Master-Arbeit

# Verteilung eines Plugin-basierten Systems innerhalb einer Grid-Umgebung

Doreen Seider
DLR Simulations- und Softwaretechnik, Universität Leipzig

DLR

Universität Leipzig
Institut für Informatik
Lehrstuhl für Angewandte Telematik / e-Business
Prof. Dr. Volker Gruhn

Masterarbeit

# Verteilung eines Plugin-basierten Systems innerhalb einer Grid-Umgebung

## Anhand eines Fallbeispiels im Projekt SESIS

Autor: Doreen Seider
Matrikelnummer: 9921379
Studiengang: Informatik (M.Sc.)
5. Semester

Erstgutachter: Prof. Dr. Volker Gruhn
Zweitgutachter: Dipl.-Math. Andreas Schreiber

Eingereicht am: 31.10.2007

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Leipzig, 31. Oktober 2007

# Abstract

Ein Grid ermöglicht die Nutzung beliebig verteilter Ressourcen, um lokal fehlende Ressourcen zu kompensieren. So müssen in der Wissenschaft oft große Mengen an Daten gespeichert und analysiert, rechenintensive Simulationen durchgeführt oder kurzzeitig Anwendungen verwendet werden, die in ihrer Anschaffung auf Grund hoher Lizenzgebühren zu teuer sind. Ressourcen (Rechenleistung, Datenbanken, Anwendungen, Messgeräte) ermöglichen Dienste und werden daher auch als Grid Services in das Grid eingebunden. Ein Grid Service ist ein zustandsbehafteter Web Service.

Ein Plugin-basiertes System ist bis auf eine minimale Laufzeitumgebung ausschließlich aus Plugins aufgebaut, die beliebig hinzugefügt und entfernt werden können. Ein solches System verfügt über großes Potential. Durch das individuelle Hinzufügen von Plugins kann das System zum einen beliebig mächtig werden und zum anderen ist es dadurch möglich, ein benutzerspezifisches System aufzubauen.

Plugins können rechen- und/oder speicherintensiv sein. Sie können auch in ihrer Anschaffung teuer sein. Vergleicht man dies mit der Motivation für ein Grid, kommt man zu der Idee, Plugins als Grid Services im Grid zu verteilen. Da die Technologie des Grid und der Plugin-basierten Systeme relativ neu sind, ist die Frage noch offen, wie Plugins sinnvoll in ein Grid eingebunden werden können.

Das Ziel dieser Masterarbeit ist die Verteilung von Plugins des Plugin-basierten Systems RCE (Reconfigurable Computing Environment) in einer Grid-Umgebung. Die Plugins sollen dabei als Grid Services realisiert werden. Weiterhin soll die clientseitige Integration dieser Grid Services in RCE betrachtet werden.

Um die Aufgabenstellung umzusetzen, werden vier Realisierungsmöglichkeiten für den geforderten Grid Service erarbeitet und bewertet. Es stellt sich die Frage, inwieweit RCE serviceseitig integriert wird und wo sich die Logik des verteilten Plugin und gegebenenfalls die von RCE befindet. Als praktikabelste Möglichkeit wird die Variante erkannt, bei der RCE zusammen mit dem integrierten und zu verteilenden Plugin aus dem Grid Service ausgelagert wird. Der Grid Service dient bei dieser Realisierungsvariante als Kommunikationsschnittstelle zwischen Client und RCE beziehungsweise den Plugins.
Beim Entwurf dieser Möglichkeit wurden neben dem Konzept des Grid Service weitere Aspekte wie die Abbildung von RCE-Funktionen auf Grid-Technologien, die Sicherheit, die Service Discovery oder die Realisierung des Verteilprozesses erschlossen und betrachtet.
Bei der clientseitigen Integration der Grid Services in RCE wird das Stub-Konzept verwendet, mit dem sowohl eine transparente als auch eine nicht-transparente Integration realisiert werden kann.
Abschließend erfolgt eine Beschreibung der Implementierung an Hand eines Beispielplugins und deren Test.

# Inhaltsverzeichnis

**1 Einleitung** **9**

1.1 Motivation und Stand der Forschung . . . . . 9

1.2 Aufgabenstellung . . . . . 11

1.3 Vorgehen und Aufbau der Arbeit . . . . . 12

**2 Grundlagen** **14**

2.1 Grid Computing . . . . . 14

2.1.1 Begriffsdefinition . . . . . 15

2.1.2 Open Grid Services Architecture . . . . . 16

2.1.3 Web Services Resource Framework . . . . . 17

2.1.4 Globus Toolkit . . . . . 19

2.2 Plugin-basierte Systeme . . . . . 22

2.2.1 OSGi-Technologie . . . . . 22

2.2.2 Eclipse-Plattform . . . . . 24

2.3 Projekt SESIS . . . . . 25

2.3.1 Architektur von RCE . . . . . 26

**3 Anforderungsanalyse** **28**

3.1 Anwendungsfälle . . . . . 28

3.2 Anforderungen . . . . . 29

3.3 Abgrenzung der Implementierung . . . . . 30

**4 Entwurf** **32**

4.1 Realisierungsmöglichkeiten für einen Plugin Service . . . . . 32

4.1.1 Vorüberlegungen . . . . . 34

4.1.2 Plugin als Grid Service . . . . . 35

4.1.3 RCE als Grid Service . . . . . 36

4.1.4 Reduziertes RCE als Grid Service . . . . . 38

4.1.5 Grid Service als Kommunikationsschnittstelle zu RCE und Plugin . . . 39
4.1.6 Fazit . . . 40
4.2 Entwurf der Realisierungsmöglichkeit: Grid Service als Kommunikationsschnittstelle zu RCE und Plugin . . . 42
4.2.1 Plugin Service . . . 43
4.2.2 Abbilden von RCE-Funktionen auf Grid-Technologien . . . 48
4.2.3 Sicherheit . . . 49
4.2.4 Service Discovery . . . 51
4.2.5 Verteilprozess . . . 53
4.3 Clientseitige Integration von Plugin Services in RCE . . . 55

**5 Implementierung und Test** **58**
5.1 Simplorer Service . . . 58
5.1.1 Implementierung . . . 59
5.1.2 Beschreibung mit WSDL . . . 60
5.1.3 Deployen . . . 62
5.2 Sicherheitsdeskriptoren . . . 64
5.3 UDDI-Verzeichnis . . . 66
5.4 Stub für RCE . . . 66
5.5 Test . . . 67
5.6 Entwicklung . . . 68
5.6.1 Modultests . . . 68
5.6.2 Werkzeuge . . . 70

**6 Schlussbetrachtung** **71**
6.1 Zusammenfassung . . . 71
6.2 Ausblick . . . 71

**A WSDL-Datei** **73**

**B Service-Sicherheitsdeskriptor** **75**

**C WSDD-Datei** **76**

**D JNDI-Datei** **77**

**E Glossar** **78**

**Abbildungsverzeichnis** **81**

**Tabellenverzeichnis** **83**

**Literatur** **84**

**Index** **87**

# 1 Einleitung

In diesem Kapitel wird die Motivation der Arbeit geklärt. Hierfür wird zunächst der Stand der Forschung im Bereich des Grid Computing und der Plugin-basierten Systeme dargestellt. Weiterhin wird die Aufgabenstellung spezifiziert und darauf aufbauend ein kurzer Überblick über den Aufbau dieser Arbeit gegeben. Dies beinhaltet auch die Beschreibung des Vorgehensmodells, welches bei der Realisierung der Arbeit verfolgt wurde.

## 1.1 Motivation und Stand der Forschung

Die Motivation für die Entwicklung des Grid ist der Wunsch nach einer gemeinschaftlichen Nutzung von Ressourcen, die beliebig verteilt sein können. Ressourcen können unter anderem Rechenleistung, Speicher, Netzwerke, Anwendungen oder Messgeräte sein [Fos03]. Allen Ressourcen gemeinsam ist, dass sie Dienste ermöglichen. Daher können Ressourcen im Grid auch als Dienste repräsentiert werden, welche man dann als Grid Services bezeichnet [FKNT02].

Das Grid Computing ist ein großer Forschungs- und Entwicklungsbereich. Es existiert weltweit eine Vielzahl an Projekten mit unterschiedlicher Zielsetzung. Einige Grid-Projekte beschäftigen sich mit der Entwicklung von Softwaretools, welche die Architektur des Grid realisieren und damit die Grid-Middleware darstellen. Andere konzentrieren sich auf die Optimierung der Grid-Infrastruktur, dem zu Grunde liegendem Netzwerk. Weiterhin gibt es Projekte, die Grid-Anwendungen für das wissenschaftliche Umfeld entwickeln. Andere wiederum konzentrieren sich auf die Realisierung der Grid Services.

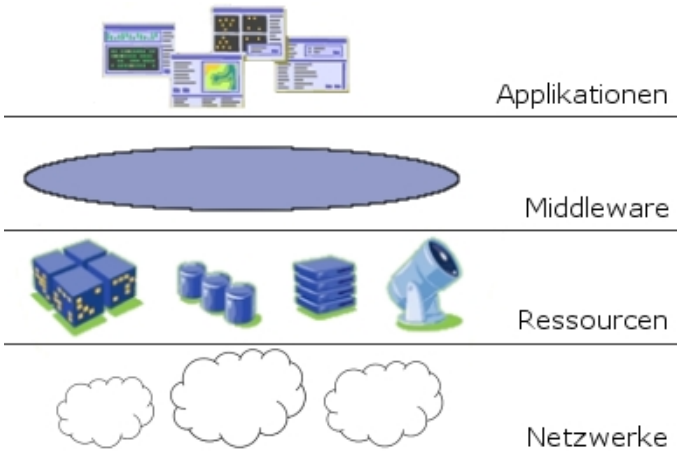


Abbildung 1: Schichten des Grid

In Abbildung 1 sind die Schichten eines Grid anschaulich dargestellt. Diese spiegeln gleichzeitig die Bereiche wieder, in denen die einzelnen Grid-Projekte angesiedelt sind. Viele Grid-Projekte erstrecken sich dabei auch über mehrere dieser Bereiche.

Im wissenschaftlichen Bereich fallen immer größere Datenmengen an. Experimente im Bereich der Hochenergiephysik produzieren alleine am Forschungszentrum CERN Terrabytes an Daten pro Tag [CER]. Wissenschaftler von verschiedenen Forschungseinrichtungen, Laboratorien und Universitäten aus der ganzen Welt nutzen die in den Daten enthaltenen Informationen. Hierfür müssen sie auf die Daten zugreifen und sie analysieren können.
Weiterhin existieren Projekte wie SDSS (Sloan Digital Sky Survey) [SDS], die eine dreidimensionale Karte des Himmels erstellen wollen. Hierfür werden Aufnahmen vom Himmel gemacht, die in zahlreichen Datenbanken gespeichert und darüber auch anderen zur Verfügung gestellt werden.
Auch die Simulationen in der Klimaforschung erzeugen große Datenansammlungen, die es zu speichern und zu verarbeiten gilt. Darüber hinaus sind solche wissenschaftlichen Simulationen sehr rechenintensiv. Dies trifft auch auf Simulationen aus anderen Bereichen zu. Dazu gehören beispielsweise die Simulation einer Supernovaexplosion, eines Autounfalls oder der Weltwirtschaft. Die lokal verfügbaren Ressourcen sind für diese Simulationen nicht immer ausreichend.

Ein Grid bietet die Möglichkeit ortsverteilte Daten zu sammeln, zu speichern und zu analysieren. Außerdem ermöglicht es, entfernte Rechenressourcen zum Beispiel für Simulationen nutzen zu können. Daher beschäftigen sich eine Vielzahl von Projekten damit, Grids für den Forschungs- und Wissenschaftsbereich zu entwickeln. Das Projekt EGEE (Enabling Grids for E-sciencE) [EGE] hat sich beispielsweise zum Ziel gesetzt, ein Grid für den europäischen Forschungsraum zu schaffen. Weiterhin wird zum Beispiel für die Analyse von Beobachtungsdaten von Wetterdiensten, die in ortsverteilten Archiven gesammelt werden, im gleichnamigen Projekt ein so genanntes C3-Grid (Collaborative Climate Community Data and Processing Grid) entwickelt [CGP].

Der Ursprung der Grid-Technologie liegt im wissenschaftlichen Bereich. Aber auch die Wirtschaft hat immer mehr Interesse an der Entwicklung von Grids. Prozesse, an denen verschiedene Arbeitsgruppen beteiligt sind, sind ein Beispiel für ein Anwendungsgebiet des Grid Computing in der Wirtschaft. So wurde ein Grid-Projekt von der Universität Siegen in Zusammenarbeit mit zwei Partnern aus Industrie und Wissenschaft gestartet, dessen Ziel es ist, eine Grid-basierte Simulation für die Gießerei-Industrie aufzubauen [BS06]. An einer solchen Simulation sind verschiedene Gruppen beteiligt, die an verschiedenen Orten arbeiten. Außerdem ist die Simulation sehr rechenaufwändig. Über die nötigen Ressourcen verfügt ein mittelständisches Unternehmen in der Regel nicht. Mittels eines Grid soll es möglich sein, einen Gießprozess mit Hilfe verteilter Ressourcen zu simulieren und gleichzeitig interaktiv zu analysieren und zu optimieren.
Weiterhin können kleine und mittelständische Unternehmen von einem Grid profitieren. Sie können Dienste in Anspruch nehmen, die von Ressourcen bereitgestellt werden, deren Anschaffung sich für diese Unternehmen nicht rentieren würde. Ein Grund dafür sind zum Beispiel hohe Lizenzgebühren für wenig genutzte Anwendungen.
Ein weiteres Einsatzgebiet der Grid-Technologie gibt es in Unternehmen, die ihre vorhandene IT-Infrastruktur so ausbauen wollen, dass im Unternehmen verteilte Ressourcen transparent genutzt werden können.

In einem Grid werden allgemein Dienste bereitgestellt. Wenn es sich bei den Diensten um Anwendungen handelt, wird oft von einer serviceorientierten Architektur (SOA) gesprochen. Nach [RHS05] definiert SOA zwar noch keine normierte Architektur oder ein klar umrissenes Begriffsgebäude. Aber auf Grund der in [RHS05] nachzulesenden Punkte, die eine SOA identifizieren, kann man ein Grid in vielen Einsatzgebieten als serviceorientierte Architektur bezeichnen. Dies gilt ebenso für Plugin-basierte Systeme wie sie im Kontext dieser Arbeit verwendet werden.

Plugins können Ergänzungen zu einer Applikation sein. Darüber hinaus kann eine Applikation auch selbst ein Plugin sein. Weiterhin kann eine Anwendung ausschließlich aus Plugins aufgebaut sein. Ein minimales Laufzeitsystem verwaltet in diesem Fall lediglich den Lebenszyklus der Plugins. Bekannte Anwendungsbeispiele, die dieses Konzept verwenden, sind NetBeans und die Eclipse-Plattform. Im Kontext dieser Arbeit wird nur diese Umsetzung eines Plugin-basierten Systems relevant sein.
Durch die Verwendung einer Plugin-basierten Architektur verfügt eine Applikation über großes Potential. Eine Anwendung, die zum Beispiel nur eine Basisfunktionalität implementiert, kann durch Plugins zu einer mächtigen Anwendung erweitert werden. Durch das individuelle Hinzufügen von Plugins ist es auch möglich, ein auf die Bedürfnisse des Nutzers zugeschnittenes System aufzubauen.

Plugins können in Abhängigkeit von ihrer Funktionalität rechen- und/oder speicherintensiv sein. Weiterhin können kommerzielle Plugins hohe Lizenzgebühren haben. Es kann auch sein, dass ein Plugin nur innerhalb eines bestimmten Vertrauensbereichs verfügbar ist. Das direkte Ausführen des Plugin außerhalb dieses Bereichs ist damit nicht möglich. Diese Aspekte führen zu der Idee, Plugins in einem Grid als Dienste verfügbar zu machen.
Sowohl die Grid-Technologie als auch das Prinzip der Plugin-basierten Systeme, bei dem eine Anwendung ausschließlich aus Plugins (und einer Laufzeitumgebung) aufgebaut ist, sind noch relativ neue Entwicklungsfelder. Es ist daher die Frage noch offen, wie man Plugins sinnvoll mit Hilfe von Grid Services in einem Grid einbinden kann. Da Plugins aufeinander aufbauen und sich gegenseitig aufrufen können, bleibt beispielsweise zu klären, wie ein auszuführendes Plugin auf andere Plugins zugreifen kann, die im Grid verteilt sind.

## 1.2 Aufgabenstellung

Im Projekt SESIS wird ein schiffbauliches Entwurfs- und Simulationssystem entwickelt. Dessen Basissystem RCE (Reconfigurable Computing Environment) baut auf einer Plugin-basierten Architektur auf. Sowohl das Projekt SESIS als auch dessen Basissystem werden in Kapitel 2.3 beziehungsweise 2.3.1 vorgestellt.

Im Rahmen von RCE soll es möglich sein, dass im Grid verteilte Plugins ausgeführt und verwendet werden können. Dafür sollen Strategien und Ideen entwickelt werden, wie man Plugins in einem Grid als Dienst einbetten kann. Ebenso soll deutlich werden, wie man auf diese Plugins in einem Grid zugreifen kann.
Im Rahmen dieser Masterarbeit soll ein Konzept dafür mit Hilfe von Grid Services er-

stellt werden. Die Realisierungsmöglichkeiten sollen verglichen und eine implementiert werden. Die Implementierung dient im Kontext dieser Arbeit dazu, den Entwurf für die Realisierung der Verteilung eines Plugin-basierten Systems zu verifizieren.

Alle Implementierungen, die innerhalb dieser Arbeit realisiert werden, müssen auf den vorhandenen Technologien aufsetzen. So sollen alle Implementierungen auf dem jetzigen RCE aufbauen und in Java implementiert werden. Des Weiteren ist die Verwendung des Globus Toolkit für die Entwicklung der Grid Services vorgesehen.

Der Schwerpunkt dieser Arbeit liegt auf der Konzeptionierung der Verteilung der Plugins im Grid und auf der serverseitigen Realisierung. Darüber hinaus soll aber auch betrachtet werden, welche Funktionalität clientseitig in die RCE-Plattform integriert werden kann, um dort die im Grid verteilten Plugins zu verwenden.

Aus der Aufgabenstellung ergeben sich neben der Einarbeitung in die Technologien des Grid Computing und der Plugin-basierten Systeme folgende Teilaufgaben:

- Definition der Anforderungen
- Erarbeitung und Vergleich von Realisierungsmöglichkeiten
- Entwurf der als praktikabelste Lösung erkannten Möglichkeit
- Implementierung und Test

## 1.3 Vorgehen und Aufbau der Arbeit

Die Gliederung der Arbeit orientiert sich an dem Vorgehensmodell, welches bei der Realisierung der Arbeit verfolgt wurde und in Abbildung 2 dargestellt ist.

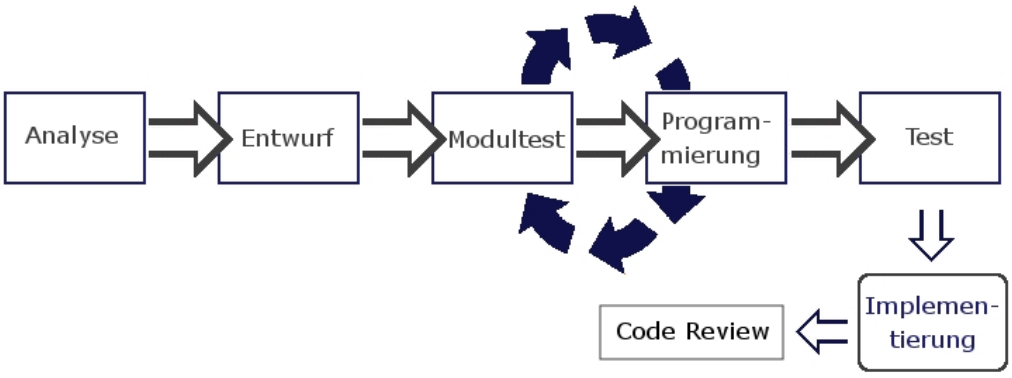


Abbildung 2: Vorgehen bei der Entwicklung

Das Modell ist nach dem Wasserfallmodell in Phasen gegliedert, bei dem die Ergebnisse einer Phase in die folgende einfließen. Das verwendete Modell umfasst die Phasen Analyse, Entwurf, Implementierung und Test. Dabei teilt sich die Implementierung wiederum in die Phasen Modultest und Programmierung auf, welche bis zur

Fertigstellung iterativ durchlaufen werden. Nach der Entwicklung wird der entstandene Quelltext einem Code Review unterzogen, an dem mehrere Personen beteiligt sind. Das Ziel ist die Verbesserung und Qualitätssicherung der Implementierung.

Aus dem beschriebenen Vorgehensmodell ergibt sich die Grobgliederung der Masterarbeit. Nach dieser einleitenden Beschreibung erfolgt die fachliche Einbettung der Arbeit, indem die notwendigen theoretischen Grundlagen gelegt werden. Aus der Aufgabenstellung und den resultierenden Anwendungsfällen werden daraufhin die Anforderungen abgeleitet und beschrieben, welche schließlich in den Entwurf einfließen. Nach einer ausführlichen Darstellung des Entwurfs folgt die Beschreibung der Implementierung und des Tests. Abschließend wird eine Zusammenfassung der Arbeit und ein Ausblick auf weiterführende Tätigkeiten gegeben. Im Anhang ist der Inhalt verschiedener Dateien enthalten, die bei der Implementierung entstanden sind. Sie dienen dem besseren Verständnis einiger Umsetzungsaspekte.

# 2 Grundlagen

Dieses Kapitel behandelt die Grundlagen dieser Masterarbeit. Dabei wird auf die beiden Themenschwerpunkte Grid Computing und Plugin-basierte Systeme eingegangen. Auf Grund des verwendeten Fallbeispiels erfolgt anschließend eine kurze Einführung in das Projekt SESIS und dessen Basissystem RCE.

RCE baut auf den gleichen Technologien und Implementierungen auf, auf denen auch die Eclipse-Plattform basiert. Aus diesem Grund liegt der Fokus bei der Beschreibung Plugin-basierter Systeme auf die darin verwendeten Technologien und Implementierungen.

