der hierfür notwendigen Metadaten (Policies).

Im Service Repository finden sich hingegen diejenigen Daten wieder, die von ‘allgemeinerem‘ Interesse sind, wo sowohl zur Entwicklungs- als auch zur Laufzeit relevante Daten (wie Codeversionen) oder Informationen über ausgetauschten Nachrichten abgelegt sein können. Weil die Grenzen zwischen Service-Registry und Repository fließend sind, werden diese zunehmend als Einheit gesehen [Mat08, Seite 29].

Abbildung 2.3 zeigt wie das System, das in der vorherigen Abbildung 2.1 beschrieben wurde, die Service Registry 2.1.5 zur Zeit umsetzt. Um die Darstellung zu Vereinfachen wurde als Beispiel der Geschäftsprozess der Bearbeitung von Benutzerdaten (EditUserData) gewählt. Wenn sich ein Benutzer im System einloggt und dann seine Daten, z.B Aussehen, Motto, Hobbys, ändern will, müssen die Daten zuerst aus der Datenbank abgefragt werden. Die Abfrage der Datenbank nach Informationen wird durch den Service UserServiceImpl, der auf einen der Business Servern läuft, durchgeführt.

Die Services werden in einer CORBA Registry (siehe 2.1.4) mit eigenen Erweiterungen (Vincent) registriert und können dort auch von Webservern gesucht und gefunden werden. Der Ablauf für die Kommunikation zwischen Webservern und Services findet wie folgt statt:

1. Beim Start des ganzen Systems registriert UserServiceImpl sein Interface (UserService) anhand des IOR (Interoperable Object Reference) in der CORBA Registry. Die IOR beschreibt eine Objektreferenz auf ein CORBA-Objekt [Hen99].

2. Der Benutzer ruft die EditUserData 2.1 auf der Plattform auf.

3. Der Webserver sucht den UserService in der Service Registry anhand des Namens "UserService“. Wenn der Service gefunden wird, bekommt der Webserver die IOR des UserServices.

4. Anhand des IORs kann der Webserver den UserService ansprechen und den Benutzer dann anschließend auf der Webseite anzeigen.

2.1.6 Lastverteilung

Das Portal bietet den Benutzern viele Funktionalitäten an, angefangen von den verschiedenen Suchmöglichkeiten nach Partnern, der Anzeige von Benutzern, die online sind, Informationnen über Messaging (Austausch von Nachrichten) und Chat zwischen Benutzern bis hin zum Hochladen von Fotos, Beschreibung der eigenen Person und vieles mehr.

In Höchstzeiten steigt die Anzahl der parallel eingeloggten Benutzer (User) bis in den mittleren 5-stelligen Bereich. Um den Benutzern auch zu diesen Zeiten ein schnelles, stabiles und zuverlässiges System zu gewährleisten werden verschiedene Lastverteilungs-Strategien gebraucht. Alle verschiedene Strategien in dieser Arbeit zu beschreiben würde den Rahmen dieser Arbeit sprengen, deshalb wird die Lastverteilungsstrategie am Beispiel eines spezifischen Services, dem UserService beschrieben (siehe Abbildung 2.4).

Vorraussetzung für eine Lastverteilung wie in der Abbildung 2.4 ist, dass der Pool (Cache) von UserService Instanzen mit User Objekten gefüllt ist. Wenn der Webserver die Daten eines Benutzers (Users) vom UserService anfordert, wird anhand einer Service Selection Strategy (SSS) entschieden auf welchen UserService Proxy die Anfrage gestellt wird. Die SSS leitet alle User mit ungerade UserId (eine eindeutige System ID) auf den UserServiceProxyA, ungerade UserIds auf den UserServiceProxyB. Wenn der Proxy die angefragten Userdaten

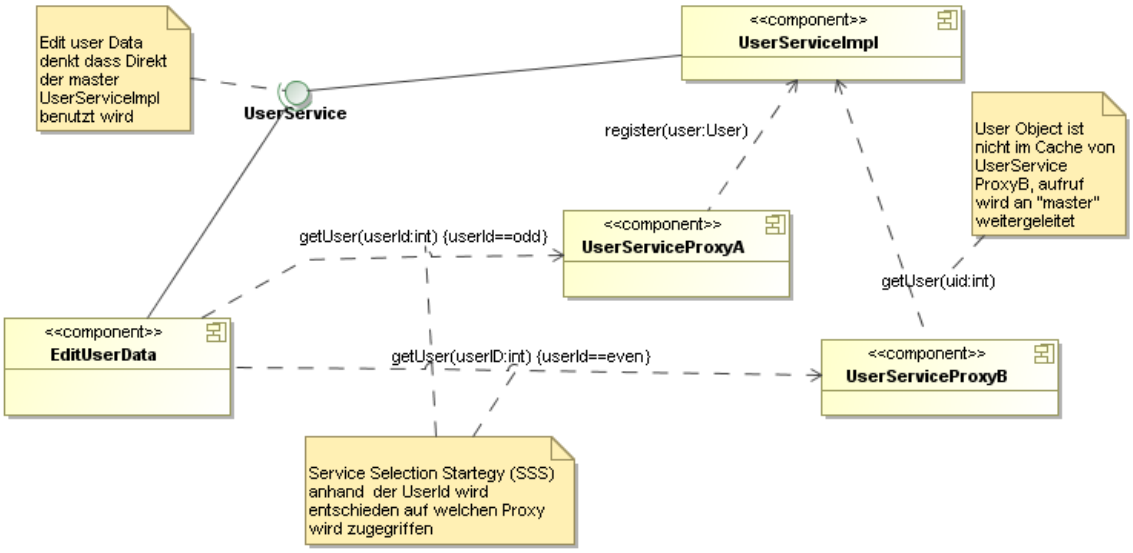

Abbildung 2.4: Caching des UserServices im System

im Cache hat, werden sie an den Webserver zurückgeschickt. Falls nicht wird die Anfrage an den "Master" UserService geschickt, welcher Zugriff auf die Daten aus der Datenbank hat. Die Antwort läuft über den Proxy wieder zurück, der dann die Daten in seinem Cache speichert und an den Webserver weiterleitet.

Somit verläuft das Caching für den Webserver völlig transparent, da die Clients (Webserver und Admintool) auf die Daten zugreifen und merken nicht ob die Objekte direkt vom "Master" UserService kommen oder aus dem Cache. Der "Master" UserService hat als einziger den Zugriff auf die Datenbank wo die Userdaten gespeichert sind. Dies ist nur eine der angewandten Lastverteilungsstrategien. Weitere Strategien sind implementiert für das Verschicken und Empfangen von Nachrichten, Chatten, usw..

2.2 Analyse des Systems

Aufbauend auf der Beschreibung des Systems in Kapitel 2.1 wird im diesen Abschnitt nun das System analysiert, sowie Vorteile und Nachteile des Systems werden genannt.

Die Vorteile des bestehenden Systems liegen in der drei Schichten Architektur, welche das System leicht erweiterbar macht. Die einzelnen Schichten sind von einander lose gekoppelt, was die Unabhängigkeit der Schichten gewährleistet. Somit kann man leicht z.B. die Datenbank gegen Datenbanken anderer Hersteller austauschen. Das Verhalten des Systems bei hoher Last, welche zu diesen Zeitpunkt höchstens erreicht wird, ist zufriedenstellenend und es existieren hier keine hier nennenswerten Nachteile.

Verbesserungsmöglichkeiten bestehen hingegen beim Release Prozess und Deployment Prozess:

- Eine Änderung (Release) von neuen Funktionen und Verbesserungen der bestehenden Funktionen bringt einen Neustart des ganzen Systems mit sich. Da Kunden und Partnern 24h am Tag Verfügbarkeit zu gewährleisten ist, ist es wichtig, dass das System

ohne Unterbrechung läuft. Um den Kunden eine Verfügbarkeit von 99,99 % zu bieten ist es wichtig, dass ein Release ohne Unterbrechung durchgeführt wird.

- Services im System sind derzeit nicht versionierbar, dass heißt es ist immer nur eine Version jedes Services deployed. Damit ist verbunden, dass Services immer im Rahmen eines Gesamtrelease getestet und in Betrieb genommen werden müssen. Ausserdem ist derzeit eine Historie der Service-Releases über ein sauberes Konfigurationsmanagement nur eingeschränkt nachzuvollziehen.

Das System ist an Standards angelehnt. benutzt jedoch ein eigenentwickeltes Framework, welches das DII (Dynamic Invocation Interface) nicht implementiert. Somit ist die Installation neuer Service-Versionen in vielen Fällen mit einem Neustart des ganzen Systems verbunden. Da Stubs und Skeletons der Services daher neu generiert werden müssen, ist es notwendig, dass alle Clients auf die neue Service-Version umsteigen und die Webserver neu gestartet werden, damit die neuen Services genutzt werden können.

Wie in Kapitel 2.1 basiert das bestehende System auf einer Service Oriented Architekture (SOA). Dieser Standard wird im Abschnitt 3.1.1 näher betrachtet.

3 Anforderungen

In diesem Kapitel werden die Anforderungen an das System anhand der Analyse aus Kapitel 2.2 detailliert erklärt. Dazu werden die grundlegenden Begriffe und der heutige Stand des offenen Standards, der dynamische und versionierbare Services ermöglicht, näher erläutert. Anhand der Erkenntnisse wird ein Kriterienkatalog zusammengestellt, der die Anforderungen an das umzusetzende Framework beinhaltet. Dann werden drei entsprechende Frameworks ausgewählt, analysiert und anhand des aufgestellten Kriterienkatalogs verglichen. Nach Analyse und Abgleich mit dem Kriterienkatalog wird eine Empfehlung ausgesprochen mit welchem Framework das neue System umgesetzt werden kann.

3.1 Grundlegende Begriffe und Anforderungen

Im folgenden Kapitel werden die Anforderungen die aus der Analyse aus Kapitel 2.2 hervorkommen, beschrieben. Die Service-Architektur basiert auf einer SOA, die im folgenden näher erklärt wird.

3.1.1 SOA - Service-orientierte Architektur

In einer service-orientierten Architektur handelt es sich um ein generisches Entwurfsmuster für IT-Systemarchitekturen oder -um einen neudeutschen Begriff zu benutzen - um ein Systemarchitektur-Pattern. Die SOA beschreibt eine Methode so wie ein lose gekoppeltes, verteiltes Software-System zu entwerfen und zu realisieren, sowie seine Einzelbausteine (vor allem die Services) untereinander und auf verschiedenen Abstraktions- und Skalierungsebenen in eine sinnvolle Wechselwirkung zu setzen [Mat08, Seite 8].

Die in Kapitel 2.1 beschriebenes Architektur basiert auf einer SOA, die die Services dem Webfrontend und anderen Clients zur Verfügung stellt. Die SOA Architektur sollte soweit wie möglich beibehalten werden. Es sollte durch den Einsatz eines Standard Frameworks, die Service-Verwaltung, die Weiter- und Neuentwicklung von Services und durch die Service-Administration vereinfacht werden.

Im Folgenden wird genauer erklärt was ein Framework ist und was von ihm erwartet wird.

3.1.2 Framework

Der Begriff Framework kann nach [Bos04] folgendermaßen definiert werden: Ein Framework ist kein fertiges Softwaresystem, sondern stellt einem Entwickler ein Rahmenwerk zur Verfügung, mit dessen Hilfe er schnell und zielgerecht eine Softwareaufgabe lösen kann. Oftmals liefert ein Framework bereits Module und Bausteine mit, die ein Entwickler optional verwenden oder erweitern kann.

Nicht jedes Framework ist für ein Projekt geeignet, deshalb sollten bei der Suche nach einem passenden Framework folgende Fragen geklärt werden [Bos04]:

- Welche Art von Anwendung soll entwickelt werden (Webanwendung, Client-/ Server - Anwendung oder Desktop - Anwendung)?
- In welcher Programmier- bzw. Skriptsprache soll die Anwendung entwickelt werden (z.B, Java, C#, Perl, ...) ?
- Wie groß ist das Entwicklerteam des Projekts?
- Welche Funktionalitäten sollten durch ein Framework nach Möglichkeit abgedeckt werden (z.B. Oberflächenkonzept, Datenbankanbindung, Benutzer- und Session-Verwaltung, ...)?
- Welche Unterstützung seitens des Herstellers oder seitens einer Community wird benötigt?
- Wie wichtig ist eine ständige und langfristige Weiterentwicklung des Frameworks (auch hinsichtlich der Investitionssicherheit)?
- Wie verbreitet ist das Wissen über das Framework? Kann davon ausgegangen werden, dass sich Entwickler mit diesen Framework schon befasst haben?
- Soll das Framework auf einem Standard basieren? Ist das Framework an einen offenen Standard gebunden und sind somit viele alternative Implementierungen verfügbar?
- Sind Schulungsmöglichkeiten für Entwickler vorhanden? Ist dieses Framework weit verbreitet und somit viele Supportmöglichkeiten verfügbar?

Im Kontext dieser Arbeit liegt der Fokus darauf ein Framework zu finden, das für die Entwicklung im Bereich Client-Server Anwendung geeignet ist. Clients können dabei Webserver, externe Partner und mobile Anwendungen sein. Das Framework soll JAVA kompatibel und für ein kleines Team von 15 Entwicklern geeignet sein.

Die bestehenden Services müssen weiterhin im vollem Umfang den Clients zur Verfügung stehen. Außerdem sollen die Services versionierbar sein (siehe Abschnitt 3.1.4). Neue Releases der Plattform sollen keinen Neustart des ganzen System mit sich bringen. Das Framework sollte weit verbreitet sein, damit schneller Support im Netz gefunden werden oder vom Betreiber des Frameworks kommen kann.

Eine Absicherung, dass das Framework auch langfristig weiterentwickelt wird, muss gewährleistet sein, da immer neue Anforderungen an das System gestellt werden. Die Komponenten der Frameworks sollten so entwickelt werden, dass man ohne großen Aufwand auf andere Frameworks migrieren kann. Somit hätte man keine Abhängigkeit zur einem bestimmten Framework. Eine Zusammenfassung aller Anforderungen wird im Abschnitt 3.2 näher betrachtet.

3.1.3 SoaML - Service oriented architecture Modeling Language

Bei der Einführung einer service-orientierten Architektur (SOA) stellt sich die Frage, wie Service, Service-Provider und Service-Nutzer modelliert werden sollen, da in der UML keine speziellen Elemente für die Modellierung der Elemente einer SOA zu finden sind. Mit der SoaML als Erweiterung der UML gibt die OMG (Object Management Group) eine Antwort auf diese Frage [Dee09]. Tabelle 3.1 listet die SoaML Stereotypen, die in den nächsten Abschnitten in Form von UML Diagrammen dieses Dokuments benutzt werden.

SoaML Stereotypen	UML Metaklasse	Beschreibung
<<ServiceInterface>>	Class oder Interface	Definiert die Schnittstelle zu einem ServicePoint oder einem RequestPoint und ist der Typ der Rollen in einem ServiceContract. Legt fest, wie ein Participant interagieren muss, um einen Service anzubieten oder zu nutzen.
<<ServicePoint>>	Port	Angebot eines Services typisiert mit einem ServiceInterface oder Interface.
<<RequestPoint>>	Port	Nutzung eines Services typisiert mit einem ServiceInterface oder Interface.
<<Participant>>	Component	Bietet einen Service über einen ServicePoint an oder nutzt ihn über einen RequestPoint.

Tabelle 3.1: Die wichtigsten SoaML Stereotypen [Dee09]

3.1.4 Versionierung von Services

Große, verteilte Systeme unterliegen ständigen Anpassungsbedarf. Neue Anforderungen werden an das System gestellt oder bereits umgesetzte Anforderungen werden modifiziert. Um die parallele Entwicklung und auch den Betrieb von mehreren Versionen effizient zu gestalten ist insbesondere innerhalb einer service-orientierten Architektur eine handfeste Strategie zur Versionierung von Services unabdingbar.[HS09]

Granularität der Versionierung Die drei am weitesten verbreiteten Varianten von Service-Versionierung:

1. Jede Änderung entspricht einem neuen Service - Dies Entspricht der einfachsten Versionierung. Jede Änderung an einem Service wird als neuer Service behandelt. Somit entstehen sehr viele Service-Versionen, die den Clients die gleiche Funktionalität bieten. Diese Art von Versionierung ist von der Seite des Service-Anbieters sehr einfach zu handhaben, aber bringt schlechte Wiederverwendbarkeit vom Code mit. Der Client kann nicht zwischen kompatiblen und nicht kompatiblen Änderungen der Services unterscheiden. Dadurch sind häufige Änderungen bei Clients nötig.

2. Die Versionierung erfolgt auf der Ebene von Operationen - Jede einzelne Operation eines Services wird versioniert. Dem Service können leicht zusätzliche Operationen und neue Versionen vorhandener Operationen hinzugefügt werden. Veraltete Operationen werden über die Zeit hin entfernt. Der Service selbst bleibt in einer Version bestehen. Bei dieser Art der Versionierung sind kompatible Änderung ohne Anpassung der Clients möglich aber die Adressierung ist sehr komplex, da jede Operation mit der Version adressiert werden muss. Die Gefahr durch die ansteigende Anzahl der Methoden darf man auch nicht vernachlässigen.

3. Die Versionierung erfolgt auf der Ebene von Services - Es besteht die Möglichkeit, einen Mittelweg zwischen den beiden oben genannten Varianten zu gehen und einen

Service als ganzes zu versionieren. Beim Hinzufügen, Entfernen, oder Ändern von mehreren oder einer Operation entsteht eine neue Service-Version. Somit können mehrere Änderungen eines Services zu einer neuen Version des Services führen. Dies betrifft insbesondere Service Interface Änderungen oder wichtige Funktionalität Änderungen. Ein Service bildet bei dieser Variante eine geschlossene Deployment-Einheit. Bei dieser Variante muss man zwischen kompatiblen und inkompatiblen Versionen unterscheiden. Hierbei entstehen weniger Versionen als bei der ersten Versionierung.

Deployment und Betrieb Beim Deployment und Betrieb wird die Versionierungsstrategie festgelegt wie die Zuordnung zwischen Endpoints und verschiedenen Service-Versionen durchgeführt werden soll. Dabei gibt es drei verschiedene Varianten:

- Ein gemeinsamer Service Endpoint für alle Service-Versionen - Dieses Vorgehen ist für den Client sehr einfach, problematisch ist das Routing im Service Endpoint, da jeder Request erst geparsed werden muss, um zu ermitteln zu welcher Service-Version er durchgereicht werden muss.

- Ein Serviceendpoint zu jeder Service-Major-Version - Neue Service-Major-Version bedeutet, dass der Service inkompatibel zur vorherigen Version ist. Minor Service Änderungen sind dagegen kompatibel. Wenn eine Minor Änderung publiziert wird, passt man nur den ServiceEndpoint and die neue Version an.

- Ein Endpoint pro Service-Version - Jede Version des Services hat einen Endpoint. Der Vorteil ist, dass kein Routing und keine Beeinflussung durch neue Service-Versionen auftritt. Als Folge muss das Framework sehr viele Endpoints verwalten es muss eine konfigurierbare Bindung eines Services an mehrere Endpoints unterstützen.

Abbildung 3.1 zeigt anhand eines SoaML 3.1.3 Diagramms wie das Framework Services in verschiedenen Versionen bereitstellen sollte. Als Beispiel wurde der UserService genommen. UserService ist einer der wichtigsten Services der FRS24 Plattform. Dieser Service wird von sehr vielen Komponenten des FRS24 Systems benutzt, z.B. die Anzeige der User, Bearbeiten der Userdaten, Suchindex, usw. Dabei sollten verschiedene Clients (Webserver Member, Mobile-Frontend-Anwendung) auf verschiedene Service-Versionen zugreifen können. Dabei ist die Versionierung zweistufig, die Interfaces (UserServiceV1, UserServiceV2) können verschieden Versionen haben. Aber auch die Service Implementierungen können verschiedene Versionen haben, aber das Interface in der gleichen Version implementieren.

Es muss beachtet werden, dass die Versionierung im Rahmen des Konfigurationsmanagements auch entsprechend für die Anforderungsspezifikation und Dokumentation der Entwicklung der Services entsprechend verwaltet werden muss. In dieser Arbeit wird nur die Machbarkeit der Service-Versionierung betrachtet. Wenn das Konzept der Versionierung umgesetzt wird, muss die Versionierung der Dokumentation, Anforderungsspezifikation, Konfigurationen und der Test Cases beachtet werden.

3.2 Kriterienkatalog

Um die Frameworks leichter vergleichen zu können wird in diesen Kapitel der Kriterienkatalog definiert. Mit Hilfe dessen werden alle Frameworks nach den einzelnen Kriterien untersucht und bewertet. Mit den definierten Kriterien werden die zwei wichtigsten Anforderungen

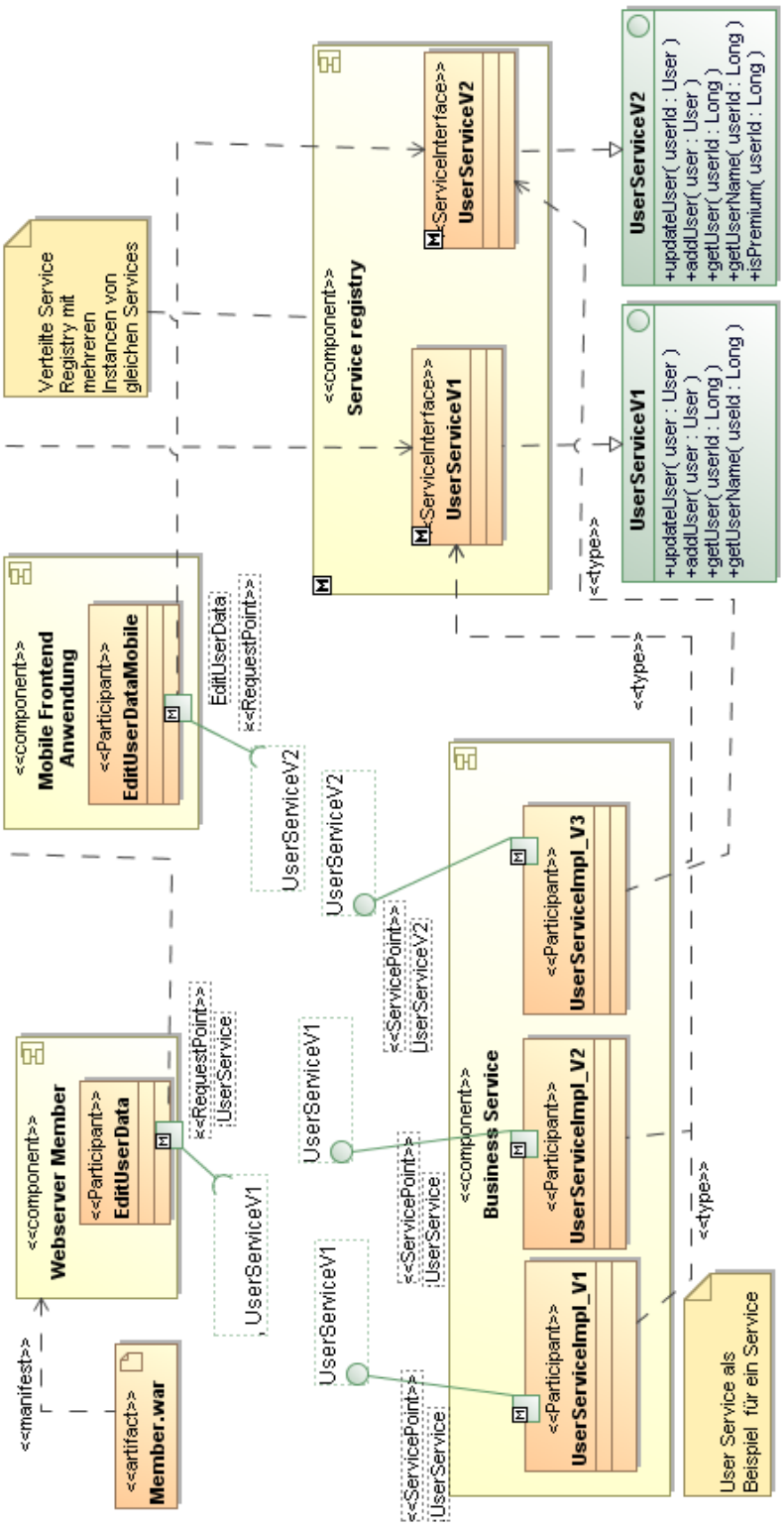

Abbildung 3.1: Mehrere Service-Versionen in der Verteilungssicht

an das Framework wiedergegeben (siehe Analyse Abschnitt 2.2): die hohe Verfügbarkeit und die Möglichkeit die Releases ohne Downtime realisieren zu können. Die Bewertung der Kriterien wird am Ende des Abschnitts vorgestellt. Die Kriterien, die die Anforderungen detailliert beschreiben, stellen sich aus folgenden Punkten zusammen:

- Verteilung
- Dynamische Versionierung
- Monitoring
- Dynamische Rekonfiguration
- Standardisierung
- Business - Relevanz

3.2.1 Verteilung (VT)

Wie in Kapitel 2.1 beschrieben laufen die Services verteilt auf verschiedenen Servern. Um die Verteilung der Services zu ermöglichen sind die Kriterien Verteilte Registry, Zugriff über ORB, Zugriff über Webservices und Asynchrone Kommunikation wichtig.

Verteilte Registry (VT1)

Services laufen in verschiedenen Java Virtual Mashines (JVMs), die auf viele verschiedenen Servern verteilt sind. Das Framework soll alle Services an einer Stelle zur Verfügung stellen, ohne dass die Clients genau wissen müssen wo die eigentliche Service Implementierung läuft. Die Umsetzung könnte nach einer der folgenden zwei Möglichkeiten realisiert werden:

- Das Framework könnte in jeder JVM laufen. Mehrere Instanzen wären auf einem Server möglich.
- Jeweils eine Instanz des Frameworks könnte auf jedem Server laufen.

Jede Instanz des Frameworks sollte dann an einer zentralen Stelle (einer “verteilten Registry“) den Clients zur Verfügung gestellt werden. Durch diese verteilte Registry können Clients (Webserver, Mobile-Clients) entsprechende Services suchen und dann eine Verbindung zu dem Server aufbauen, auf dem der Service läuft.

Zugriff über ORB (VT2)

Das Framework muss eine Methode zum Zugriff auf verteilte Services unterstützen, zum Beispiel eine der ORB Implementierungen wie RMI, JacORB, CORBA. Der Zugriff über ORB wird dazu genutzt um die interne Kommunikation zwischen den Services über verteilte Methodenaufrufe Objekte, mit bestimmten Daten, auszutauschen. Dabei können die Objekte verschiedene Userdaten beinhalten, z.B. wie Onlinestatus, Alter, Geschlecht usw. In der Literatur wird bei der losen Koppelung die Service-Kommunikation mit einfachen Datentypen wie Zeichen, Zahlen und boolean Werten empfohlen. Bei FRS24 ist dies nicht der Fall. Da die Services jedoch alle innerhalb einer Serverfarm laufen, ist die Abweichung nicht gravierend.

Zugriff über Webservice (VT3)

Das Framework soll eine Webservice Schnittstelle für die Services unterstützen, damit externe Partner auf diese Zugreifen können. Somit können auch Mobile-Anwendungen (Android, iPhone) die gleichen Services benutzen wie die Webanwendung. Für den Zugriff von Mobilen-Anwendungen sollte ein Konzept für die Komprimierung der zu übertragenden Daten vorgeschrieben werden, da Mobile-Endgeräte über Netze mit beschränkter Bandbreite verfügen. Außerdem soll das Framework ein mögliches Sicherheitskonzept für den Zugriff über Webservice bieten, wie z.B. eine XML-Signatur.

Asynchrone Kommunikation (VT4)

Es soll die Möglichkeit einer asynchronen Kommunikation über Message Queuing (Warteschlangen) Technologie z.B. eine Implementierung der JMS (Java Message Service [jms09]) vorhanden sein. Wie bei JMS sollen zwei verschiedene Nachrichtenmodelle zu Verfügung stehen [Ben04]:

1. **Point-to-Point** Modell (PTP, 1:1 - Kommunikation): Der Erzeuger erzeugt eine Nachricht und verschickt diese über einen virtuellen Kanal, der Queue genannt wird. Eine Warteschlange (Queue) kann viele Empfänger haben, aber nur ein Empfänger kann die Nachricht konsumieren.

2. **Publish/Subscribe** Modell (Publish/Subscribe, 1:m Kommunikation) Ein Erzeuger produziert die Nachricht und verschickt diese über einen virtuellen Kanal, der Topic genannt wird (publish). Ein oder mehrere Empfänger können sich zu einen Topic verbinden (subscribe) und die Nachricht erhalten. Jeder der zu dem Topic registrierten Empfänger erhält dann eine Kopie der Nachricht.

Die Queuing Technologie bietet auch die Möglichkeit mehrere Nachrichten zu bündeln und dann zusammen über das Netz zu schicken. Damit wird das Netz weniger beansprucht und ein größerer Durchsatz kann erreicht werden.

3.2.2 Dynamische Versionierung einzelner Services (DV)

Services sollen gleichzeitig in mehreren Versionen laufen können, damit bei z.B. Releases alte Services bis zum kompletten Update weiterlaufen können, während neue Services hinzukommen. Auf diese Weise entsteht keine Downtime der Plattform. Außerdem können Services mit alten Interfaces für externe Partner einfach weiterlaufen. Der Entwicklungsprozess der Services wird zusätzlich erleichtert wenn eine Versionierung der einzelnen Services möglich ist. Und auch das Testen und das Deployment der Services wird weniger Aufwand mit sich bringen. Die Clients können dadurch nacheinander auf die Benutzung der neuen Versionen der Services konfiguriert werden. Die Versionierung könnte wie in der Abbildung 3.1 aussehen.

3.2.3 Monitoring - Passive Beobachtung von Phänomenen (MT)

Die Services sollen periodisch Statusangaben an eine zentrale Stelle kommunizieren, die diese den Systemadministratoren sichtbar macht. Hier werden die zwei Kriterien Ausfallbehandlung und Lastverteilung, die Ziel des Monitorings sind, betrachtet.

Ausfallbehandlung (MT1)

Wenn ein Service oder eine Komponente ausfällt, sollte eine Benachrichtigung automatisch an die Verantwortlichen geschickt werden. Falls ein ganzer Server ausfällt, soll das Framework alle Anfragen an gleiche Services auf andere Server verteilen.

Last (MT2)

Das Framework soll bei hoher Last die Anfragen auf alle Services verteilen. Bei kritischer Last sollen Warnmeldung verschickt werden. Administratoren sollen eine Vorschau der Last angezeigt bekommen. Maßnahmen zur Behebung der Last wie das Starten neuer Service-Instanzen (Skalierung) müssen den Administartoren zur Verfügung stehen.

3.2.4 Dynamische Rekonfiguration (DR)

Die Plattform hat einen hohen Bedarf an Anpassungen und Bereitstellungen von neuen Funktionen in kurzer Time to Market Iterationen. Beide Punkte werden in den zwei Kriterien Steuerung über zentrale Administrationskonsole und Deployment der Services zur Laufzeit des ganzen Systems betrachtet. Zum Beispiel sollte es zur Laufzeit möglich sein, Änderungen an Bezahlmethoden vorzunehmen oder sie ein- oder abschalten.

Steuerung über zentrale Administration Console - GUI (DR1)

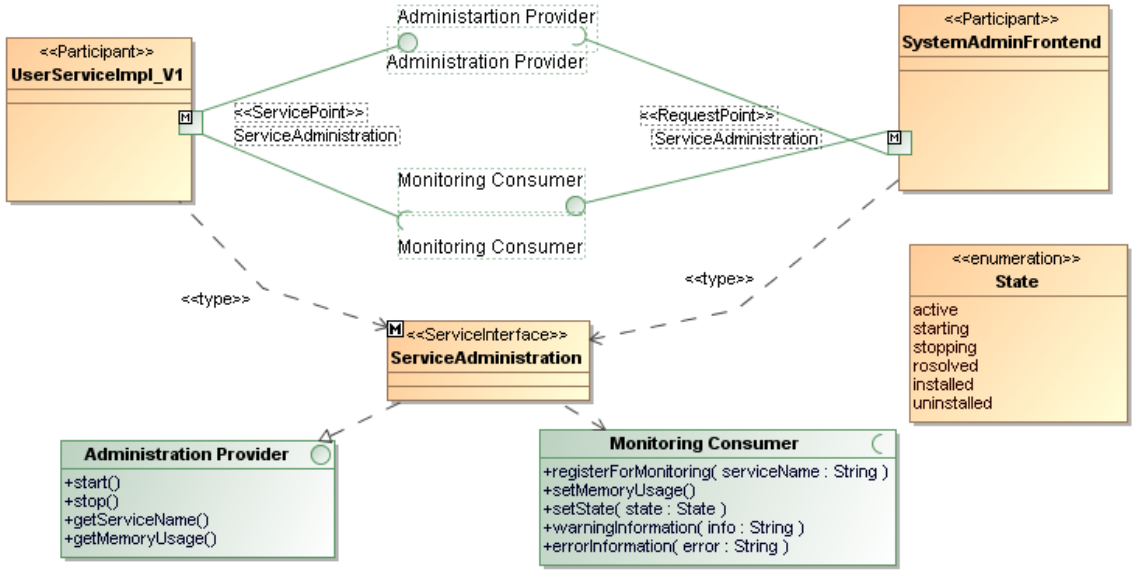

Abbildung 3.2: Monitoring und Administration der Services durch eine zentrale Oberfläche

Das Framework soll eine graphische Oberfläche und eine Console für die Administration der Services auf allen Servern bereitstellen. Auf der Console soll die Arbeit auf Servern ermöglicht werden und durch verschiedene Scripte automatisierte Befehle ausführen. Die GUI soll alle Services graphisch darstellen und Informationen über Service-Status, Ort, Aus-

lastung sowie verfügbare Änderungen am Service machen. Somit könnten Administratoren ganz einfach:

- Services starten und stoppen
- Status und Auslastung von Services anzeigen
- Services zur Laufzeit konfigurieren

Abbildung 3.2 zeigt anhand eines SoaML 3.1.3 Diagramms wie die Service-Steuerung und Monitoring realisiert werden kann. Als Beispiel wurde der UserService gewählt. Die Abbildung zeigt die Klasse UserServiceImpl, die den ServicePoint ServiceAdministration bereitstellt. Der UserServiceImpl muss das Interface Administration Provider verfügbar machen, welcher wiederum von der zentralen Administrationsoberfläche vorausgesetzt wird um Services zu Administrieren. Damit der UserService wiederum Status Informationen (Monitoring) an die Administrationsoberfläche verschicken kann, muss diese das Interface MonitoringConsumer realisieren.

Deployment der Services zur Laufzeit des ganzen Systems (DR2)

Um den Kunden 99,99 % Verfügbarkeit zu gewährleistet, sollen die Kunden beim Deployment von neuen Service-Versionen keine Beeinträchtigung des Systems wahrnehmen. Während neue Versionen installiert werden, können alte Service-Versionen normal weiter verfügbar sein. Erst wenn alle Services aktualisiert sind, können alte Service-Versionen abgeschaltet werden. Somit wären die Services immer verfügbar und neue Funktionen könnten schnell released werden.

Authentifizierung für Administratoren (DR3)

Ein Sicherheitskonzept für den Zugriff auf die Console und die Administrationsoberfläche sollte existieren. Das Sicherheitskonzept soll verschiedene Rollen mit verschiedenen Rechten bieten. Somit gäbe es die Möglichkeit, dass die Entwickler den Status des Systems nur betrachten können, aber die Administratoren das Recht zur Veränderung haben.

3.2.5 Standardisierung (ST)

Die Frameworks und ihre Features sollten sich an offene Standards halten, um Unabhängigkeit vom Framework zu schaffen. Vom Standard könnten die Schnittstellen vorgegeben sein die jeder Service realisieren muss. Durch die Realisierung der Schnittstellen werden die Service im Framework eibgebunden.

Die Anforderungen an das Portal ändern sich ständig und somit könnte ein anderes Framework die Anforderung besser erfüllen. Das Wechseln zwischen Frameworks, die sich an den gleichen offenen Standard halten, würde dann einen angemessenen Aufwand mit sich bringen.

Verbreitung (ST1)

Das Framework soll weit verbreitet und länger am Markt vorhanden sein. Es soll eine stabile, keine "Beta“ Version genommen werden, somit könnte eine Sicherheit gegeben werden, dass das Framework weiterentwickelt wird und somit neue Verbesserungen mit sich bringt. Falls

das Framework nicht stabil genug ist, ist die hochverfügbarkeit der FRS24 Plattform gefärdet, was auf keinen Fall in kauf genommen werden kann.

Standardisierte Komponenten /Vermeidung von Vendor Lock in (ST2)

Das Framework sollte sich an einen offenen Standard anlehnen, damit die Komponenten leicht in andere Frameworks integriert werden können, die den gleichen Standard entsprechen. Idealerweise mit angemessenen Programmieraufwand und wenig Konfigurationsaufwand. Somit ist man nicht an eine Implementierung des Framework gebunden und kann bei Bedarf schnell eine Framework-Migration vollziehen. Die Services sollen nur die von den Standard vorgeschriebenen Schnittstellen realisieren. Es sollen möglichts wenige eigene Schnittstellen der Framework von den Services realisiert werden, um eine Lose Koppelung von zwischen Framework und Komponenten die die FRS24 Services repräsentieren.

Standardisierter Zugriff (ST3)

Die Kommunikation zwischen einzelnen Komponenten (Services) soll standardisiert sein und die Kriterien in Abschnitt 3.2.1 erfüllen. Wenn z.B. Verbesserungen an dem JMS Standard herauskommen, könnten die Services ohne Implementierungsaufwand auf die neue Version des JMS Standards umsteigen. Updates der anderer Software (z.B. Java, Webservice, ORB) sollten beim Umstieg keine größeren Probleme mit sich bringen. Ziel dieses Kriteriums ist, dass das Fremework etablierte Kommunikationsprotokolle unterstützen, um eine sichere und performante Kommunikation zwischen den Services zu erreichen.

3.2.6 Business - Relevanz (BR)

Das Framework soll auch in wirtschaftlicher Hinsicht umsetzbar sein. Das Framework könnte fachlichen Anforderungen erfüllen, aber wenn die Nutzung bezogen auf die Entwickleranzahl nicht groß genug ist oder Schulungsaufwände und Migrationsaufand zu hoch sind, passt das Framework nicht zum System. Deshalb sind folgende Kriterien von Bedeutung:

Nutzung - Entwickleranzahl (BR1)

Die Weiterentwicklung des Systems mit dem neuen Framework ist für eine Anzahl von bis zu 15 Entwicklern leichter und schneller. Die Wartung des Frameworks soll keinen großen Aufwand mit sich bringen.

Schulungsaufwände (BR2)

Der Schulungsaufwand für Entwickler und Administratoren sollte gering sein. Es soll eine gute und ausführliche Dokumentation über das Framework vorhanden sein. Trainings und Coaching Möglichkeiten sollten gegeben sein. Somit könnten die Entwickler und Administratoren schnell angelernt werden.

Migrationsaufwand (BR3)

Der Entwicklungsaufwand für die Migration auf ein bestimmtes Framework soll angemessen sein. Die verschiedenen bestehenden Patterns zur Lastverteilung und zum Caching sollten ohne großen Aufwand unterstützt werden oder sinnvoll adaptiert werden können.

Lizenzkosten (BR4)

Die Lizenzkosten müssen als Kriterium beachtet werden, da die Entscheidung welches Framework eingesetzt wird, die Höhe der Kosten bestimmt. Wenn das Framework alle Kriterien erfüllt aber die Lizenzkosten das Budget von FRS24 übersteigen, kommt das Framework nicht zum Einsatz.

3.2.7 Skalierbarkeit (SK)

Das FRS24 System wächst ständig weiter und den Benutzern immer mehr Funktionen, deshalb sollte das Framework leicht erweiterbar sein bezogen auf die Anzahl der verschiedenen Services. Bei hoher Last soll die Möglichkeit gegeben sein, neue Service-Instanzen problemlos starten zu können, um das ganze System zu entlasten. Genauso sollte es möglich sein, während der Laufzeit des Systems, neue Server und darauf neue Service-Instanzen dem Framework hinzuzufügen.

3.2.8 Performance (PE)

In Abschnitt 2.1.6 wurden die Pattern zur Lastverteilung im System vorgestellt, die weiterhin in den Frameworks realisiert werden sollen. Das Framework soll natürlich Performance Eigenschaften mit sich bringen. Dieser Aspekt wird in dieser Arbeit nicht genauerer betrachtet, da es den Rahmen sprengen würde.

3.3 Relevanz der Eigenschaften

Um die Kriterien leichter zu bewerten und wichtige Kriterien genauerer zu analysieren, wird in diesem Abschnitt die Wichtigkeit der Kriterien aufgeführt. Dann werden die Wichtigkeiten der Kriterien begründet und die Beurteilung genauerer beschrieben.

3.3.1 Wichtigkeit der Anforderungen

Die Anforderungen werden nach der Wichtigkeit bewertet, siehe hierzu die zweite Spalte in der Kriterientabelle 4.1. Dazu wird eine Skala von nicht erforderlich (1) bis unbedingt erforderlich (5) werwendet. Beschreibung der Skala:

- **5** steht für unbedingt erforderlich, Eigenschaften mit der Wichtigkeit 5 müssen erfüllt werden
- **4** steht für erforderlich, Eigenschaften mit der Wichtigkeit 4 sollten erfüllt werden müssen aber nicht unbedingt
- **3** steht für gewünscht, Eigenschaften mit der Wichtigkeit 3 sind gewünscht, aber nicht zwingend erforderlich
- **2** steht für nicht unbedingt erforderlich, Eigenschaften mit der Wichtigkeit 2 stören nicht wenn vorhanden, aber sind nicht richtig relevant
- **1** steht für nicht erforderlich, Eigenschaften mit der Wichtigkeit 1 sind nicht relevant

3.3.2 Begründung der Bewertung und Evaluierungsaufwand

Wie schon im Kapitel 3.1.2 kurz beschrieben sind einige Kriterien für den Vergleich ausschlaggebend und einige sind nicht unbedingt erforderlich.

Kriterien mit der **Wichtigkeit 5** sind ausschlaggebend und falls nicht vorhanden ist das Framework nicht geeignet.

Dazu steht in der dritten Spalte der Tabelle 4.1 der Aufwand für die jeweilige Evaluierung eines einzelnen Kriteriums. Der Aufwand kann leicht, mittel oder schwer sein. Leicht bedeutet, dass die Beurteilung für das Kriterium ganz leicht ist und wenig Zeit beansprucht, z.B. reicht eine Implementierung eines "Hello World" - Programms um es beurteilen zu können. Ein mittlerer Aufwand beansprucht ein Implementierung, die sich zeitlich in Grenzen hält. Schwerer Aufwand bedeutet, es wird sehr viel Zeit und Entwicklung benötigt, um eine Aussage machen zu können. Im Folgenden werden die Kriterienbewertungen angewandt:

- **VT1 Verteilte Registry** - Dieses Kriterium ist ausschlaggebend, da die Services auf verschiedenen Servern laufen und an einer Stelle verfügbar sein sollten. Der Aufwand ist "leicht bis mittel", da man mehrere Services auf verschiedenen Rechnern starten muss und dann schaut wie sich die Services verhalten.

  Wichtigkeit: 5 Testaufwand: mittel

- **VT2 Zugriff über ORB** - Ist erwünscht, aber nicht unbedingt notwendig. Die verteilte Kommunikation zwischen Servern sollte in baldiger Zukunft auf eine performantere Technologie wie JMS umgestellt werden. Falls vorhanden, könnten die Objekte der Benutzer auf verschiedene Server, wie schon vorhanden, verteilt werden und die Umstellung auf JMS könnte im Nachhinein passieren. Dies ist leicht zu Testen, da man nur ein einfaches Java Objekt erstellen muss und zwischen 2 Servern verschicken muss.

  Wichtigkeit: 3 Testaufwand: leicht

- **VT3 Zugriff über Webservice** - Ist wünschenswert, da man derzeit über die Eigenentwicklung einige Services anbietet (siehe Abschnitt 2.1.2). Die Standardisierung wurde den Prozess der Bereitstellung um einiges erleichtern. Eine Alternative wäre, die Services über einen ESB (Enterprise Service Bus [ESB09]) zur Verfügung zu stellen. Die Evaluierung ist "mittel", da abzuwarten ist wie der Service mithilfe des Frameworks als Webservice bereitgestellt werden kann. Dann kann ein entsprechender Client erstellt werden.

  Wichtigkeit: 4 Testaufwand: mittel

- **VT4 Asynchrone Kommunikation (JMS)** - Ist ausschlaggebend, da die Umstellung der Kommunikation zwischen Servern über JMS geplant ist. Durch JMS werden Verbesserungen an der Performance des Systems erleichtert. Der Aufwand für die Evaluierung ist ähnlich wie beim vorherigen Kriterium. Zuerst muss herrausgefunden werden wie das Framework JMS implementiert ist, falls nicht gibt es eine Aalternative?

  Wichtigkeit: 5 Testaufwand: mittel

- **DV Dynamische-Versionierung einzelner Services Wichtigkeit** - Ist eine der Hauptanforderungen und trägt dazu bei, dass ein Release der Plattform ohne Downtime realisiert werden kann. Außerdem erleichtert es die Entwicklung, den Test und den

Betrieb. Der Evaluierungsaufwand ist nicht zu groß, da man die Test-Services nur leicht verändert in der Registry starten muss und dann darauf Zugreifen kann.

Wichtigkeit: 5 Testaufwand: leicht

- **MT1 Monitoring Ausfallbehandlung** - Dieses Kriterium ist erforderlich aber nicht ausschlaggebend, da man über eine Eigenentwicklung Ausfallbehandlung nachziehen könnte. Dies bringt einen enormen Aufwand mit sich weshalb dieses Kriterium von großer bedeutung ist. Die Evaluierung ist anspruchsvoller, da man den Fehlerfall konstruieren muss und dann nachprüfen muss ob alles fehlerfrei lief.

  Wichtigkeit: 4 Testaufwand: hoch

- **MT2 Monitoring Last** - Die Anzeige der Last auf den Services und dessen Grenzen ist erwünscht, da man bei kritischer Last schnell eingreifen könnte um Ausfälle zu vermeiden. Der Aufwand um diese Kriterium beurteilen zu können ist hoch, da erst der Services implementiert und konfiguriert werden müssen. Dann sehr viel Last auf Services erzeugen und die Verteilung evaluieren.

  Wichtigkeit: 3 Testaufwand: hoch

- **DR1 Zentrale GUI** - Eine übersichtliche Darstellung der Services ist erwünscht damit Fehler schnell gefunden werden können, wenn z.B. ein Service fehlt oder sich in einen unerwünschten Zustand befindet. Die Evaluierung ist leicht, da nur zu prüfen ist ob die Services grafisch angezeigt werden und ob sie über die GUI ansteuerbar sind.

  Wichtigkeit: 3 Testaufwand: leicht

- **DR2 Deployment der Services zur Laufzeit Wichtigkeit 5** - Die Aufgabe des Portals ist es, Kunden und Partnern eine 99,9 % Verfügbarkeit zu gewährleisten. Deshalb ist dieses Kriterium eines der ausschlaggebenden. Die Evaluierung ist einfach da man nur in neuen Versionen starten muss und prüfen muss ob sie automatisch ansteuerbar sind.

  Wichtigkeit: 5 Testaufwand: leicht

- **DR3 Authentifizierung für Administratoren** - Ist erwünscht, wobei der Zugang zu den Servern bereits über die Betriebsystemrechte realisiert wurde. Der Evaluierungsaufwand ist mittel, da man die Authentifizierung erst für jedes Framework einschalten und konfigurieren muss was nicht einschätzbar ist.

  Wichtigkeit: 3 Testaufwand: mittel

- **ST1 Verbreitung** - Das Framework muss länger auf dem Markt verfügbar sein und es darf nicht nur eine Beta Version existieren. Beta Versionen sind sehr fehleranfällig und dürfen nicht in Live-Betrieb genommen werden. Das Ziel ist es auf ein Framework umzustellen welches auch eine langwierige Weiterentwicklung gewährleisten kann. Evaluierung hat die Schwierigkeit mittel, da man sich gründlich über das Framework informieren muss.

  Wichtigkeit: 5 Testaufwand: mittel

- **ST2 Standard Komponenten 5** - Ist sehr wichtig, da abhängigkeit zu einem bestimmten Framework unbedingt vermieden werden sollte. Der Anteil an Eigenetwicklung muss gering gehalten werden. Der Aufwand ist mittel, da man den Standard mit der wirklichen Implementierung vergleichen muss.

  Wichtigkeit: 5 Testaufwand: mittel

- **ST3 Standard Zugriff** - Es sind keine proprietären Erweiterungen beim Zugriff auf Komponenten erwünscht, da dabei Wartungs- und Weiterentwicklungsaufwand entsteht. Der Aufwand ist mittel wie beim vorherigen Kriterium.

  Wichtigkeit: 4 Testaufwand: mittel

- **BR1 Nutzung - Entwickler** - Ist verlangt, da den Entwicklern die Weiterentwicklung der Plattform so weit wie möglich erleichtert wird und damit auch schneller etwickelt wird. Der Evaluierungsaufwand ist mittel, da man erst am Ende der Evaluierung seine Meinung äußern kann. Dazu muss das Framework einigermaßen bekannt sein.

  Wichtigkeit: 4 Testaufwand: mittel

- **BR2 Schulungsaufwände** - Schulungsaufwände sollen gering wie möglich gehalten werden und es soll ausreichend Literatur vorhanden sein, damit schneller Umstieg gewährleistet ist. Die Schulungsaufwände für das Framework sollten vorhersehbar sein. Es besteht der gleicher Aufwand wie im vorigen Kriterium.

  Wichtigkeit: 4 Testaufwand: mittel

- **BR3 Migrationsaufwand** - Der Migrationsaufwand ist nicht zu vernachlässigen. Allerding muss nur einmalig realisiert werden. Der Migrationsaufwand kann nur geschätzt werden. Dazu ist ein sehr genaues Wissen über das Framework nötig.

  Wichtigkeit: 3 Testaufwand: mittel

- **BR4 Lizenzkosten** - Die beste Methode zur Behandlung der Lizenzkosten wäre, wenn das Framework als Open Source bestehen würde und man profesionellen Support einkaufen könnte. Falls die Lizenzkosten für eine kommerzielle Lizenz nicht zu hoch sind und man dafür qualitativen Support bekommen kann, ist Open Source allerdings keine zwigende Anforderung. Der Evaluierungsaufwand ist zu vernachlässigen.