## 2.1 Grid Computing

Die Vision des *Grid* ist, weltweit verteilte Ressourcen für Problemlösungen einfach nutzen zu können. Um die Einfachheit zu gewährleisten, muss der Zugriff einheitlich erfolgen.
Das Vorbild für diese Technologie ist die elektrische Stromversorgung (engl. *power grid*). Für den Betrieb eines elektrischen Geräts ist es technisch irrelevant, wie der Strom gewonnen wurde (Solar-, Windkraft-, Kohle-, Kernenergie) und wie dieser in die Steckdose gelangt. Der Benutzer des Geräts möchte unter Umständen aber entscheiden, von wem und damit auch aus welcher Quelle er den Strom bezieht. Genauso ist es für den Benutzer einer Grid-Anwendung möglicherweise relevant, mit welchen Rechenressourcen seine Aufgabe gelöst wird. Unbedeutend ist allerdings für ihn, wie die Kommunikation mit diesen Ressourcen funktioniert. Im Nachhinein muss für die Menge an genutztem Strom gezahlt werden. Analog zur Stromabrechnung erfolgt im Grid die Abrechnung an Hand der genutzten Menge an Ressourcen. Diese Vorstellung des Grid ist in Abbildung 3 visualisiert.

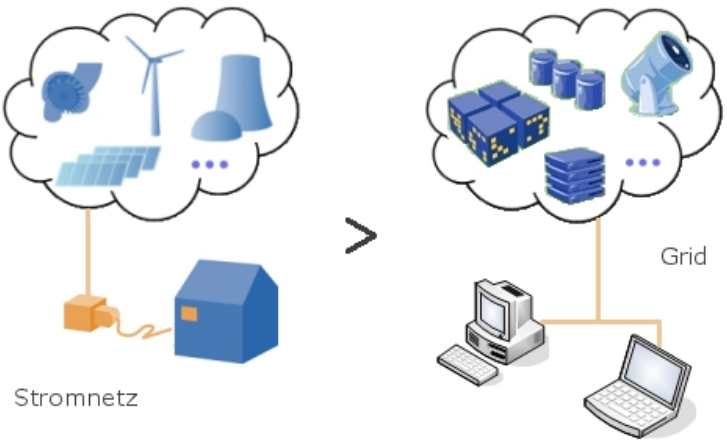


Abbildung 3: Vision des Grid

Es existieren heute eine Vielzahl an Grids unterschiedlicher Dimensionen, die mit verschiedenen Zielen entwickelt wurden und werden. Gemeinsam ist ihnen der

Wunsch nach einer gemeinsamen Problemlösung innerhalb *virtueller Organisationen*. Eine virtuelle Organisation ist ein dynamischer Zusammenschluss von Individuen und/oder Institutionen, die gemeinsame Ziele bei der Nutzung des Grid verfolgen [FKT01].
*Das* eine weltumspannende Grid in der oben beschriebenen Form ist allerdings immer noch Vision. Gründe dafür sind mangelndes Vertrauen gegenüber unbekannten Grid-Teilnehmern und ungenügende Interoperabilität der verschiedenen Grids.

### 2.1.1 Begriffsdefinition

Um den Begriff des Grid definieren und besser verstehen zu können, werden kurz die Anfänge des *Grid Computing* beschrieben.

Das Grid Computing hat seinen Ursprung in den frühen neunziger Jahren. Das Ziel damaliger Experimente war es, Hochleistungsrechner mittels Hochgeschwindigkeitsnetzen so zu koppeln, dass rechen- und/oder speicherintensive Aufgaben gelöst werden können. In diesem Bereich des so genannten Meta Computing entstanden 1995 zwei Projekte.

- Die Zielstellung des Projekts Factoring via Network-Enabled Recursion (*FAFNER*) war es, das Problem der Primfaktorzerlegung großer Zahlen effizient zu lösen. Die Faktorisierung ist leicht parallelisierbar. Die resultierende einfache Implementierung der Teilalgorithmen erlaubte es, auch weniger rechenstarke Computer für die Problemlösung zu nutzen. [DRBJS03]

- Das Projekt Information Wide Area Year (*I-WAY*) hatte zum Ziel, innerhalb eines Jahres 17 existierende Rechenzentren, bestehend aus Hochleistungsrechnern, miteinander zu verbinden [DFP+96]. Die resultierende Rechenkapazität sollte gebündelt von Hochleistungsanwendungen genutzt werden.
  Das Globus-Projekt (Kapitel 2.1.4) hat seinen Ursprung im I-WAY-Projekt [FK97].

Die Anforderungen der Projekte gleichen rudimentären Anforderungen an ein Grid - Nutzen verteilter Ressourcen ohne zentrale Verwaltung. Beide Projekte verliefen sehr erfolgreich. Sie bereiteten damit den Weg für neue Projekte, in denen die gewonnenen Erkenntnisse genutzt werden konnten und welche die Grid-Technologie weiter vorangetrieben haben. Das Konzept des Grid Computing wurde dann erstmals 1998 in dem Buch „The Grid: Blueprint for a New Computing Infrastructure“ [FK98] vorgestellt. Dieses Buch wird auch als die *Grid-Bibel* bezeichnet.

Systeme aus Bereichen des Cluster Computing, Peer-to-Peer Computing, Distributed Computing und andere weisen viele Aspekte der Grid-Technologie auf. *Ian Foster*, ein bedeutender Mitbegründer der Grid-Technologie, stellte schließlich in [Fos02] die Frage:

*„If by deploying a scheduler on my local area network I create a „Cluster Grid“, then doesn’t my Network File System deployment over that same network provide me*

*with a „Storage Grid?“ Indeed, isn't my workstation, coupling as it does processor, memory, disk, and network card, a „PC Grid?“ Is there any computer system that isn't a Grid?“*

Daraufhin hat Foster eine *3-Punkte-Checkliste* aufgestellt, mit der er die Eigenschaften eines Grid spezifiziert. Ein Grid ist nach [Fos02] ein System, dass:

1. Ressourcen koordiniert, die keiner zentralen Instanz untergeordnet sind,
2. standardisierte, offene, allgemeingültige Protokolle und Schnittstellen verwendet,
3. um eine nicht triviale Dienstgüte (QoS) bereitzustellen.

Im Gegensatz dazu gibt [CER] eine kurze und intuitive Antwort auf die Frage: Was ist das Grid?: Während das Web eine Anwendung ist, mit der man Informationen über das Internet austauschen kann, ist das Grid eine Anwendung mit der man Ressourcen über das Internet austauschen kann.
Diese Antwort macht zum einen deutlich, dass die Infrastruktur für das Grid bereits existiert: das Internet. Zum anderen kann man aus dieser Antwort schlussfolgern, dass für das Grid Standards definiert werden müssen, wenn es den Status des Web erreichen soll. Denn dieses konnte sich in der jetzigen Form nur durchsetzen, weil es auf offenen Standards basiert. Die drei grundlegenden sind HTTP als Protokoll, HTML als Dokumentenbeschreibungssprache und URLs als eindeutige Adressbezeichner eines Dokuments.

In den beiden folgenden Kapiteln werden zwei der wichtigsten, bereits definierten Standards für das Grid vorgestellt.

### 2.1.2 Open Grid Services Architecture

Eine Grid-Anwendung besteht in der Regel aus verschiedenen Komponenten, die unterschiedliche Dienste zur Verfügung stellen. Typische Dienste sind zum Beispiel:

- Virtual Organisation Management Service
- Resource Discovery und Management Service
- Job Management Service
- Security Service
- Data Management Service

Die Komponenten verschiedener Grid-Anwendungen interagieren miteinander. So wird beispielsweise der Job Management Service einer Anwendung mit dem Resource Discovery Service einer anderen Anwendung kommunizieren, um Ressourcen zu finden, mit denen eine bestimmte Aufgabe gelöst werden kann.

Das *Open Grid Forum* ist ein Zusammenschluss von Anwendern, Entwicklern und Handelsvertretern, mit dem Ziel Grid-spezifische Standards zu erarbeiten. Einer dieser Standards ist die Open Grid Services Architecture (*OGSA*). Neben anderen Festlegungen definiert OGSA zum einen eine Menge von Services, die eine Grid-Anwendung bereitstellen soll. Zum anderen findet eine Standardisierung dieser Grid Services statt, indem für die Services einheitliche Schnittstellen festgelegt werden. Weiterhin wird durch OGSA definiert, dass die Middleware, auf dem die Architektur basiert, zustandsbehaftet sein muss. Das bedeutet, dass Informationen zwischen zwei Methodenaufrufen eines Service gespeichert werden können.

Obwohl OGSA theoretisch auf jede verteilte Middleware (z.B. CORBA, RMI, selbst RPC) aufbauen kann, wurde schließlich bestimmt, das OGSA auf der Web-Service-Technologie basieren soll. Die Entscheidung fiel unter anderem auf Grund der Tatsache, dass die Schnittstelle von Web Services unabhängig von genutzten Protokollen und auch unabhängig von der Implementierung des Service definiert wird. Darüber hinaus haben sich Web Services bei der Realisierung von internetbasierten Anwendungen mit lose gekoppelten Clients und Server bereits etabliert. Dies ist eine gute Voraussetzung für die Verwendung dieser Technologie für Grid-basierte Anwendungen.

Web Services sind im Allgemeinen zustandslos und genügen demnach nicht den Anforderungen, die OGSA an die Middleware stellt. Im nächsten Kapitel wird die Spezifikation vorgestellt, die beschreibt, wie die zustandsbehafteten Web Services realisiert werden können.

Für weitere Informationen zur Spezifikation von OGSA wird auf [FKS+05] sowie [FKNT03] verwiesen.

### 2.1.3 Web Services Resource Framework

Die Spezifikation des Web Services Resource Framework (*WSRF*) wurde gemeinsam von der Globus Alliance (siehe Kapitel 2.1.4) und IBM entwickelt. Inzwischen ist es von *OASIS* (Organization for the Advancement of Structured Information Standards) als Standard anerkannt worden. OASIS ist die treibende Organisation bei der Entwicklung von Web-Service-Standards. Die Anerkennung als Standard ist wichtig für die Akzeptanz von WSRF in der Web-Service-Gemeinschaft, die noch nicht vollständig erreicht ist.

Die Abbildung 4 verdeutlicht den Zusammenhang zwischen den beiden Spezifikationen OGSA und WSRF, den benötigten zustandsbehafteten Web Services (*Grid Services*) und den zu Grunde liegenden (zustandslosen) Web Services.

Das Web Services Resource Framework beschreibt demnach wie Web Services erweitert werden sollen, damit sie zustandsbehaftet sind und damit dem Anspruch eines Grid Service genügen. Bevor auf die Realisierung dieser Erweiterung eingegangen wird, soll zunächst kurz die Web-Service-Technologie allgemein vorgestellt werden.

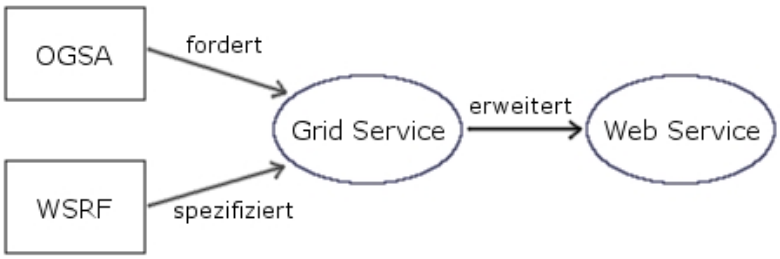


Abbildung 4: Zusammenhang zwischen OSGA, WSRF und Web Services

Ein *Web Service* ist nach der Definition des W3C in [BHM+04] ein System, welches mit dem Ziel entwickelt worden ist, interoperable Interaktion zwischen Maschinen über ein Netzwerk zu unterstützen.
Ein Web Service wird eindeutig über einen Uniform Resource Identifier (URI) identifiziert. Die Schnittstelle eines Web Service wird mit *WSDL* (Web Service Description Language) beschrieben und ist maschinenlesbar. Mit dieser Schnittstellenbeschreibung ist es möglich, *Stubs* sowohl für den Service als auch für den Client zu generieren. Die Aufgabe eines Stub ist es, die Kommunikation zwischen Client und Service abzuwickeln. Damit werden die Client- und die Service-Anwendung auf die Funktionalität reduziert, die sie bereitstellen wollen, weil die notwendige Kommunikationslogik in die Stubs ausgelagert wird. Die Abbildung 5 verdeutlicht die Funktion eines Stub.

Wenn der Client eine Anfrage an den Web Service stellen will, so wird er immer seine Anfrage an seinen Stub delegieren. Dieser wird wiederum den Stub des Service ansprechen, der daraufhin die Anfrage an den eigentlichen Service weiterleitet. Die Antwort des Web Service erfolgt analog.


Abbildung 5: Web-Service-Aufruf durch Client

Die Kommunikation zwischen den Stubs wird im Allgemeinen mit *SOAP* realisiert (ursprünglich Simple Object Access Protocol, inzwischen eigenständig verwendet). Denkbar ist auch XML-RPC. SOAP definiert, wie die auszutauschenden Daten innerhalb einer XML-basierten Nachricht repräsentiert werden. Es kann auf jedes beliebige Transportprotokoll aufsetzen. In der Regel wird auf Grund der Kompatibilität mit üblichen Netzwerkarchitekturen HTTP (Hyper Text Transfer Protocol) benutzt. SOAP hat im Allgemeinen Nachteile im Übertragungsvolumen und Rechenaufwand, was bei der Entwicklung von Grid Services berücksichtigt werden muss [HGS+05].

Die serverseitige Software einer Web-Service-Anwendung ist in Abbildung 6 dargestellt. Die Web Services sind hier in einer SOAP-Engine eingebettet. Es ist üblich, anstatt für jeden einzelnen Web Service Stubs zu generieren und zu nutzen, eine allgemeine Engine zu nutzen, die die Aufgabe der Stubs übernimmt [SC05]. Die SOAP-Engine wiederum wird innerhalb eines Applikationsservers ausgeführt. Stellt dieser keine HTTP-Funktionalität bereit, wird als letzte Instanz ein HTTP-Server benötigt. Beim Globus Toolkit, welches in dieser Arbeit verwendet wird, wird als SOAP-Engine Apache Axis und als Applikationsserver eine Eigenentwicklung von Globus eingesetzt. Diese serverseitige Software wird zusammen als *Container* bezeichnet.

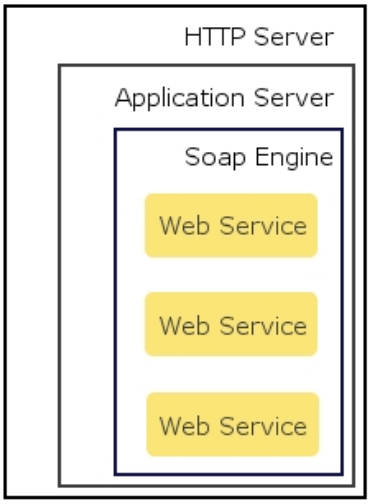


Abbildung 6: Serverseitige Software mit eingebetteten Web Services

Wie bereits erwähnt wurde, sind Web Services zustandslos. Grid Services müssen nach OGSA allerdings in der Lage sein, Informationen zu speichern. Da man die Web-Service-Technologie an sich als Grid-Middleware nutzen will, wäre es nicht sinnvoll, diese im Grundsatz zu ändern. Der Ansatz nach WSRF sieht daher vor, dass der Web Service selbst zustandslos bleibt. In der WSRF-Spezifikation wird eine weitere Komponente definiert: die *Resource*. Innerhalb der Resource werden die Zustandsinformationen in Form von *Resource Properties* (RP) gespeichert. Bei einer Anfrage an einen zustandsbehafteten Web Service muss diesem mitgeteilt werden, welche Resource er nutzen soll. In Abbildung 7 ist das Konzept von WSRF zusammengefasst. Demnach kann für einen Web Service eine beliebige Anzahl an Resources existieren. Ein zustandsloser Web Service bildet zusammen mit einer Resource die *WS-Resource*.

### 2.1.4 Globus Toolkit

Das *Globus Toolkit* ist ein von der *Globus Alliance* entwickeltes, frei verfügbares Softwareprodukt, welches die Entwicklung Grid-basierter Anwendungen unterstützt. Die aktuelle Version 4.0 basiert auf den oben beschriebenen Standards OGSA und WSRF und stellt eine Referenzimplementierung dieser dar.

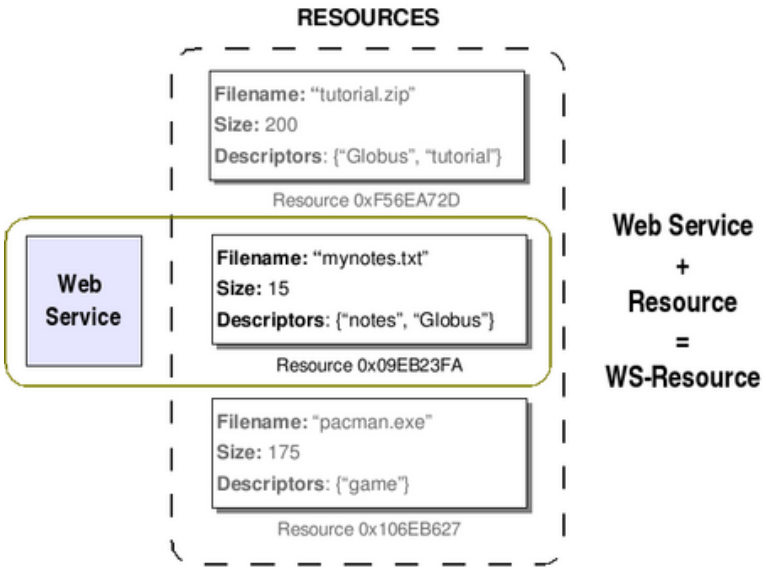


Abbildung 7: WS-Resource [SC05]

Das Globus Toolkit 4.0 (GT4) besteht aus einer Vielzahl von Komponenten. Ein Großteil der Komponenten ist Web-Service-basiert. GT4 stellt mit diesen WS-Komponenten fast alle von OGSA geforderten Grid Services bereit. Damit werden vom Globus Toolkit die notwendigen Bausteine geliefert, die zum Erstellen von Grid-Anwendungen notwendig sind und individuell und unabhängig von einander benutzt werden können.
Die wichtigsten *GT4-Komponenten* sind in Abbildung 8 dargestellt. Dort ist zu sehen, dass eine Einteilung in fünf Gruppen mit folgenden Aufgaben erfolgt [Ins07]:

- **Security**: Authentifizierung und Autorisierung von Benutzern und Ressourcen; Verwaltung von Zugriffsrechten; abgesicherte Kommunikation
- **Data Management**: Lokalisierung, Verschiebung und Verwaltung von im Grid verteilten Daten
- **Execution Management**: Initialisierung, Beobachtung, Planung und Koordination von verteilten Aufgaben
- **Information Services**: Überwachung und Identifikation von Ressourcen und Diensten im Grid
- **Common Runtime**: Laufzeitumgebung zur plattformunabhängigen Implementierung der oben genannten Dienste; Bibliotheken

Neben dem Globus Toolkit gibt es noch weitere Grid-Middleware-Lösungen wie UNICORE [ES01] oder gLite. Sie unterscheiden sich grundsätzlich darin, ob und wie definierte Grid-Standards umgesetzt werden. Während für gLite beispielsweise auch zukünftig keine WSRF-Implementierung vorgesehen ist, wird mit der neuesten Version 6 von UNICORE das WSRF inzwischen implementiert. Allerdings handelt es sich bei UNICORE 6 wiederum bereits um WSRF 1.2 während GT4 noch auf

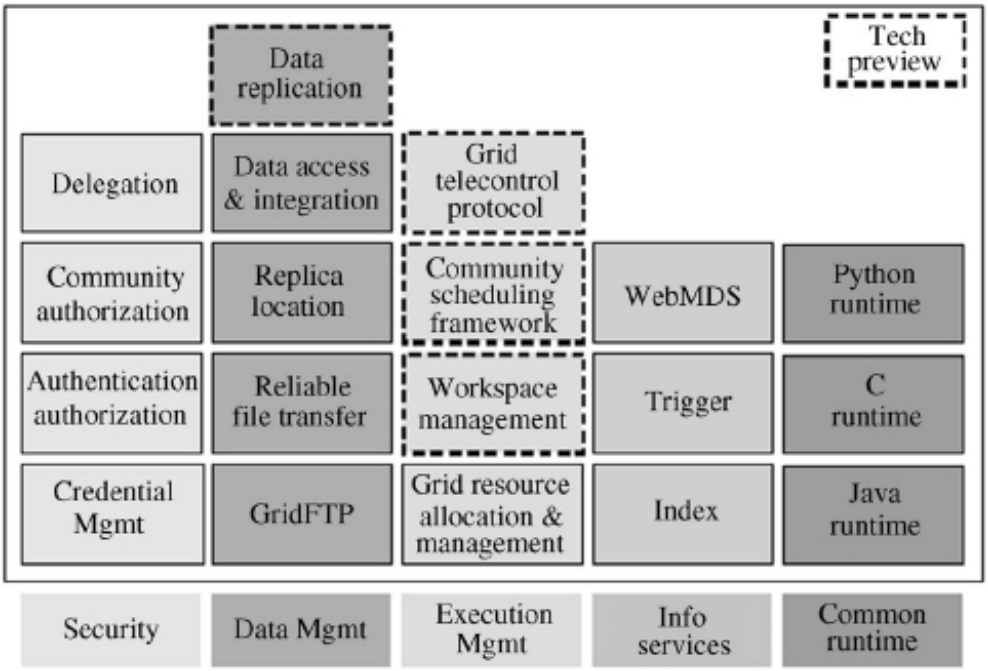


Abbildung 8: Komponenten von Globus Toolkit 4 [Fos06]

WSRF 1.0 basiert. Daraus resultiert, dass die einzelnen Grid-Middleware-Lösungen zum großen Teil nicht interoperabel sind, so dass die Kommunikation zwischen Grids mit unterschiedlicher Middleware schwierig ist. Auch wenn der gleiche Grid-Standard verwendet wird, kann die jeweilige Umsetzung innerhalb der Middleware so unterschiedlich sein, dass wiederum Interoperabilität nicht gegeben ist. Ein Grundgedanke des Grid sieht vor, dass ein Benutzer des Grid sich nicht um das Grid selbst zu kümmern braucht. Das beinhaltet auch, dass es für ihn irrelevant sein soll, welche Grid-Middleware genutzt wird, um auf Ressourcen zuzugreifen. Voraussetzung dafür ist, dass die einzelnen Grid-Middleware-Lösungen interoperabel sind [AHLZ04]. Mit dem bereits abgeschlossen Projekt GRIP (Grid Interoperability Project) wurde die Problematik beispielsweise in Bezug auf Globus und UNICORE angegangen [GP].
In dieser Arbeit wird auf Grund der Aufgabenstellung (siehe Kapitel 1.2) ausschließlich das Globus Toolkit eingesetzt. Folglich wird an dieser Stelle nicht weiter auf andere Grid-Middleware eingegangen.