  Wichtigkeit: 3 Testaufwand: leicht

- **SK Skalierbarkeit** - Eine Skalierbarkeit sollte vorhanden sein, da immer neue Services dazukommen werden. Der Evaluierungsaufwand ist sehr hoch, da man viele Services hinzufügen muss.

  Wichtigkeit: 5 Testaufwand: hoch

- **PE Performance** - Damit die Plattform performant bleibt, ist Performance sehr wichtig. Der Fokus liegt in dieser Arbeit auf Verteilung und Dynamik der Services. Um Performance genau zu evaluieren muss man das Framework erst mit allen System Komponenten umsetzen und dann in einer Testumgebung simulieren. Dieser Aufwand sprengt den Rahmen dieser Arbeit, weshalb die Performance gesondert betrachtet werden muss.

  Wichtigkeit: 4 bis 5 Testaufwand: hoch

3.3.3 Beurteilung der Eigenschaft

Die Eigenschaften werden anhand einer Skala von 4 oder ++ (vollständig vorhanden) und 0 oder - - (nicht vorhanden) bewertet. Die Beurteilungsbeschreibungen:

- ++ (4) Die Eigenschaft ist vollständig implementiert und es besteht nur Konfigurationsaufwand um diese auch mit den Services umzusetzen.
- + (3) Die Eigenschaft (Logik) ist implementiert, aber die Services müssen bestimmte Schnittstellen noch implementieren und es besteht noch viel Konfigurationsaufwand.
- **0** (2) Die Eigenschaft ist theoretisch vorhanden, aber die Implementierung der Logik muss vollzogen werden.
- **-** (1) Die Eigenschaft ist nicht vorhanden, aber für die kommenden Versionen geplant.
- **- -** (0) Die Eigenschaft ist nicht vorhanden und auch nicht geplant oder nicht möglich.

Die letzte Zeile von Tabelle 4.1 stellt die Summe der Wichtigkeitsbeurteilung dar. Diese wird mit Hilfe folgender Formel berechnet: $\sum w_i * f_i$. Wobei w_i für die Wichtigkeit des Kriteriums steht und f_i für die Beurteilung der jeweiligen Eigenschaft (siehe oben).

Kriterien	Wichtigkeit	Testaufwand	Framework 1
VT1 Verteilte Registry	5	mittel	
VT2 Zugriff über ORB	3	leicht	
VT3 Zugriff über Webservice	4	mittel	
VT4 Asynchrone Komm (JMS)	3	mittel	
DV Dynamische Versionierung	5	leicht	
MT1 Monitoring Ausfall.	4	hoch	
MT2 Monitoring Last	3	hoch	
DR1 Zentrale GUI	3	leicht	
DR2 Deployment zur Laufzeit	5	leicht	
DR3 Authentifizierung der Admins	3	mittel	
ST1 Verbreitung	5	mittel	
ST2 Standard Komponenten	5	mittel	
ST3 Standard Zugriff	4	mittel	
BR1 Nutzung - Entwickler	4	mittel	
BR2 Schulungsaufwände	4	mittel	
BR3 Migrationsaufwand	3	mittel	
BR4 Lizenzkosten	3	leicht	
SK Skalierbarkeit	5	hoch	
PE Performance	4	hoch	
SUMME	w_i		

Tabelle 3.2: Kriterienkatalog Überblick

4 Technologie und Frameworks

In diesem Kapitel wird die zu verwendende Technologie, welche die Frameworks benutzen sollen, vorgestellt. Dann werden entsprechende Frameworks vorgestellt. Abschnitt 3.2 beschreibt die Kriterien, die das Framework implementieren soll. Dabei sind die Kriterien Verteilte Registry, Asynchrone Kommunikation über JMS, dynamische Versionierung, Deployment zur Laufzeit, Verbreitung und Standard Komponenten ausschlaggebend. Um diese Kriterien auch alle möglichst mit einer Technologie zu unterstützen, werden die Frameworks die den OSGi Standard realisieren, verglichen.

4.1 OSGi

OSGi ist eine Hardware-unabhängige, dynamische Software-Plattform, die es erleichtert, verteilte Anwendungen und ihre Dienste zu modularisieren und über den gesamten Lebenszyklus hinweg zu verwalten. Die OSGi-Plattform setzt eine Java Virtual Maschine (JVM) voraus und bietet darauf aufbauend ein Framework an. Die OSGi-Allianz (Open Service Gateway Initiative [osg09a] ist ein Industriekonsortium, bestehend aus einer Vielzahl von Herstellern aus verschiedenen Branchen, die die Plattform ursprünglich für den Einsatz im Embedded-System-Bereich vorgesehen hat [LSL+08, Seite 67]. Heute wird die OSGi Service-Plattform in ganz unterschiedlichen Bereichen verwendet wie Anwendungen für Mobilfunkgeräte, Client-Anwendungen (Eclipse IDE) und Server-Applikationen.

Das OSGi Framework ist die Basiskomponente der OSGi Service Plattform. Es stellt einen Container bereit, in dem Services zur Laufzeit installiert und zur Ausführung gebracht werden können. Die Schnittstellen und das Verhalten des OSGi Frameworks sind innerhalb der OSGi Service Plattform Core Spezifikation festgelegt.

Ein Bundle ist eine fachlich oder technisch zusammenhängende Einheit von Klassen und Ressourcen, die eigenständig im OSGi Framework installiert und deinstalliert werden kann. Bundles können Services verwenden, die Systemweit zur Verfügung gestellt werden. Bundles sind JAR-Archive in den die Manifest Datei die OSGi-Eigenschaften des Bundles beschreibt.

Ein Service ist ein Java Objekt das unter einem Interface Namen bekannt gemacht wird. Die Registrierung eines Services erfolgt über die Service Registry, die zentral und bundle übergreifend bereitsteht, und an der die angemeldeten Dienste von beliebigen Bundles abgefragt werden können. Bundles, die einen bestimmten Dienst benötigen, können den entsprechenden Service dann von der Service Registry abfragen, ohne konkret wissen zu müssen, welche Implementierung dahinter steht oder welches Bundle den Dienst anbietet [WHKL08].

Die Framework Spezifikation ist in mehrere logische Schichten aufgeteilt. Jede Schicht fasst zusammengehörige Aufgaben bzw. Verantwortlichkeiten zusammen [WHKL08] (vgl. Abbildung 4.1):

- **Module Layer:** Die Module-Schicht definiert das Bundle als grundlegende Modularisierungseinheit innerhalb der OSGi Service Plattform.

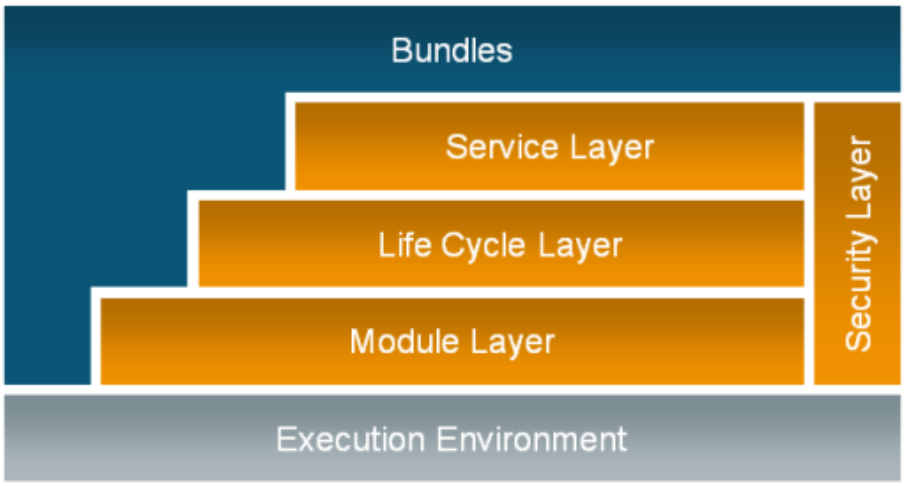

Abbildung 4.1: Schichtenarchitektur des OSGI Frameworks [See08]

- **Life Cycle-Layer:** Die Life Cycle-Schicht spezifiziert, welche Zustände ein Bundle während seines Lebenszyklus innerhalb der OSGi Service Plattform besitzen kann.
- **Service-Layer:** Die Service-Schicht spezifiziert ein allgemeines Service-Modell, in dem Services systemweit über eine Service Registry verfügbar gemacht werden können
- **Security-Layer:** Die Security-Schicht spezifiziert sicherheitsrelevante Aspekte wie z.B. den Umgang mit signierten Bundles und die Möglichkeit zur Einschränkung der Ausführungsrechte einzelner Bundles.

In dieser Arbeit werden verteilte Services betrachtet. Der nächste Abschnitt stellt daher die Distributed OSGi Spezifikation vor.

4.1.1 Distributed OSGi

Das OSGi-Service-Modell beschränkte sich bisher auf Dienste innerhalb eines Containers. Die OSGi Allianz beschloss die Spezifikation so zu erweitern, dass auch eine verteilte Kommunikation über Containergrenzen hinweg möglich ist. Seit September 2009 ist auch die Finale Version der OSGi 4.2 Spezifikation verfügbar [osg09b].

Distributed OSGi ermöglicht es, einen OSGi Service sowohl für lokale als auch für entfernte Clients auffindbar und nutzbar zu machen. Distributed OSGi bietet eine standardisierte Methode eine Middleware wie JMS, CORBA, RMI, Webservice in das OSGi Framework zu integrieren. Service Provider können über ihre Attribute mitteilen, ob sie auch remote aufrufbar sein sollen. Service Consumer bestimmen über Filter, ob sie Services nur lokal oder auch außerhalb ihres Containers aufrufen wollen. Hinter den Kulissen übernehmen zwei neue optionale Framework Erweiterungen, Distribution Provider und Discovery, die Arbeit.

Der Distribution-Provider übernimmt die eigentliche Kommunikation zwischen Service-Provider und -Consumer. Er nutzt ein oder mehrere Kommunikationsprotokolle, um Services auch remote anzubieten. Dazu erzeugt er protokollspezifische Service-Endpunkte. Ein Beispiel ist ein Webservice-Endpunkt, der an einem bestimmten Port lauscht, eingehen-

de SOAP-Nachrichten in Serviceaufrufe überführt und Rückgabewerte wieder als SOAP-Nachricht zurückschickt.

Damit die Consumer die Service-Endpunkte finden können, kann der Distribution Provider diese mit Hilfe der Discovery Dienste im Netzwerk bekannt machen. Dazu übergibt der Distribution Provider dem Discovery Dienst neben den registrierten Serviceeigenschaften zusätzliche Informationen, die notwendig sind, um mit den Service-Endpunkten zu kommunizieren. Darunter fallen URL, benutztes Protokoll, protokollspezifische Schnittstellendefinition. Ein Discovery Dienst kann, transparent für den Distribution-Provider, beliebige Mechanismen zur Publikation des OSGi Dienstes und für dessen Suche verwenden. Beispiele sind UDDI Registry und das Service Location Protokoll (SLP).

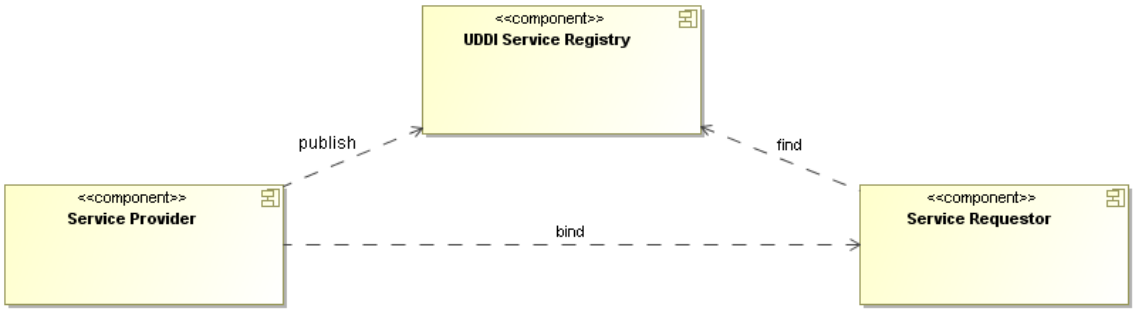

Abbildung 4.2: Komponenten der UDDI Dienstfindung

UDDI steht für Universal Description, Discovery and Integration [udd09]. UDDI stellt eine standardisierte Methode für das Publizieren und das Auffinden von (Web-) Services zur Verfügung. UDDI wurde gebildet durch eine Initiative aus der Industrie mit dem Ziel der Schaffung eines offenen und plattformunabhängigen Frameworks für die Beschreiben, Registrieren und für das Auffinden von (Web-basierten) Diensten. Wie in Abbildung 4.2 werden die Nutzer einer UDDI Registry "Service Provider" und "Service Requestor" genannt. Der Service Provider legt die Beschreibung seiner Dienstangebote (z.B. WSDL) in der Registry ab (publish). Der Service Requestor sucht in der Registry nach passenden Angeboten (find). So wird die Integration von Diensten (bind) vereinfacht.

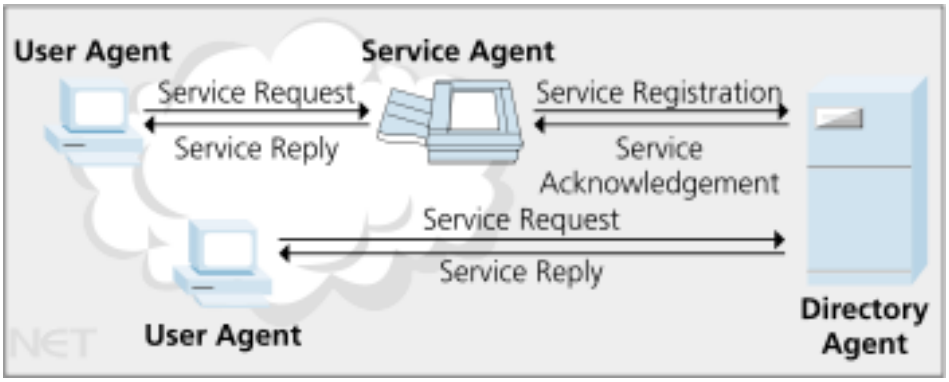

Abbildung 4.3: Schematischer Ablauf des SLP [Det04]

Das **Service Location Protocol (SLP)** ist im RFC 2608 [Gro99] definiert. Die SLP-

Architektur besteht aus drei Agenten die miteinander kommunizieren (siehe Abbildung 4.3):

- Der User Agent (UA) übernimmt auf der Clientseite das Service Discovery und ist für Anfrage und Auswertung der Antworten zuständig.
- Der Service Agent (SA) vertritt den Dienst und macht "Werbung". Er kann direkt auf Anfragen eines UA antworten und ihm die nötigen Informationen zukommen lassen.
- Der Directory Agent (DA) sammelt alle Informationen von allen vorhandenen SAs und speichert diese in einer Datenbank. Auch er kann die Anfragen eines UA bearbeiten.

  Wenn ein neuer Service hinzukommt, muss der SA diesen beim DA anmelden (Service Registration) und bekommt die erfolgreiche Registrierung bestätigt. Sucht ein Benutzer nach einem bestimmten Service, fragt er den DA und durchsucht dessen Services nach bestimmten Kriterien.

Clientseitig kann ein Distribution Provider den Discovery Service nutzen, um die vom Consumer gesuchten Dienste im Netzwerk ausfindig zu machen. Findet sich ein geeigneter Dienst, so kann ein Distribution Provider basierend auf den gefundenen Metadaten einen Service Proxy erzeugen und im Container als Services registrieren. Dies wird auch als Import eines remote Service bezeichnet. Ein Service Proxy sieht von außen wie ein lokaler Service aus, leitet aber alle Aufrufe intern an den remote Service weiter und übernimmt somit die eigentliche Kommunikation [KRW09].

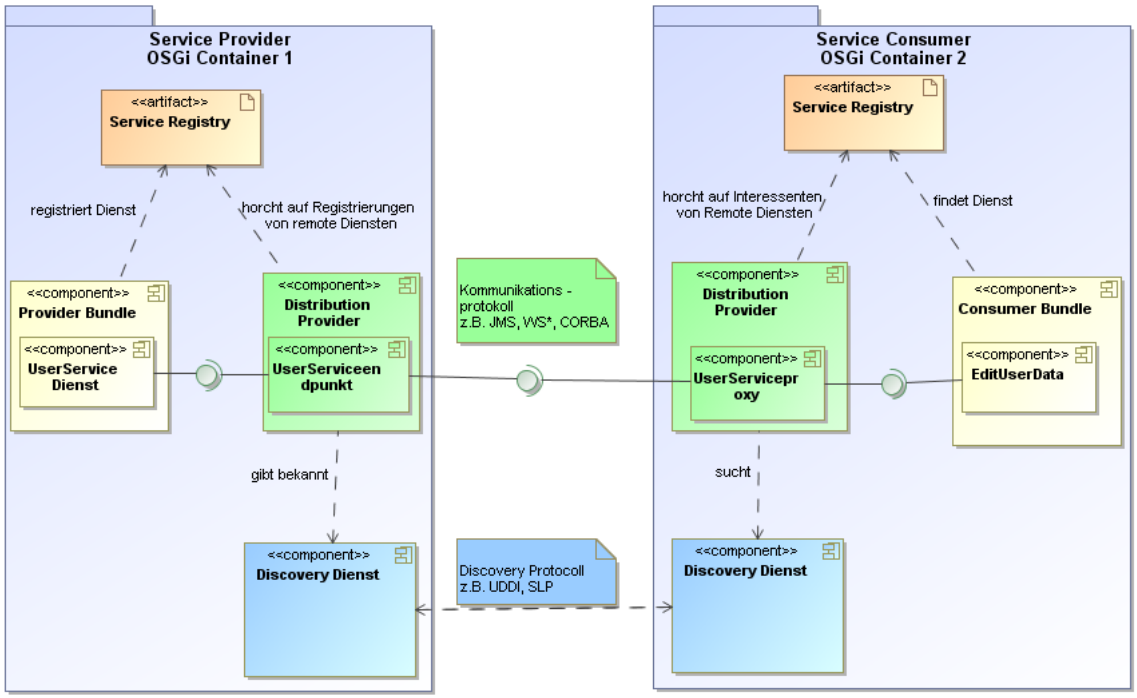

Abbildung 4.4: Mögliches UserService Modell anhand von OSGi 4.2

In Abbildung 4.4 wird gezeigt, wie der UserService (siehe Abbildung 2.3, implementiert nach dem OSGi 4.2 Standard aussehen könnte. Wie in der Abbildung gezeigt, läuft in jedem

Container eine lokale OSGi Registry, welche alle Services, die in diesen Container laufen, beinhaltet. Der Distribution Provider im Container 1 horcht auf Registrierungen von Remote Diensten in der Registry und verbreitet sie über den Discovery Dienst. Der Distribution Provider vom Client (Container 2) wartet auf Suchen nach remote Services. Die Abbildung zeigt nur einen Distribution Provider und Discovery Service, falls erwünscht können jedoch mehrere gleichzeitig in einem Container betrieben werden. Der Standard erlaubt es, dass die einzelnen Container Implementierungen verschieden sein können. Somit wäre es möglich das z.B. der Apache Tomcat unter einem OSGi Container laufen zu lassen, den Apache Tomcat unterstützt werden und die Services in anderen Container laufen könnten.

4.2 Frameworks

Die Framework Spezifikation ist nutzlos, wenn davon nicht auch produktiv einsetzbare Implementierungen existieren. Derzeit gibt es mehrere Implementierungen der OSGi Service Plattform, die bereits lange existieren und auch etabliert sind.

Die OSGi Spezifikation wird seit September 2009 in der Version 4.2 auf der Seite [osg09a] zur Verfügung gestellt. Diese Version der Spezifikation stellt Schnittstellen für eine verteilte Registry zur Verfügung. Diese Eigenschaft ist eine dringende Anforderungen an das Framework (siehe voriger Abschnitt 3.2). Bis alle Frameworks ihre Implementierungen an den OSGi 4.2 Standard angepasst haben wird es seine Zeit brauchen.

Bei den OSGi Implementierungen können zwei Gruppen unterschieden werden, je nachdem ob sie den Dienst für die Service Discovery (verteilte Registry) integrieren oder nicht. Folgende Open Source Implementierungen haben keinen Discovery Dienst implementiert, wobei die Discovery von anderen Komponenten realisiert sein kann und mit dem Frameworks verbunden werden kann ([WHKL08]):

- **Eclipse Equinox** [equ09] ist die Basis aller Eclipse-Anwendungen, allem voran der Eclipse-IDE. Damit ist Equinox die wohl am weitesten verbreitete laufende OSGi Plattform. Die Equinox Implementierung entspricht den Vorgaben der OSGi 4.2 Spezifikation seit der Version 3.5.

- **Prosyst mBedded Server Equinox Edition** [pro09] wird von der Firma Prosyst entwickelt und ist seit März 2007 als Open Source Variante verfügbar. Die kommerzielle Variante ist mBedded Server Professional Edition, die auf der Equinox Edition basiert aber unter anderem zusätzliche Services bereitstellt. Sowohl die freie, als auch die kommerzielle Version des mBedded Servers ist noch nicht OSGi 4.2 kompatibel.

- **Knopflerfish** [kno09] ist die Open Source Variante der kommerziellen OSGi- Implementierung Knopflerfish Pro. Beide werden derzeit von der schwedischen Firma Makewave entwickelt. Die Knopflerfish Implementierung 3.0, die den OSGi 4.2 Standard unterstützt, ist derzeit noch (Dezember 2009) in der “Beta“ Phase, also noch nicht für den produktiven Einsatz geeignet.

- **Apache Felix**[fel09] ist seit Anfang 2007 ein Top Level Projekt bei Apache. Seit Oktober 2008 ist es von der OSGi Allianz OSGi-zertifiziert. Bei der Entwicklung der OSGi 4.2 Spezifikation wurde Apache Felix als Beispiel Framework benutzt. Apache Felix ist seit Version 2.0 OSGi 4.2 kompatibel.

Einige Frameworks unterstützen nicht nur die verteilte Registry, sondern bieten auch eine Implementierung an, um die Services auch auf verteilten Servern ausfindig zu machen (Distribution). Die Frameworks die zum Zeitpunkt der Erstellung dieser Arbeit weiter verbreitet waren (siehe Kriterienkatalog 3.2.5) , werden in folgenden Kapiteln detaillierter evaluiert:

- **Eclipse Communication Framework (ECF)** [ecf09] Das Hauptaugenmerk des ECF-Projekts gilt der Bereitstellung von APIs zur Interprozesskommunikation innerhalb des Java Stacks. Es wird nicht nur auf die reine Anwendungsebene beschränkt, sondern es werden auch Werkzeuge zur Benutzerkommunikation geboten. Ein großer Teil von ECF ist die Einführung von Distributed OSGi, ebenfalls bekannt unter RFC-119. Dieses Projekt ist derzeit (Dezember 2009) noch in der Entwicklungsphase. Dieses Framework wird nicht weiter analysiert, aber die Weiterentwicklung sollte beobachtet werden.
- **Swordfish** [swo09] Swordfish setzt auf bekannten Open-Source-Projekten wie der OSGi-Implementierung Eclipse Equinox und dem Apache ESB ServiceMix auf. Es setzt Standards aus dem SOA-Umfeld wie Java Business Integration (JBI), Service Component Architecture (SCA) und SOAP (Simple Object Access Protocol) um und unterstützt Techniken wie ein verteiltes Deployment und eine Laufzeit-Service-Registry für die lose Koppelung von Services. Darüber hinaus enthält Swordfish ein Monitoring-Framework, über das man die Nachrichten verfolgen kann. Über einen sogenannten Remote Configuration Agent lassen sich verteilte Server von einem zentralen Repository aus konfigurieren. Leider ist dieses Framework noch in der Inkubations -Phase. Ein stabile Version von Swordfish ist erst für Mitte 2010 geplant.
- **Newton Framework**, siehe Kapitel 4.3.
- **The Paremus Service Fabric**, siehe Kapitel 4.4.
- **Apache CXF DOSGi**, siehe Kapitel 4.5

4.3 Newton

Newton ist ein Open Source Framework, das für verteilte, komponentenbasierte, serviceortientierten Anwendungsmodelle konzipiert wurde. Die Ziele von Newton sind Unterstützung dynamischer, verteilter Systeme und eine einfache Entwicklung von Komponenten in Form von POJOs.

In Newtons Kern werden folgende Technologien verwendet [new09]:

- OSGi(siehe 4.1)
- JINI - Jini definiert eine serviceorientierte Anwendungsarchitektur, welche sich als Ziel setzt, skalierbare, leicht veränderbare verteilte Systeme durch ihr Programmiermodell zu unterstützen. In Jini wird eine Service Registry verwendet, über die sich im Netzwerk verteilte Services gegenseitig finden können. Newton verwendet diese Technologie, um verteilte Komponenten zu verknüpfen
- Service Component Architecture (SCA) [oso09] - Unter SCA werden unterschiedlichste Spezifikationen zur Standardisierung serviceorientierte Architekturen zusammengefasst. SCA baut auf bewährten Ansätzen, wie z.B. Web Services oder RMI, auf und

erweitert diese. SCA spezifiziert Service-Komponenten, die zu Service Kompositionen zusammengefasst sind. Newton bietet eine dynamische Implementation der SCA Spezifikation.