Im Zusammenhang mit der Abstraktion des Grid sind außerdem die Commodity Grid Kits (*CoG Kits*) zu nennen. Nach [LFGL01] definiert und implementiert ein CoG Kit eine Menge von allgemeinen Komponenten, welche Grid-Funktionalitäten auf weit verbreitete Umgebungen und Frameworks abbilden. *Commodity* bezeichnet hier allgemein Technologien, die außerhalb des Grid-Kontextes oft verwendet werden. CoG Kits bieten demnach einen abstrahierten Zugang zu einem Grid.
Es existieren verschiedene Versionen von CoG Kits (Python, CORBA, Web/CGI, DCOM, Java). In dieser Arbeit wird das Java CoG Kit verwendet. Es unterstützt vor allem die clientseitige Anbindung an das Globus Toolkit, indem es den Zugriff auf verschiedene GT4-Komponenten ermöglicht.

Abschließend ist zu sagen, dass das Globus Toolkit durch die frühzeitige starke Orientierung an definierten Grid-Standards wie OGSA und WSRF eine große Akzeptanz und Verbreitung findet.

## 2.2 Plugin-basierte Systeme

Dieses Kapitel behandelt den zweiten Themenschwerpunkt dieser Arbeit, die Plugin-basierten Systeme. Es beschreibt die OSGi-Technologie, welche die Grundlage der Plugin-basierten Eclipse-Plattform darstellt. Diese wiederum bildet die Basis von RCE, welches in dieser Arbeit als Beispiel eines Plugin-basierten Systems innerhalb einer Grid-Umgebung verteilt werden soll. Aus diesem Grund wird in diesem Kapitel auch eine Einführung in die Technologien der Eclipse-Plattform gegeben.

### 2.2.1 OSGi-Technologie

Die *OSGi*-Spezifikationen definieren eine Serviceplattform, deren Architektur in Abbildung 9 veranschaulicht ist. Die Plattform baut auf einer Java Virtual Machine (JVM) auf. Das Kernstück der Plattform bildet das OSGi-Framework. Seine Aufgabe ist, den Lebenszyklus der integrierten Softwarekomponenten - die so genannten *Bundles* - zu verwalten. Das Hinzufügen und Entfernen von Bundles zur beziehungsweise von der Serviceplattform kann vollständig dynamisch zur Laufzeit und ferngesteuert über Netzwerke erfolgen. Weiterhin bietet die Serviceplattform eine Reihe von Standarddiensten. Weitere Informationen zur OSGi-Serviceplattform und deren Funktionalität findet man in [OSG05].

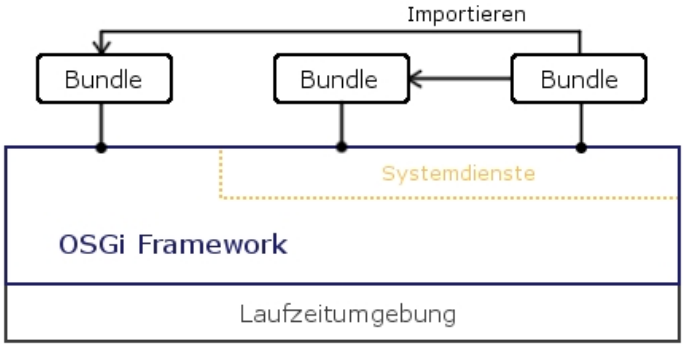


Abbildung 9: Schichtenaufbau einer OSGi-Architektur

Ein Bundle bildet im Rahmen der OSGi-Spezifikationen die Einheit der Modularisierung. Dienste können demnach ausschließlich als Bundles in die Serviceplattform integriert werden. Diese Integration erfolgt als Verzeichnis, welches meist ein Javaarchiv (JAR-Datei) ist. Dieses Verzeichnis beinhaltet Ressourcen, um Funktionalitäten bereitzustellen. Diese Ressourcen können dabei Java-Klassendateien, HTML-Dateien, Bilder, Textdateien und so weiter sein. Darüber hinaus enthält das

Verzeichnis eine Manifestdatei, welche den Inhalt des Verzeichnisses beschreibt und Informationen über das Bundle bereitstellt. Diese Informationen beziehen sich beispielsweise auf den Classpath eines Bundle. Da ein Bundle eine eigenständige, abgeschlossene Applikation darstellt, hat auch jedes Bundle seinen eigenen Classpath. Mit der Manifestdatei ist es dem Framework möglich, alles Wissenswerte über ein Bundle zu erhalten, ohne dessen Ressourcen laden zu müssen.
Wie in Abbildung 9 zu sehen ist, können Bundles auf Funktionen anderer Bundles zugreifen. Ein Bundle kann Java-Packages exportieren (Funktionalität anderen Bundles zur Verfügung stellen) oder importieren (Funktionalität anderer Bundles nutzen), insofern diese dort exportiert werden. Informationen darüber stehen ebenfalls im Manifest des Bundle.

Die Abbildung 10 zeigt den Lebenszyklus eines Bundle. Nach der Installation befindet sich ein Bundle im Zustand *Installed*. Das Framework liest jetzt die Manifestdatei. Können alle dort definierten Abhängigkeiten aufgelöst werden, nimmt das Bundle den Zustand *Resolved* an. Bis hierhin wurden noch keine Ressourcen des Bundle sondern nur sein Manifest geladen. Aus dem Zustand *Resolved* kann es gestartet werden. Dies geschieht aber auch erst, wenn das Bundle tatsächlich benötigt, also seine Funktionalität vom Benutzer oder von anderen Bundles gefordert wird. Wenn es nicht mehr benötigt wird, kann ein Bundle wieder gestoppt und daraufhin auch deinstalliert werden.

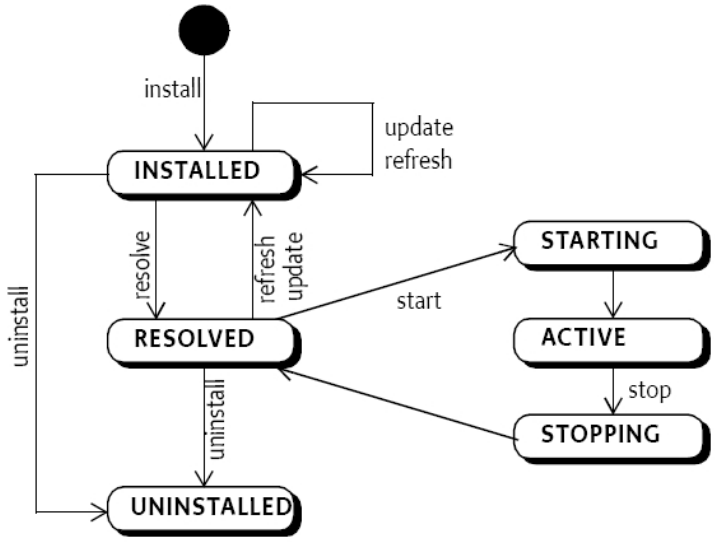


Abbildung 10: Lebenszyklus eines Bundle [OSG05]

Die Anwendungsgebiete der OSGi-Serviceplattform existieren überall dort, wo Dienste in einer standardisierten und einfachen Weise bereitgestellt, dynamisch zusammengestellt und verwaltet werden sollen. Das ursprüngliche Anwendungsgebiet der OSGi-Technologie sind eingebettete Systeme, bei denen Dienste nachträglich zur Laufzeit eingebunden und ausgeführt werden. Es existiert somit die Möglichkeit der Fernsteuerung, -diagnose und -wartung dieser Geräte.

Der Schwerpunkt in der Softwareentwicklung wird zukünftig immer mehr auf der Integration existierender Software in neue Systeme liegen, anstatt diese vollständig

neu zu entwickeln. Mit den OSGi-Spezifikationen kann der Integrationsprozess von Software standardisiert werden [OSG05].
Ein Anwendungsbeispiel dieses Integrationskonzeptes ist die integrierte Entwicklungsumgebung Eclipse, welche auf der OSGi-Implementierung Equinox basiert und im folgenden Kapitel vorgestellt wird. Neben Equinox existieren weitere Implementierungen wie Knopflerfish oder Oscar.

### 2.2.2 Eclipse-Plattform

Die *Eclipse*-Plattform ist eine integrierte Entwicklungsumgebung mit Plugin-basierter Architektur, welche schematisch in Abbildung 11 dargestellt ist. Sie basiert auf dem von der OSGi Alliance spezifizierten Framework und deren Implementierung *Equinox*. Im Kontext von OSGi wird ein Bundle als kleinste funktionstragende Einheit bezeichnet. Equinox erweitert dieses Modell und führt das Konzept der erweiterbaren Plugins ein. Um den Bezug zu OSGi nicht zu verlieren, werden die Plugins logisch als Bundles gruppiert. Ein Bundle kann dabei aus ein bis n Plugins bestehen. Diese Gruppierung ermöglicht es, dass die Implementierung des OSGi-Frameworks, welches nur das Konzept von Bundles kennt, problemlos durch eine andere ersetzt werden kann.

Für ein Bundle im OSGi-Framework sind andere Bundles unbekannt. Folglich kann es diese in ihrem Verhalten nicht beeinflussen. Equinox hebt diese Beschränkung bei Plugins auf, indem es das Konzept der *Extensions* und *Extension Points* implementiert. Die Idee ist, dass sich Plugins gegenseitig funktionell erweitern können. Ein Beispiel ist ein Plugin, welches eine GUI für Benutzereinstellungen realisiert. Es kann sein, dass der Benutzer für die von einem anderen Plugin bereitgestellte Funktionalitäten Einstellungen vornehmen können soll. Um dies zu ermöglichen, muss dieses Plugin das GUI-Plugin so erweitern, dass seine Einstellungen in der GUI mit berücksichtigt werden.

Ein Plugin definiert einen Extension Point, wenn es anderen Plugins ermöglicht werden soll, dieses zu erweitern. Die Definition des Extension Point gibt dabei vor, wie die Erweiterung realisiert werden muss. Andere Plugins, die das Ziel haben, dieses Plugin zu erweitern, müssen ihrerseits eine entsprechende Extension definieren. Die Beziehung zwischen Extensions und Extension Points folgt dem Schlüssel-Schloss-Prinzip. Ein Plugin kann beliebig viele Extension Points und Extensions definieren. Es existieren nun neben den Abhängigkeiten durch exportierte/importierte Java-Packages (auf Ebene der Bundles) auch Abhängigkeiten durch Extension und Extension Points (auf Ebene der Plugins). Informationen darüber werden für jedes Bundle in einer weiteren Beschreibungsdatei (*plugin.xml*) gespeichert.

In Abbildung 11 ist als Basis der Eclipse-Plattform die Laufzeitumgebung zu sehen, welche die Plugins verwaltet. Weiterhin sind die Plugins der Plattform abgebildet, welche dessen Standardfunktionen bereitstellen (*Workspace*, *Debug*, *Team*, ...). Darüber hinaus sind die Plugins dargestellt, welche die Eclipse-Plattform individuell erweitern (*JDT*, *PDE*, *Anwendungen*, ...). Dabei wird das Konzept der Extensions und Extension Points durch die angedeuteten Anschlüsse visualisiert.

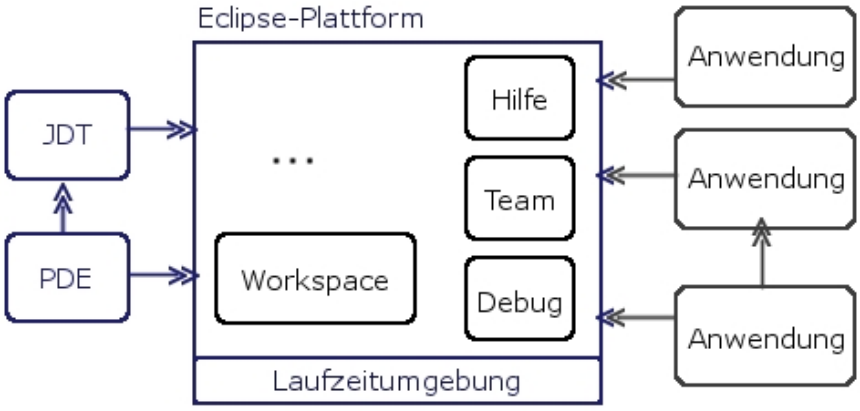


Abbildung 11: Architektur von Eclipse

Durch die Plugin-basierte Architektur der Eclipse-Plattform verfügt diese über großes Potential. Durch das individuelle Hinzufügen von Plugins ist es möglich, ein auf die Bedürfnisse des Nutzers zugeschnittenes System aufzubauen, welches (limitiert durch die Anzahl verfügbarer Plugins) beliebig mächtig sein kann.
Inzwischen existieren für die Eclipse-Plattform zahlreiche Plugins aus den verschiedensten Anwendungsbereichen (Applikationsserver, Datenbanken, UML, Web Services, Entwicklung in Teams, ...)[1].

Für weitergehende Informationen zur Eclipse-Plattform wird auf [Obj03] verwiesen. Einen tieferen Einblick in die Anwendung der OSGi-Technologie bei der Eclipse-Plattform ist in [GHM+05] zu finden.

## 2.3 Projekt SESIS

Ziel des Projektes *SESIS* ist, ein schiffbauliches Entwurfs- und Simulationssystem zu entwickeln, welches in verteilten Umgebungen bestehend aus Unix-, Linux- und Windowssystemen einsetzbar ist [BCE+07]. Das System soll die Zusammenarbeit von Werft und Zulieferern während des Entwurfsprozesses eines Schiffes vereinfachen, indem komplexe, gemeinschaftliche Simulationen ermöglicht werden.

Auf jedem teilnehmenden Rechner innerhalb des (verteilten) Systems läuft ein identisches Basissystem. Dieses Basissystem erhält durch das Nachladen von unterschiedlichen Plugins eine bestimmte Ausprägung.
Die Abbildung 12 zeigt zwei verschieden ausgeprägte Systeme. Es ist jeweils als unterste Schicht das Basissystem namens RCE zu sehen. Darauf aufbauend sind die Plugins abgebildet. Um eine grundlegende Gleichartigkeit der Systeme und damit dessen Funktionalität zu erreichen, sind einige davon Teil jedes Basissystems. Der Rest bestimmt die Ausprägung des Systems.
Auf der rechten RCE-Installation sind (schiffbauspezifische) Anwendungen mittels Wrapper-Technologie als Plugins eingebunden. Rechts ist ein GUI-Plugin installiert,

[1] Eine Referenz für Eclipse-Plugins bietet http://www.eclipseplugincentral.com (Juni 2007)

welches beispielsweise Ergebnisse der links zu sehenden Anwendung Simplorer[2] visualisiert. In diesem Fall hat das rechte System eine Ausprägung als Client und das linke eine als Applikationsserver.

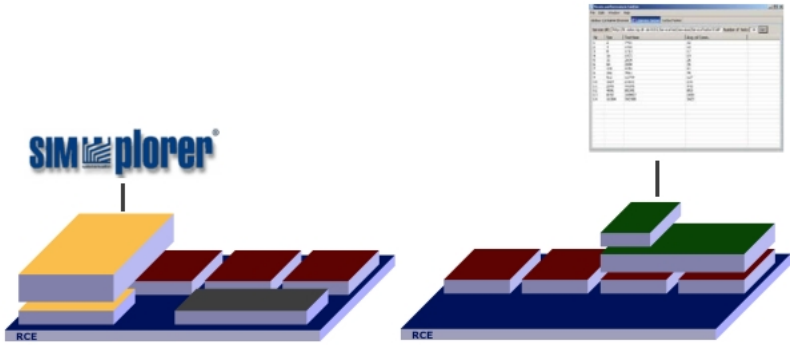


Abbildung 12: Ausprägungen zweier RCE-Installationen

Die Einrichtung Simulations- und Softwaretechnik des Deutschen Zentrums für Luft- und Raumfahrt[3] (DLR) in Köln übernimmt im Rahmen des Projekts SESIS zusammen mit dem Fraunhofer-Institut für Algorithmen und wissenschaftliches Rechnen[4] (SCAI) die Entwicklung des Basissystems, auf dessen Architektur im folgenden Kapitel genauer eingegangen wird.

### 2.3.1 Architektur von RCE

Das Basissystem *RCE* (Reconfigurable Computing Environment) von SESIS stellt die Dienste bereit, die für das Ausführen des gesamten Systems notwendig sind. Es baut auf der Implementierung des OSGi-Framework auf, auf dem auch die in Kapitel 2.2.2 beschriebene Eclipse-Plattform basiert. Alle für RCE notwendigen Erweiterungen gegenüber dieser Implementierung werden in einer zusätzlichen Schicht bereitgestellt, die dem Basissystem seinen Namen gegeben hat [KKFM+07]. Die Architektur von RCE ist in Abbildung 13 gezeigt.

Die Basis der Architektur bilden die Java Virtual Machine (JVM) und die Implementierung Equinox sowie dessen Erweiterung, die RCE-Schicht. Die darauf aufsetzenden, einzelnen Plugins werden gemäß ihrer Einbettung und Funktionalität wie folgt unterschieden:

- **SESIS-Plugins (Grau)**
  Sie bauen auf der RCE-Schicht auf und nutzen Funktionalitäten des gesamten Systems. Sie sind schiffbauspezifisch.
  (z.B. Wrapper für externe Anwendungen)

[2] Analyse- und Simulationsprogramm für technische Systeme
[3] http://www.dlr.de/sc
[4] http://www.scai.fraunhofer.de

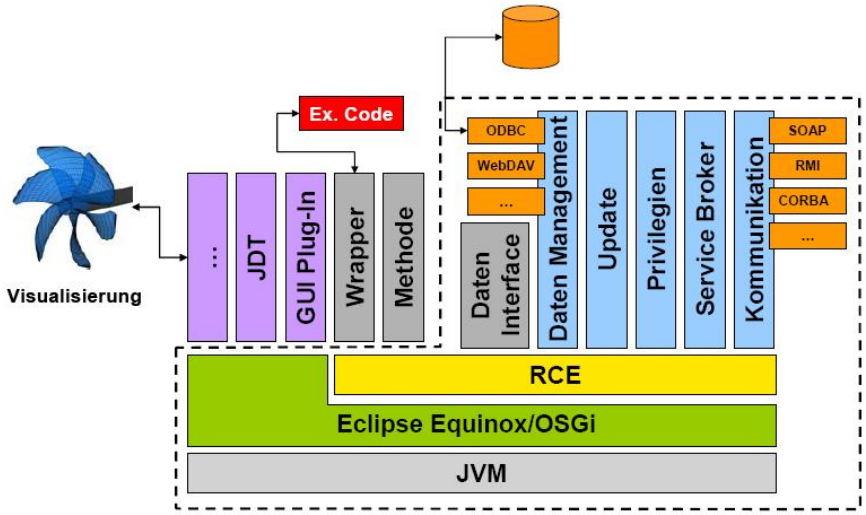


Abbildung 13: Architektur von RCE [KKFM+07]

- **Eclipse-Plugins (Violett)**
  Sie bauen auf der Equinox-Schicht auf und brauchen keine Funktionalitäten des gesamten Systems. Es können demnach vorhandene Plugins für die Eclipse-Plattform in das Gesamtsystem integriert werden.
  (z.B. Java Development Tools (JDT))

- **RCE-Plugins (Blau)**
  Sie bauen auf der RCE-Schicht auf und gehören zum Basissystem. Sie werden zur Laufzeit nur einmal gestartet und sind nicht schiffbauspezifisch.
  (z.B. Plugin zur Kommunikation zwischen den RCE-Instanzen)

Die RCE-Plattform stellt eine *API* (Application Programming Interface) bereit, über die integrierte Plugins auf Funktionen von RCE-Plugins zugreifen können. Die RCE-Plugins stellen die *Basisdienste* Kommunikation, Update, Privilegien, Service Broker und Datenmanagement bereit.

# 3 Anforderungsanalyse

Dieses Kapitel beschäftigt sich mit den Anforderungen, die an die Entwicklung gestellt werden. Dabei wird eine Unterscheidung zwischen funktionalen, nichtfunktionalen und qualitativen Anforderungen vorgenommen. Im Vorfeld werden die bestehenden Anwendungsfälle betrachtet, aus denen sich die Anforderungen ergeben. Abschließend wird eine Abgrenzung hinsichtlich der Umsetzung der Anforderungen bei der Implementierung vorgenommen.

## 3.1 Anwendungsfälle

Das Anwendungsfalldiagramm in Abbildung 14 beschreibt die existierenden Anwendungsfälle und deren Beziehungen untereinander und zu den Akteuren. Mit *Plugin Service* wird im Folgenden das als Grid Service realisierte, verteilte Plugin bezeichnet.

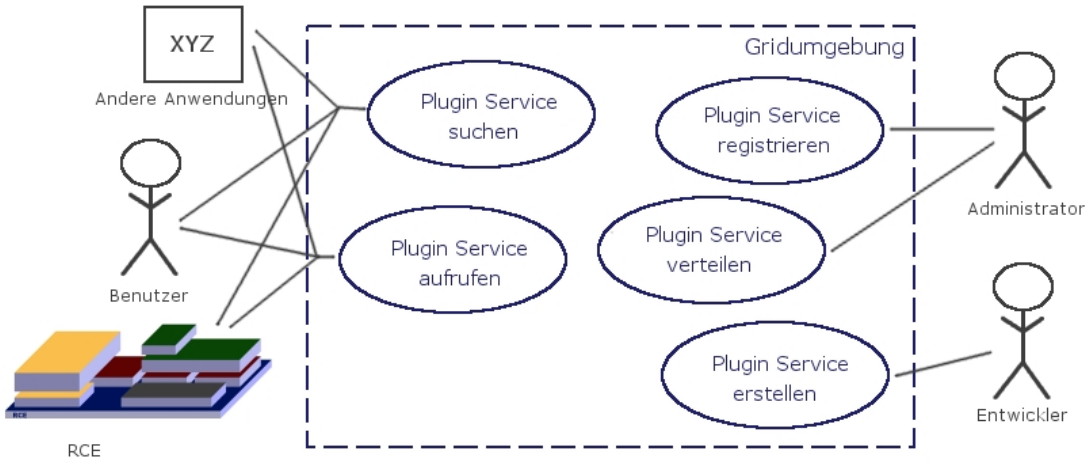


Abbildung 14: Anwendungsfalldiagramm

Es gibt verschiedene Arten von Akteuren:

- **Benutzer**: Person, die Funktionen des Plugin Service nutzt.
- **Entwickler**: Person, die Plugin entwickelt hat.
- **Administrator**: Person, die Plugin Service administriert.
- **RCE**: System, welches Funktionen des Plugin nutzt.
- **Andere Anwendungen**: Systeme, welche auf Plugin Service zugreifen (z.B. GridSphere-Portale und Grid-Clients).

Ein Benutzer kann beispielsweise ein RCE-Anwender sein. In diesem Fall interagiert der Benutzer an sich mit der Grid-Umgebung (mittels RCE). RCE fungiert selbst als

Akteur, wenn zum Beispiel eine entfernte RCE-Instanz eine Anfrage nach einem Plugin stellt, welches für das angefragte RCE nur im Grid als Plugin Service zur Verfügung steht. In diesem Fall interagiert RCE selbstständig mit der Grid-Umgebung. Die Akteure stehen mit verschiedenen Anwendungsfällen in Beziehung.

- **Plugin Service suchen**: Suche nach einem Plugin Service in einem Verzeichnisdienst
- **Plugin Service aufrufen**: Aufruf eines Plugin Service über seine Schnittstellenmethoden
- **Plugin Service registrieren**: Registrieren eines Plugin Service bei Verzeichnisdienst
- **Plugin Service verteilen**: Deployen eines Plugin Service im Globus-Container
- **Plugin Service erstellen**: Implementieren des Plugin Service

## 3.2 Anforderungen

Um die Anwendungsfälle zu realisieren und um der Aufgabenstellung zu entsprechen, müssen die folgenden funktionalen, nicht-funktionalen und qualitativen Anforderungen beim Entwurf berücksichtigt werden.

**Funktional (F)**

- Ein Plugin, welches für die RCE-Plattform entwickelt wurde, muss im Grid mit Hilfe von Grid Services verteilt werden.
- Ein Plugin Service muss alle Funktionalitäten bereitstellen, die das entsprechende Plugin zur Verfügung stellt, wenn es in der RCE-Plattform integriert ist.
- Über die API der RCE-Plattform bereitgestellte Funktionalitäten, die das Plugin nutzt, wenn es dort integriert ist, müssen auch von einem im Grid verteilten Plugin verwendet werden können.
- Ein Plugin Service muss im Grid gefunden werden können.
- Ein Plugin Service muss auch von anderen Anwendungen genutzt werden können.
- Vom zu verteilenden Plugin abhängige Sicherheitsanforderungen müssen berücksichtigt werden.

**Nicht-funktional (NF)**

- Für die Entwicklung von Grid Services muss das Globus Toolkit verwendet werden.

**Qualitativ (Q)**

- Der Prozess des Verteilens eines Plugin im Grid muss so allgemein wie möglich gestaltet sein, damit ein hoher Grad an Wiederverwendbarkeit erreicht wird.
- Es soll gewährleistet werden, dass ein Entwickler mit wenig Kenntnis über Grid-Technologien in der Lage ist, *sein* Plugin als Grid Service zu realisieren.

In der folgenden Tabelle 1 sind die Anforderungen zusammengefasst. Dabei wird jede Anforderung mit einer ID versehen, um in folgenden Kapiteln auf diese referenzieren zu können.

| ID | Anforderung | Art |
|---|---|---|
| 1 | Plugin Service | F |
| 2 | Funktionalität des RCE-Plugin = Funktionalität des Plugin Service | F |
| 3 | RCE-API von Plugin Service bei Bedarf nutzbar | F |
| 4 | Plugin Service auffindbar | F |
| 5 | Plugin Service von anderen Systemen nutzbar | F |
| 6 | Sicherheit gewährleisten | F |
| 7 | Verteilprozess - hoher Grad an Wiederverwendbarkeit | Q |
| 8 | Verteilprozess auf andere Plugins intutitiv anwendbar | Q |
| 9 | Globus Toolkit nutzen | NF |

Tabelle 1: Liste der Anforderungen

## 3.3 Abgrenzung der Implementierung

Bei der Implementierung wird nicht der gesamte Entwurf umgesetzt. Die Implementierung wird die elementaren Anforderungen erfüllen. Einige andere Anforderungen werden nicht berücksichtigt. Zum einen ist dies nicht notwendig, um den Entwurf zu verifizieren. Zum anderen liegt der Schwerpunkt dieser Arbeit auf der Konzeptionierung der Verteilung eines Plugin-basierten Systems innerhalb einer Grid-Umgebung. Letztendlich fehlt auch der notwendige zeitliche Rahmen.

Die Implementierung wird wie folgt abgegrenzt:

- ✓ Ein RCE-Plugin wird als Service im Grid verteilt.
- ✓ Der Plugin Service stellt alle Funktionalitäten des Plugin bereit.
- ✓ Der Plugin Service kann auch von anderen Anwendungen genutzt werden.
- ✓ Bestehende Sicherheitsanforderungen werden berücksichtigt.

- ✗ Über die API der RCE-Plattform bereitgestellte Funktionalitäten können von einem im Grid verteilten Plugin nicht zwangsläufig genutzt werden.
- ✗ Ein Plugin Service wird nicht über einen Verzeichnisdienst im Grid auffindbar sein. Im Rahmen der Implementierung ist die Adresse des Service bekannt.

Die Tabelle 2 fasst die Abgrenzung der Implementierung an Hand der gestellten, funktionalen Anforderungen zusammen.

| ID der Anforderung | Umsetzung |
|---|---|
| 1 | ✓ |
| 2 | ✓ |
| 3 | ✗ |
| 4 | ✗ |
| 5 | ✓ |
| 6 | ✓ |

Tabelle 2: Abgrenzung der Implementierung

# 4 Entwurf

Nachdem die Anforderungen definiert wurden, kann nun der Entwurf erfolgen. Nach Anforderung 1 und 4 aus der Tabelle 1 auf Seite 30 soll die Verteilung eines Plugin mittels eines Plugin Service realisiert werden, der durch einen Verzeichnisdienst gefunden werden kann. Wie in Kapitel 2.1.3 beschrieben wurde, sind Grid Services in einem (Globus-)Container eingebettet. Daraus ergibt sich für die Verwendung des Plugin Service im Allgemeinen das in Abbildung 15 dargestellte Einsatzszenario. Es wird hier der Fall betrachtet, dass von einer RCE-Installation ein im Grid verteiltes Plugin verwendet werden soll. Web Services haben die Eigenschaft, dass bei deren Verwendung Implementierungsdetails sowohl client- als auch serverseitig verborgen bleiben (siehe Kapitel 2.1.3). Folglich hat es keine Auswirkungen auf das dargestellte Szenario, wenn die RCE-Installation durch eine beliebige andere Anwendung ersetzt wird. Somit ist mit diesem Szenario die Anforderung 5 unwillkürlich erfüllt.

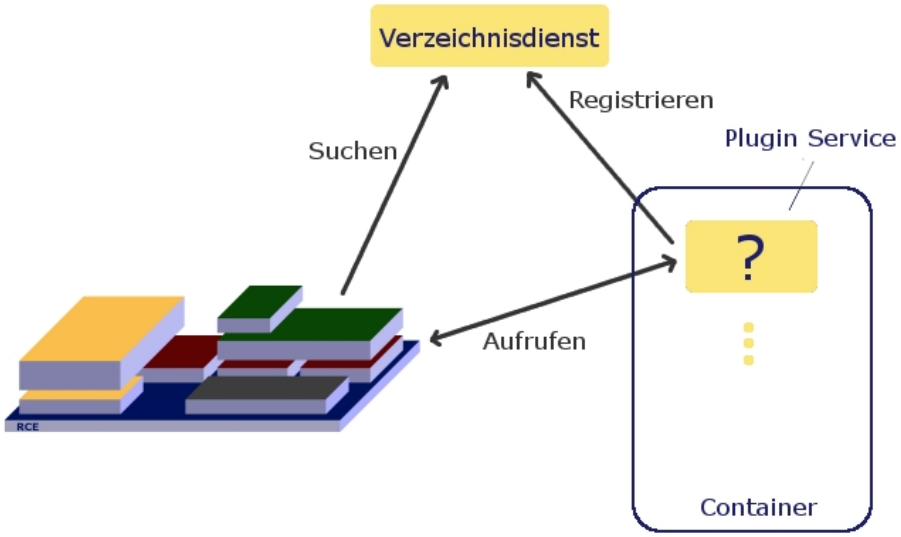


Abbildung 15: Einsatzszenario

Falls die Adresse des Plugin Service nicht bekannt ist, muss diese zunächst bei einem Verzeichnisdienst erfragt werden. Daraufhin kann die Kommunikation mit dem Plugin Service erfolgen und auf Funktionen des Plugin zugegriffen werden. Wie in der Abbildung deutlich wird, ist grundsätzlich die Frage noch offen, wie die serverseitige Umsetzung konkret aussieht. Aus diesem Grund wird im Folgenden die serverseitige Realisierung betrachtet.

## 4.1 Realisierungsmöglichkeiten für einen Plugin Service

Wie schon in der Aufgabenstellung in Kapitel 1.2 erwähnt wurde, gibt es verschiedene Möglichkeiten für die Realisierung des Plugin Service.

1. **Plugin als Grid Service**
   Das zu verteilende Plugin wird als Grid Service realisiert. Die Service-Implementierung enthält die Logik des Plugin.

2. **RCE als Grid Service**
   RCE inklusive der integrierten und zu verteilenden Plugins wird als Grid Service realisiert. Die Service-Implementierung enthält die Logik von RCE und der integrierten Plugins.

3. **Reduziertes RCE als Grid Service**
   Das zu verteilende Plugin und ein Teil von RCE, der vom Plugin benutzt wird, wird als Grid Service realisiert. Die Service-Implementierung enthält die Logik des reduzierten RCE und des zu verteilenden Plugin.

4. **Grid Service als Kommunikationsschnittstelle zu RCE und Plugin**
   RCE und die integrierten Plugins werden aus der Service-Umgebung ausgelagert. Die Kommunikation zwischen Client und Plugin beziehungsweise RCE wird über den Grid Service realisiert. Die Service-Implementierung enthält daher weder die Logik des zu verteilenden Plugin noch die von RCE. Es dient ausschließlich als Kommunikationsschnittstelle.

In den nachfolgenden Kapiteln werden die vier Realisierungsmöglichkeiten genauer vorgestellt. Neben individuellen Betrachtungen werden bei jeder Möglichkeit folgende Aspekte untersucht:

- **RCE-API**: Bei diesem Aspekt stellt sich die Frage, ob die API von RCE genutzt werden kann.
- **Resource Properties**: Es wird die Anzahl und die Komplexität der notwendigen Resource Properties für den Plugin Service betrachtet. Mit den Resource Properties wird der Zustand eines Grid Service gespeichert. Dieses Konzept wurde in Kapitel 2.1.3 erläutert.
- **Plugin-Implementierung**: Die Frage hierbei ist, inwieweit die Implementierung des Plugin bei der Realisierung des Plugin Service verändert werden muss. Dies kann zum Beispiel notwendig sein, wenn innerhalb des Plugin auf Resource Properties zugegriffen werden muss.
- **Plugin-Service-Schnittstelle**: Bei diesem Aspekt wird die Komplexität der Schnittstelle des Plugin Service betrachtet, die mittels WSDL definiert wird.

Die Auswahl der Aspekte erfolgte unter anderem auf Basis der in der Tabelle 1 definierten Anforderungen. Dabei sind von den funktionalen Anforderungen nur diejenigen relevant, die vom Plugin Service erfüllt werden müssen. Das sind Anforderung 2 und 3. Um die Anforderung 3 zu erfüllen, muss die Frage geklärt werden, ob die API von RCE vom verteilten Plugin genutzt werden kann. Ist dies nicht möglich, muss die Implementierung des Plugin angepasst werden, um die Anforderung 2 erfüllen zu können.

Aus der geforderten Intuitivität und Wiederverwendbarkeit des Verteilprozesses (nicht-funktionale Anforderungen 7 und 8) folgt, dass die Schnittstelle des Plugin Service nicht zu komplex und nicht zu sehr auf einen Plugin Service zugeschnitten sein sollte. Weiterhin ist es aus diesem Grund erstrebenswert, dass sich die zu definierenden Resource Properties nicht zu stark von Plugin Service zu Plugin Service unterscheiden, um erneute Definitionen zu vermeiden. Auch die Implementierung des Plugin sollte bei der Realisierung des Plugin Service so wenig wie möglich verändert werden müssen, um Intuitivität und Wiederverwendbarkeit zu erreichen. Demzufolge ist auch die Anzahl der Resource Properties möglichst klein zu halten.

Mit den Realisierungsaspekten kann zum einen untersucht werden, ob Anforderungen erfüllt werden. Zum anderen sind sie dazu da, den Aufwand und die Praktikabilität der Realisierung abzuschätzen. Die Reihenfolge bei der obigen Aufzählung der Aspekte gibt dabei deren Gewichtung an. So ist die Möglichkeit, die RCE-API nutzen zu können, grundlegend. Die Definition der Service-Schnittstelle ist dagegen weniger kritisch, da sie grundsätzlich einem Schema folgt und nicht unrealisierbar komplex werden kann. Die Relevanz der Aspekte der Resource Properties und der Plugin-Implementierung hängen vom Umfang der Properties beziehungsweise vom Ausmaß der notwendigen Änderungen der Implementierung ab. Je größer der Umfang oder das Ausmaß ist, desto kritischer ist der Aspekt zu bewerten.

### 4.1.1 Vorüberlegungen

Um die Gedanken zu den einzelnen Realisierungsmöglichkeiten zu verstehen, ist es notwendig, zuvor einige Vorüberlegungen zu betrachten.

Der Einstiegspunkt jedes Plugin ist seine Schnittstelle. Zurzeit liegen die Schnittstellenbeschreibungen für die RCE-Plattform ausschließlich als Java-Interfaces vor. Um eine Programmiersprachenunabhängigkeit zu erreichen, ist es angedacht, dass die Plugin-Schnittstelle zusätzlich mit WSDL beschrieben wird. Das bedeutet für die Realisierung eines Plugin Service, dass nur über die Methoden der Plugin-Schnittstelle hinaus gehende Servicemethoden zusätzlicher Aufwand bei der Definition der Serviceschnittstelle sind.

Es ist bei der Betrachtung der Realisierungsmöglichkeiten zu berücksichtigen, dass es verschiedene Arten von Plugins gibt. Dies betrifft zum einen die bereitgestellten Funktionen und die damit zusammenhängenden zu speichernden Zustandsinformationen. Für einen Plugin Service, dessen zu Grunde liegendes Plugin im einfachsten Fall die Grundrechenarten als unabhängige Funktionen anbietet, sind Überlegungen zur Definition der Resource Properties vernachlässigbar. Zum anderen beziehen sich die Unterschiede auf das Nutzen von RCE-API-Funktionalitäten. Eine externe Anwendung, die mittels eines Wrapper in RCE als Plugin eingebunden wird, wird kaum RCE-API-Funktionen nutzen, da diese der externen Anwendung ursprünglich nicht zur Verfügung standen. Demnach ist es beispielsweise unwahrscheinlich, dass von einem solchen Plugin aus andere Plugins aufgerufen werden. Dafür ist die Anwendung zu eigenständig und wegen des externen Code zu sehr gekapselt.

Im Endeffekt geht es bei der Realisierung eines Plugin Service darum, die Technologie der Grid Services mit der Plugin-Technologie zu vereinen. Dabei wird es letztendlich eine Frage der Philosophie sein, welches technologische Konzept dominieren soll.

### 4.1.2 Plugin als Grid Service

Die nahe liegendste Idee, um das Ziel zu erreichen, ein Plugin als Service im Grid zu verteilen, ist, die vorhandene Plugin-Implementierung direkt als Serviceimplementierung einzusetzen. Das bedeutet, das Plugin wird zum Grid Service. Die mit WSDL zu beschreibende Serviceschnittstelle entspricht folglich exakt der bereits definierten Plugin-Schnittstelle. In Abbildung 16 wird die Idee aufgezeigt.

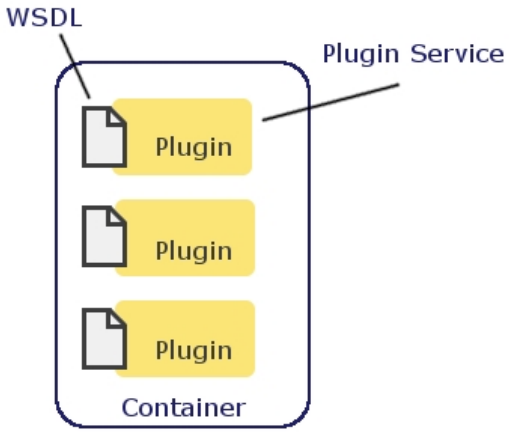


Abbildung 16: Plugin als Grid Service

Es wird deutlich, dass das Plugin bei dieser Realisierung von der RCE-Plattform isoliert wird. Demzufolge kann auf die API von RCE nicht zugegriffen werden. Beinhaltet die Plugin-Implementierung API-Aufrufe, so hat dies zur Folge, dass die Implementierung geändert werden muss. Die Aufrufe müssen entfernt und gegebenenfalls ersetzt werden.
Handelt es sich um ein zustandsbehaftetes Plugin, müssen Resource Properties definiert werden. Diese sind für jeden Plugin Service individuell, weil der Zustand jedes Plugin durch andere Informationen repräsentiert wird. Auch der Umfang variiert von Plugin zu Plugin und kann unter Umständen sehr groß sein. Ist die Definition von Resource Properties notwendig, muss innerhalb der Plugin-Implementierung das Laden und Speichern der Properties realisiert werden.

Die Tabelle 3 fasst die Aspekte dieser Realisierung zusammen.

Bei dieser Variante dominiert das Konzept der Web-Service-Technologie. Das Plugin wird selbst zum Grid Service und verliert die Verbindung zu *seiner* Plattform und anderen Plugins. Damit geht der Plugin-Charakter verloren.

| Aspekt | Ausprägung |
|---|---|
| RCE-API | nicht nutzbar |
| Resource Properties | individuell, u.U. sehr umfangreich |
| Plugin-Implementierung | wegen API und RP eventuell anzupassen |
| Schnittstelle (WSDL) | Plugin-Schnittstelle |

Tabelle 3: Zusammenfassung der Realisierungsaspekte für: Plugin als Grid Service

### 4.1.3 RCE als Grid Service

Wenn Plugins auf die API von RCE nicht zugreifen können, steht dies unter Umständen in Konflikt mit der Anforderung 3. Um dieses Problem zu beheben, kann man die RCE-Plattform zusammen mit dem Plugin als Grid Service realisieren. Dargestellt ist die Idee in Abbildung 17.

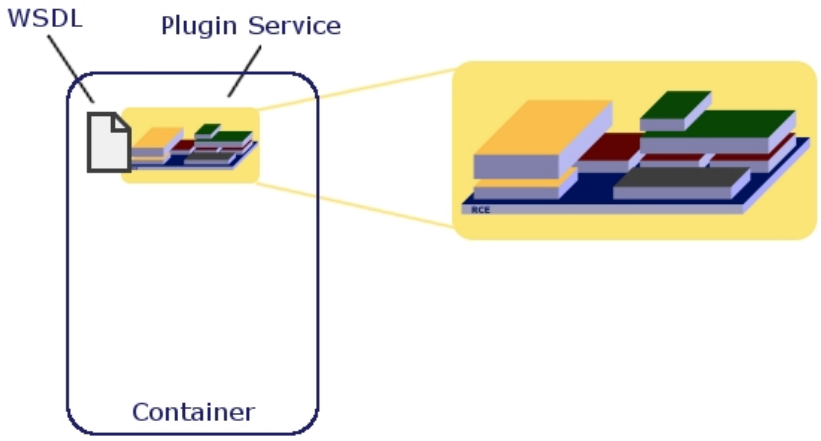


Abbildung 17: RCE als Grid Service

Bei dieser Realisierung verlässt das Plugin *seine Welt* nicht. Es bleibt in RCE integriert. Folglich ist die Nutzung der RCE-API gewährleistet. Die Plugin-Implementierung bleibt diesbezüglich daher unbeeinflusst.
Die zu definierenden Resource Properties sind auf Grund von RCE umfangreich. So müssen zum Beispiel für den Notificationdienst eine Liste der angemeldeten Erzeuger und Verbraucher von Notifications vorgehalten werden. Es ist allerdings zu berücksichtigen, dass die Properties für RCE statisch sind und nur einmal definiert werden müssen unter der Voraussetzung, dass RCE wenig bis keinen Änderungen dauerhaft unterworfen sein wird.
Mit den Properties für das Plugin verhält es sich genauso wie bei der ersten Realisierungsvariante. Sie sind individuell und können unter Umständen umfangreich

sein. Es muss somit die Implementierung des Plugin und in diesem Fall auch die von RCE hinsichtlich des Ladens und Speicherns der Resource Properties angepasst werden.