4.3.1 Kriterienbewertung Newton

VT1 Verteilte Registry - Die Abbildung 4.5 zeigt wie UserService Findung der FRS24

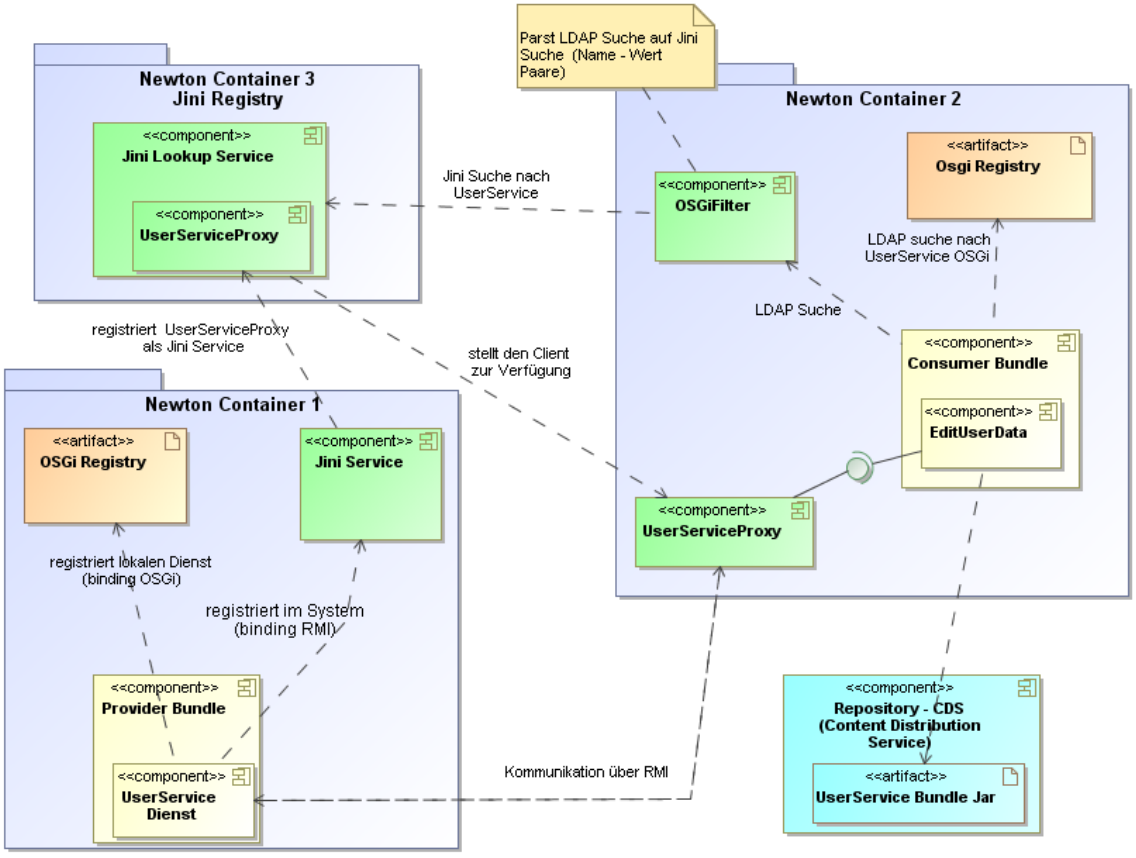

Abbildung 4.5: Mögliches UserService Modell in Newton Registry und Repository

Plattform in der Newton verteilten Registry realisiert werden könnte. Das Provider Bundle (siehe Container 1), registriert den UserService in der lokalen OSGi und der Jini Registry (oder Container 3). Die Registrierung in der Jini Registry wird vom Newton automatisch erledigt. Dazu muss in der Service Beschreibung über SCA stehen, dass dieser Service auch anderen Containern zu Verfügung stehen soll.

Wenn ein Client nach dem UserService sucht, schickt er eine Suche mit der Syntax von The Internet Engineering Task Force (IETF) Requests for Comments 1960 (RFC) [lda96] ab. Die Suche wird an der lokalen OSGi Registry ausgeführt. Falls die Suche nicht erfolgreich war, wird über den OSGi Filter in der Jini Registry gesucht. Jini unterstützt den RFC1960 Standard nicht, aber Newton stellt einen sogenannten OsgiFilter bereit, der die LDAP Suchanfragen an die Jini Registry entsprechend umwandelt. Somit werden die Suchanfragen in Newton an die Lokale OSGi und an die Jini Registry nach der gleichen Syntax ausgeführt. Der Client

bekommt nicht mit wo genau gesucht wird. Wenn die Suche erfolgreich war, wird ein UserServiceProxy an den Client Container übergeben. Dann werden entsprechende Informationen über den Service aus der Repository geholt.

Die Repository ist in Newton mit Hilfe eines sogenannten Content Distribution Services (CDS) realisiert, welches eine Eigenentwicklung von Newton ist. Dort stehen alle von Administratoren öffentlich gestellten Bundles (Artefakte) zur Verfügung und können auch geändert werden.

Wenn der Client die UserService Bundle gefunden hat, kann er den UserService auch aufrufen, dabei verläuft eine Kommunikation zwischen den Client und dem UserServiceProxy. Von der Verbindung zwischen UserServiceProxy und dem eigentlichen UserService merkt der Client nichts. Der Ablauf der Dienstfindung von Newton kann auch dem Sequenzdiagramm in der Abbildung 4.6 entnommen werden.

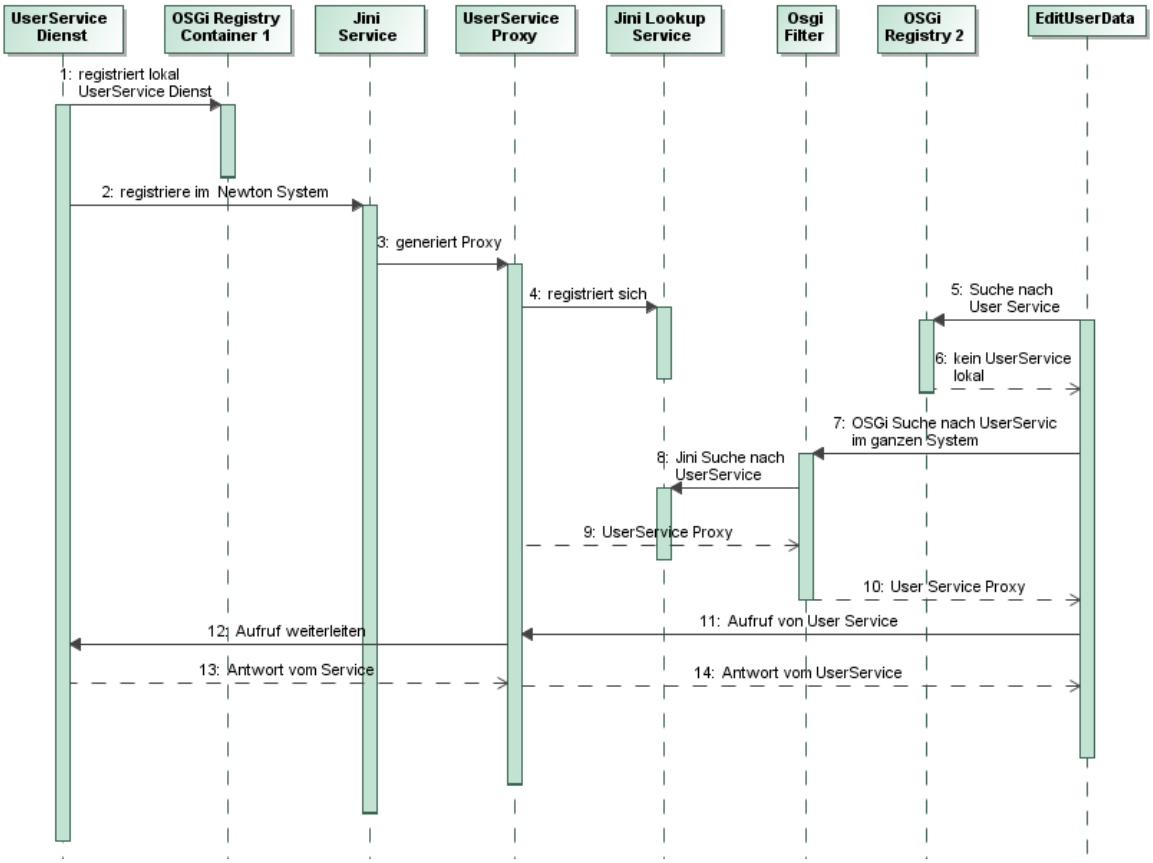

Abbildung 4.6: Ablauf der Service Registrierung und Findung in Newton

Die Findung der einzelnen Newton Containers geschieht automatisch per Broadcast anhand der gleichen Gruppen ID des Containers (FabricName Parameter). Es müssen keine IP Adressen und Ports angegeben werden um anzudeuten auf welchen Servern die Services laufen. Die Service-Findung geschieht also völlig transparent. Als einzige Voraussetzung muss im Netzwerk der Broadcast zwischen den Servern möglich sein.

VT2 Zugriff über ORB - Die Kommunikation zwischen zwei Services innerhalb des selben Containers wird in Newton per OSGi realisiert. Falls Services in verschiedenen Newton

Containern laufen wird die Kommunikation per RMI ermöglicht. Als Beispiel zur Verifizierung wurde die UserService Implementierung, die in Kapitel 5.1 detaillieren beschrieben wird, gewählt.

Zuerst wurde ein Beispiel-Client und Server in verschiedenen Newton Containern auf einen Server installiert und gestartet. Die Container wurden unter dem gleichen FabricName gestartet und der Server so konfiguriert, dass er seinen Service auch anderen Containern zur Verfügung stellt. Server und Client können über die Kommunikation UserObjekte austauschen. Das gleiche Beispiel wurde mit Newton auf verschiedenen Servern, die im selben Netzwerk laufen, erfolgreich ausprobiert. Es gab keinen Unterschied ob die Kommunikation innerhalb eines oder zwischen verschiedenen Servern realisiert wird.

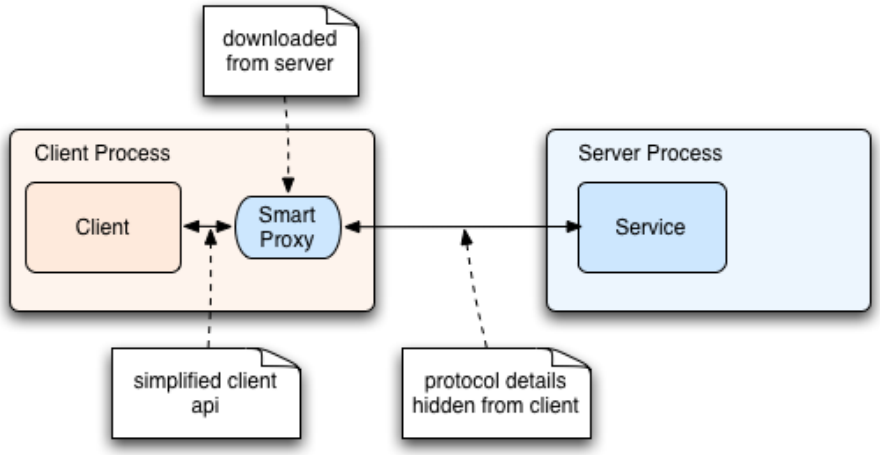

Abbildung 4.7: Smart Proxy Modell [new09]

Newton bietet die Möglichkeit der Einbindung der Jini Smart Proxys (Abbildung 4.7). Die Jini Smart Proxys repräsentieren die eigentlichen Services. Diese müssen in Java implementiert sein. Dies hat den Nachteil, dass für bestehende Dienste erst Proxys implementiert werden müssen um sie Jini-fähig zu machen. Die Proxys können dynamisch zum Client heruntergeladen werden und das Java-Typsystem kann verwendet werden um später passende Dienste einzusetzen. Durch die Jini Smart Proxys ergibt sich die Möglichkeit die Timeout Behandlung, das Caching, und Batching sowie weitere Eigenschaften des Clients transparent zu halten und alles im Service Anbieter zu behandeln.

Es gibt auch die Möglichkeit außerhalb von Newton auf die Objekte und Schnittstellen zuzugreifen. Die Applikation muss eine bestimmte Newton JAR und die Multicast Adresse, Port und FabricName der Container angeben.

VT3 Zugriff über Webservice- Es existiert keine vorgegebene Webservice Implementierung in Newton. Aber durch die dynamische Unterstützung von SCA in Newton ist die Anbindung durch Webservice theoretisch möglich. Diese Eigenschaft ist in Newton nicht Integriert, sondern Newton muss erweitert werden. Die Erweiterung wird über die Realisierung von Schnittstellen, die von Newton vorgegeben in das Newton Framework auch eingebunden werden. In nächster Zukunft ist keine Webservice Anbindung von Newton vorgesehen.

Es existiert ein Open Source Projekt Mule4Newton [mul09], welches Newton und Mule, den bekannten Open Source ESB (Enterprise Service Bus), verbindet. Dadurch wird die

Anbindung auf die Newton Services über Webservice Schnittstellen oder andere Protokolle wie z.B. JMS ermöglicht. Derzeit (September 2009) hat dieses Projekt die Versionsnummer "0.1", somit nicht für das System von FRS24 relevant.

VT4 Asynchrone Kommunikation (JMS) - Durch die Unterstützung von SCA durch Newton ist die Kommunikation durch JMS möglich. Dazu muss die entsprechende Anbindung implementiert werden und durch Realisierung bestimmter Schnittstellen in das Newton Framework integriert werden. Eine JMS Implementierung von Newton ist derzeit nicht vorgesehen. Eine andere Möglichkeit die JMS Kommunikation zu gewährleisten besteht durch das Open Source Projekt Mule4Newton.

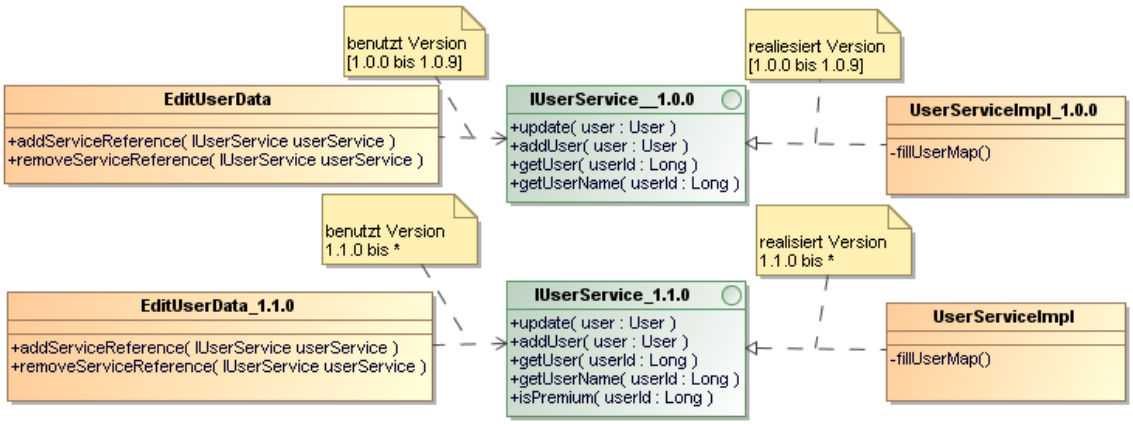

Abbildung 4.8: Versionierung des UserService in Newton

DV Versionierung - Als Beispiel für die Evaluierung der Versionierung wurde das UserService Beispiel so erweitert wie in Abbildung 4.8 dargestellt. Die Versionierung von Services wird in OSGi innerhalb der Manifest Dateien definiert und dann vom Framework interpretiert. Die Manifest Datei vom UserServiceImpl (Bundle-Version: 1.0.0) sieht wie in Listing 4.1 aus. Die Zeile `Import - Package:` gibt an, welche Packages in welcher Version falls erforderlich von der UserService Implementierung gebraucht werden.

UserServiceImpl importiert das Package `de.frs24.business.data;version="[1.0.0 ,1.0.9]"`, in dem sich die Schnittstelle IUserService 5.2 befindet. Das import besagt, dass die UserService Schnittstelle in der Version 1.0.0 bis 1.0.9 im CDS vorhanden sein muss. In der IUserService Manifest Datei muss wiederum folgende Zeile stehen:
`Export-Package:de.frs24.business.data;version="1.0.0"`.

Der Client (EditUserData), der den UserService benutzt, muss auch das Interface IUserService in eine der Versionen von 1.0.0 bis 1.0.9 importieren, um den Service benutzen zu können. In der Manifest Datei von EditUserData (Version 1.0.0) sollte dann folgenden Zeile stehen: `Import-Package: de.frs24.business.data;version="[1.0.0,1.0.9]"`. Bundles IUserService, UserServiceImpl und Edituser Data wurden in den oben genannten Versionen in Newton installiert und funktionieren.

Um zu verifizieren, dass die Bundles in Newton in verschieden Versionen gleichzeitig installiert werden und sich finden können, wurde das Interface IUserService um die Methode `isPremium(Long userID)` erweitert und dann auch die Version der Schnittstelle auf 1.1.0 gesetzt. Die Java Klassen UserServiceImpl und EditUserdata wurden so angepasst, dass die IUserService Schnittstelle in Version 1.1.0 importiert werden kann. Die Versionen der Bund-

les wurden im Newton installiert und es konnte beobachtet werden, dass die Clients jeweils die richtige Version benutzen. Falls die erforderliche Version von UserService nicht installiert war, wurde auch keine andere genommen.

Die Repository (CDS - Content Distribution Service) unterstützt auch mehrere Versionen der jars im System. Somit ist außerdem die Versionierung in der Verteilten Registry möglich und funktioniert.

Listing 4.1: UserService OSGi Manifest.MF Datei

```
Manifest-Version: 1.0
Installable-Component: true
Installable-Component-Templates: META-INF/newton/UserService.composite
Bundle-Version: 1.0.0
Bundle-Name: UserService
Bundle-Activator: org.cauldron.newton.installer.component.client.Activator
Bundle-ManifestVersion: 2
Created-By: sigil.codecauldron.org
Bundle-Description: OSGi implementation of UserService
Bundle-SymbolicName: UserService
Import-Package: de.frs24.business.data;version="[1.0.0,1.0.9]",org.cau
ldron.newton.framework,org.osgi.framework,org.osgi.util.tracker
```

MT1 Ausfallbehandlung - Die Ausfallbehandlung wird mit Hilfe von Newton Replication Handlern unterstützt. Listing 4.2 zeigt wie in Newton `FixedSizeReplicationHandler` zum einen Service hinzugefügt wird. Dazu musst die Definition des Replication Handlers innerhalb des `<composite>` Elements des UserService.composite (siehe Listing 5.6) eingefügt werden. Dann werden beim Starten des UserServices automatisch 2 Instanzen des Services installiert und gestartet. Wenn eine Service Instanz abstürzt, wird in der verteilten Registry automatisch nach einem passenden Service gesucht. Wenn ein passender Service gefunden wird, werden automatisch weitere Anfragen an ihn weitergeleitet. Bei Ausfall von einzelnen Services werden keine neuen Services vom Framework gestartet.

Listing 4.2: Replication Handler von Newton

```
<replication.handler type="org.cauldron.newton.system.replication.
    FixedSizeReplicationHandler">
  <property name="size"
            value="2"
            type=integer" />
</replication.handler>
```

MT2 Last - In Newton existiert kein Tool, das Warnmeldungen bei bestimmter Last anzeigen oder verschicken könnte. Für die Lastverteilung existiert ein Vorschlag der über Java Spaces erreicht werden kann, siehe dazu Beispiel ScatterGatherDemo auf der Newton Webseite [new09]. Welcher zeigt, wie durch viele gleiche Services die Last auf alle verteilt und damit die Performance gesteigert werden kann.

DR1 Zentrale GUI - Console vollständig vorhanden, erlaubt scripting und Verteilung von Instanzen in allen Containern des Newton Systems. Die Console bietet außerdem die Möglichkeit den Status einzelner Services über die Console zu ändern. Die Statusanzeige der Services innerhalb einer verteilten Umgebung ist aus der Console auch möglich.

Newton unterstützt Knopflerfish [kno09] und Eclipse Equinox [equ09] OSGi - Implementierungen. Damit ist es möglich alle Consolenbefehle aus Equinox OSGi und Knopflerfish auch in Newton auszuführen.

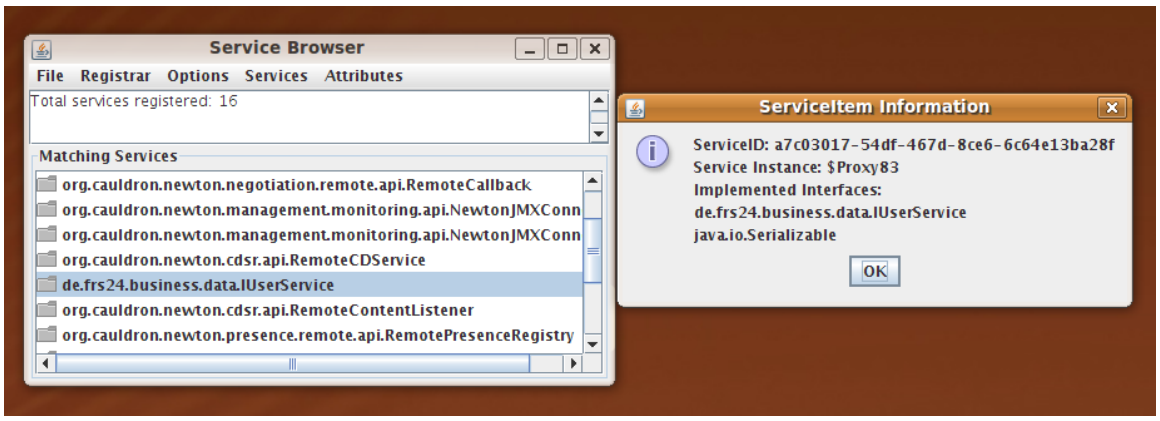

Abbildung 4.9: Anzeige der Jini Service Registry von Newton (links) und Informationen über den IUserService (rechts)

GUI - Eine graphische Anzeige der Newton Registry ist über Jini Service Console [jin09], wie in Abbildung 4.9, möglich. Dort werden alle installierten verteilten Services einer Newton Gruppe (FabricName) nur angezeigt, Änderungen an den Services können nicht vorgenommen werden. Abbildung 4.9 stellt die Eigenschaften der Schnittstelle IUserServices vor. Durch Anzeigen der Service Instanz wird die Information auf welchem Server der bestimmte Service läuft, angezeigt.

DR2 Deployment zur Laufzeit - Newton bietet ein dynamisches Deployment zur Laufzeit des ganzen Systems. In Containern können Services, auch in verschiedenen Versionen, zur Laufzeit installiert werden, ohne dabei eine Unterbrechung des Systems hervorzurufen.

Um diese Eigenschaft zu verifizieren wurde das Beispiel, das im Kapitel 5.2.1 detaillierter beschrieben wird, in Newton installiert und gestartet. Die Newton Consolenausgabe ist in Abbildung 4.10 dargestellt. In den ersten vier Zeilen sieht man den Status der Services und sieht, dass ein Client (EditUserData) und 2 Services (UserService) in verschiedenen Versionen laufen. Der Service Ausfall wurde so simuliert, dass der Service angehalten wurde, mit dem Befehl "stop 113". Man sieht, dass der Client automatisch die andere Service Instanz findet und benutzt, ohne dabei Fehlermeldungen von sich zu geben.

```
112     RESOLVED    IUserService_1.0.0
113     ACTIVE      UserService_1.0.0
114     ACTIVE      EditUserData_1.0.0
115     ACTIVE      UserService_1.0.1
>
 USER 2:  user2 [2] 13001(user2@frs.de) , bd: 02.12.1980, zip: 12345, country: 0, city: munich

 USER 2:  user2 [2] 13001(user2@frs.de) , bd: 02.12.1980, zip: 12345, country: 0, city: munich

 USER 2:  user2 [2] 13001(user2@frs.de) , bd: 02.12.1980, zip: 12345, country: 0, city: munich
stop 113
Stopped: UserService (#113)
>  Pseudonym User 1: pseudonym1

 USER 2:  pseudonym2 [2] 13001(user2@frs.de) , bd: 02.12.1980, zip: 12345, country: 0, city: munich

 USER 2:  pseudonym2 [2] 13001(user2@frs.de) , bd: 02.12.1980, zip: 12345, country: 0, city: munich
```

Abbildung 4.10: Consolenausgabe von UserService in Newton

DR3 Authentifizierung der Admins - Ist in Newton nicht vorhanden.

ST1 Verbreitung - Newton ist seit 2006 als Open Source vorhanden. Newton bietet die Grundlage für das kommerzielle Produkt Paremus Service Fabric [par09]. Die Entwickler sind die selben wie bei Paremus Service Fabric. Somit ist keine vollständige Sicherheit gegeben, dass Newton immer weiterentwickelt wird und neue Features hinzukommen. Derzeit existiert keine große Open Source Community, durch welche schnell Hilfe und Dokumentation gefunden werden kann. Die Dokumentation existiert nur auf der Newton Webseite, welche einige Bespiele beinhaltet wie man mit Newton arbeitet. Aber es existieren zur Zeit (September 2009) keine expliziten Beispiele für den Zugriff über JMS oder Webservice mit Newton.

ST2 Standard Komponenten - Newton unterstützt vollständig OSGi und Spring Dynamic Modules [spr09]. Für Komponenten ist einiges an Konfigurationsaufwand nötig damit um sie in Newton deployen zu können, da erst alle Komponenten über OSOA Standards [oso09] konfigurieren werden müssen.

Durch die Unterstützung von Knopflerfish und Eclipse Equinox OSGi-Implementierungen können Komponenten in Newton installiert werden, auf die allerdings nur innerhalb eines Containers zugegriffen werden kann. Für den verteilten Zugriff ist die SCA-Erweiterung von Newton nötig.

ST3 Standard Zugriff - Wie schon bei den Kriterien VT1-VT4 beschrieben, ist nur der Zugriff per RMI oder eventuell über Jini Smart Proxys möglich. JMS und Webservice werden derzeit und auch in absehbarer Zeit nicht unterstützt.

BR1 Nutzung - Entwickler - Es existiert ein Eclipse Plugin unter dem Namen Sigil [sig09], welches die Entwicklung von OSGi Komponenten in Eclipse für Newton erleichtert. Dieses Plugin bietet auch die Möglichkeit Newton direkt in Eclipse einzubinden und es aus Eclipse zu starten und zu steuern. Somit können Komponenten schnell in den Newton Container deployed werden. Es existieren auch noch Addons zum Testen der Services mit Hilfe von Mock Objekten oder Addons für die Ivy Einbindung. Die Entwicklung der Services wird, im Vergleich zu heutigen Stand, um einiges erleichtert werden.

BR2 Schulungsaufwände - Entwickler müssen erst mit Umgebungen wie SCA, OSGi und Spring angelernt werden. Administratoren müssen in die Arbeit mit dem Newton Container eingearbeitet werden. Die Schulungsaufwände für Entwickler sind nicht zu vernachlässigen. Für Administratoren ist der Aufwand noch größer da viele start-stopp Scripte an Newton angepasst werden müssen.

BR3 Migrationsaufwand - Zuerst müssen Services als OSGi Bundles umgesetzt werden und dann in Newton konfiguriert werden. Gegebenenfalls muss man Erweiterungen für Newton implementieren, wie z.B eine JMS oder ein Webservice Anbindung für die Services. Die Verwaltung der einzelnen Komponenten (Bundles) muss auf geeignete weise gesteuert werden. Der Aufwand ist daher ziemlich groß.

SK Skalierbarkeit - Die Skalierbarkeit wird von Newton unterstützt. Während der Laufzeit des ganzen Systems können neue Service Instanzen und auch neue Container dynamisch zum ganzen System hinzugefügt werden, oder weggenommen werden.

PE Performance - Anhand des Beispiels Scatter Gather Demo auf der Newton [new09] Webseite, steigt die Performance wenn die Implementierung mit JavaSpaces realisiert wird. Von Newton heraus wird keine bestimmte Performance versichert.

Lizenzkosten- Keine, es ist aber auch kein kommerzieller Support vorhanden.

4.3.2 Fazit

Newton ist ein Framework, das die Skalierbarkeit, Versionierung und dynamisches Deployment vollständig unterstützt. Der Zugriff über Webservice oder JMS muss selbst implementiert werden. Aber es gibt die Erweiterung für Newton Mule4Newton, die JMS und Webservice für Newton theoretisch unterstützt. Leider ist die Implementierung von Mule4Newton erst in der Version 0.1 und derzeit ist nur ein Entwickler auf der Webseite angegeben. Daher ist Mule4Newton für den Produktiveinsatz nicht zu empfehlen.

Newton ist für den Produktiveinsatz der FRS24 Plattform nicht geeignet, da für einige Anforderungen eine Alternative gesucht werden muss. Die Dokumentation fällt sehr mager aus, ein Kommerzieller Support ist nicht vorhanden. Die Community von Newton ist sehr klein, dies ist sehr schlecht da auf Hilfe gewartet werden muss. Diese Eigenschaften sind von äußerster Bedeutung, damit das bestehende System migriert werden kann und ein stabiles FRS24 System mit Newton zu realisiert wird. Es wäre ein sehr großer zusätzlicher Implementierungsaufwand nötig um mit Hilfe von Newton ein stabiles und produktives System zu realisieren. Der OSGi 4.2 Standard wir nicht auf der Webseite von Newton erwähnt und damit ist die Anpassung an diesen auch in Frage gestellt.

4.4 Paremus Service Fabric (Kommerziell)

Das Konzept der Service Fabric beruht auf der Verwaltung einer verteilten, komplexen Service Umgebung. Die Service Fabric bietet verteilten Anwendungen einen sehr hohen Grad an Abstraktion von der Hardware und des Betriebssystems für die Services. Der Service Fabric Kern besteht aus Newton und bietet zahlreiche Erweiterungen von Newton 4.3:

- Persistente Speicherung des ganzen verteilten Systemzustands
- Verwaltung aller Instanzen von Service Fabric von einer Instanz aus
- Entfernen und Hinzufügen von Servern zum Service Fabric System.
- Graphische Oberfläche zur Verwaltung von Service Fabric
- Authentifikation von Administratoren
- Unterstützung von WAR Dateien in Service Fabric

Paremus Service Fabric setzt sich strukturell aus folgenden Komponenten zusammen:

- **Atlas Agenten** laufen lokal auf Servern und wünschen sich die Teilnahme in einer Service Fabric Umgebung. Jeder Agent ist zuständig für das Herunterladen der Service Fabric Software, den Bundels die Services beinhalten. Service Fabric konfiguriert und installiert Software auf den Atlas Agenten.
- **Nimble Nodes** beinhalten einen oder mehrere Atlas Agenten und bilden einen Teil oder ein ganzes Service Fabric. Nimble Nodes repräsentiert ein Gruppe von Computer Knoten auf denen Atlas Agenten laufen.
- **System** repräsentiert in Service Fabric eine Business Applikation. Im System von FRS24 könnte UserService eine Business Applikation sein inklusive der Logik und persistenten Datenspeicherung

- **System Composites** beschreibt einen Teil des Systems. So könnte zum Beispiel die persistente Datenspeicherung von Usern im FRS24 System eine Composite sein.
- **Services & Bindings** - System Composites exportieren Services mit Hilfe von Service Bindings. Service Bindings stellen fest wie zugehörige Services von Clients gefunden und anschließend benutzt werden.

4.4.1 Kriterienbewertung Service Fabric

VT1 Verteilte Registry - Die verteilte Registry ist wie in Newton realisiert (Abbildung 4.5). Um die verteilte Registry zu verifizieren wurden auf zwei verschiedenen Servern Atlas Agenten gestartet. Ein Nimble Node wurde mit Hilfe der Posh-Console (siehe Kriterium DR1) auf einem der Server gestartet und die Atlas Agenten wurden dem System hinzugefügt. Dann wurde das Beispiel des UserServices welches im Kapitel 5.2 detaillierter beschrieben wird, wurde auf einem Server der Client(EditUserData) und auf dem anderem der UserService. Es konnte beobachtet werden, dass der Client den Server finden und ohne Konfigurationsaufwand aufrufen konnte. Die installierten Beispiele haben nichts von der Verteilung bemerkt. Damit ist dieses Kriterium vollständig in Paremus Service Fabric erfüllt.

VT2 Zugriff über ORB - Da Service Fabric auf Newton basiert ist, wird der Zugriff über ORB wie auch in Newton ermöglicht. Das oben genannte Beispiel, überträgt User Objekte zwischen verschiedenen Atlas Instanzen von Paremus Service Fabric. Auch wenn diese sich auf verschiedenen Servern befinden, somit ist dieses Kriterium auch vollständig vorhanden. Wie in Newton sind folgende Zugriffsmöglichkeiten möglich: OSGi, RMI und JINI Smart Proxys.

VT3 Zugriff über Webservice Wie bei Newton nicht vorhanden, aber nach Aussagen des Paremus Service Fabric Supports werden in den kommenden Versionen von Service Fabric weitere Anbindungen, darunter auch Webservice, implementiert sein.

VT4 Asynchrone Kommunikation (JMS) Nicht vorhanden, aber wie Newton durch SCA erweiterbar durch Eigenentwicklung.

DV Versionierung - Wird wie in Newton unterstützt, das gleiche Beispiel mit verschiedenen Versionen hat die gleichen Ergebnisse auch in Paremus Service Fabric erzielt. Der Unterschied in der Versionierung nicht zwischen Paremus Service Fabric und Newton konnte nicht festgestellt werden.

MT1 Ausfallbehandlung Newton Application Handler werden auch von Paremus unterstützt und ermöglichen das mehrere Instanzen vom gleichen Service in der Service Fabric Nimble Umgebung laufen und somit eine Ausfallsicherheit geben. Wenn man dem UserService Beispiel aus Listing 5.6 den Replication Handler aus 4.2 hinzufügt und dann während der Laufzeit des EditUserData Clients die benutzte UserService Instanz herunter fährt wird automatisch eine andere Instanz genommen. Im Gegensatz zu Newton bietet Service Fabric noch die Eigenschaft, dass Service Fabric einen bestimmten Zustand während der Laufzeit des ganzen Systems beibehalten kann. Paremus Service Fabric kann so gestartet werden, dass es folgende Zustände beibehält:

- Simple Service Fabric - besteht aus einem infra Knoten und beliebig vielen einfachen Knoten. Innerhalb eines Infra Knoten laufen alle Services die für die Infrastruktur von Service Fabric zuständig sind. Die Business Services können sowohl innerhalb infra als auch innerhalb von einfachen Knoten laufen. Wenn innerhalb einer simple Service Fabric ein UserService Beispiel läuft und der Infra Knoten des Service Fabric ausfällt,

muss Service Fabric neu gebildet und alle CDS Dateien müssen wieder installiert und die Services gestartet werden.

- Durable Service Fabric - hat wie simple Service Fabric einen Infra Knoten und beliebig viele einfache Knoten. Aber durable Service Fabric speichert den Systemzustand inklusive der Software Artefakte, den Status der installierten Software und die Security Einstellungen (Benutzter und Gruppen) im CDS. Die Informationen über den Systemzustand werden auf den Infra Knoten gespeichert und beim Neustart werden die Einstellungen automatisch geladen. Der infra-Node wird in der Konfigurationsdatei fest definiert und muss beim Neustart verfügbar sein.

- Robust Service Fabric - besteht aus mindestens drei infra Knoten und beliebigen einfachen Knoten. Robust Service Fabric speichert wie die durable Konfiguration den Systemzustand. Service Fabric Infrastruktur Services werden dynamisch installiert, gestartet und falls notwendig auf den Infrastruktur Knoten neu installiert. Beim installierten UserService Beispiel in einer robusten Service Fabric Umgebung, wirkte sich der Ausfall einzelner infra Knoten nicht negativ auf das System aus.

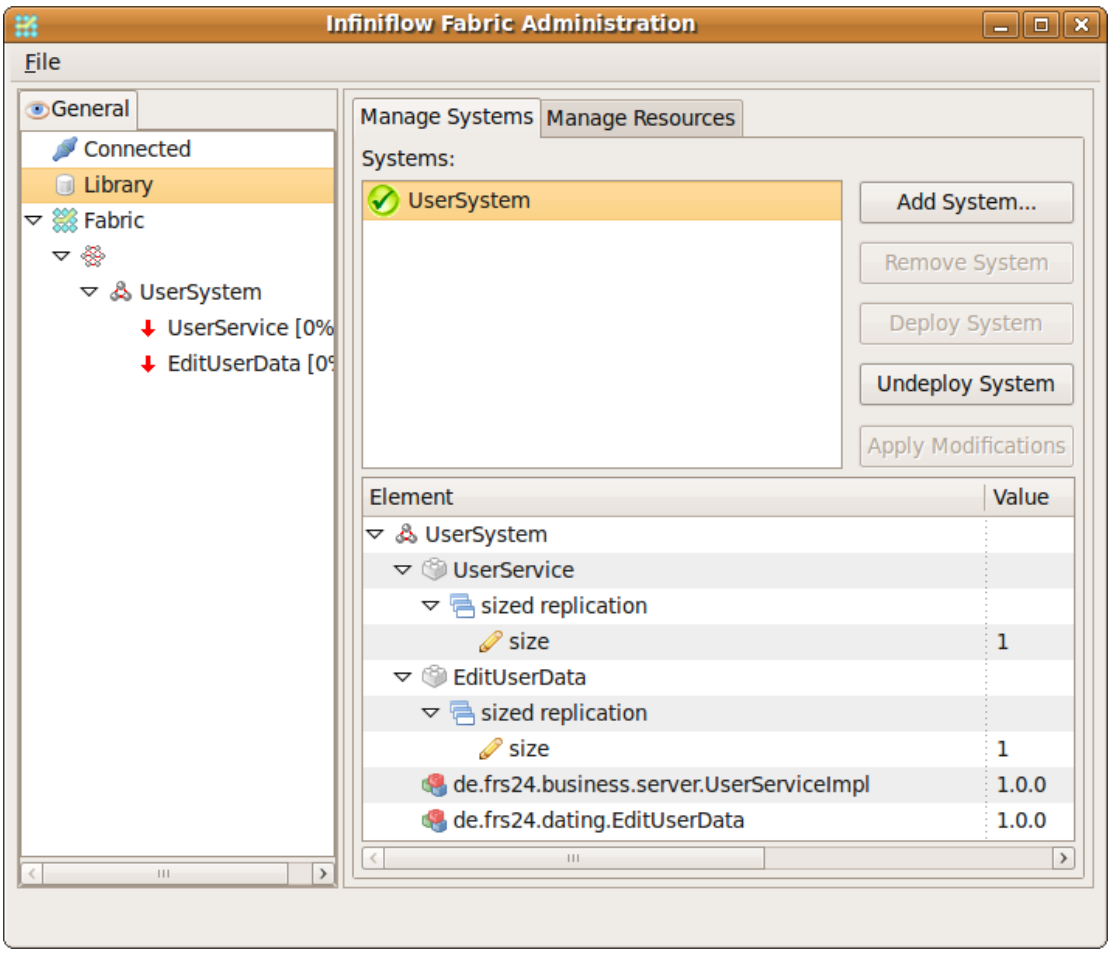

Abbildung 4.11: Administrationsoberfläche in Paremus Service Fabric

MT2 Last - Paremus Service Fabric kann bei kritischer Last Warnmeldungen an der GUI 4.11 anzeigen. Für die Lastverteilung existiert wie in Newton der Ansatz zur Implementierung mit Java Spaces.

DR1 Zentrale GUI - Die Console, in Service Fabric Posh genannt, ist wie in Newton vollständig vorhanden. Posh basiert auf den OSGi 4.2 RFC132 Command Line Interface Standard, und bietet eine bash- ähnliche Console mit Unterstützung für OSGi Frameworks Equinox, Felix und Knopflerfish. Einige Funktionen aus der bash - Console werden unterstützt wie unter anderem cat, cd. cp, each, if, not. Aus einer Posh Console können auf allen Servern, auf denen Service Fabric läuft, Services installiert, konfiguriert und Statusänderungen vorgenommen werden.

Abbildung 4.11 zeigt die Paremus Service Fabric Administrationsoberfläche mit den installierten UserService Beispiel. Sie verbindet sich zum Service Fabric und bietet die Möglichkeit Informationen über das laufende verteilte System anzuzeigen und auch Änderungen am System vorzunehmen. Zu den Funktionen der GUI gehören Userverwaltung, Loganzeige, Installation von neuen Services und Statusanzeige des Systems. In der Abbildung 4.11 ist die Installation von Systemen angezeigt. Als Beispiel wurde das System UserService mit Client installiert. und ist in Listing 4.3 beschrieben.

Listing 4.3: SCA UserService Beschreibung für Newton und SCA

```
<system name="UserSystem">
  <description>An example static system</description>
<system.composite name="EditUserData"
                  bundle="de.frs24.dating.EditUserData"
                  version="1.0.0" >
  <description>Created by lleovac</description>


  <reference name="interface" multiplicity="0..1">
    <interface.java interface="de.frs24.business.data.IUserService"/>
    <binding.rmi/>
  </reference>

  <component name="client">
    <reference name="ServiceReference"/>
    <implementation.java.callback impl="de.frs24.dating.EditUserData"/>
  </component>

  <wire>
    <source.uri>client/ServiceReference</source.uri>
    <target.uri>interface</target.uri>
  </wire>
</system.composite>

<system.composite name="UserService"
                  bundle="de.frs24.business.server.UserServiceImpl"
                  version="1.0.0" >
  <description>Created by lleovac</description>

  <service name="interface">
  <interface.java interface="de.frs24.business.data.IUserService"/>
    <binding.rmi/>
  </service>

```

```
35   <component name="implementation">
36   <implementation.java.callback impl="de.frs24.business.server.
         UserServiceImpl"/>
37   </component>
38
39   <wire>
40   <source.uri>interface</source.uri>
41     <target.uri>implementation</target.uri>
42   </wire>
43
44   </system.composite>
45
46 </system>
```

Derzeit ist die Administrator GUI als Java Applikation vorhanden, eine Webbasierte GUI befindet sich derzeit in der Entwicklung.

DR2 Deployment zur Laufzeit - Das gleiche Beispiel wie in Newton DR2 wurde auch bei Paremus ausgeführt. Es lieferte die gleichen Ergebnisse wie in Newton.

DR3 Authentifizierung der Admins - Authentifizierung wird unterstützt und ist über die Administrations-GUI, siehe Abbildung 4.11, und über Konfigurationsdateien verwaltbar. Es können User in beliebigen Gruppen mit bestimmten Rechten für eine Service Fabric Umgebung angelegt werden. Die User können sich dann über ssh mit “Nimble Node“ verbinden und das laufende System verändern. Dies betrifft nicht nur die Services die auf einem Nimble Node laufen, sondern auch die Services, die auf allen Atlas Agenten auf verschiedenen Servern laufen und zu diesem Nimble Node gehören.

ST1 Verbreitung - Die Firma Paremus existiert seit 2001 und ihr erstes kommerzielles OSGi Framework ist seit 2005 (damals unter dem Namen Infiniflow) verfügbar. Seit Juni 2009 wird Infiniflow unter dem Namen Paremus Service Fabric weiterentwickelt. Da es ein kommerzielles Produkt ist, ist die Dokumentation und Hilfe nur auf der Paremus Seite verfügbar. Im Rahmen dieser Arbeit kamen veraltete und fehlerhafte Dokumentationen zum Vorschein. Der Support von Paremus reagierte auf Fragen dagegen schnell mit klaren Antworten.

ST2 Standard Komponenten - Wie Newton unterstützt Service Fabric vollständig OSGi und Spring Dynamic Modules. In Service Fabric ist das Verwalten und Deployen von Apache Tomcat WAR Dateien möglich. Dies ist ein sehr großer Vorteil, da damit leicht das FRS Portal innerhalb von Paremus Service Fabric integriert werden könnte. Die Administration von Apache Tomcats könnte auch, über die POSH Console, wie andere Service von einer Stelle aus geändert werden.

ST3 Standard Zugriff - Die Kommunikation lehnt sich an die Standards OSGi und RMI an. Java JVM Versionen 5.0 und 6.0 werden von aktuellen Service Fabric (Version 1.5) unterstützt. Seit September 2009 ist Paremus Vollmitglied der OSGi Alliance. Damit ist sichergestellt, dass Service Fabric auch neue OSGi Spezifikationen unterstützen wird.

BR1 Nutzung Entwickler - Wie in Newton ist der Nutzen nach der Einarbeitung sehr groß, da das Eclipse Plugin Sigil sehr viel Hilfe bei der Entwicklung verschafft. Für die Administratoren würde der Nutzen auch nach der Umsetzung groß sein, da ein schneller Überblick über das System und den Systemstand möglich wäre. Die Services können als POJOs entwickelt werden, was das ”Unit Testen um einiges erleichtert, da man sich nicht um die Abhängigkeiten der Services nicht kümmern muss.

BR2 Schulungsaufwände - Wie in Newton sind für die Entwickler die Schulungsaufwände sehr groß, da das Anlernen an OSGi, SCA und Spring nicht zu vernachlässigen ist.

Für die Administratoren könnte die Wartung der verteilten Umgebung um einiges erleichtert werden. Der Nachteil ist, dass sich die Administratoren erst mit dem Paremus Service Fabric vertraut machen müssen.

BR3 Migrationsaufwand - Der Migrationsaufwand ist sehr groß, da die bestehende Software erst als OSGi Bundles implementiert werden muss und dann mit Spring und SCA in Service Fabric integriert und konfiguriert werden muss. Der Aufwand für die Administratoren ist auch nicht zu vernachlässigen, da die Service Fabric Umgebung mit vielen Servern konfiguriert werden muss.

SK Skalierbarkeit - Die Anforderungen an die Skalierbarkeit werden über den System Provisioner vollständig unterstützt. Der System Provisioner ist ein hoch-skalierbares asynchroner Service Fabric Sub-System. Der System Provisioner baut und wartet ein Runtime-Modell für jedes laufende System anhand der Beschreibung des Systems und anhand horizontaler Skalierungsanforderungen an jedes System.

PE Performance - Paremus verspricht sehr gute Performance. Ob es auch für das System von FRS24 auch eine Verbesserung der Performance mit sich bringt kann erst nach der Umsetzung festgestellt werden.

Lizenzkosten - Da Paremus Service Fabric eine kommerzielles Produkt ist, fallen für jeden Server Lizenzkosten pro Jahr in Höhe von 1,695 $ mit einem normalen Support von 5 Tage / Woche à 9 Stunden an. Für einen 24h / 7 Tage die Woche Support sind die Kosten mit 2095 $ pro Server vergleichsweise hoch.

4.4.2 Fazit

Paremus Service Fabric bietet einige Erweiterungen von Newton und damit auch Eigenschaften, die für den Produktiveinsatz bei FRS24 relevant wären, wie z.B. der WAR Dateien Support, Sicherheit für Administratoren oder persistente eine Speicherung des ganzen verteilten Systemzustands. Die Dokumentation ist verbesserungswürdig, aber der Support ist sehr schnell und gut.

4.5 Apache CXF Distributed OSGi

Apache CXF [cxf09] ist ein Open Source Framework für die Entwicklung von Web-Services-Anwendungen. Das Projekt ist aus den Projekten Celtix und XFire hervorgegangen. Die Funktionen von Apache CXF sind [ZLMG08]:

- Unterstützung für JAX-WS. Dies ist die Java API zum Erstellen von Web Services, siehe [jax09a]. Das umfasst unter anderem die Generierung von WSDL aus Java-Klassen und Java-Klassen aus WSDL.
- Komfortable Spring-Integration.
- Unterstützung einer Reihe von WS-Standards, z.B. SOAP, WSDL und weitere.
- Ist unter anderem im Apache Tomcat oder in Spring-basierenden Containern lauffähig.

Das **Distributed OSGi (DOSGi)** ist ein Unterprojekt von Apache CXF [dos09] und stellt eine Referenz-Implementierung des Distribution Provider der OSGi Remote Services Spezifikation (Kapitel 13 in der OSGi-Spezifikation 4,2 [osg09b]). CXF DOSGi setzt die Remote Services-Funktionalität über Web Services um.

Apache CXF hat keine eigene OSGi Framework Implementierung, sondern stellt Bundles für bestehende OSGi Frameworks bereit. Diese Bundels werden in die Frameworks installiert und diese Frameworks können dann remote Services konsumieren bzw bereitstellen. Zur Zeit unterstützen zwei OSGi Frameworks CXFs Distributed OSGi und zwar Eclipse Equinox [equ09] und Apache Felix [fel09] OSGi Container, siehe auch Kapitel 4.2.

Um die Suche nach Services zwischen verschiedenen OSGi Containern zu ermöglichen (Discovery) wird Apache Zookeeper [zoo09] verwendet. Der Name Zookeeper entstand aus der Idee, dass die Koordination von von Verteilten Systemen ein Zoo entspricht und Zookeeper versucht, diesen Zoo zu verwalten. Zookeeper wurde 2006 von Yahoo entwickelt und ist der Apache Lizenz unterstellt.

Zookeeper erlaubt verteilten Prozessen die Koordination untereinander über einen gemeinsamen hierarchischen Namensraum von Datenregistern (auch Znodes genannt), ähnlich wie bei einem Dateisystem. Im Gegensatz zu normalen Dateisystemen bietet Zookeeper einen hohen Durchsatz, niedrige Latenz, hohe Verfügbarkeit und einen streng geordneten Zugang zu den Znodes. Die Performance von Zookeeper erlaubt es, in großen verteilten Systemen verwendet zu werden. Die Zuverlässigkeit verhindert Single Point of Failure in großen Systemen. Die strenge Ordnung ermöglicht anspruchsvolle Synchronisierungsmaßnahmen auf Client Implementierungen. Zookeeper kann unter anderem für das Konfigurationsmanagement, das Server Cluster Management und für load Balancing in einer verteilten Umgebung verwendet werden. Weitere Informationen über Apache Zookeeper können unter [zoo09] gefunden werden.