Die Schnittstelle des Plugin Service entspricht bei dieser Realisierung nicht direkt der des Plugin. Sie umfasst einerseits Methoden für den Lebenszyklus des Plugin. Da das Plugin in die RCE-Plattform integriert ist, muss es explizit für die Benutzung gestartet und danach gestoppt werden. Andererseits müssen in der Serviceschnittstelle die Schnittstellenmethoden des Plugin repräsentiert sein. Man kann sich dabei an dem Vorgehen orientieren, welches angewandt wird, wenn Plugins einer RCE-Instanz Plugin-Methoden einer entfernten RCE-Installation aufrufen. Alle Aufrufe von entfernten Methoden erfolgen über eine *call*-Methode, mit deren Parameter alle notwendigen Informationen wie Methodenname, Methodenparameter und so weiter übergeben werden. Adaptiert man dieses Vorgehen, so besteht die Serviceschnittstelle aus Methoden für den Lebenszyklus und einer *call*-Methode. Der Nachteil dieser Schnittstellenumsetzung ist, dass man die Schnittstelle des Plugin im Vorhinein kennen muss, um die *call*-Methode korrekt aufrufen zu können. Da diese Voraussetzung bei anderen Anwendungen, die den Plugin Service nutzen können sollen, nicht gegeben sein muss, ist die Anforderung 5 nicht erfüllt. Demnach muss die Plugin-Schnittstelle zwangsläufig direkt in der Serviceschnittstelle berücksichtigt werden. In diesem Fall müssen die Aufrufe des Plugin Service intern in Aufrufe der *call*-Methode transformiert werden, um RCE schließlich damit anzusprechen.
Eine Zusammenfassung der Aspekte für diese Realisierung zeigt die Tabelle 4.

| Aspekt | Ausprägung |
|---|---|
| RCE-API | nutzbar |
| Resource Properties | sehr umfangreich (RCE: statisch, Plugins: individuell) |
| Plugin-Implementierung | wegen RP eventuell anzupassen |
| Schnittstelle (WSDL) | Lebenszyklus + Plugin-Schnittstelle o. *call*-Methode |

Tabelle 4: Zusammenfassung der Realisierungsaspekte für: RCE als Grid Service

Bei dieser Möglichkeit der Realisierung kann der Plugin Service eine Menge an Implementierung enthalten, die nicht benötigt wird, da für die Ausführung eines Plugin meist nur ein Teil von RCE benötigt wird. Dieser Teil ist allerdings für jedes Plugin individuell. Während ein Plugin über die API auf das Datenmanagement zugreifen will, benötigt ein anderes Plugin die API, um entfernte Plugins aufzurufen. Indem mehrere Plugins in einer RCE-Plattform integriert werden, wird der Anteil an überflüssiger Implementierung reduziert. Es werden folglich verschiedene Plugins zu einem Plugin Service zusammengefasst. Der Nachteil dabei ist, dass eine Vermischung von unabhängigen Anwendungen beziehungsweise deren Schnittstellenmethoden innerhalb der Serviceschnittstelle stattfindet. Folglich ist auch das Ziel, ein Plugin als Plugin Service zu verteilen, nur unvollständig erreicht.

### 4.1.4 Reduziertes RCE als Grid Service

Die bei der vorherigen Realisierung bemängelte Vermischung verschiedener Plugins innerhalb eines Plugin Service wird mit der Idee beseitigt, die in Abbildung 18 visualisiert ist.

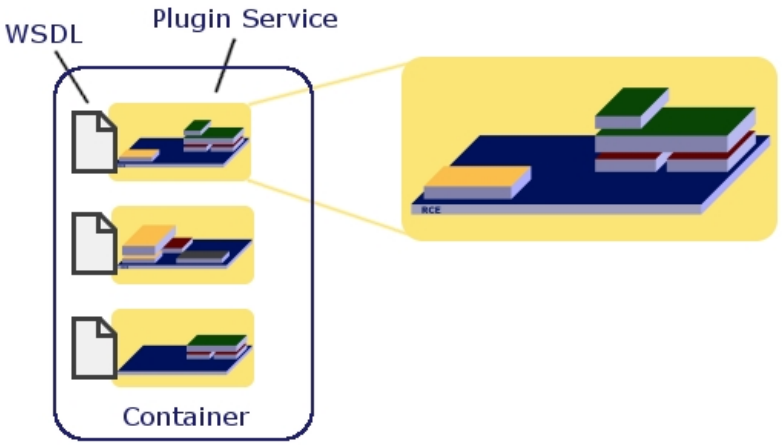


Abbildung 18: Reduziertes RCE als Grid Service

Anstatt die gesamte Plattform als Grid Service zu realisieren, wird hier ein reduziertes RCE, ein so genanntes *RCE** eingesetzt. Jedem Plugin Service wird nur ein Plugin zugeordnet. Beim RCE* sind damit nur noch die RCE-Plugins enthalten, die von dem integrierten Plugin tatsächlich benötigt werden. Die Reduzierung des RCE führt dazu, dass gleichzeitig der Umfang der Resource Properties reduziert wird. Damit verringern sich auch die notwendigen Änderungen innerhalb der Implementierung von RCE*. Der Umfang der Änderungen innerhalb der Plugin-Implementierung bleibt im Vergleich zur vorherigen Variante natürlich unverändert.
Da die vom Plugin benötigten API-Funktionen durch RCE* zur Verfügung stehen, bleibt dessen Implementierung auch hier diesbezüglich unbeeinflusst.
Die Schnittstellenthematik stellt sich ähnlich dar wie bei der vorherigen Realisierungsmöglichkeit. Da RCE so reduziert wurde, dass es nur noch Funktionen für das Plugin bereitstellt und keine Verwaltung von Plugins mehr übernimmt, sind Methoden für den Lebenszyklus des Plugin in der Serviceschnittstelle allerdings nicht mehr notwendig.

Wiederum werden die Aspekte der Realisierung in der Tabelle 5 zusammengefasst.

Bei dieser Realisierungsvariante ist zu berücksichtigen, dass für jedes zu verteilende Plugin ein entsprechendes, reduziertes RCE zur Verfügung gestellt werden muss. Dieser zusätzliche Aufwand kann abhängig vom Grad der Reduzierung hoch sein.

| Aspekt | Ausprägung |
|---|---|
| RCE-API | Relevantes nutzbar |
| Resource Properties | umfangreich (RCE*: statisch, Plugin: individuell) |
| Plugin-Implementierung | wegen RP eventuell anzupassen |
| Schnittstelle (WSDL) | Plugin-Schnittstelle |

Tabelle 5: Zusammenfassung der Realisierungsaspekte für: Reduziertes RCE als Grid Service

### 4.1.5 Grid Service als Kommunikationsschnittstelle zu RCE und Plugin

Ein Nachteil aller bisher betrachteten Realisierungsmöglichkeiten ist die notwendige und teils sehr aufwändige Definition der Resource Properties und die damit verbundenen Änderungen von bereits vorhandenen Implementierungen. Dieser Nachteil kann nur ausgeräumt werden, wenn sowohl RCE als auch die Plugins ausgelagert werden und nicht mehr Teil des Grid Service sind. In Abbildung 19 wird das Prinzip dieser Idee veranschaulicht.

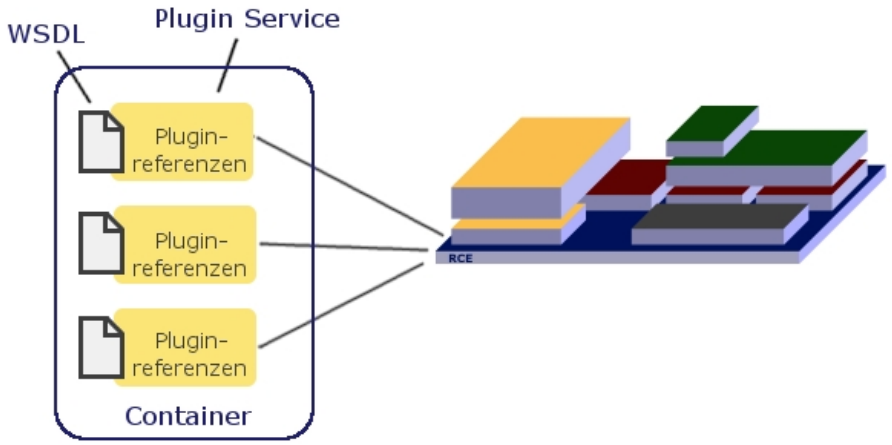


Abbildung 19: Grid Service als Kommunikationsschnittstelle zu RCE und Plugin

Der Plugin Service übernimmt hier die Funktion einer Kommunikationsschnittstelle zwischen aufrufenden Clients und dem ausgelagerten RCE sowie den dort integrierten Plugins. Innerhalb des Plugin Service werden lediglich Referenzen zu dem Plugin gehalten. Der Zustand eines Plugin Service wird jetzt ausschließlich über diese Referenz beschrieben. Somit beschränkt sich die Definition der Resource Properties auf eine Zustandsinformation.

Wie bei der zweiten Realisierungsmöglichkeit (RCE als Grid Service) verbleibt das Plugin in *seiner Welt*. Demzufolge ist die RCE-API nutzbar und die Plugin-

Implementierung bleibt unbeeinflusst.
Die Schnittstelle des Plugin Service besteht aus Methoden für den Lebenszyklus der Plugins, die wieder notwendig sind, weil die Plugins in die RCE-Plattform integriert sind. Weiterhin enthält die Schnittstelle die Schnittstellenmethoden des Plugin.

In Tabelle 6 sind die Aspekte der Realisierung zusammengestellt.

| Aspekt | Ausprägung |
|---|---|
| RCE-API | nutzbar |
| Resource Properties | übersichtlich |
| Plugin-Implementierung | unbeeinflusst |
| Schnittstelle (WSDL) | Plugin-Schnittstelle + Lebenszyklus |

Tabelle 6: Zusammenfassung der Realisierungsaspekte für: Grid Service als Kommunikationsschnittstelle zu RCE und Plugin

Der Container und RCE können auf zwei JVMs (Java Virtual Machine) laufen. In diesem Fall muss für die Kommunikation zwischen Plugin Service und der RCE-Plattform gesorgt werden. Ein denkbares Protokoll hierfür ist RMI (Remote Method Invocation).
Um diese zusätzliche Kommunikation über ein Protokoll einzusparen, können der Container und RCE auf einer JVM ausgeführt werden. In diesem Fall kann der Plugin Service direkt RCE ansprechen.

Im Gegensatz zu der ersten Realisierung (Plugin als Grid Service) tritt bei dieser letzten Möglichkeit das Konzept der Web Services etwas in den Hintergrund. Die Implementierung der Funktionen, die der Service über seine Schnittstelle anbietet, befindet sich nicht im Service selbst, sondern ist in eine andere Anwendung ausgelagert worden.

### 4.1.6 Fazit

Es gilt eine Realisierungsmöglichkeit auszuwählen, welche auf Grund ihrer Praktikabilität im Folgenden detaillierter entworfen und schließlich umgesetzt wird. Hierzu werden die vorgestellten Aspekte der einzelnen Möglichkeiten verglichen und ausgewertet. Bevor die ausführliche Betrachtung folgt, bietet die Tabelle 7 einen ersten Überblick, in der die vier Aspekte für die vier Möglichkeiten kurz symbolisch bewertet werden.

Bei der Realisierung 1 (Plugin als Grid Service) und 3 (Reduziertes RCE als Grid Service) entspricht die Schnittstelle des Plugin Service der des Plugin. Dadurch stellt die Definition der Serviceschnittstelle keinen zusätzlichen Aufwand dar, wenn

| Aspekt / Realisierung | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| RCE-API | - - | ++ | ++ | ++ |
| Resource Properties | - - | - - | - | ++ |
| Plugin-Implementierung | o | o | o | ++ |
| Plugin-Service-Schnittstelle | ++ | - | + | + |

Tabelle 7: Vergleich der Realisierungsmöglichkeiten hinsichtlich der Aspekte

die Plugin-Schnittstelle zukünftig auch als WSDL-Beschreibung vorliegt. Bei den Möglichkeiten 2 (RCE als Grid Service) und 4 (Grid Service als Kommunikationsschnittstelle zu RCE und Plugin) beinhaltet die Serviceschnittstelle zusätzlich Methoden für den Lebenszyklus des Plugin. Diese beschränken sich auf eine Art *start*- und *stop*-Methode. Sie sind für jedes Plugin identisch und müssen folglich nur einmal mit WSDL definiert werden. Demnach ist der zusätzliche Aufwand für diese Serviceschnittstellen vernachlässigbar und damit die Realisierungsmöglichkeiten 2 und 4 weiterhin genauso praktikabel wie 1 und 3.

Ein weiterer Aspekt bei der Realisierung ist die Möglichkeit, dass Plugins die RCE-API nutzen können. Aus der Anforderung 3 folgt, dass die API für Plugins zur Verfügung stehen muss, welche auf dessen Funktionalität zugreifen.
Unter den Realisierungsmöglichkeiten ist die erste die Einzige, bei der die API-Funktionen nicht nutzbar sind. Folglich ist es ausschließlich bei der ersten Realisierung notwendig, dass die Plugin-Implementierung auf Grund von API-Aufrufen verändert werden muss. Nach Anforderung 7 und 8 soll der Verteilprozess wiederverwendbar und intuitiv sein. Dies kann mit der Notwendigkeit von individuellen Änderungen der Plugin-Implementierung nicht erfüllt werden. Demzufolge ist die Realisierungsmöglichkeit 1 nur für Plugins praktikabel, die auf keine von RCE bereitgestellten Funktionen zugreifen. Dies betrifft vor allem externe Anwendungen, welche mittels Wrapper in RCE integriert sind.

Der letzte Aspekt, der ausgewertet werden muss, betrifft die Resource Properties, welche für einen Plugin Service definiert werden müssen, um Zustände des Service zu speichern. Resource Properties werden innerhalb der WSDL-Beschreibung des Service definiert und müssen in XML-Format repräsentiert werden. Um komplexe Datentypen (z.B. Java-Objekte) als Resource Properties zu speichern und zu laden, sind jeweils Serialisierer beziehungsweise Deserialisierer zu definieren. Weiterhin muss an jeder Stelle in der Plugin-Implementierung, an der Zustandsdaten benötigt oder verändert werden, Quelltext zum Laden beziehungsweise Speichern der Resource Properties eingefügt werden. Bei der Realisierungsvariante 2 und 3 betrifft dies auch die Implementierung von RCE beziehungsweise RCE*.
Es wird deutlich, dass Resource Properties mit einem erhöhten Arbeitsaufwand verbunden sind. Je umfangreicher und komplexer die Properties sind und je öfter auf

sie innerhalb der Implementierung zugegriffen werden muss, desto größer ist der Aufwand und desto unpraktikabler ist die Realisierungsmöglichkeit.
Während bei der Realisierungsmöglichkeit 4 die Resource Properties vernachlässigbar sind, sind sie bei der Variante 2 und 3 umfangreich. Die Properties für RCE und RCE* sind statisch. Ebenso müssen Implementierungsänderungen für RCE sowie für RCE* nur einmal vorgenommen werden. Aus diesem Grund bleiben beide Möglichkeiten trotz des Umfangs der Properties weiterhin praktikabel. Die für die Plugins notwendigen Resource Properties sind dagegen für jedes Plugin individuell. Weiterhin muss auch die Änderung der Plugin-Implementierung individuell vorgenommen werden. Mit Verweis auf die Anforderungen 7 und 8 sind diese beiden Realisierungsmöglickeiten nur für Plugins denkbar, die entweder zustandslos sind oder deren Zustand mit wenig Informationen beschrieben werden kann. Bei der Realisierungsvariante 1 verhält es sich mit den für ein Plugin notwendigen Resource Properties genauso. Demnach gilt diese Aussage ebenso für die erste Möglichkeit.

Durch Ausschlussverfahren hat sich die vierte Realisierungsmöglichkeit (Grid Service als Kommunikationsschnittstelle zu RCE und Plugin) hinsichtlich der allgemeinen Verwendbarkeit als die Praktikabelste erwiesen. Sie ist unabhängig von der Art eines Plugin sinnvoll verwendbar. Außerdem wird die Anforderung, dass der Verteilprozess wiederverwendbar und intuitiv ist, am Besten erfüllt, da für die Verteilung eines Plugin kaum individuelles Vorgehen gefordert ist.

Für eigenständige Plugins, die nicht auf externe Funktionalität angewiesen und dazu zustandslos sind, ist die erste Realisierungsmöglichkeit allerdings am Geeignetsten. Bei vorliegender WSDL-Beschreibung der Plugin-Schnittstelle ist kein zusätzlicher Aufwand bei der Umsetzung notwendig.

Der Aufwand für die Umsetzung der Realisierungsmöglichkeiten 2 und 3 ist auch bei zustandslosen Plugins wegen der notwendigen Resource Properties für RCE und RCE* und wegen des Implementierungsaufwands für RCE* groß. Daher ist in jedem Fall den beiden Möglichkeiten die Variante 4 vorzuziehen, auch wenn es sich um ein eigenständiges Plugin handelt.

Da sich die Realisierung der Verteilung eines Plugin nicht auf bestimmte Arten von Plugins beschränken soll, wird die Realisierungsmöglichkeit, Grid Service als Kommunikationsschnittstelle zu RCE und Plugin, ausgewählt und im Folgenden detaillierter entworfen.

## 4.2 Entwurf der Realisierungsmöglichkeit: Grid Service als Kommunikationsschnittstelle zu RCE und Plugin

In diesem Kapitel wird der Entwurf der im vorherigen Kapitel ausgewählten Realisierungsmöglichkeit (Grid Service als Kommunikationsschnittstelle zu RCE und Plugin) beschrieben. Dabei wird zunächst die Realisierung des Plugin Service betrachtet. In diesem Zusammenhang wird auf die Problematik eingegangen, wie RCE-Funktionen auf Grid-Technologien abgebildet werden können, so dass diese auch clientseitig nutzbar sind. Weiterhin wird auf die Frage nach der Sicherheit einge-

gangen. Außerdem werden die Möglichkeiten für das Bekanntmachen eines Plugin Service untersucht. Abschließend werden die einzelnen Schritte des Verteilprozesses genannt und erläutert.

### 4.2.1 Plugin Service

Zwischen Plugin Service und zugehörigem, verteilten Plugin besteht eine 1:1-Beziehung. Zwischen Plugin Service und der Instanz eines Plugin besteht dagegen eine 1:n-Beziehung. Die einzelnen Plugin-Instanzen werden innerhalb des Service durch die Resources referenziert. Jede Resource enthält eine Referenz auf eine Plugin-Instanz in Form einer Resource Property. Zwischen Plugin Service und Resources existiert folglich auch eine 1:n-Beziehung, so dass ein Plugin Service beliebig viele Resources haben kann. Daraus ergeben sich für die serviceseitige Realisierung zwei Aufgaben. Zum einen müssen Resources erzeugt werden können, wenn Anfragen an den Plugin Service gestellt werden. Zum anderen müssen die bereits bestehenden Resources verwaltet werden, so dass bei einem späteren Aufruf des Plugin Service eine bestimmte Resource wiederverwendet werden kann.

#### *4.2.1.1 WS-Resource Factory Pattern*

Im Bereich der Softwareentwicklung gibt es verschiedene Entwurfsmuster (*Design Patterns*). Einige davon beschreiben den Konstruktionsprozess von Objekten in objektorientierten Programmiersprachen. Sie werden unter dem Begriff Erzeugungsmuster (*Creational Patterns*) zusammengefasst. Das Ziel von Erzeugungsmustern ist, die Konstruktion von Objekten zu generalisieren, so dass eine Systemunabhängigkeit erreicht wird [GHJV95].
Nach [SC05] gibt die WSRF-Spezifikation mit dem *WS-Resource Factory Pattern* (WSRFP) vor, dass für die Erzeugung der Resources das Muster *Fabrikmethode* verfolgt werden soll. Bei der Fabrikmethode wird eine Schnittstelle für die Erzeugung eines Objekts definiert. Diese Schnittstelle ist der einzige Zugriffspunkt, mit dem ein bestimmtes Objekt konstruiert werden kann. Es erfolgt eine Trennung von Konstruktion und Repräsentation eines Objekts. Diese grobe Beschreibung des Erzeugungsmusters Fabrikmethode reicht aus, um das Prinzip des WS-Resource Factory Pattern nachvollziehen zu können. Für eine ausführliche Beschreibung von Erzeugungs- und Entwurfsmustern allgemein wird auf [GHJV95] verwiesen.

Die Umsetzung der Fabrikmethode als WS-Resource Factory Pattern zeigt die Abbildung 20.

Wie in Kapitel 2.1.3 beschrieben wurde, bildet der Plugin Service zusammen mit einer Resource die WS-Resource. Die WS-Resource wird über eine so genannte *Endpoint Reference* (EPR) adressiert. Die EPR beinhaltet einerseits die URL des Plugin Service und andererseits den eindeutigen Bezeichner einer Resource. Damit der Plugin Service beim Aufruf weiß, welche Resource er verwenden soll, muss ein Client den Service immer mit einer EPR ansprechen. Um eine EPR und damit eine neue Resource für einen Aufruf des Plugin Service zu erhalten, muss der Client

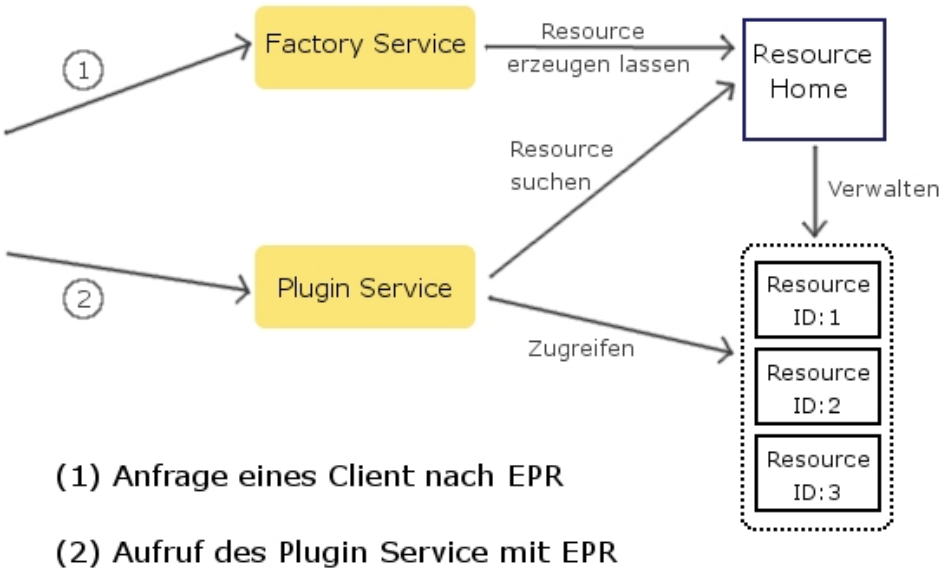


Abbildung 20: WS-Resource Factory Pattern nach [SC05]

zunächst die Methode *createResource* des zum Plugin Service gehörenden *Factory Service* aufrufen. Die Methode erzeugt mittelbar eine neue Resource und liefert eine entsprechende EPR zurück. Der Factory Service bildet nach dem Erzeugungsmuster Fabrikmethode die Schnittstelle für die Erzeugung der Resources.