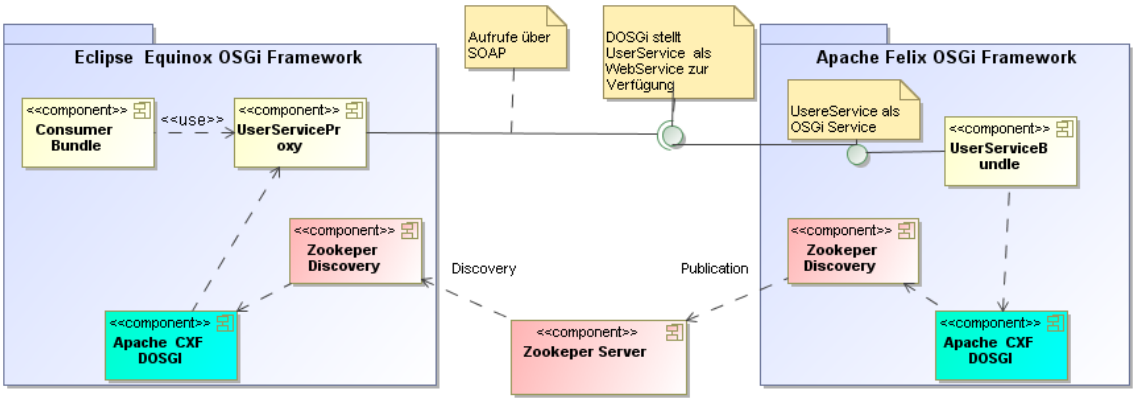

Abbildung 4.12: Apache CXF Distributed OSGi Registry

VT1 Verteilte Registry - Die verteilte Registry in CXF DOSGi wird anhand des UserService Beispiels wie in Abbildung 4.12 dargestellt. Lokal, innerhalb von Apache Felix, wird der UserServiceBundle als OSGi Service publiziert. Falls der UserService so konfiguriert wurde, dass ihn Services aus anderen Containern aufrufen können, stellt CXF DOSGi den UserService als Webservice Endpunkt bereit. Dieser Endpunkt kann die Anfragen über SOAP vom Client annehmen und verwandelt sie zur einem Java Aufruf. Dadurch wird der entsprechende Service aufgerufen.

Das Finden von Services wird mit Hilfe von Apache Zookeeper ermöglicht. UserService wird als ein Znode beim Starten in Apache Zookeeper registriert und gespeichert. Der Client (EditUserData) kann dann nach UserService Endpunkten im Apache Zookeeper suchen und bekommt einen UserServiceProxy des UserServices geliefert. Dieser Proxy nimmt die Aufrufe

des Consumer Bundels so entgegen als würde dieser Service lokal im Equinox Container laufen. Der Proxy konvertiert die Aufrufe in SOAP und schickt sie an den Container, wo der eigentliche Service läuft. Weitere Informationen über SOAP können unter [soa09] gefunden werden.

VT2 Zugriff über ORB - Durch die Kommunikation über SOAP sind Zugriffe über ORB möglich. Die Java Objekte werden in SOAP Nachrichten von CXF DOSGi an der Client Seite konvertiert und an der Serverseite von SOAP wieder in Java Objekte umgewandelt. Im Listing 4.4 wird das User Objekt, das der Java Klasse aus Listing 5.1 entspricht, dargestellt. Die Umwandlung der Java Objekte in XML wird von DOSGi übernommen. Der Entwickler muss sich nicht darum kümmern.

Listing 4.4: User Objekt als SOAP dargestellt

```
<xsd:complexType name="User">
<xsd:sequence>
<xsd:element minOccurs="0" name="birthday" nillable="true" type="xsd:string"/
    >
<xsd:element minOccurs="0" name="city" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="country" type="xsd:int"/>
<xsd:element minOccurs="0" name="email" nillable="true" type="xsd:string"/>
<xsd:element minOccurs="0" name="gender" type="xsd:int"/>
<xsd:element minOccurs="0" name="userId" type="xsd:long"/>
<xsd:element minOccurs="0" name="userName" nillable="true" type="xsd:string"/
    >
<xsd:element minOccurs="0" name="zip" nillable="true" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
```

VT3 Zugriff über Webservice - Wie schon beim Kriterium VT1 beschrieben, werden die Service Endpunkte als WebServices bereitgestellt und über SOAP ansprechbar. Wenn der UserService (siehe Beispiel in Listing 5.3 mit Hilfe von CXF Distributed OSGi zur Distribution freigegeben wird erstellt CXF automatisch den Service Endpunkt. Dieser wird aus dem zu realisierenden Interface erstellt. Das WSDL zum UserService sieht wie in Abbildung 4.13 aus.

Ab der CXF DOSGi Version 1.1 (Dezember 2009) können Java-Schnittstellen als REST JAX-RS Webservices den Clients bereitgestellt werden. Der Begriff Representational State Transfer (REST) bezeichnet einen Softwarearchitektur-Stil für verteilte Hypermedia - Informationssysteme wie das World Wide Web. Dessen Architektur kann durch den URI-Standard und HTTP beschrieben werden, wogegen der REST-Architekturstil nahe legt, jede Ressource mit einer eigenen URI anzusprechen. Der Begriff wird auch im weiteren Sinne verwendet, um grundsätzlich einfache Schnittstellen zu kennzeichnen, die Daten via HTTP übertragen, ohne etwa eine zusätzliche Transportschicht wie SOAP oder Sitzungsverwaltung via Cookies einzusetzen. Weitere Informationen und Literaturhinweise zu REST können auf der Webseite [RES09] gefunden werden.

Java API for RESTful Web Services (JAX-RS) ist eine Java Bibliothek, um Web Services auf Representational State Transfer (REST) - Basis entwickeln zu können. Wie andere Java EE-APIs, benutzt auch JAX-RS Annotationen, um die Entwicklung und das Deployment von Webservice-Clients und Service-Endpunkten zu vereinfachen. Die vollständige Spezifikation kann [jax09b] entnommen werden.

VT4 Asynchrone Kommunikation (JMS) - Derzeit existiert kein JMS Support in DOSGi. Eine alternative wäre die Implementierung von ApacheMQ [act09]. ActiveMQ stellt

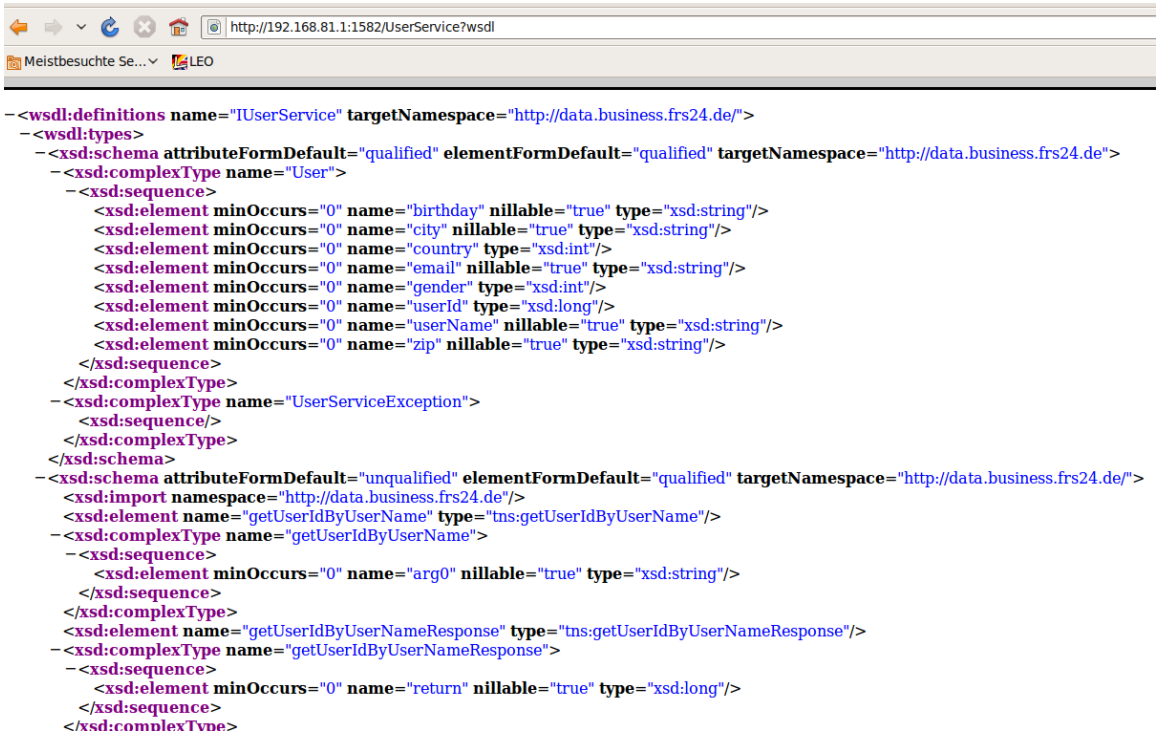

Abbildung 4.13: WSDL des UserService Endpunkts mit Hilfe von Apache CXF

OSGi Bundels bereit, die im OSGi Container installiert werden. Die Distribution und Discovery wird von DOSGi nicht unterstützt. Sie müsste für jeden Client und Server definiert werden.

DV Versionierung - Die Versionierung innerhalb der OSGi Container wird vollständig unterstützt. Das Beispiel aus Abbildung 4.8 wird auch innerhalb der einzelnen Containern unterstützt. DOSGi hat die OSGi Versionierung derzeit nicht in Zookeeper integriert, allerdings unterstützt Zookeeper eine eigene Versionierung der Znodes.

MT1 Ausfallbehandlung - Innerhalb vom Zookeeper ist es erlaubt, mehrere Service Endpoints (Znodes), die den gleichen Service bereitstellen, zu halten. Zookeeper teilt diesen Endpunkten eine unique ID mit, um diese zu unterscheiden. Wenn ein Service ausfällt, entfernt der Zookeeper den ServiceProxy vom Client (Service Consumer). Der Service Consumer fängt die Fehlermeldung ab und fragt den Zookeeper Server nach anderen passenden Services.

Listing 4.5: Znodes von zwei registrierten UserService in Apache Zookeeper Console

```
1 [zk: localhost:2181(CONNECTED) 35] ls /osgi/service_registry/de/frs24/
      business/data/IUserService
2 [r021048#1582##UserService, r021048#2138##UserService]
3 [zk: localhost:2181(CONNECTED) 36]
```

Um den Fehlerfall abzufangen, dass ein Zookeeper Server ausfällt und somit die ganze verteilte Registry Informationen verloren gehen, kann Zookeeper Server in Clustern betrieben werden. Wie die verteilten Prozesse koordiniert Zookeeper die Replikation über eine Gruppe von Hosts, siehe Abbildung 4.14. Die Server, die den Zookeeper Cluster bilden, müssen alle voneinander wissen. Sie halten alle Daten über Znodes und Events im Hauptspeicher

und halten Schnappschüsse in einem persistenten Speicher. Die Daten zwischen den Servern werden bei Änderungen ab-geglichen [zoo09].

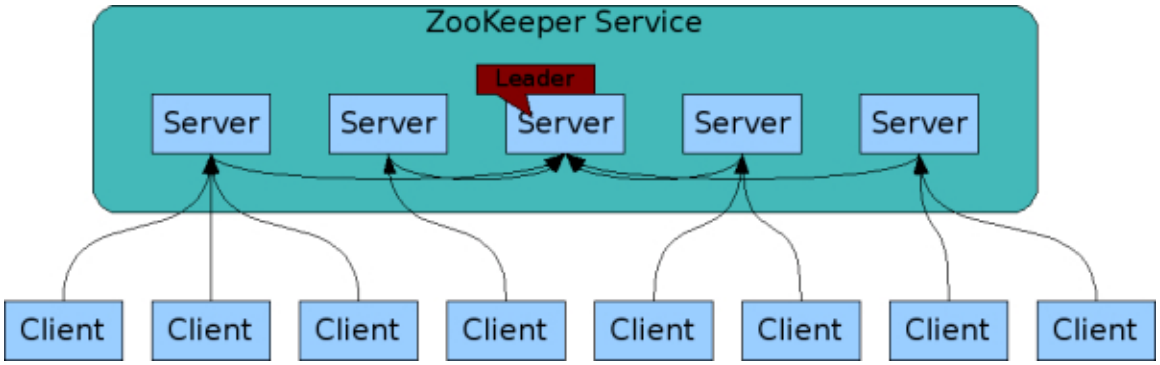

Abbildung 4.14: Replikation von Zookeeper Servern [zoo09]

MT2 Last - Abbildung 4.15 zeigt das Daten Modell und den hierarchischen Namensraum in Apache Zookeeper, welcher dem Standard File System sehr ähnelt. Zookeeper könnte die Anfragen an die Services so verteilen, dass sie anhand der UserId den jeweiligen Service-Proxy zurückgeben. Anhand der Abbildung könnten beispielsweise alle User mit der UserId Endziffer 1 den Service "/app/p_1" bekommen und User die mit der Endziffer 2 den "/app/p_2" und so weiter. Während der Laufzeit wäre es möglich, bei Engpässen einfach neue Service Instanzen zu starten. Die Auslastung der einzelnen Services kann derzeit nicht angezeigt werden.

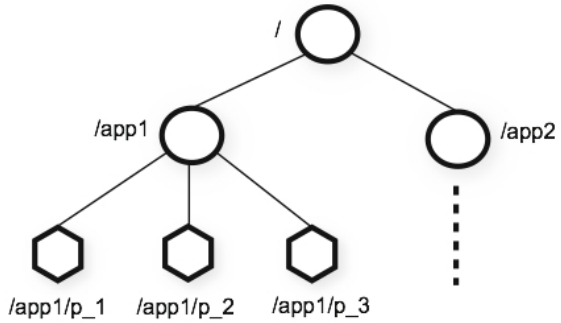

Abbildung 4.15: Daten Modell und hierarchischen Namensraum in Apache Zookeeper [zoo09]

DR1 Zentrale GUI - Eine grafische Oberfläche gibt es in Zookeeper, Apache Felix und auch in Eclipse Equinox nicht.

Zu jeden Zookeeper Server kann eine Verbindung über den Zookeeper Client hergestellt werden, der eine Console zur Bearbeitung der Znodes zur Verfügung stellt. Die Abbildung 4.16 zeigt wie, sich in der Zookeeper Registry die Details des Services anzeigen lassen. Es existieren viele weitere Funktionen in der Zookeeper Console, wie u.a. das Entfernen und Anzeigen von Ordnerinhalten. Über Zookeeper ist es nicht möglich den Status der Services zu ändern, dazu muss man sich lokal in den Container (Felix oder Equinox) einloggen. Dort kann man neue Services hinzufügen, starten, stoppen usw.

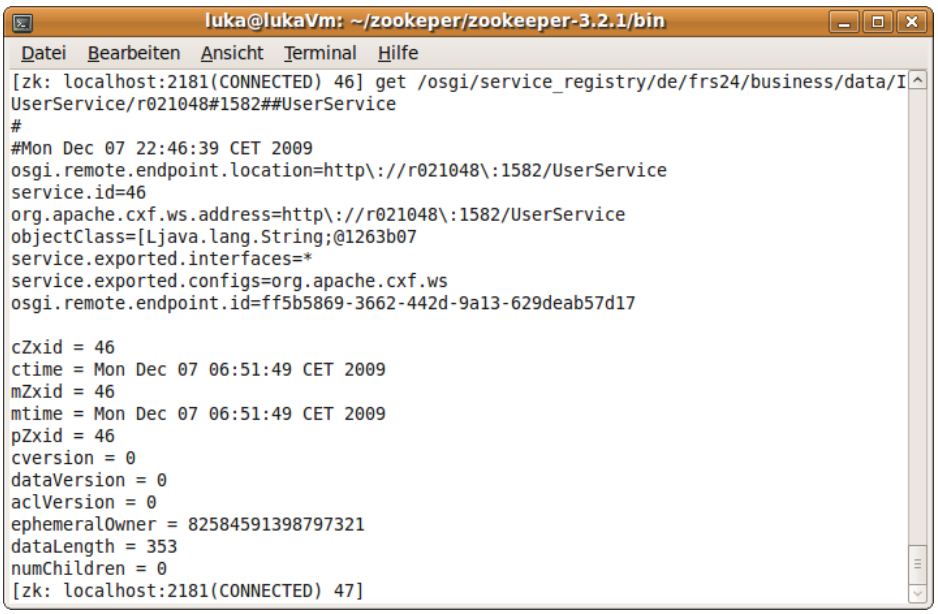

Abbildung 4.16: Anzeige der UserServices Znodes in Zookeeper Console

DR2 Deployment zur Laufzeit - Services können innerhalb von Felix oder Equinox gestartet, gestoppt oder gelöscht werden ohne andere Services zu beeinflussen. Wie schon beim Kriterium MT1 beschrieben, können auch im Zookeeper mehrere gleiche Services registriert sein, um Ausfallsicherheit zu gewährleisten.

DR3 Authentifizierung der Admins - Für Felix und Equinox existiert derzeit keine Authentifizierung für Administratoren. Für den Zookeeper existiert ein Authentifizierungsmechanismus mit verschiedenen Rollen. Die Authentifizierung wird über den Zookeeper Client realisiert.

Außerdem existiert in Zookeeper ein Authentifizierungsmechanismus für einzelne Znodes. Die Authentifizierung wird mit Hilfe von Access Control Lists (ACL) realisiert, die für jeden Znode definiert werden können. Die Prüfung kann dann über in Zookeeper integrierte Authentifizierungsmechanismen geschehen, wie z.B. dass nur Clients mit bestimmten IP-Adressen auf einen Znode zugreifen können. Es gibt die Möglichkeit, für Zookeeper eigene Authentifizierungsmechanismen zu entwickeln und zu integrieren.

ST1 Verbreitung - Apache Felix (seit 2007) und Equinox (seit 2003) sind als OSGi Container sehr weit verbreitet. Die Eclipse Entwicklungsumgebung basiert auf Equinox. Die Weiterentwicklung dieser beiden Frameworks ist gewährleistet. Apache Zookeeper ist schon länger existent und wird bei einigen großen Unternehmen wie Yahoo im produktiven Bereich eingesetzt. CXF DOSGi ist erst seit April 2009 in der Version 1.0 herausgekommen. Im letzten halben Jahr hat es sich verbreitet und wird auch ständig weiterentwickelt.

ST2 Standard Komponenten - In CXF DOSGi werden Bundles, die nach dem OSGi 4.2 Standard entwickelt sind, vollständig Unterstützt. Um die Services die die Bundles bereitstellen über die Container grenzen hinweg zugänglich zu machen sind nur kleine Änderungen in den Konfigurationsdateien nötig. Das Framework unterstützt auch Webservice Standards, die in VT3 beschrieben sind.

ST3 Standard Zugriff - Die Anbindung von WebServices bietet viele Möglichkeiten, die OSGi Services außerhalb vom OSGi Frameworks zu benutzen. Leider ist die Anbindung per JMS in CXF DOSGi noch nicht realisiert. Da das reine CXF Framework jedoch JMS unterstützt, wird es auch bei DOSGi nicht lange dauern.

BR1 Nutzung Entwickler - Für die Entwickler wäre der Nutzen nach der Einarbeitung sehr hoch. Durch die Dynamik und die lose Kopplung von DOSGi könnten einzelne Komponenten schnell und unabhängig voneinander entwickelt werden.

BR2 Schulungsaufwände - Die Schulungsaufwände sind nicht zu vernachlässigen. Da aber Technologien wie Equinox und Felix weit verbreitet sind könnten einige Entwickler Wissen mitbringen. Der Unterschied zwischen DOSGi und OSGi ist meistens nur in der Konfiguration der Webservice Anbindung. Daneben ist die Einarbeitung in Apache Zookeeper nicht zu vernachlässigen, da dies einen mächtiges Tool ist.

BR3 Migrationsaufwand - Der Migrationsaufwand ist sehr hoch, da alle Services als OSGi Komponenten umgeschrieben werden müssen. Die Konfiguration, die die Verteilung der OSGi Komponenten ermöglicht, ist nicht zu vernachlässigen. Dazu kommt die Konfiguration von Apache Zookeeper, die den Zugriff und die Replikation der Zookeeper Server beinhaltet.

SK Skalierbarkeit - Durch die Unabhängigkeit der Komponenten ist die Skalierung bei DOSGi hochwertig, da Komponenten unabhängig voneinander zum ganzen System hinzugefügt und entfernt werden können.

PE Performance - Apache Zookeeper ist ein sehr hochperformantes Tool. Die Performance wird anhand der Abbildung 4.17 gezeigt. Das Tool könnte man dadurch nicht nur für die Discovery benutzen, sondern z.B. auch für die Konfiguration von Services.

Beim Evaluieren fiel auf, dass die Container Felix und Equinox bei der Installation von DOSGi Bundels einige Zeit brauchen bis sie alles initialisiert haben. Die Anfragen an die

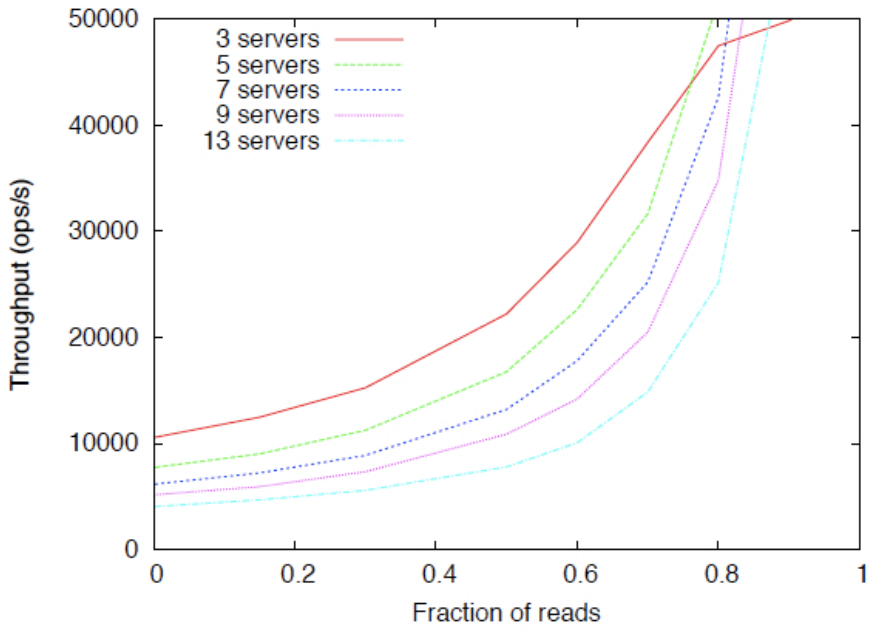

Abbildung 4.17: Performance von Zookeeper Servern bei Requests pro Sekunde bezogen auf das Verhältnis der Lese- und Schreibvorgängen [zoo09]

Zookeeper Registry werden schnell abgearbeitet, was den Zahlen aus Abbildung 4.17 zu entnehmen ist.

Lizenzkosten Bei diesen Komponenten fallen keine Kosten an, da alle Open Source sind. Die Community ist stetig wachsend, da dynamische Systeme immer mehr nachgefragt werden.

4.5.1 Fazit

CXF DOSGi bietet viele Möglichkeiten und Hilfen bei der Entwicklung von dynamischen, verteilten Systemen. Dieses Framework besteht aus Komponenten, die auf ihren Gebieten etabliert und weit verbreitet sind. Die OSGi Container bieten eine stabile Umgebung für die OSGi Komponenten innerhalb einer JVM. Apache Zookeeper bietet ein etabliertes System für die Kommunikation zwischen Prozessen in einer verteilten Umgebung. CXF ist eine Komponente die für die Anbindung der Services über Webservice oder JMS bereitstellen kann. Leider ist das Framework noch nicht vollständig ausgereift und noch nicht für den produktiven Einsatz geeignet. Dies ist nicht erstaunlich, da dieses Framework erst seit einen halben Jahr auf dem Markt ist. DOSGi lehnt sich an den OSGi 4.2 Standard an. Seit Dezember 2009 bietet CXF DOSGi eine verteilte Registry mit Hilfe von Apache Zookeeper, diese ermöglicht die Discovery des OSGi 4.2 Standards. Es ist nur eine Frage von ein paar Monaten bis dieses Framework ausgereift und für die Entwicklung großer Systeme bereit ist.

4.6 Vergleich der Frameworks

Die ersten zwei Frameworks Newton und Paremus Service Fabric hängen sehr stark voneinander ab. Im Prinzip bietet Service Fabric nur Erweiterungen für Newton an. Wenn man die beiden vergleicht, ist Service Fabric deutlich im Vorteil, da er im Kriterienkatalog /siehe Tabelle 4.1) deutlich mehr Anforderungen wie Ausfallsicherheit, Authentifizierung der Administratoren, usw. erfüllt. Newton vermittelt den Eindruck, als ob dieses Framework nur da sei um die Verteilung von OSGi Services einfacher zu realisieren, ohne Augenmerkmale auf die Sicherheit und Robustheit des Frameworks zu legen. Somit ist Newton nicht ein geeignetes Framework um ein System, wie es von FRS24 verlangt ist, zu realisieren. Für Anwendungen bei denen die Hochverfügbarkeit keine Rolle spielt, ist dieses Framework durchaus zu empfehlen. Bei der Evaluierung wurde der Eindruck geweckt, dass Newton nur dafür da ist das Paremus Service Fabric zu vermarkten und zu verkaufen. Für hoch-verfügbare Systeme, wie bei FRS24, ist ein Framework wie das von Paremus Services Fabric durchaus zu empfehlen. Der große Nachteil von Paremus Service Fabric ist die Dokumentation und die Tatsache, dass es sich um ein kommerzielles Framework handelt. Bei der Evaluierung wäre eine bessere und größere Dokumentation von Paremus Service Fabric wünschenswert gewesen.