In Abbildung 20 wird neben dem Erzeugen von Resources auch deren Verwaltung durch das *Resource Home* verdeutlicht. Es hat zum einen die Aufgabe, auf Anfrage des Factory Service eine neue Resource zu erzeugen. Zum anderen dient es dem Plugin Service dazu, eine bestehende Resource zu finden, welche für einen Aufruf genutzt werden soll. Das erfolgt an Hand des eindeutigen Bezeichners, welcher in der EPR enthalten ist. Für solche Zwecke hält das Resource Home eine Liste aller instanziierten Resources vor.

#### *4.2.1.2 Objektorientierter Entwurf*

Die Anforderung 7 verlangt einen hohen Grad an Wiederverwendbarkeit des Verteilprozesses. Das betrifft auch die Realisierung des Plugin Service. Die Implementierung des vorgestellten WSRFP unterscheidet sich für verschiedene Plugin Services nicht grundsätzlich. Damit sich der individuell zu erstellende Quelltext auf ein Minimum reduziert, wird so viel Funktionalität wie möglich in abstrakte Klassen ausgelagert. Durch Ableiten kann diese für jeden Plugin Service wiederverwendet werden.

Das Klassendiagramm in Abbildung 21 zeigt, wie das WS-Resource Factory Pattern implementiert wird. Es sind zum einen die abstrakten Klassen (helles Orange) dargestellt, welche zum Großteil auf dem Java-WS-Core von Globus aufbauen. Die Globus-spezifischen Oberklassen und Schnittstellen sind in dem Klassendiagramm der Übersicht wegen nicht berücksichtigt. Zum anderen sind am Beispiel des Plugin Simplorer die Unterklassen der abstrakten Klassen (Orange) dargestellt, die für die Realisierung des Simplorer Service notwendig sind. Das Klassendiagramm enthält

eine Klasse *SimplorerService* beziehungsweise deren generalisierte Klasse *PluginService*. Die Benennung resultiert aus dem umgesetzten WSRFP. Mit der Realisierung des Plugin Service ist allerdings weiterhin die gesamte serviceseitige Realisierung gemeint.

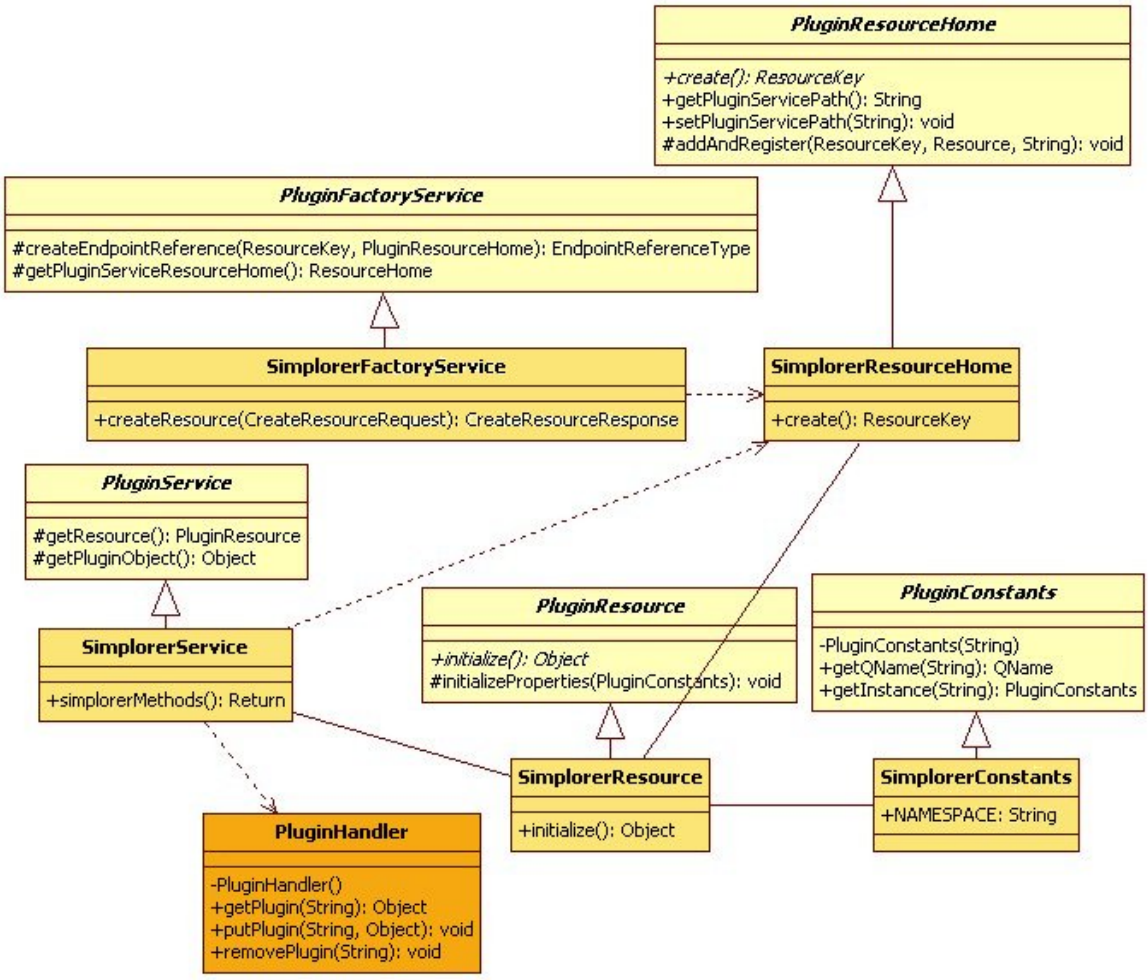


Abbildung 21: Klassendiagramm für die Implementierung des WS-Resource Factory Pattern

Im linken Teil sieht man die Klassen für den Simplorer Service und dessen Factory Service. Die Klassen beinhalten die Implementierungen der jeweiligen Serviceschnittstelle. Diese beschränkt sich beim Factory Service auf die Methode *createResource*, welche eine neue Resource kreiert und eine entsprechende EPR zurückliefert. Beim Plugin Service sind die Methoden abhängig von der Schnittstelle des Plugin.
Gleichzeitig erben beide Klassen von *ihren* abstrakten Klassen Methoden, deren Funktionalität von den Service-Klassen beziehungsweise deren Schnittstellenmethoden benötigt werden. So muss der Plugin Service die zu verwendende Resource erfragen und darüber die in den Resource Properties gespeicherte Referenz der Plugin-Instanz erhalten können. Für den Factory Service ist es wichtig, dass er das für das Erzeugen von Resources zuständige Resource Home erfragen und nutzen sowie eine EPR erstellen kann.

Die Klasse *SimplorerFactoryService* ist folglich abhängig von der Klasse *SimplorerResourceHome*. Sie enthält die Implementierung, die für das Erzeugen von Simplorer Resources notwendig ist. Für die allgemeine Verwaltung von Resources erbt sie entsprechende Methoden von der abstrakten Klasse *PluginResourceHome*.

Die Klasse *SimplorerService* hat eine Assoziation zur Klasse *SimplorerResource*, da der Simplorer Service bei Anfragen auf eine bestimmte Simplorer Resource zugreifen muss. Die Klasse *SimplorerResource* stellt eine Methode für die Initialisierung seiner Resource Properties bereit. Falls die Resource Properties nur die Plugin-Instanzreferenz enthalten sollen, kann die geerbte Methode *initializeResourceProperties* genutzt werden, um die Properties zu initialisieren.

Mit den vorgestellten Klassen ist das WS-Resource Factory Pattern implementiert. Darüber hinaus existieren für die serviceseitige Realisierung zwei weitere Klassen. Eine davon ist der *PluginHandler*, welche alle aktuell genutzten Plugin-Instanzen als Liste vorhält. Jede Plugin-Instanz wird über einen eindeutigen Schlüssel innerhalb der Liste identifiziert. Dieser Schlüssel stellt die Referenz einer Plugin-Instanz dar, welche wie oben beschrieben als Resource Property gespeichert wird. Um die zu verwendende Plugin-Instanz zu erhalten, muss die Klasse *SimplorerService* mit der in der Resource gespeicherten Referenz die Methode *getPlugin* der Klasse *PluginHandler* aufrufen.
Theoretisch ist es möglich, die Plugin-Instanz direkt als Resource Property zu speichern und den Umweg über eine Referenz und einen Handler einzusparen. Das würde allerdings bedeuten, dass die Resource Properties und damit die Implementierungen der Klasse *Resource* für jeden Plugin Service zwangsläufig unterschiedlich sind. Dies widerspricht dem geforderten hohen Grad an Wiederverwendbarkeit. Aus diesem Grund wurde das Konzept des Handler entwickelt. Darüber hinaus entfällt damit auch die unter Umständen aufwändige Definition der Resource Properties in XML Schema innerhalb der WSDL-Beschreibung des Plugin Service.

Der Instanziierungsprozess der Klasse *PluginHandler* folgt dem Erzeugungsmuster *Singleton*. Das bedeutet, dass es nur eine Instanz der Klasse gibt. Die Klasse bietet einen globalen Zugriffspunkt, über den auf ihre Instanz zugegriffen werden kann [GHJV95]. Somit wird erreicht, dass es eine zentrale Liste der Plugin-Instanzen gibt. Der *PluginHandler* ist so allgemein gehalten, dass er für die Realisierung jedes Plugin Service verwendet werden kann. Aus diesem Grund ist die Klasse separat mit einem dunklen Orange dargestellt.
Die andere Klasse, *SimplorerConstants*, definiert Full Qualified Names (FQN), die innerhalb des Simplorer Service verwendet werden. Benutzt ein Plugin Service einen Standardsatz an FQN, so kann die Oberklasse *PluginConstants* genutzt werden. Der Standardsatz umfasst die Bezeichnung für das Resource-Properties-Dokument und die für die Resource Property (Plugin-Instanzreferenz).

#### *4.2.1.3 Lebenszyklusverwaltung von Resources und Plugin-Instanzen*

Wie oben beschrieben wurde, wird eine Resource über den Factory Service erzeugt. Es stellt sich die Frage, wann sie wieder zerstört wird. Gleichzeitig muss der Lebenszyklus der Plugin-Instanzen gesteuert werden. Eine Plugin-Instanz steht in

unmittelbarer Verbindung zu einer Resource, in der die Referenz auf die Instanz gespeichert wird. Demnach ist es sinnvoll, den Lebenszyklus einer Plugin-Instanz an den Lebenszyklus der zugehörigen Resource zu koppeln. Folglich wird eine Plugin-Instanz bei der Erzeugung der Resource kreiert und bei dessen Zerstörung automatisch gelöscht.

Wenn der Globus-Container gestoppt wird, werden zwangsläufig alle Resources zerstört. Da es in den meisten Fällen allerdings erstrebenswert ist, Resources individuell zu zerstören, ist diese Möglichkeit nicht praktikabel. WSRF stellt zwei unterschiedliche Konzepte zur Lebenszyklusverwaltung bereit. Das eine Konzept sieht vor, dass eine Resource durch Aufrufen einer *destroy*-Methode durch den Client explizit zerstört wird. Beim anderen wird beim Erzeugen der Resource eine Zeit festgelegt, nach der die Resource zerstört wird. Diese Zeit muss seitens des Client mittels der Methode *setTerminationTime* immer wieder erneuert werden, wenn die Resource erhalten bleiben soll.

Der Lebenszyklus von Plugins wird innerhalb von RCE explizit durch Aufrufen von den Methoden *start* und *stop* gesteuert. Es spricht nichts dagegen, dieses Konzept für verteilte Plugins zu adaptieren und eine Resource explizit zu zerstören, wenn die entsprechende Plugin-Instanz nicht mehr verwendet wird.
Um dieses Lebenszykluskonzept zu realisieren, muss lediglich die Plugin-Service-Schnittstelle um die Methoden erweitert werden, die WRSF für diese Lebenszyklusverwaltung zur Verfügung stellt. WSRF bietet darüber hinaus weitere Schnittstellenerweiterungen an, um typische Problemstellungen eines Grid-Service-Entwicklers abzudecken. Bis auf die Verwaltung des Lebenszyklus von Resources sind die Problemstellungen für jeden Plugin Service individuell, so dass an dieser Stelle nur auf die Möglichkeit der Schnittstellenerweiterung seitens WSRF verwiesen wird.

#### *4.2.1.4 Zusatz*

In der Schnittstellenbeschreibung des Plugin Service werden mit WSDL zum einen die Methoden definiert, welche der Service anbietet. Dies beinhaltet auch die Beschreibung der Übergabeparameter und Rückgabewerte. Zum anderen werden die Resource Properties definiert. Ein Beispiel für die WSDL-Beschreibung des Simplorer Service ist im Anhang A auf Seite 73 zu finden. Eine detaillierte Erläuterung des WSDL-Schnittstellendokuments erfolgt in Kapitel 5.1.2.

Wie in Kapitel 4.1.5 angedeutet wurde, können RCE und der Globus-Container auf zwei JVMs oder gemeinsam in einer ausgeführt werden. Bei der ersten Variante müsste ein Protokoll für die Kommunikation genutzt werden. Ein Kommunikationsprotokoll impliziert, dass zwei entfernte Systeme interagieren. Da es vorgesehen ist, dass RCE und der Globus-Container auf einem Rechner laufen, ist die Nutzung eines Kommunikationsprotokolls überflüssiger Overhead. Aus diesem Grund wird die Variante mit einer JVM vorgezogen.

Im Folgenden wird der Einfachheit halber nicht explizit zwischen Factory Service und Plugin Service unterschieden, sondern stattdessen weiterhin nur der Begriff Plugin Service verwendet.

### 4.2.2 Abbilden von RCE-Funktionen auf Grid-Technologien

Ein Service bietet Leistungen an, die von Clients abgefragt werden können. Ein Service agiert daher nicht, sondern reagiert immer nur. Sendet ein Client eine Anfrage (*Request*), reagiert der Service mit einer Antwort (*Response*). Die Kommunikation ist somit bidirektional und vom Client abhängig.

Nutzt ein Plugin über die API von RCE dessen Basisdienste, so kann es notwendig sein, dass serviceseitig eine Kommunikation initiiert werden muss. Ein Beispiel ist der Notification-Service von RCE, bei dem sich Plugins als Produzenten von Notifications registrieren lassen können. Wird von einem Produzenten eine Notification versendet, so wird diese an alle Plugins weitergeleitet, die sich für sie als Konsumenten angemeldet haben.
Fungiert das Plugin eines Plugin Service als Produzent, so sollten dessen Notifications auch clientseitig verfügbar gemacht werden. Die *RCE-Welt*, aus der die Notification hervorgeht, bleibt hinter dem Plugin Service verborgen. Demnach müssen solche RCE-Funktionen, die sich über den Plugin Service hinaus erstrecken, auf Grid-Technologien abgebildet werden. Im Folgenden soll das Vorgehen am Beispiel des Notification-Service erläutert und daraufhin für andere Anwendungsfälle verallgemeinert werden.

Das Globus Toolkit realisiert ein Konzept, mit dem ein Grid Service Notifications an dafür registrierte Clients senden kann. Es gilt nun, das Konzept der RCE-Notifications auf dieses abzubilden. Die Problematik veranschaulicht die Abbildung 22.

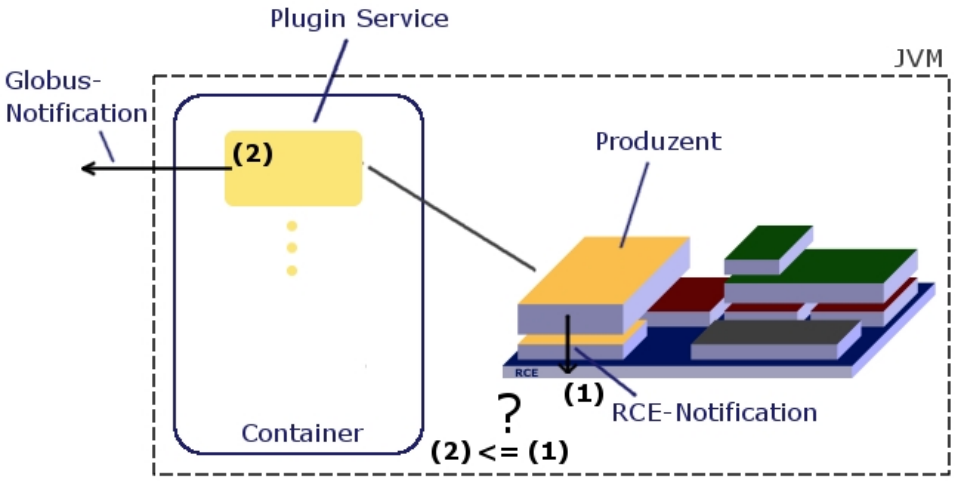


Abbildung 22: Abbilden von RCE-Notifications auf Globus-Notifications

Es wird angenommen, dass sich das Plugin des Plugin Service als Produzent von Notifications bei RCE registriert hat und mittels einer API-Methode Notifications versendet. Die erste Möglichkeit, das Problem zu lösen, ist, dass sich der Plugin Service bei RCE als Konsument für diese Notification anmeldet. Empfängt er eine RCE-Notification, so wird er diese als Globus-Notification weiter versenden. Der Vorteil dieser Lösung ist, dass die Implementierung der API unverändert bleibt. Der

Nachteil ist, dass eine Notification erst über Umwege den endgültigen Konsumenten erreicht.
Die zweite Möglichkeit ist, die Implementierung der API anzupassen. Dabei bleibt die API selbst unverändert, so dass sich für das Plugin nichts ändert. Es wird die API-Methode *sendNotification* so umgeschrieben, dass bereits an dieser Stelle eine Globus-Notification generiert wird.

Ein weiteres Beispiel, bei dem RCE-Funktionen auf Grid-Technologien abgebildet werden müssen, ist das Datenmanagement. Fordert das Plugin eines Plugin Service über den Datenmanagement-Service von RCE eine Datei an, welche außerhalb dieser RCE-Instanz gespeichert ist, kann diese Datei beispielsweise mittels Grid-FTP besorgt werden. Die Abbildung beider Technologien kann wiederum durch Überschreiben der API-Implementierung von RCE an entsprechender Stelle erfolgen.

Das Prinzip, die API-Implementierung zu ändern, ist allgemein anwendbar, um RCE-API-Funktionen auf Grid-Technologien abzubilden. Bei der Realisierung eines Plugin Service wird daher dieses Vorgehen verfolgt. Abbildung 23 bietet zusammenfassend eine Veranschaulichung.

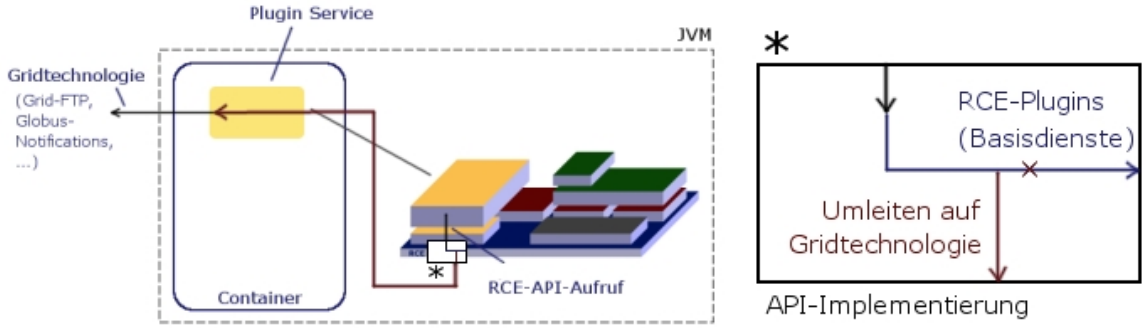


Abbildung 23: Abbilden von RCE-Funktionen auf Grid-Technologien

Wie in Kapitel 3.3 bereits festgelegt wurde, wird bei der Implementierung ein im Grid verteiltes Plugin nicht zwangsläufig alle RCE-Funktionalitäten nutzen können. Der Grund ist, dass das oben entworfene Prinzip, die API-Implementierung anzupassen, zunächst nicht implementiert wird. Ein Grund ist, dass die API noch Veränderungen unterworfen sein wird, so dass zu diesem Zeitpunkt keine vollständige Umsetzung erfolgen kann. Außerdem beschränkt der begrenzte Zeitrahmen der Arbeit den Umfang der Implementierung.