Bei Apache CXF handelt sich um ein Framework, das aus einzelnen Komponenten besteht und zwar aus OSGi Containern Felix oder Equinox, ECF DOSGi und Apache Zookeeper. Diese Komponenten sind auf ihren Teilgebieten etabliert und bieten viele Möglichkeiten an. Der große Vorteil von Paremus Service Fabric verglichen mit Apache CXF ist, dass Paremus Service für den produktiven Einsatz sofort geeignet ist. Bei Service Fabric wird dies noch eine Weile dauern. Der Vorteil von Apache CXF liegt in der Offenheit gegenüber der Anbindung von anderen Systemen und Plattformen. Es existieren bei CXF DOSGi viel mehr Blog - Einträge, Hilfe Seiten und Foren wo Hilfe zu finden als in Newton. Die Community ist bei CXF DOSGi nicht vergleichbar im Gegensatz zu der Community von Newton und

Paremus Service Fabric. Im Kriterienkatalog schneidet CXF DOSGi besser ab als Paremus Service Fabric und alles noch wegen der Offenheit gegenüber anderen Protokollen, z.B. wie Webservice. Aber im Prinzip sind die beiden Frameworks mindestens ebenbürtig was die Funktionalität angeht, aber der großer Vorteil vom Apache CXF DOSGi ist dass es Open Source ist.

Kriterien	**Wichtigkeit**	**Testaufwand**	**Newton**	**Service Fabric**	**CXF DOSGi**
VT1 Verteilte Registry	5	mittel	++	++	++
VT2 Zugriff über ORB	3	leicht	++	++	0
VT3 Zugriff über Webservice	4	mittel	-	-	++
VT4 Asynchrone Komm (JMS)	3	mittel	- -	- -	-
DV Dynamische Versionierung	5	leicht	++	++	+
MT1 Monitoring Ausfall.	4	hoch	0	++	+
MT2 Monitoring Last	3	hoch	-	+	-
DR1 Zentrale GUI	3	leicht	- -	++	-
DR2 Deployment zur Laufzeit	5	leicht	++	++	++
DR3 Authentifizierung der Admins	3	mittel	- -	++	++
ST1 Verbreitung	5	mittel	-	0	+
ST2 Standard Komponenten	5	mittel	0	0	++
ST3 Standard Zugriff	4	mittel	0	0	++
BR1 Nutzung - Entwickler	4	mittel	0	++	+
BR2 Schulungsaufwände	4	mittel	0	0	+
BR3 Migrationsaufwand	3	mittel	- -	0	-
BR4 Lizenzkosten	3	leicht	++	-	++
SK Skalierbarkeit	5	hoch	+	++	+
PE Performance	4	hoch	+	+	++
SUMME	w_i		157	218	225

Tabelle 4.1: Kriterienkatalog Überblick

5 Implementierung

Eines der wichtigsten Ziele von FRS24 ist es, dass die Komponenten (Bundles) so weit wie möglich an den OSGi Standard angelehnt sind, um die Unabhängigkeit zwischen Frameworks zu gewährleisten und somit die Möglichkeit für die FRS24 Plattform zu bieten, schnell OSGi Frameworks auszutauschen. Dann könnten die Komponenten mit minimalem Konfigurationsaufwand ausgetauscht werden. Um diese Anforderung bei den Frameworks zu untersuchen, wird in diesem Kapitel zuerst ein Implementierungsbeispiel vorgestellt, dass sich an OSGi 4.2 Standard anlehnt. Danach werden die Unterschiede zu den Implementierungen in den jeweiligen Frameworks verglichen. Am Ende des Kapitels wird auf die zukünftigen Lösungen eingegangen und eines der Frameworks für die Implementierung vorgeschlagen.

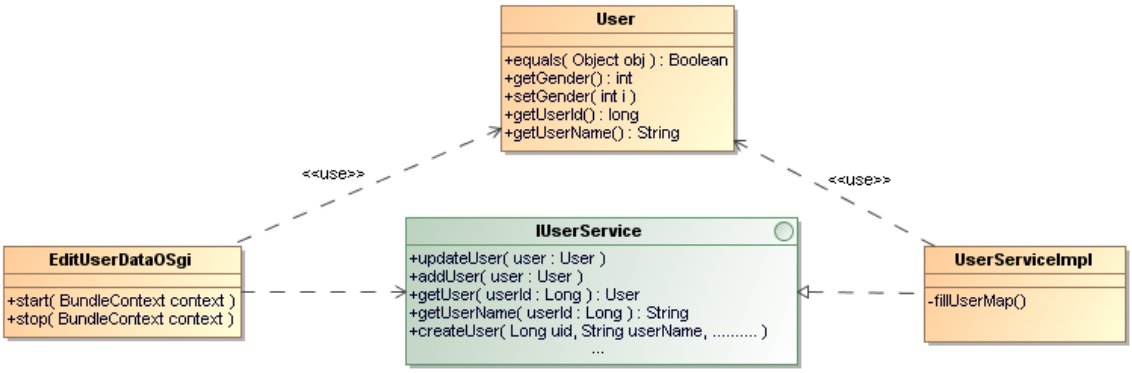

Abbildung 5.1: UML Klassendiagramm der UserService Implementierung

5.1 Distributed OSGi Implementierungsbeispiel

Im folgenden Abschnitt wird eine vereinfachte UserService Implementierung des FRS24 Systems nach dem OSGi 4.2 Standard vorgestellt. Die Entscheidung für die Implementierung den UserService zu nehmen beruht darauf, weil UserService einer der wichtigsten Services der FRS24 Plattform ist. Dieser Service wird von sehr vielen Komponenten des FRS24 Systems benutzt, z.B. der Anzeige der User, Bearbeiten der Userdaten, Suchindex, usw. Somit müsste bei einer Migration UserService als erstes migriert werden.

In der Abbildung 5.1 wird die vereinfachte UserService Implementierung angezeigt. EditUserDataOSGi repräsentiert den Client, IUserService das Interface, welches den UserService darstellt. UserServiceImpl realisiert das Interface und stellt Informationen von Usern bereit.

Listing 5.1 stellt ein Teil des User Objektes aus der Abbildung 5.1 dar, das Objekt ist eigentlich eine einfache JavaBean. Weitere Informationen über JavaBeans können unter [Ull09, Kapitel 7.4] gefunden werden. Einige Getter- und Setter-Methoden wurden im Listing 5.1 ausgelassen. Die User Klasse implementiert das Interface Serializable, um den Transfer von

diesen Objekt zwischen verschiedenen JVMs zu ermöglichen. Somit könnten die Informationen für die Benutzer der FRS24 Plattform in der Webseite dargestellt werden.

Listing 5.1: Java Klasse, die das vereinfachte Objekt des Users im FRS24 System repräsentiert

```
1  package de.frs24.business.data;
2  import java.io.Serializable;
3  public class User implements Serializable {
4    private static final long serialVersionUID = -5518041606867706319L;
5    private Long userId;
6    private String userName;
7    private String email;
8    private int gender;
9    private int country;
10   private String city;
11   private String zip;
12   private String birthday;
13
14   public User(long anId) {
15     this.userId = anId;
16   }
17   public int getGender() {
18     return this.gender;
19   }
20   public void setGender(int i) {
21     this.gender = i;
22   }
23   public long getUserId() {
24     return this.userId;
25   }
26   public String getUserName() {
27     if (this.userName == null) return "";
28     return this.userName;
29   }
30   public void setUserName(String string) {
31     this.userName = string;
32   }
33   //... weitere getter und setter Methoden anderer Atribute
34   @Override
35   public boolean equals(Object obj) {
36     //User vergleich logik...
37   }
38 }
```

Das Listing 5.2 stellt das Interface des UserServices da. Der UserService bietet Funktionen zum Erstellen von neuen Usern (`createUser(..)`), zum Heraussuchen von User Objekten anhand der Pseudonyme oder UserIds (`getUser(Long userId)`, `getUserName(String userName)`) und zum Hinzufügen und Verändern von User Objekten (`addUser(...)`, `updateUser(...)`).

Listing 5.2: Java Interface, welches den vereinfachten UserService repräsentiert

```
1  package de.frs24.business.data;
2
3  public interface IUserService {
4
5    public User createUser(Long uid, String userName, String zip,
6        int gender, String email, String city, String birthday);
```

```
7
8   public User getUser(Long userId) throws UserServiceException;
9
10  public void updateUser(User user) throws UserServiceException;
11
12  public String getUserName(Long userId) throws UserServiceException;
13
14  public void addUser(User user) throws UserServiceException;
15
16 }
```

Im Listing 5.3 ist der vereinfachte Code der eigentlichen UserService Realisierung dargestellt. Zur Vereinfachung werden die User zu Testzwecken in einer `HashMap userMap` gespeichert, auf die die UserService Clients zugreifen können. Der UserServiceImpl ist ein einfaches POJO.

POJO ist die Abkürzung für "Plain Old Java Object". Der Ausdruck wurde von Martin Fowler, Rebecca Parson und Josh Mackenzie mit der Absicht geprägt, einfache Java Objekte von Objekten mit vielfältigen externen Abhängigkeiten unterscheiden zu können. Solche externen Abhängigkeiten können z.B. zwingend zu implementierende Interfaces, einzuhaltende Namenskonventionen oder notwendige Annotationen sein. Weitere Informationen können bei Wikipedia nachgelesen [poj09] werden. Hier kann man noch beobachten, dass weder in der Beschreibung des Interfaces, noch in deren Implementierung eine Abhängigkeit zu einem Kommunikationsprotokoll besteht. Die Angabe von Servernamen oder Ports wird auch nicht direkt im Java Code definiert. Die Services können dann lokal auf den Entwickler Rechnern entwickelt und mit Unit-Tests getestet werden, ohne sich um die Konfiguration kümmern zu müssen.

Listing 5.3: Die vereinfachte UserService Implementierung im FRS24 System

```
1  package de.frs24.business.server;
2  // imports weggellassen
3  public class UserServiceImpl implements IUserService {
4
5    private HashMap<Long, User> userMap;
6
7    public UserServiceImpl() {
8      userMap = new HashMap<Long, User>();
9      fillUserMap();
10   }
11
12   private void fillUserMap() {
13     userMap.put(new Long(1), createUser(new Long(1), "user1",
14         "12345", 13001, "user1@frs.de", "munich", "01.12.1980"));
15     userMap.put(new Long(2), createUser(new Long(2), "user2",
16         "12345", 13001, "user2@frs.de", "munich", "02.12.1980"));
17     userMap.put(new Long(3), createUser(new Long(3), "user3",
18         "12345", 13000, "user3@frs.de", "munich", "03.12.1980"));
19     userMap.put(new Long(4), createUser(new Long(4), "user4",
20         "12345", 13000, "user4@frs.de", "munich", "04.12.1980"));
21     userMap.put(new Long(5), createUser(new Long(5), "user5",
22         "12345", 13000, "user5@frs.de", "munich", "05.12.1980"));
23   }
24
25   @Override
26   public User createUser(Long uid, String userName, String zip,
```

```
27        int gender, String email, String city, String birthday) {
28      User user = new User(uid);
29      user.setUserName(userName);
30      user.setZip(zip);
31      user.setGender(gender);
32      user.setEmail(email);
33      user.setCity(city);
34      user.setBirthday(birthday);
35      return user;
36    }
37
38    @Override
39    public User getUser(Long userId) throws UserServiceException {
40      if (userId.equals(null)) {
41        throw new UserServiceException("UserId is null ");
42      }
43      User usr = userMap.get(userId);
44      if (usr == null) {
45        throw new UserServiceException("No user with UserId " + userId);
46      }
47      return usr;
48    }
49
50    @Override
51    public String getUserName(Long userId) throws UserServiceException {
52      return getUser(userId).getUserName();
53    }
54
55    @Override
56    public void updateUser(User user) throws UserServiceException {
57      userMap.put(user.getUserId(), user);
58    }
59  }
```

Zum Starten des Bundles, das den UserService beinhaltet, gibt es mehrere Möglichkeiten, Bundleaktivierungs und -deaktivierungslogik zu definieren. Eine Möglichkeit bietet die Nutzung des Blueprint Services der OSGi 4.2 Spezifikation [osg09b]. Der Blueprint Service ist aus dem Spring-Dynamic-Modules-Projekt hervorgegangen und definiert den sogenannten Blueprint Container, der XML-Definitionen innerhalb der Bundles benutzt, um Anwendungsobjekte über ein Dependency-Injection-Framework zu erzeugen und zu verknüpfen. Dabei kommt insbesondere die potenzielle Dynamik der (OSGi) Services zum Vorschein. Durch die Verwendung des Blueprint Services können die zu entwickelnden Anwendungs-Bundles frei von OSGi-spezifischem Code gehalten werden, was die Verwendbarkeit in Nicht-OSGi-Anwendungen und die Testbarkeit der Bundles deutlich verbessert. Für die Anwendungen von FRS24 würden die Abhängigkeiten zwischen den Services nur an einer in den Blueprint XML Dateien vorhanden sein, nicht wie jetzt in XML Dateien als auch in Java-Code.

Listing 5.4 veranschaulicht die Nutzung des Blueprint Services. Diese Konfigurationsdatei muss in den Ordner `OSGI_INF/blueprint` innerhalb des Bundles abgelegt werden, damit das OSGi Framework sie interpretiert. In diesem Beispiel erzeugt der Blueprint Service eine Instanz der `UserServiceImpl` - Klasse und registriert sie als Service mit der Schnittstelle `IUserService`.

Erst bei der Registrierung des Dienstes kommt das Distributed OSGi zum Vorschein. Das Attribut `service.exported.interface` definiert, ob die festgelegten Services auch remote,

also über die Grenzen des Containers hinaus, verfügbar sein sollen. Falls ''*'' angegeben wird, heißt dies, dass diese Schnittstelle auch remote verfügbar sein soll. Wenn ein Service mit diesem Attribut registriert wird, steht der Distribution Provider (Abbildung 4.2) auch remote für Clients in anderen Containern zur Verfügung. Diese Eigenschaft wird von dem Kriterium ST2 verlangt, um Abhängigkeit nur zu einem Framework zu minimieren.

Listing 5.4: Definition des zu registrierenden Services als OSGi 4.2 Blueprint Standard

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd">

  <!-- create the bean -->
  <bean id="UserService" class="de.frs24.business.server.UserServiceImpl" />

  <!-- export the bean on the service registry -->
  <service ref="UserService" interface="de.frs24.business.data.IUserService"
      >
   <service-properties>
      <entry key="service.exported.interface" value="*"/>
   </service-properties>
  </service>

</blueprint>
```

Das Listing 5.5 zeigt, dass der Client nicht mitbekommt, ob er den `IUserService` lokal, also innerhalb eines Containers, oder remote nutzt. Die Client Seite bekommt von den Distributed OSGi nichts mit. `EditUserData` ist ein ganz einfacher Client, der nach Usern mit den IDs 1 und 2 sucht und sie auf der Console ausgibt. In Listing 5.5 ist die OSGi Bundle Aktivierung im Quellcode der Java Klasse gelöst.

Listing 5.5: Einfache UserService Client Implementierung

```

public class EditUserOSGi implements BundleActivator {
  IUserService userService = null;

  @Override
  public void start(BundleContext context) throws Exception {
    ServiceReference serviceRef = context
        .getServiceReference(IUserService.class.getName());
    userService = (IUserService) context.getService(serviceRef);
    try {
      System.out.println("User Name User 1: "
          + this.userService.getUserName(new Long(1)));
      User usr = this.userService.getUser(new Long(2));
      System.out.println("**USER 2:  " + usr.toString() + "***");
    } catch (UserServiceException e) {
      System.err.println(e.getMessage());
    }

  }

  @Override
```

```
22   public void stop(BundleContext context) throws Exception {
23     userService = null;
24
25   }
26 }
```

5.2 Implementierungsbeispiel in Newton und Paremus Service Fabric

Das Beispiel aus dem vorigen Abschnitt 5.1 kann nicht einfach in Newton installiert und gestartet werden. Da diese beiden Frameworks schon vor der Verabschiedung des OSGi 4.2 Standards eine verteilte Registry bereitgestellt haben, konnten sich diese bei der Implementierung nicht an den Standard halten. In diesem Abschnitt wird kurz beschrieben, welche Änderungen an der OSGi 4.2 Standard Implementierung vorgenommen werden müssen, damit es in Newton implementiert werden kann.

In Newton werden Komponenten nicht nach dem dem OSGi 4.2 Standard, sondern nach dem SCA (siehe [oso09]) Standard beschrieben. Der UserServiceImpl Code aus Listing 5.3 bleibt identisch, aber die Bundle Aktivierungslogik ist die aus Listing 5.6. Um diese SCA Datei vom Newton Framework zu interpretieren wird in der OSGi Manifest Datei 4.1 auf die Newton Composite Datei verwiesen, in der dritten Zeile steht dann folgendes: `Installable-Component-Templates:META-INF/newton/UserService.composite`. Beim Installieren der Komponente in Newton muss die UserService.composite Datei im Order "META-INF/newton/UserService.composite" des UserService Jars liegen.

Listing 5.6: UserService.composite - SCA Definition für Newton

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <composite name="UserService">
3  <description>Created by lleovac</description>
4  <bundle.root bundle="UserService" version="1.0.0"/>
5
6    <service name="interface">
7    <interface.java interface="de.frs24.business.data.IUserService"/>
8      <binding.rmi/>
9    </service>
10
11   <component name="implementation">
12   <implementation.java.callback impl="de.frs24.business.server.
         UserServiceImpl"/>
13   </component>
14
15   <wire>
16   <source.uri>interface</source.uri>
17     <target.uri>implementation</target.uri>
18   </wire>
19
20 </composite>
```

Wenn man die Bundle Konfigurationsdateien miteinander vergleicht (Listing 5.6 und Listing 5.4), stellt man fest, dass folgende Elemente im Wesentlichen das Gleiche beschreiben:

- `<blueprint>` (OSGi 4.2) - `<composite>` (SCA)

- `<bean>` (OSGi 4.2) - `<component>` (SCA)
- `<service>` ist bei OSGi 4.2 und SCA gleich
- `<wire>` existiert nur bei SCA Standard und gibt die Verknüpfung an, welcher Service von welche Implementierung realisiert wird. Bei OSGi wird die Verbindung so erreicht, dass die ID der `<bean>` im `<service>` Element referenziert wird.
- `<binding.rmi/>` ist äquivalent zum `<entry key="service.exported.interface"value="*"/>` Element in OSGi. Wie bei OSGi 4.2 merkt die Java Implementierung des Services nicht, ob er nur innerhalb eines Containers oder remote zur Verfügung gestellt wird.

Der Unterschied zwischen Newton und dem OSGi 4.2 Standard liegt nur in den XML Konfigurationsdateien. Die Service Implementierungen können sowohl in Newton als auch in OSGi 4.2 Standard Framework Implementierungen eingesetzt werden.

5.2.1 Deployment zur Laufzeit

Um das Kriterium "Deployment zur Laufzeit" (DR2) zu verifizieren, wurde das obige Beispiel folgendermaßen angepasst: UserService (Version 1.0.0) aus Listing 5.3 wurde mit entsprechender composite Datei aus Listing 5.6 ins Newton installiert. Der Client ist im Listing 5.7 dargestellt. Der Client holt sich den UserNamen vom User mit der ID 1 und gibt den Namen auf der Console aus. Das Objekt vom User 2 wird alle 30 Sekunden vom UserService geholt und die toString() Methode des User Objekts wird auf der Console ausgegeben (siehe Abbildung 4.10).

Listing 5.7: Client des UserService aus Listing 5.3

```
1  package de.frs24.dating;
2
3  import de.frs24.business.data.IUserService;
4  import de.frs24.business.data.User;
5  import de.frs24.business.data.UserServiceException;
6
7  public class EditUserData {
8
9    IUserService userService = null;
10
11   public void addServiceReference(IUserService userService) {
12     this.userService = userService;
13
14     try {
15       System.out.println(" Pseudonym User 1: "
16           + this.userService.getUserName(new Long(1)));
17       while(true){
18         User usr = this.userService.getUser(new Long(2));
19         System.out.println("\n USER 2:  " + usr.toString());
20         Thread.sleep(30000);
21       }
22     } catch (UserServiceException e) {
23       System.err.println(e.getMessage());
24     } catch (InterruptedException e) {
25       e.printStackTrace();
26     }
```

```
27   }
28
29   public void removeServiceReference(IUserService userService) {
30     this.userService = null;
31   }
32
33 }
```

Die Methode fillUsermap() in UserServiceImpl.java wurde wie in Listing 5.8 verändert und UserService wurde die Versionsnummer 1.0.1 zugewiesen. Diese Version wurde in Newton installiert und gestartet. Wenn man sich die installierten Bundles in Newton anzeigen lässt, sieht man die ersten vier Zeilen aus Abbildung 4.10. Hier Stellt man fest, dass beide Versionen (1.0.0 und 1.0.1) vom UserService installiert sind. Die Consolenausgabe "USER 2: user2 [2]..." zeigt, dass der Client den UserService in Version 1.0.0 benutzt. Dann wird mit dem Befehl: "stop 113" in der Console das Bundle User Service Version 1.0.0 gestoppt und stellt fest, dass die Consolenausgabe vom UserService Version 1.0.1 herausgeschrieben wird. Damit wurde gezeigt, dass ohne Eingreifen von außerhalb nach anderen UserServices gesucht wird, wenn einige ausfallen oder ausgeschaltet werden.

Listing 5.8: UserService (Version 1.0.1) fillUsermap Methode

```
1  private void fillUserMap() {
2      userMap.put(new Long(1), createUserAtOnce(new Long(1), "pseudonym1",
3          "12345", 13001, "user1@frs.de", "munich", "01.12.1980"));
4      userMap.put(new Long(2), createUserAtOnce(new Long(2), "pseudonym2",
5          "12345", 13001, "user2@frs.de", "munich", "02.12.1980"));
6      userMap.put(new Long(3), createUserAtOnce(new Long(3), "pseudonym3",
7          "12345", 13000, "user3@frs.de", "munich", "03.12.1980"));
8      userMap.put(new Long(4), createUserAtOnce(new Long(4), "pseudonym4",
9          "12345", 13000, "user4@frs.de", "munich", "04.12.1980"));
10     userMap.put(new Long(5), createUserAtOnce(new Long(5), "pseudonym5",
11         "12345", 13000, "user5@frs.de", "munich", "05.12.1980"));
12   }
```

5.2.2 Newton und Paremus mit Spring Dynamic Modules

Newton und Paremus Service Fabric bieten auch die Möglichkeit, Service mit Hilfe von Spring Dynamic Modules zu beschreiben und dann vom Framework zu interpretieren. Das mit Spring Dynamic Modules konfigurierte UserService Beispiel sieht wie im Listing 5.9 aus. Die Konfiguration stellt sich aus den Dateien bundle-context.xml, bundle-context-osgi.xml und springchaincli.template zusammen. In bundle.context.xml und bundle-context-osgi.xml stehen die Spring Dynamic Modules Definitionen, welche den OSGi Blueprint Dateien aus Listing 5.4 sehr ähneln. Die springchaincli.template Datei ist dazu da, um die Spring Definitionen von SCA zu interpretieren.