### 4.2.3 Sicherheit

Wird ein Grid Service nicht nur in geschlossenen Anwendungsumgebungen wie Intranets oder Extranets verwendet, stellt sich zwangsläufig die Frage nach der Sicherheit beim Aufruf des Grid Service bezüglich der Daten, die ausgetauscht werden.
Bei der Anforderungsanalyse wurde mit der Anforderung 6 festgelegt, dass Sicherheit bei der Verwendung des Plugin Service gewährleistet werden muss. Die Sicherheitsanforderungen an einen Plugin Service sind abhängig vom Plugin. Während ein

Plugin sensible Daten verarbeitet, ist es für ein anderes Plugin wichtig, dass es nur von bestimmten Personen genutzt wird. Für ein drittes Plugin, welches beispielsweise als einfacher Taschenrechner fungiert, ist die Frage nach Sicherheit dagegen eher unkritisch. Aus diesem Grund kann an dieser Stelle kein allgemeingültiges Sicherheitskonzept entworfen werden, sondern lediglich vorgestellt werden, was im Rahmen von Globus Toolkit 4 mit dessen Implementierung der *Grid Security Infrastructure* (GSI) [Glo05] an Sicherheit garantiert werden kann. In der Literatur werden in der Regel drei Säulen definiert, auf denen sich eine sichere Kommunikation stützt:

- Datenschutz: Informationsgehalt von Daten darf einem Dritten nicht zugänglich sein.
- Datenintegrität: Empfangene Daten müssen widerspruchsfrei sein.
- Authentifizierung: Korrektheit der Identität aller Beteiligten muss gewährleistet sein.

Darüber hinaus findet im Kontext einer sicheren Kommunikation das Konzept der Autorisierung Anwendung. Dabei geht es darum, ob ein authentifizierter Benutzer über notwendige Berechtigungen verfügt.

GSI stellt drei Sicherheitsmodelle bereit, die in Tabelle 8 verglichen werden. Alle drei Modelle basieren auf unterschiedlichen, bereits existierenden Technologien aus den Bereichen der Web Services (WS-Security, WS-SecureConversation) und der sicheren Kommunikation im Internet (SSL). Dabei ermöglicht GSI Sicherheit auf zwei verschiedenen Ebenen: auf Nachrichten- und Transportebene. Während auf Transportebene die gesamte Kommunikation (alle ausgetauschten Daten) sicher ist, beschränkt es sich auf Nachrichtenebene auf den Nutzinhalt der Nachrichtenpakete. Da für jede einzelne Nachricht aufwändige Berechnungen zur Verschlüsselung notwendig sind, ist die Performanz auf Nachrichtenebene schlechter als auf Transportebene. Allerdings ist auf Nachrichtenebene die Authentizität einzelner Nachrichten durch Dritte überprüfbar und auch eine individuelle Verschlüsselung einzelner Nachrichten möglich. Im Gegensatz zu GSI Secure Message baut GSI Secure Conversation zunächst einen *Security Context* zwischen Client und Server auf, welcher von folgenden Nachrichten wiederverwendet wird. Daraus resultiert eine höhere Performanz als bei GSI Secure Message, falls der Overhead für den Aufbau des Security Context hinsichtlich der Anzahl der Nachrichten vernachlässigbar ist.
Beide Sicherheitsebenen basieren in GSI auf asymmetrischer Verschlüsselung mit öffentlichen und privaten Schlüssel. Dieses Prinzip garantiert die oben genannten drei Säulen einer sicheren Kommunikation. Wie dies ermöglicht wird, ist unter anderem in [SC05] nachzulesen. GSI unterstützt verschiedene Authentifizierungsmethoden. Eine davon sieht Credentials auf Grundlage von *X.509-Zertifikaten* zur Authentifizierung vor. Genauer gesagt werden *Proxy-Zertifikate* benutzt, die auf Basis von X.509-Zertifikaten erstellt und dem Globus-Container übergeben werden. Ein Proxy-Zertifikat ist ein Stellvertreter-Zertifikat für ein X.509-Zertifikat. Es wird mit dem privaten Schlüssel des Benutzers generiert und hat eine begrenzte Lebensdauer. Es ermöglicht das Prinzip des Single Sign-On, wonach ein Benutzer nach einer einmaligen Authentifizierung (Übergabe des Proxy-Zertifikats) auf verschiedene, vorgesehene Anwendungen, Dienste, Ressourcen zugreifen kann, ohne sich

jedes Mal erneut authentifizieren zu müssen (geschieht implizit über vorliegendes Proxy-Zertifikat). Im Rahmen dieser Arbeit wird nur diese Methode relevant sein. Die Delegation von Credentials, das heißt, dass jemand im Namen von jemand anderem agieren kann, wird nur von GSI Secure Conversation unterstützt. Im Zusammenhang mit der Authentifizierung unterstützt GSI sowohl serverseitige als auch clientseitige Autorisierung.

| | GSI Secure Conversation | GSI Secure Message | GSI Transport |
|---|---|---|---|
| *Ebene* | Nachrichten | Nachrichten | Transport |
| *Technologie* | WS-SecureConversation | WS-Security | SSL |
| *Delegation* | Ja | Nein | Nein |
| *Performanz* | gut bei vielen Nachr. | gut bei wenigen Nachr. | am Besten |

Tabelle 8: Vergleich von GSI-Sicherheitsmodellen nach [SC05]

Sicherheit kann in GT4 für den Client, für den Service, für deren Resources und für den Container individuell mittels *Sicherheitsdeskriptoren* konfiguriert werden. Auch wenn beide Sicherheitsebenen und damit auch alle drei Sicherheitsmodelle Datenschutz, Datenintegrität und Authentifizierung beinhalten, müssen nicht alle zusammen verwendet, sondern können innerhalb der Deskriptoren beliebig kombiniert werden. Ein Beispiel eines solchen Deskriptors ist im Anhang B zu finden. Im Zusammenhang mit der Implementierung wird in Kapitel 5.2 beispielhaft der Aufbau einer solchen Deskriptordatei beschrieben.

Weiterführende Informationen zu GSI bietet [SC05].

### 4.2.4 Service Discovery

Wenn ein Plugin im Grid verteilt wird, ist es wichtig, dass man den entsprechenden Plugin Service im Grid finden kann. Es kann auch Anwendungsfälle geben, bei denen die Adresse des Plugin Service im Vorhinein bekannt ist. Das kann der Fall sein, wenn zwischen Teilnehmern einer virtuellen Organisation eine enge und statische Zusammenarbeit besteht. Wenn also Programm A auf den Grid Service B angewiesen ist. Nichtsdestotrotz muss auch der allgemeine Fall betrachtet werden, dass das Programm A einen Grid Service mit bestimmten Funktionalitäten sucht, von dem er noch nicht weiß, wo er diesen finden kann.

Auch bei den Web Services, auf denen das Konzept der Grid Services beruht, stellt sich die Frage, wie Services im Web gefunden werden können. Viele Web-Service-Anbieter, vor allem wenn es sich um große Unternehmen handelt, publizieren ihre Services über ihre Internetpräsenz. Diese Möglichkeit ist aus Sicht des Konsumenten von Web Services allerdings nicht ideal. Wenn der Konsument beispielsweise

einen Web Service sucht, der Informationen über das Wetter anbietet, ist es sinnvoller, wenn er mit entsprechenden Suchbegriffen bei einem Verzeichnisdienst nach einem solchen Web Service suchen kann. OASIS hat hierfür einen Verzeichnisdienst namens *UDDI* (Universal Description, Discovery and Integration) [OAS] als Standard definiert. Ein UDDI-Verzeichnis fungiert selbst als Web Service. Es hat eine spezifizierte Schnittstelle, über welche Clients Informationen abrufen und registrieren können. Hierfür wurde von OASIS ein Datenmodell definiert, welches die Zusammenhänge zwischen Diensten, deren Anbietern und deren Schnittstellen beschreibt.
In Abbildung 24 wird das Zusammenspiel von Konsument, UDDI-Verzeichnis und Service verdeutlicht.

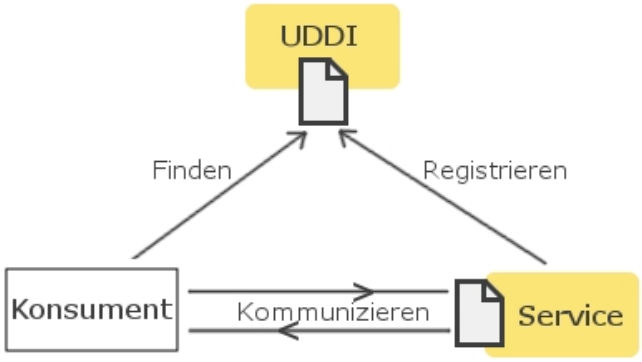


Abbildung 24: Beziehung zwischen Web-Service-Konsument, UDDI-Verzeichnis und Web Service

Das Globus Toolkit 4 stellt mit *MDS* (Monitoring and Discovery System) [Glo] eine Reihe von Web Services bereit, um Ressourcen im Grid überwachen und suchen zu können. Es liegt hierbei der Schwerpunkt darauf, Informationen über Ressourcen im Grid zu erhalten und nicht nach Anwendungen zu suchen, die eine bestimmte Dienstleistung erbringen. Eine zentrale Rolle spielt dabei der *Index Service*, der vom Prinzip mit einem UDDI-Verzeichnis vergleichbar ist. Der Index Service sammelt aus verschiedenen Quellen Informationen zu Ressourcen, die wiederum als Resource Properties gespeichert werden. In Abbildung 25 registrieren sich beispielsweise zwei Grid Services beim Index Service, um die Inhalte deren Resource Properties zu publizieren. Mittels *WebMDS* können Resource Properties in einem Browser visualisiert werden. Demnach kann WebMDS als Frontend für den Index Service benutzt werden, um dort gespeicherte Informationen einzusehen.

Ein Plugin Service zeichnet sich in erster Linie dadurch aus, dass er die Funktionalität einer Anwendung anbietet. Im Kontext des Grid sind allerdings auch Informationen relevant, die hinter dem Plugin Service in Form von Resource Properties stecken. Daraus ergibt sich, dass für das Suchen von im Grid verteilten Plugins ein Verzeichnisdienst wie UDDI verwendet werden kann, da dort der Fokus auf die angebotene Dienstleistung liegt. Um einem Nutzer des Plugin Service Informationen zugänglich zu machen, die in den Resource Properties einer WS-Resource gespeichert sind, bietet es sich an, das MDS von GT4 zu verwenden.

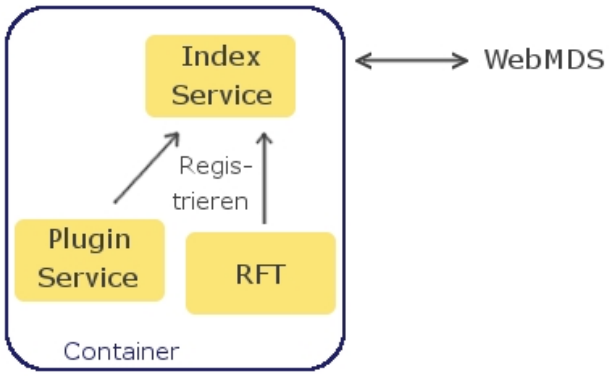


Abbildung 25: MDS-Registrierung von Plugin Service und RFT Service (Reliable File Transfer)

Es gibt kein zentrales UDDI-Verzeichnis, sondern nur unternehmensspezifische Implementierungen. Folglich muss für das Registrieren von Plugin Services zunächst ein UDDI-Verzeichnis realisiert werden. Es gibt zahlreiche Projekte, die sich mit der Entwicklung von UDDI-Verzeichnissen und Client-APIs beschäftigen. Allerdings sind nur einige Implementierungen davon frei verfügbar. Als zweckmäßige Projekte mit frei verfügbaren Implementierungen sind hier *UDDI4J* für die Clientseite und *jUDDI* für die Serverseite erkannt worden. Im Rahmen dieser Arbeit wird mit jUDDI ein Test-UDDI-Verzeichnis erstellt, welches mit UDDI4J angesprochen wird, um das Konzept, ein UDDI-Verzeichnis für das Registrieren von Plugin Services zu nutzen, zu verifizieren.
[BBG+05] beschäftigt sich damit, eine skalierbare Grid Service Discovery auf Basis von UDDI zu realisieren. Es kann für zukünftige Entwicklungen im Bereich der Plugin Service Discovery als Referenz genutzt werden.

Weiterhin wird innerhalb dieser Arbeit jede Resource eines Plugin Service beim entsprechenden MDS-Web-Service registriert, so dass die Inhalte deren Resource Properties für Benutzer zugänglich gemacht werden. Das Resource Home verwaltet die Resources. Demnach ist es sinnvoll, wenn das Resource Home auch diese Registrierung übernimmt.

### 4.2.5 Verteilprozess

Nachdem die einzelnen Aspekte der Verteilung eines Plugin im Grid betrachtet wurden, soll an dieser Stelle dieser Prozess zusammenfassend beschrieben werden. Das Aktivitätsdiagramm in Abbildung 26 stellt hierfür den Verteilprozess schematisch dar. Die darin enthaltenen, einzelnen Schritte des Prozesses werden anschließend kurz erläutert. Es soll deutlich werden, dass der Verteilprozess wie gefordert intuitiv und wiederverwendbar ist.

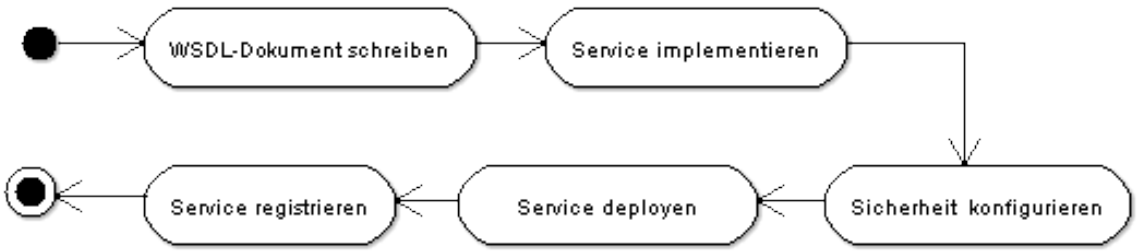


Abbildung 26: Aktivitätsdiagramm des Verteilprozesses

1. **Plugin Service mit WSDL beschreiben**
   Es muss die Schnittstelle des Plugin beschrieben werden. Wie in Kapitel 4.1.1 erwähnt wurde, wird diese zukünftig in WSDL gegeben sein. Falls WSRF-Schnittstellenerweiterungen verwendet werden, müssen diese (lediglich) referenziert werden, wie es in der WSDL-Beschreibung der Implementierung für die Verwaltung des Lebenszyklus von Resources gemacht wird. Darüber hinaus müssen die Resource Properties beschrieben werden. Bestehen diese lediglich aus den notwendigsten Properties (Referenz), kann die Beschreibung aus dem WSDL-Dokument der Implementierung übernommen werden.

2. **Plugin Service implementieren**
   Durch die in Kapitel 4.2.1 erläuterten abstrakten Klassen ist das Grundgerüst für die Implementierung vorhanden. Damit ist bereits vorgegeben, welche Klassen benötigt werden. Die notwendigen individuellen Implementierungen müssen in den abzuleitenden Klassen vorgenommen werden.

3. **Sicherheit konfigurieren**
   Mittels der beschriebenen Deskriptoren muss die Sicherheit so konfiguriert werden, dass die individuellen Anforderungen des Plugin Service erfüllt werden. Das Vorgehen bei der Konfiguration der Sicherheit wird im Kapitel 5.2 an Hand der Implementierung deutlich.

4. **Plugin Service deployen**
   Bis auf die Definition einiger Parameter wird der Deploy-Prozess durch Tools unterstützt und ist somit unkompliziert. Die detaillierte Beschreibung dieses Prozesses erfolgt in Kapitel 5.1.3.

5. **Service registrieren**
   Über definierte Schnittstellen wird der Plugin Service bei einem UDDI-Verzeichnis registriert. Das Vorgehen dabei ist auf Grund der gegebenen Schnittstellen statisch. Die Resource Properties werden automatisch durch das Resource Home beim MDS registriert.

Falls die Realisierung des Verteilprozesses für ein Plugin einem Standard folgt, den es noch genauer zu definieren gilt, sind alle für das Deployen notwendigen Dateien (WSDL, WSDD, JNDI, JAVA, GAR) (siehe Kapitel 5.1.3) in gewisser Weise statisch. Sie unterscheiden sich lediglich bei individuellen Benennungen. Es liegt daher die Idee nahe, in einem solchen Fall alle notwendigen Dateien zu generieren. Dies erspart zum einen Aufwand. Zum anderen wird damit auch eine Fehlerquelle beseitigt,

die ein menschlicher Entwickler immer darstellt. Für die Generierung sind lediglich Informationen zur Plugin-Schnittstelle und Benennungen (z.B. der Resource Properties) bereitzuhalten. Dies kann zum Beispiel in Form eines XML-Dokuments erfolgen.

## 4.3 Clientseitige Integration von Plugin Services in RCE

In diesem Kapitel soll betrachtet werden, wie verteilte Plugins in RCE integriert und verwendet werden können. Dabei können grundsätzlich zwei Konzepte verfolgt werden. Zum einen kann die Verteilung vor dem Benutzer verborgen werden, so dass er die Plugin-Funktionalitäten nutzt, als wenn das Plugin lokal in RCE installiert ist. Zum anderen kann dem Benutzer explizit die Möglichkeit gegeben werden, aus einer Liste aller verfügbaren Plugin Services auszuwählen, welches Plugin er verwenden oder welche seiner Methoden er aufrufen möchte. Einerseits ist die Entscheidung für eines der beiden Konzepte eine Frage der Transparenz und andererseits der Philosophie. Während beim ersten Konzept die Verteilung eines Plugin dem Benutzer gegenüber völlig transparent gehalten wird, verfolgt das zweite Konzept eher die Idee eines Grid, bei dem ein Benutzer die Ressourcen, die er nutzen möchte, frei wählen kann (vergleiche 2.1). Im Folgenden wird der Entwurf beider Konzepte kurz betrachtet.

Der Globus-Container verwendet standardmäßig den Port 8443, welcher eine verschlüsselte und authentifizierte Kommunikation erfordert. SOAP-Nachrichten an den Container müssen daher mit dem Protokoll HTTPS (HTTP mit Verschlüsselung (SSL/TLS)) gesendet werden. Wie in Kapitel 4.2.3 erwähnt wurde, werden als Credentials Proxy-Zertifikate benutzt. Für RCE wird dieses Konzept der Authentifizierung ebenfalls verwendet. Ein Benutzer muss ein X.509-Zertifikat besitzen. Beim Anmelden von RCE wird durch Eingabe des privaten Schlüssels des Benutzers ein Proxy-Zertifikat erstellt, welches den Benutzer für die Lebensdauer des Proxy-Zertifikats bei allen lokalen und entfernten RCE-Komponenten authentifiziert.
Es ist sinnvoll, für die Credentials, welche für den Aufruf der Plugin Services notwendig sind, auf die Credentials vom RCE-Benutzer zurückzugreifen. Zum einen sollten nicht ein und dieselben Credentials zweifach im System vorgehalten werden. Zum anderen wird so das in RCE verfolgte Prinzip des Single Sign-On konsequent weitergeführt. Dies ist vor allem beim ersten Integrationskonzept wichtig. Nur wenn sich der Benutzer nicht erneut authentifizieren muss, wenn er das Plugin über den Plugin Service aufruft, kann die gewünschte Transparenz der Verteilung erreicht werden.

Für beide Integrationskonzepte wird das Prinzip von Stubs angewendet. Ein Stub bezeichnet einen lokalen Anknüpfungspunkt, über den entfernte Softwarekomponenten einfach angesprochen werden können. Damit wird die dahinter steckende Komplexität verborgen. Er lässt letztendlich entfernte Aufrufe lokal aussehen.
Das verteilte Plugin wird in RCE durch einen Stub ersetzt, der auf Grund der Architektur von RCE erwartungsgemäß selbst ein Plugin ist. Dieser Stub gibt vor, ein verteiltes Plugin (z.B. Simplorer) zu sein, indem er die Schnittstelle dieses Plugin implementiert. Anstatt die Funktionalitäten des Plugin zu beinhalten, initialisiert er eine Verbindung zu einem entsprechenden Plugin Service und leitet Aufrufe seiner

Schnittstellenmethoden an die zugehörigen Methoden des Service weiter. Das in Abbildung 27 gezeigte Sequenzdiagramm beschreibt das Verhalten des Stub innerhalb eines Lebenszyklus.

Um den Lebenszyklus eines Plugin zu steuern, werden vom Plugin die Methoden *start* und *stop* implementiert. Beim Stub bietet es sich an, innerhalb dieser Methoden gleichzeitig den Lebenszyklus der serviceseitig zu nutzenden Resource und damit gleichzeitig der des verteilten Plugin zu verwalten. Eine Resource soll nur dann existieren, wenn der Stub ausgeführt wird. Das bedeutet im Einzelnen, dass in der Methode *start* die Methode *createResource* des Factory Service des Plugin Service aufgerufen wird, um eine Resource zu erstellen und die EPR zu erhalten. Mit der EPR wird daraufhin die Verbindung zum Plugin Service initialisiert, so dass im Folgenden die Methoden des Service aufgerufen werden können.
Um in der Methode *stop* des Stub die benutzte Resource zu zerstören, wird die Methode *destroy* des Plugin Service aufgerufen. Wie in Kapitel 4.2.1 erläutert wurde, handelt es sich dabei um eine WSRF-Methode, welche die Schnittstelle des Plugin Service zum Zweck der Lebenszyklusverwaltung von Resources erweitert.

Beim ersten Konzept wird ein Plugin, welches als Plugin Service im Grid verteilt wurde, lokal in RCE statisch durch ein Plugin ersetzt, welches auf Grund seiner Stubeigenschaften, das verteilte Plugin imitiert. Damit wird eine vollständige Transparenz der Verteilung erreicht.
Beim zweiten Konzept erfolgt die Integration von verteilten Plugins auch über Stubs. Der Unterschied ist, dass der Benutzer zur Laufzeit von RCE explizit die Plugin Services auswählen kann, die er nutzen möchte. Hierzu wird in RCE eine Liste aller verfügbaren Plugin Services zur Verfügung gestellt. Die Auflistung beinhaltet neben Informationen zu dem Plugin, welches dahinter steckt, auch Informationen zum Service selbst, die für die Auswahl der Plugin Services wichtig sind. Dazu können der Serviceanbieter, die verfügbare Rechenleistung, Lizenzgebühren und so weiter zählen. Solche Informationen müssen von RCE über entsprechende Informationsdienste (siehe Kapitel 4.2.4) allokiert werden. Zu jedem Plugin Service existiert ein Stub, der bei Bedarf gestartet wird.