Listing 5.9: UserService eingebunden in Newton mit Hilfe von Spring Dynamic Modules

```
1 Datei: bundle-context.xml
2 <beans>
3   <bean name="UserService"
4   class="de.frs24.business.server.UserServiceImpl" />
5 </beans>
6
7 Datei: bundle-context-osgi.xml
```

```
<beans>
  <osgi:service ref="UserService"
    interface="de.frs24.business.data.IUserService"/>
</beans>

Datei: springchaincli.template
<component name="Service">
    <service name="UserService" />
    <implementation.spring />
</component>
```

5.3 Implementierungsbeispiel mit Apache Felix und DOSGi

In diesem Abschnitt wird beschrieben, welche Änderungen man an der Implementierung aus Kapitel 5.1 vornehmen muss, um diese in das CXF DOSGi Framework zu installieren und zu starten. Die Container Felix und Equinox unterstützen alle Bundles, die nach dem OSGi 4.2 Standard entwickelt wurden. Somit läuft das Beispiel aus dem Kapitel 5.1 ohne Änderungen in beiden Containern. Um das UserService Beispiel in einer verteilten Umgebung starten zu können, sind folgende Schritte nötig:

1. Zookeeper Server herunterladen und starten
2. Im Unterordner "load" der Felix (oder Equinox) Installation befindet sich eine Datei mit dem Namen `org.apache.cxf.dosgi.discovery.zookeeper.cfg` und mit folgendem Inhalt: `zookeeper.host=192.168.81.188` und `zookeeper.port=2181`. Host und Port sind entsprechen dem Zookeeper Server, auf dem UserServiceImpl seinen Service anmeldet.
3. Zum Starten des Felix OSGi Containers folgendes aus der Console aufrufen:
   `java -jar bin/felix.jar`.
4. Starten der CXF-DOSGi Bundles und OSGi 4.2 Bundles (osgi.compendium) im Felix Container
5. Installation vom IUserService und UserService Bundles in Felix

Nach all diesen Schritten ist die Consolenausgabe wie in Abbildung 5.2 dargestellt. In der ersten Zeile sieht man an den Logausgaben, dass sich der UserService beim Zookeeper angemeldet hat.

```
INFO: Creating Zookeeper node: /osgi/service_registry/de/frs24/business/data/IUserService/192.168.220.1
#business#data#IUserService
14.12.2009 04:28:29 org.apache.cxf.dosgi.dsw.hooks.AbstractClientHook lookupDiscoveryService
INFO: lookup discovery service: interface: org.apache.cxf.dosgi.dsw.decorator.ServiceDecorator filter:
14.12.2009 04:28:29 org.apache.cxf.dosgi.dsw.hooks.ServiceHookUtils publish
INFO: Remote de.frs24.business.data.IUserService endpoint published via Discovery service
ps
14.12.2009 04:28:40 org.apache.cxf.dosgi.dsw.hooks.AbstractClientHook lookupDiscoveryService
INFO: lookup discovery service: interface: org.osgi.service.startlevel.StartLevel filter: null
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (2.0.1)
[   1] [Active     ] [    1] Apache Felix Bundle Repository (1.4.2)
[   2] [Active     ] [    1] Apache Felix Shell Service (1.4.1)
[   3] [Active     ] [    1] Apache Felix Shell TUI (1.4.1)
[   4] [Active     ] [    1] osgi.cmpn (4.2.0.200908310645)
[   8] [Active     ] [    1] Distributed OSGi Distribution Software Single-Bundle Distribution (1.1)
[  93] [Active     ] [    1] IUserService (1.0.0)
[  94] [Active     ] [    1] UserServiceImpl (1.0.0)
```

Abbildung 5.2: Felix Consolenausgabe mit der Installation des UserService Beispiels

5.3.1 CXF DOSGi mit Spring Dynamic Modules und Blueprint konfigurieren

Um die Implementierung wie in Listing 5.5 vom OSGi Code frei zu halten, werden in CXF DOSGi Spring Dyamic Modules und Blueprint Services unterstützt. Die EditUserData Implementierung ohne OSGi spezifisches Code ist die gleiche wie die die bei Newton in Listing 5.7 zu sehen ist. Die Spring DM Datei "spring.xml" aus Listing 5.10 wird innerhalb des UserServiceImpl Bundles im Order "META-INF/spring" definiert. Im Wesentlichen stellt die "spring.xml" Datei das gleiche dar, wie die drei Dateien von Newton (Listing 5.9). Das Pflegen und Aktualisieren der drei Dateien ist somit viel leichter und anschaulicher.

Listing 5.10: UserServiceImpl Definition mit Hilfe der Dependency Injection von Spring Dynamic Modules

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <beans xmlns="http://www.springframework.org/schema/beans"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xmlns:osgi="http://www.springframework.org/schema/osgi"
6    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.
         springframework.org/schema/beans/spring-beans.xsd
7                        http://www.springframework.org/schema/osgi http://www.
                             springframework.org/schema/osgi/spring-osgi.xsd">
8    <osgi:service interface="de.frs24.business.data.IUserService">
9      <osgi:service-properties>
10       <entry key="osgi.remote.interfaces" value="*" />
11     </osgi:service-properties>
12
13     <bean class="de.frs24.business.server.UserServiceImpl" />
14   </osgi:service>
15 </beans>
```

Apache CXF DOSGi unterstützt die Dependency Injection auch durch Blueprint Services, die ein Teil der OSGi 4.2 Spezifikation sind. Mit der XML Datei aus Listing 5.4 wird der UserServiceImpl aktiviert. Für die Unterstützung der Blueprint Services innerhalb von Apache CXF DOSGi ist die Implementierung der Apache Arries Blueprint nötig. Die entsprechenden Bundles können unter dem Link [ari09] gefunden werden. Diese müssen nur in den Felix Container installiert werden und dann kann UserService Beispiel installiert werden.

5.4 Evaluierung des Implementierungsaufwandes

Es sind viele Faktoren zu berücksichtigen, wenn man den Implementierungsaufwand zu den einzelnen Frameworks abschätzen soll. In den Abschnitten 5.2 und 5.3 wurden die Anpassungen der OSGi 4.2 Implementierung des UserServices vorgestellt, damit das Beispiel auch in den Frameworks lauffähig ist. Wenn man die Anpassungen miteinander vergleicht, fällt auf, dass bei der CXF 5.3 viel weniger anzupassen ist als im Abschnitt 5.2. Die Kriterien für den Aufwand bei der Implementierung werden anhand der Punkte in der Tabelle 5.1 zusammengefasst. Dabei werden die Kriterien, wie im Kriterienkatalog 3.3.3, anhand einer Skala von ++ (vollständig vorhanden) und - - (nicht vorhanden) bewertet. Bei "kein Zugriff" kann dieses Kriterium nicht bewertet werden, da der Zugriff aus die Implementierung nicht möglich war.

Dokumentation im Framework Code - Wenn man alle Abläufe, die in einem Framework stattfinden, verstehen will, muss man auch manchmal die Details aus dem Imple-

mentierungscode herauslesen. Die Schnittstellen (Interfaces) sind im CXF DOSGi sehr gut dokumentiert, die Implementierungen jedoch weniger. Geringen Aufwand bereitet das Verstehen des CXF DOSGi Codes, da es derzeit aus ca. 30 Java Klassen besteht. CXF verbindet nur die schon etablierten Komponenten wie Zookeeper, CXF, Equinox und Felix und daher ist die CXF DOSGi Implementierung klein. Bei Newton hat Code im Verhältnis zum CXF DOSGi 20 mal so viele Java Klassen und die Kommentare fehlen sowohl in den Schnittstellen als auch im restlichen Code. Da der Paremus Service Fabric ein kommerzielles Framework ist, ist kein Zugang zum Implementierungscode möglich.

Verfügbare Bespiele - Newton und Paremus Service Fabric sind seit 2006 verfügbar. Leider existieren jedoch nicht viele Beispiele im Internet, sondern nur die auf den offiziellen Seiten der Frameworks. Im Vergleich dazu existiert CXF DOSGi erst ein halbes Jahr, hat aber mindestens genau so viele Beispiele wie die beiden anderen zusammen. Viele Hilfen zu den einzelnen Komponenten von CXF DOSGi sind frei über das Web zugänglich, z.B. die Beispiele auf der Zookeeper Webseite [zoo09].

Erweiterungsmöglichkeiten - Bei allen drei Frameworks können die Protokolle für den Zugriff auf die Services erweitert werden. Bei Newton gibt es keinen Sicherheitmechanimus, der z.B nur bestimmten Servern erlaubt, nach Services zu suchen und sie zu benutzen. Paremus Service Fabric bietet einen Zugriffsmechanismus auf die Services über SSH Zertifikate. Bei CXF DOSGi kann Apache Zookeeper einen Zugriffsmechanismus integrieren, wie in Kapitel 4.5 Kriterium DR3 beschrieben.

Vorhandene Erweiterungen - Newton und Service Fabric werden von dem Unternehmen Paremus entwickelt. Es existieren nicht viele Erweiterungen und die, die existieren, sind meistens kommerziell. Im Gegensatz dazu besteht CXF aus offenen Komponenten, für die es schon viele Erweiterungen gibt. Zum Beispiel unterstützt CXF DOSGi die OSGi 4.2 Spezifikation der Blueprint Services nicht in seiner Implementierung, sondern die Apache Arries Blueprint kann in CXF DOSGi integriert werden.

Kriterium	Newton	Service Fabric	CXF DOSGi
Dokumentation im Framework Code	-	kein Zugriff	0
Verfügbare Bespiele	-	0	+ +
Erweiterungsmöglichkeiten	0	0	+
Vorhandene Erweiterungen	-	-	+

Tabelle 5.1: Vergleich der Entwicklung in dem jeweiligen Framework

5.5 Evaluierung des Aufwandes einer Migration

Die Anforderungen ST2 und ST3 aus dem Kriterienkatalog 3.2.5 war, dass sich die Service Implementierung soweit wie möglich an einen Standard halten soll, um die Abhängigkeit zur nur einem Framework zu vermeiden. Es wird angenommen, dass das FRS24 System vollständig mit Hilfe von Newton, Service Fabric oder CXF DOSGi umgesetzt wurde. Im Fall, dass neue Anforderungen bei FRS24 durch neue Frameworks besser erfüllt werden, wird im Folgenden der Aufwand für die Migration geschätzt. Es wird angenommen, dass die Konfiguration der Bundels mit Spring Dynamic Modules gelöst wurde und das neue

Framework den OSGi 4.2 Standard vollständig unterstützt. Durch die leichte Umstellung der Komponenten, z.B. in eines der Frameworks die in Kapitel 4.2 beschrieben und noch in der Entwicklungsphase sind, wären dann gegebenenfalls folgende Anpassungen nötig:

- Änderungen der Spring Konfigurationsdateien jedes Bundles. Bei Newton und Service Fabric sind es drei Dateien pro Service, wie im Listing 5.9 dargestellt. Diese müssten dann zu einer Konfigurationsdatei zusammengeführt und konfiguriert werden. Bei Apache CXF DOSGi sind gegebenenfalls nur Änderungen an der Protokollkonfiguration nötig, aber bei CXF DOSGi ist es nur eine Konfigurationsdatei (Listing 5.10), die angepasst werden muss. Die CXF DOSGi Konfiguration muss nur angepasst werden, wenn der Zugriff geändert wird, z.B. wenn auf die Services nicht per SOAP, sondern über RMI Zugegriffen wird.

- Protokollerweiterungen in Service Fabric und Newton sind im Framework integriert, d.h. bei einer Migration müssen die Protokollerweiterungen auch migriert werden. Diese Migration würde nicht nur Konfigurations- sondern auch Implementierungsaufwand mit sich bringen, da die Protokollerweiterungen in Newton von bestimmten Schnittstellen erben müssen. In Apache CXF DOSGi können Frameworkerweiterungen als eigenständige Bundles implementiert werden, somit wäre das Umziehen nur Konfigurationsaufwand.

- Die Replicationhandler (Listing 5.6) in Newton und Paremus Service Fabric müssen beim Umzug ebenfalls beachtet werden. Sie müssen neu konfiguriert werden. Bei CXF DOSGi muss die Replicationhandler Eigenschaft in der jeweiligen Startkonfiguration des Felix oder Equinox Containers definiert werden. Diese müssen dann in den neuen Framework entsprechend auch definiert werden.

Die obige Aufzählung hebt Vorteile hervor, wie viel weniger Konfigurationsaufwanderbracht werden muss, wenn die Bundles strikt nach dem OSGi 4.2 Standard entwickelt wurden. Die Aufwände für die Migration wurden in folgender Tabelle 5.2 grob geschätzt. Für die Schätzung wird die Einheit Mannstunden (MS) verwendet, also die Menge der Arbeit, die eine Person durchschnittlich in einer Stunde schafft.

Die Werte Spring Konfiguration und Replicationhandler sind in der Tabelle jeweils pro Service geschätzt und Protokollerweiterungen pro Protokoll. Geht man davon aus, dass bei FRS24 50 Services gleichzeitig laufen, wobei einige mehrere Instanzen haben. In der letzten Zeile der Tabelle 5.2 wird die Summe für die Migration gebildet für 50 Service und 2 Protokolle. Die Ergebnisse der Summe zeigen grob, dass die Migration von OSGi 4.2 Standard Komponenten in CXF DOSGi viel schneller geschehen würde als in Newton.

Anpassungen bei Migration in Mannstunde (MS	**Newton**	**Service Fabric**	**CXF DOSGi**
Spring Konfiguration	2 MS	1.5 MS	0.5 MS
Protokollerweiterungen	10 MS	10 MS	7 MS
Replicationhandler	1 MS	1 MS	1 MS
SUMME	170 MS	145 MS	89 MS

Tabelle 5.2: Migration zu einem OSGi 4.2 Framework

5.6 Empfehlung eines Frameworks

In der Analyse im Abschnitt 2.2 wurden die Nachteile des FRS24 Systems beschrieben. Aus diesen Nachteile schloss man die eigentlichen Vorgaben, die in Kriterienkatalog 3.2 detailliert ausgeschrieben wurden, welche das Framework erfüllen sollte, um es bei FRS24 auch einzusetzen. Dabei sind die Kriterien verteilte Registry, Asynchrone Kommunikation über JMS, dynamische Versionierung, Deployment zur Laufzeit, Verbreitung und Standard Komponenten als ausschlaggebend definiert worden.

Für die Umsetzung der FRS24 Services wird die Apache CXF DOSGi Framework empfohlen. Die Entscheidung beruht auf dem Ergebnis des Vergleichs der Frameworks im Kapitel 4.6. Dieser Vergleich zeigt, dass das Paremus Service Fabric und Apache CXF DOSGi ebenbürtig sind. Der große Unterschied liegt in der Bereitstellung des Zugriffs auf die Services in den Containern. Service Fabric baut auf einer Eigenentwicklung auf, die über RMI realisiert wird und CXF DOSGi setzt auf Webservices. Apache CXF bietet viel mehr Möglichkeiten für Zugriffe von anderen Plattformen aus auf die Services, da Webservicezugriffe weit verbreitet sind und von allen Plattformen aus aufgerufen werden können.

Erst gegen Ende der Erstellung dieser Arbeit ist das Apache CXF DOSGi in Version 1.1 erschienen, diese Version integriert den Zookeeper Server als Service Discovery. Bei der Entwicklung und den Tests sind deshalb oftmals unerklärliche Fehler aufgetreten. Es wird noch eine Weile dauern, bis CXF DOSGi stabil läuft. Trotz der Tatsache, dass das Projekt erst seit einem guten halben Jahr auf dem Markt ist, ist die Entwicklung erstaunlich weit fortgeschritten und das Projekt bietet viele Möglichkeiten an.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden die Anforderungen des FRS24 Portals an eine verteilte Serviceumgebung anhand des Kriterienkatalogs beschrieben. Dieser Kriterienkatalog setzt eine dynamische, standardisierte, konfigurierbare, versionierbare und verwaltbare Service Umgebung voraus. Die Recherche nach einem geeignetem Standard hat ergeben, dass OSGi für viele dieser Anforderungen einen Lösungsweg vorgibt. OSGi wurde so konzipiert, dass es innerhalb einer JVM eine dynamische, versionierbare und konfigurierbare Umgebung für Services bereitstellt. Erst seit der OSGi 4.2 Spezifikation definiert die OSGi Allianz die OSGi Services auch für eine verteilte Umgebung. Die finale Spezifikation des OSGi 4.2 Standards ist erst im September 2009 verabschiedet worden. Dort wurden nur die Schnittstellen für die Discovery und die Distribution der Services definiert. Die einzelnen Implementierungen entscheiden selbst, wie die Discovery und Distribution gelöst wird.

Da der OSGi 4.2 Standard erst seit kurzem verfügbar ist, existieren noch nicht viele Frameworks, die diesen Standard unterstützen und auch die Discovery und Distribution für die Services bereitstellen. Viele der Projekte sind noch in der frühen Entwicklungsphase (siehe Kapitel 4.2). Somit war eine der Schwierigkeiten dieser Arbeit, geeignete Frameworks zu finden, die auch detailliert evaluiert werden können. Frameworks, deren Entwicklung sich noch in der “Beta” Phase befindet, konnten zur Evaluierung nicht genommen werden. In dieser Arbeit wurden drei spezifische Frameworks ausgewählt, um sie auf ihre Eigenschaften zu überprüfen. Die Überprüfung wurde für alle 19 Punkte des Kriterienkatalogs bei jedem Framework gemacht. Die Überprüfung wurde praktisch ausprobiert und dann in der Bewertung beschrieben, wie jeweils ein Framework dieses Kriterium auch umsetzt. Falls ein Kriterium vom Framework nicht vollständig oder gar nicht erfüllt wird, wurde nach einem alternativen Lösungsweg gesucht.

Um einige Kriterien, wie z.B. Verteilte Registry und Dynamische Versionierung, zu evaluieren, wurde in dieser Arbeit eine vereinfachte Implementierung eines Services, aus dem bestehendem FRS24 System, für jedes Framework entwickelt. Diese Implementierungen wurden auch dazu genutzt, um die Unterschiede zu OSGi 4.2 Standard Implementierung darzustellen. Da eine der Anforderungen an das Framework von FRS24 ist, sich soweit wie möglich an den Standard zu halten, um die Abhängigkeit von einem bestimmten Framework so gering wie möglich zu halten. Es stellte sich heraus, dass diese Eigenschaft am besten vom Apache CXF DOSGi gelöst wurde. Zwei der drei Frameworks unterstützen nicht den OSGi 4.2 Standard, sondern nur den OSGi 4 Standard. Mit Hilfe einer Eigenentwicklung stellen sie die OSGi 4.2 Funktionalität der Verteilung bereit.

Für die Umsetzung der FRS24 Services wird die Apache CXF DOSGi Framework empfohlen. Die Entscheidung beruht auf dem Ergebnis des Vergleichs der Frameworks im Kapitel 4.6. Dieser Vergleich zeigt, dass das Paremus Service Fabric und Apache CXF DOSGi ebenbürtig sind. Der große Unterschied liegt in der Bereitstellung des Zugriffs auf die Services in den Containern. Service Fabric baut auf einer Eigenentwicklung auf, die über RMI realisiert wird und CXF DOSGi setzt auf Webservices. Apache CXF bietet viel mehr Möglichkeiten für Zugriffe von anderen Plattformen aus auf die Services, da Webservicezu-

griffe weit verbreitet sind und von allen Plattformen aus aufgerufen werden können. Die erste offizielle Apache CXF DOSGi Version, die den OSGi 4.2 Standard inklusive der Discovery von Services implementiert, wurde gegen Ende der Erstellung dieser Arbeit verabschiedet. Die Version ist noch nicht für den produktiven Einsatz geeignet, da sie nicht stabil genug ist.

OSGi verbreitete sich in letzten Jahren sehr schnell und spielt eine immer wichtigere Rolle in der Softwareentwicklung. Der OSGi 4.2 Standard ist erst seit etwa vier Monaten verabschiedet. In dieser kurzen Zeit entstanden viele neue Projekte, die diese Features umsetzen können. Es wird voraussichtlich noch ein halbes Jahr dauern, bis alle Frameworks stabil laufen und für die Anforderungen von FRS24 geeignete Lösungswege bieten. Auch wenn sie noch nicht in den Livebetrieb genommen werden können, könnten die FRS24 Services in dieser Zeit bereits auf den OSGi Standard umgestellt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[act09] *Apache ActiveMQ*. http://activemq.apache.org/, 11 2009.

[apa09] *Apache Struts*. http://struts.apache.org/, 07 2009.

[ari09] *Apache Aries Blueprint*. http://incubator.apache.org/aries/blueprint.html, 12 2009.

[Ben04] Bengel, Günther: *Verteilte Systeme: Grundlagen und Praxis des Client-Server-Computing. Inklusive aktueller Technologien wie Web-Services*. Vieweg+Teubner, Wiesbaden, 02 2004.

[Bos04] Bosch, Andy: *Java Server Faces*. Addison-Wesley Verlag, München, 2004.

[CSS09] *Cascading Style Sheets*. http://www.w3.org/Style/CSS/, 07 2009.

[cxf09] *Apache CXF*. http://cxf.apache.org/, 11 2009.

[Dee09] Deeg, Maria: *Modellierung von Services mit SoaML*. Objekt Spektrum, Seiten 1–3, 01 2009.

[DEF+08] Dunkel, Jürgen, Andreas Eberhart, Stefan Fischer, Carsten Kleiner und Arne Koschel: *Systemarchitekturen für verteilte Anwendungen. Client-Server, Multi-Tier, SOA, Event Driven Architecture, P2P, Grid, Web 2.0*. Hanser, München, 1 Auflage, 2008.

[Det04] Detken, Kai-Oliver: *Service Discovery: Automatisches Auffinden von Diensten*. NET, Seiten 34–36, 06 2004.

[dos09] *Apache CXF Distributed OSGi*. http://cxf.apache.org/distributed-osgi.html, 11 2009.

[ecf09] *Eclipse Communication Framework (ECF)*. http://www.eclipse.org/ecf/, 12 2009.

[equ09] *Eclipse Equinox*. http://www.eclipse.org/equinox/, 10 2009.

[ESB09] *Enterprise Service Bus*. http://de.wikipedia.org/wiki/Enterprise_Service_Bus, 08 2009.

[fel09] *Apache Felix*. http://felix.apache.org/site/index.html, 11 2009.

[Gro99] Group, Network Working: *Service Location Protocol, Version 2*. http://tools.ietf.org/html/rfc2608, 06 1999.

[Haa08] Haase, Oliver: *Kommunikation in verteilten Anwendungen: Einführung in Sockets, Java RMI, CORBA und Jini*. Oldenbourg, München, 2008.

[Hen99] Henning, Michi: *Binding, Migration, and Scalability in CORBA*. http://www.triodia.com/staff/michi/cacm.pdf, 06 1999.

[HKF03] Horn, Christian, Immo O. Kerner und Peter Forbrig: *Lehr- und Übungsbuch Informatik 1. Grundlagen und Überblick*. Hanser Fachbuchverlag, München, 8 2003.

[HS09] Hildebrandt, Eduard und Christian Schneider: *Strategien zum Umgang mit neuen Service-Versionen in einer SOA*. Objekt Spektrum, Seiten 18–23, 02 2009.

[jav09] *JavaScript*. http://de.selfhtml.org/javascript/intro.htm, 07 2009.

[jax09a] *Java API for XML-Based Web Services (JAX-WS)*. http://www.jcp.org/en/jsr/detail?id=224, 11 2009.

[jax09b] *JAX-RS: The Java API for RESTful Web Services*. http://jcp.org/en/jsr/detail?id=311, 11 2009.

[jin09] *Jini Services Browser*. http://www.gigaspaces.com/docs/JiniApi/com/sun/jini/example/browser/package-summary.html, 10 2009.

[jms09] *Java Message Service*. http://java.sun.com/products/jms/, 08 2009.

[jsp09] *Java Server Pages*. http://java.sun.com/products/jsp/, 07 2009.

[kno09] *Knopflerfish*. http://www.knopflerfish.org, 08 2009.

[KRW09] Konradi, Philip, Viktor Ransmayr und Nicole Wengatz: *Distributed OSGi*. Java Spektrum, (04):13–17, 2009.

[lda96] *String Representation of LDAP Search Filters RFC 1960*. http://www.ietf.org/rfc/rfc1960.txt, 09 1996.

[LSL+08] Liebhart, Daniel, Guido Schmutz, Marcel Lattmann, Markus Heinisch, Michael Könings, Mischa Kölliker, Perry Pakull und Peter Welkenbach: *Integration Architecture Blueprint: Leitfaden zur Konstruktion von Integrationslösungen*. Hanser Fachbuchverlag, München, 2008.

[Mat08] Mathas, Christoph: *SOA intern Praxiswissen zu Servic-orientierten IT Systemens*. Hanser, München, 2008.

[mul09] *Mule4Newton*. http://mule4newton.codecauldron.org/, 08 2009.

[new09] *Newton Framework*. http://newton.codecauldron.org/site/index.html, 08 2009.

[osg09a] *OSGi Alliance*. http://www.osgi.org/, 08 2009.

[osg09b] *OSGi Service Platform Specification*. http://www.osgi.org/Download/Release4V42, 09 2009.

[oso09] *Service Component Architecture*. http://www.osoa.org, 09 2009.

[par09] *Paremus Service Fabric*. http://www.paremus.com/, 08 2009.

[poj09] *Plain Old Java Object*. http://de.wikipedia.org/wiki/Plain_Old_Java_Object, 10 2009.

[pro09] *mBedded Server Equinox Edition*. http://www.prosyst.com/products/osgi_framework.html, 10 2009.

[RES09] *Representational State Transfer (REST)*. http://de.wikipedia.org/wiki/Representational_State_Transfer, 11 2009.

[SAN09] *Storage Area Network*. http://de.wikipedia.org/wiki/Storage_Area_Network, 08 2009.

[See08] Seeberger, Heiko: *Erste Schritte mit OSGi*. http://entwickler.de/zonen/portale/psecom,id,101,online,2078,.html, 12 2008.

[sig09] *Sigil*. http://felix.apache.org/site/apache-felix-sigil.html, 09 2009.

[soa09] *Simple Object Access Protocol*. http://de.wikipedia.org/wiki/SOAP, 11 2009.

[spr09] *Spring Dynamic Modules for OSGi(tm) Service Platforms*. http://www.springsource.org/osgi, 08 2009.

[Sta09] Stal, Michael: *World Wide CORBA - Verteilte Objekte im Netz*. http://www.stal.de/Downloads/CORBA/Corba_.html, 06 2009.

[swo09] *Swordfish SOA Runtime Framework Project*. http://www.eclipse.org/swordfish/, 10 2009.

[tom09] *Apache Tomcat*. http://tomcat.apache.org/, 07 2009.

[udd09] *Unversal Description, Discovery and Intergration (UDDI)*. http://uddi.xml.org/, 10 2009.

[Ull09] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, Wiesbaden, 6 Auflage, 02 2009.

[WHKL08] Wütherich, Gerd, Nils Hartmann, Bernd Kolb und Matthias Lübken: *Die OSGi Service Platform - Eine Einführung mit Eclipse Equinox*. Oldenbourg, München, 2008.

[ZLMG08] Zeitner, Alfred, Birgit Linner, Martin Maier und Thorsten Göckeler: *Spring 2.5: Eine pragmatische Einführung*. Addison-Wesley Verlag, München, 09 2008.

[zoo09] *Apache ZooKeeper*. http://hadoop.apache.org/zookeeper/, 11 2009.