Auf der einen Seite beinhaltet ein Stub die Implementierung der Plugin-Schnittstelle. Auf der anderen Seite implementiert er die Anbindung an den Plugin Service, so dass die Schnittstellenmethoden die zugehörigen Servicemethoden aufrufen können. Die Implementierung des Stub ist also ausschließlich von der Plugin-Schnittstelle auf der einen Seite und der Serviceschnittstelle auf der anderen Seite abhängig. Dies ist eine sehr gute Ausgangssituation für die Generierung von Stubs. Es ist denkbar, dass diese Generierung beim zweiten Integrationskonzept zur Laufzeit in Abhängigkeit der Plugin-Service-Auswahl erfolgt. Damit ist die gesamte clientseitige Integration von Plugin Services in RCE automatisiert. Es müssen lediglich die Schnittstellenbeschreibungen bereitgestellt werden.

Im Rahmen der Arbeit wird das Konzept implementiert, bei dem das verteilte Plugin statisch durch ein Stub ersetzt wird und die Verteilung somit für den Benutzer transparent ist. Einerseits ist die Variante einfacher umzusetzen, was vor dem Hintergrund, dass der Schwerpunkt auf der serverseitigen Umsetzung liegt, vertretbar ist. Andererseits wird kein Verzeichnisdienst implementiert werden, der für das zweite

Konzept verwendet werden könnte. Das macht die Umsetzung dieser zweiten Variante weniger sinnvoll.

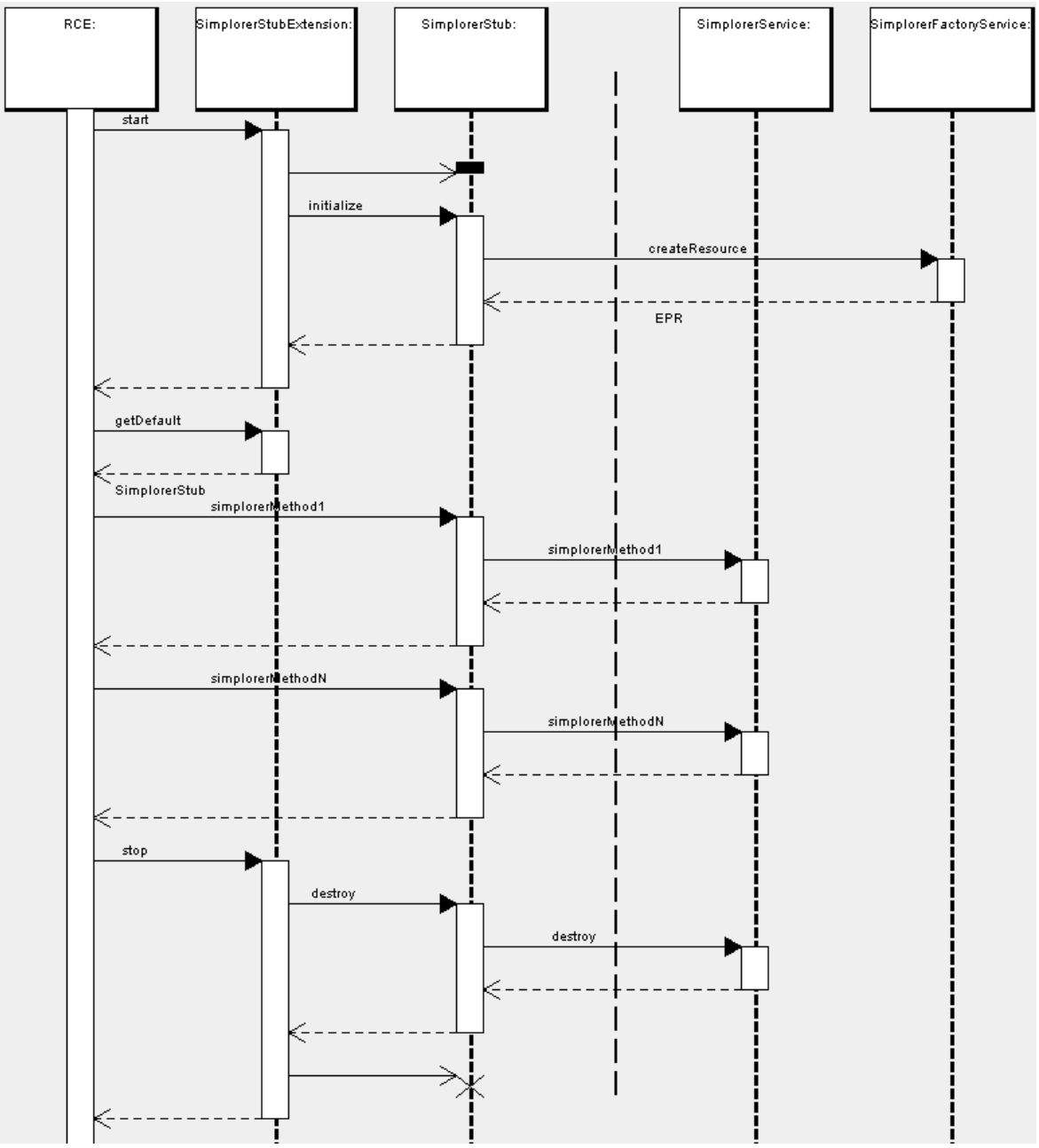


Abbildung 27: Sequenzdiagramm für den Lebenszyklus eines Stub

# 5 Implementierung und Test

In diesem Kapitel wird die Implementierung am Beispiel der Anwendung Simplorer kurz beschrieben, welche mittels Wrapper-Technologie als Plugin in RCE integriert ist. Die Motivation für die Verteilung von Simplorer als Service im Grid ist die begrenzte Anzahl an Lizenzen. Wenn die Anwendung Simplorer nur auf einem Rechner läuft, wird diese Problematik gelöst.
In diesem Kapitel wird nur auf ausgewählte Aspekte der Implementierung eingegangen, da der Großteil der Thematik beim Entwurf in Kapitel 4 bereits erläutert wurde. Weiterhin befasst sich dieses Kapitel mit dem Test der Implementierung. Dies beinhaltet Modul- und allgemeine Verhaltenstests.

## 5.1 Simplorer Service

Als erstes erfolgt die Betrachtung des Simplorer Service. Dabei wird einerseits dessen Implementierung allgemein behandelt. Andererseits wird die Service-WSDL-Beschreibung erläutert, das Vorgehen beim Deployen beschrieben sowie die Konfiguration der Sicherheit an Hand eines Beispieldeskriptors verdeutlicht.

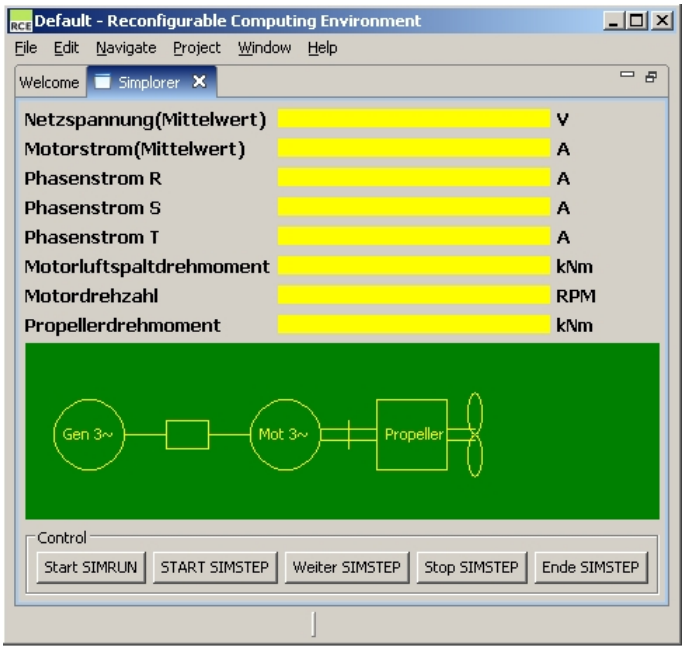


Abbildung 28: GUI des Plugin Simplorer in RCE

Das Bildschirmfoto in Abbildung 28 zeigt die grafische Benutzerschnittstelle vom Plugin Simplorer in RCE.

Mit den dargestellten Schaltflächen ist jeweils eine Methode verbunden. Sie bilden zusammen mit zwei *get*- und *set*-Methoden die Schnittstelle des Plugin Simplorer. Folglich sind sie auch Teil der Schnittstelle des Simplorer Service.

### 5.1.1 Implementierung

Für die Realisierung des Simplorer Service wird das in Kapitel 4.2.1 beschriebene Klassendiagramm umgesetzt. Die abstrakten Klassen sowie die Klasse *PluginHandler* werden im Package `de.dlr.gridsevice.common` zusammengefasst. Damit können sie für weitere Plugin Services einfach wiederverwendet werden. Die Simplorer-Service-spezifischen Klassen befinden sich im Package `de.dlr.sesis.simplorer.gridservice`.


Abbildung 29: Details der Properties einer Resource des Simplorer Service

Beim Entwurf wurde festgelegt, dass der Globus-Container und RCE in einer JVM ausgeführt werden. Hierzu muss RCE wegen der Kommandozeilenumgebung des Rechners mit der Globus-Installation ohne GUI gestartet werden. Dies ist auf Grund der Plugin-basierten Architektur von RCE grundsätzlich unproblematisch. RCE befindet sich allerdings noch in der Entwicklung, so dass dies zum jetzigen Zeitpunkt ohne zusätzlichen Implementierungsaufwand nicht möglich ist. Aus diesem Grund werden RCE und die verteilten Plugins dem Classpath des Containers hinzugefügt, indem sie in dessen `lib`-Ordner kopiert werden. Es können somit die Plugins innerhalb der Serviceimplementierung instanziiert werden. Des Weiteren ist der Zugriff auf zustandslose Funktionalitäten von RCE möglich, was für die Implementierung ausreichend ist.

Ein Plugin Service beziehungsweise dessen Resources sollen nach Kapitel 4.2.4 beim MDS von Globus registriert werden. Dies erfolgt in der Methode *addAndRegister* der Klasse *PluginResourceHome*. Das Werkzeug WebMDS von Globus bietet die Möglichkeit, über eine Web-Browser-Schnittstelle die registrierten Resource Properties grafisch darzustellen. Die Abbildung 29 zeigt beispielhaft die Darstellung der registrierten Resource Properties des Simplorer Service. Man erkennt sowohl die FQNs der Resource Properties als auch die darin gespeicherten Informationen.

### 5.1.2 Beschreibung mit WSDL

Im Folgenden wird gezeigt, wie der Simplorer Service mit WSDL beschrieben wird. WSDL basiert auf XML. Eine WSDL-Datei folgt dem in Abbildung 30 dargestellten Aufbau.

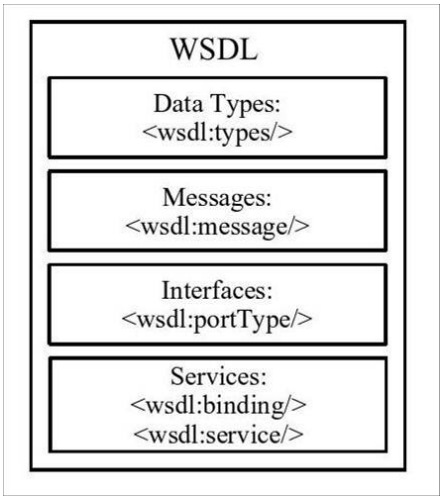


Abbildung 30: Aufbau einer WSDL-Datei

Nachfolgend werden einzelne, verwendete Elemente und Attribute kurz erläutert.

Das Wurzelelement des WSDL-Dokuments ist `definitions`. Die Attribute von `definitions` umfassen den Namen des Grid Service und die im Dokument verwendeten Namensräume.

```
<definitions name="SimplorerService"
targetNamespace="http://www.dlr.de/namespaces/SimplorerService"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
... >
...
</definitions>
```

In Kapitel 4.2.1 wurde das Prinzip der WSRF-Schnittstellenerweiterung erläutert. Die Schnittstelle des Simplorer Service wird, wie die Schnittstelle jedes Plugin Service, um eine Methode zur Lebenszyklusverwaltung erweitert. Mit dem Element `import` werden die zusätzlichen Definitionen integriert. Über die Attribute von `import` werden zum einen der Namensraum und zum anderen der Pfad zur WSDL-Datei angegeben, welche die Definitionen enthält.

```
<wsdl:import
namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
location="../wsrf/lifetime/WS-ResourceLifetime.wsdl"/>
...
```

Die vom Simplorer Service bereitgestellten Methoden werden jeweils mit dem Element `operation` beschrieben. Der Name der Methode wird als Attribut definiert. Dessen Unterelemente geben die Ein- und Ausgabeparameter an. Die `operation`-Elemente werden im Element `portType` zusammengefasst, welches demnach die

gesamte Schnittstelle beschreibt. Mit dessen Attribut `extends` kann die Schnittstelle eines Service um Methoden erweitert werden, so dass diese nicht explizit mit einem `operation`-Element beschrieben werden müssen. Im Fall des Simplorer Service geschieht dies für die Methoden zur Lebenszyklusverwaltung von Resources. Weiterhin wird mit dem Attribut `ResourceProperties` vom Element `portType` die Definition der Resource Properties referenziert.

```
<portType name="SimplorerPortType"
wsdlpp:extends="wsrlw:ImmediateResourceTermination"
...
wsrp:ResourceProperties="tns:SimplorerResourceProperties">

<operation name="start">
<input message="tns:startInputMessage"/>
<output message="tns:startOutputMessage"/>
</operation>
...

</portType>
```

Die Abbildung 31 veranschaulicht den definierten Porttype des Simplorer Service.

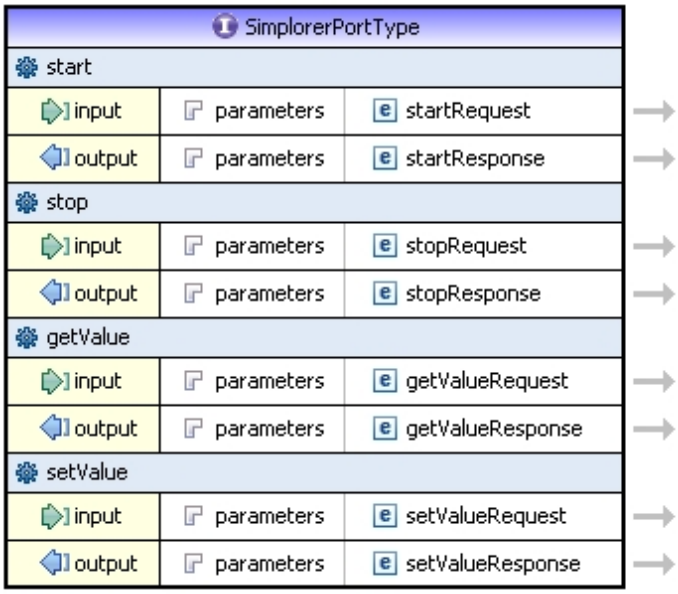


Abbildung 31: Porttype des Simplorer Service

Die Ein- und Ausgabeparameter (Inhalt der SOAP-Nachrichten) jedes `operation`-Elements werden mit dem Element `message` wie folgt beschrieben.

```
<message name="startInputMessage">
<part name="parameters" element="tns:startRequest"/>
</message>
...
```

Im Element `types` werden schließlich für sämtliche zuvor angegebenen Elemente die Datentypen definiert. Dies betrifft beim Simplorer Service einerseits die einzelnen Ein- und Ausgabeparameter. Andererseits bezieht es sich auf die Resource Properties. Beim Simplorer Service beschränken sie sich hauptsächlich auf die Referenz der Plugin-Instanz, die den Namen (FQN) *PluginKey* hat und vom Typ *String* ist.

```
<types>
<xsd:schema
targetNamespace="http://www.dlr.de/namespaces/SimplorerService"
... >

<!-- REQUESTS AND RESPONSES -->
<xsd:element name="setValueRequest" type="xsd:double"/>
<xsd:element name="setValueResponse">
<xsd:complexType/>
</xsd:element>
...

<!-- RESOURCE PROPERTIES -->
<xsd:element name="PluginKey" type="xsd:string"/>

<xsd:element name="SimplorerResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:PluginKey" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>
</types>
```

Damit ist die Schnittstelle des Simplorer Service beschrieben. Das vollständige WSDL-Dokument ist im Anhang A zu finden.

Das WSDL-Dokument wird nicht unmittelbar zum Deployen des Simplorer Service genutzt, da es auf Grund von WSRF- und Globus-spezifischen Eigenschaften kein korrektes WSDL beinhaltet. Zu diesen Eigenschaften zählt zum Beispiel das Attribut `extends` vom `portType`-Element. GT4 stellt einen WSDL-Preprozessor bereit, der zu den bereits definierten Methoden innerhalb des `portType`-Elements die referenzierten WSRF-Methoden explizit mittels `operation`-Elementen hinzufügt.
Weiterhin gehören zu einer vollständigen WSDL-Beschreibung Informationen zu den *Bindings* (Protokolldetails). Da diese für jeden Grid Service identisch sind, bietet GT4 ein Hilfswerkzeug an, mit dem sie generiert werden, so dass sie nicht manuell in das WSDL-Dokument geschrieben werden müssen.

Insofern die Implementierung des Grid Service in Java erfolgt, wird die Schnittstellenbeschreibung des Service auf ein Java-Interface basieren. Es existieren einige Hilfswerkzeuge, welche aus einem gegebenen Java-Interface ein WSDL-Schnittstellendokument erzeugen. Ein Beispiel ist *Java2WSDL* von Apache. Die Resource Properties spiegeln sich nicht im Java-Interface wieder. Daher müssen sie in jedem Fall zusätzlich definiert werden. Dies gilt ebenso für WSRF-Schnittstellenerweiterungen.

### 5.1.3 Deployen

Unter dem Deployen eines Grid Service versteht man das Einbinden eines Service in den Globus-Container, so dass dieser von Clients angesprochen werden kann. Hierfür werden die Serviceimplementierung, die WSDL-Datei und weitere Dateien, die nachfolgend beschrieben werden, als Grid-Archiv (*GAR*) zusammengefasst. Diese GAR-Datei wird mit einem Kommandozeilenbefehl von GT4 deployed.

Das GSBT-Projekt (Globus Service Build Tools) beschäftigt sich mit der Entwicklung von Hilfswerkzeugen, die den Entwicklungsprozess von Grid Services für Globus unterstützen. Eines dieser Werkzeuge ist *globus-build-service*. Es besteht aus einem Skript (Shell oder Python) und einer Ant-Build-Datei[5]. Die Build-Datei erstellt das Grid-Archiv. Damit die Build-Datei nicht an jeden Grid Service angepasst werden muss, gibt es das Skript, welches die Build-Datei mit entsprechenden Parametern aufruft, die dem Skript zum Teil selbst übergeben werden.
Dem Werkzeug globus-build-service müssen die Dateien bekannt gemacht werden, die an der Generierung der GAR-Datei beteiligt sind. In Abbildung 32 sind diese Dateien dargestellt.

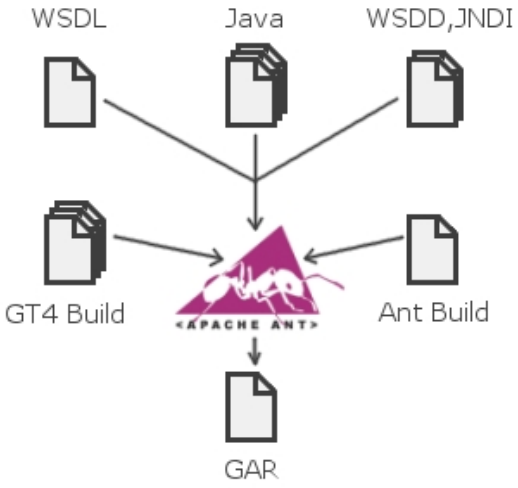


Abbildung 32: Generierung einer GAR-Datei

In der oberen Reihe sind die Dateien abgebildet, die bei der Entwicklung des Simplorer Service geschrieben werden müssen. Die Dateien rechts und links in der Mitte werden von GT4 beziehungsweise von globus-build-service bereitgestellt.

Die WSDL-Datei und die Implementierung des Service in Java wurden bereits vorgestellt. Darüber hinaus müssen die WSDD-Datei (Web Service Deployment Descriptor) und JNDI-Datei (Java Naming and Directory Interface) geschrieben werden. Die WSDD-Datei beschreibt, wie der Service deployed werden soll. Das beinhaltet zum Beispiel die URL des Simplorer Service, den Pfad zur WSDL-Datei sowie Parameter allgemein. Da Web Services keine Zustände und damit keine Resources kennen, können in der WSDD-Datei keine Informationen darüber bereitgestellt werden. Aus diesem Grund existiert die JNDI-Datei, in der Informationen zu den Resources des Simplorer Service stehen. Dies beinhaltet beispielsweise den Pfad zur Implementierung des Resource Home, den Full Qualified Name der Resource und so weiter.

Neben der Build-Datei für Ant werden für die Generierung der GAR-Datei GT4-Build-Dateien benötigt. Sie dienen unter anderem dazu, das oben erläuterte, korrekte WSDL zu generieren oder Stub-Klassen zu erstellen. Die einzelnen GT4-Build-Dateien werden von der Ant-Build-Datei aufgerufen.

[5]XML-Datei zur Steuerung des Hilfswerkzeugs Ant von Apache

Nachdem das Grid-Archiv durch Aufruf des Skripts von globus-build-service generiert und über die Kommandozeile deployed worden ist, sollte der Simplorer Service nach dem Neustart des Containers dort so eingebettet sein, dass er angesprochen werden kann. Falls dies nicht der Fall ist, kann die Fehlersuche aufwändig sein, da es wenige Informationen zur Ursache aber viele Fehlerquellen gibt. So können falsche Pfade zu den Implementierungsdateien angegeben worden sein, Full Qualified Names können nicht mit denen in der Serviceimplementierung übereinstimmen, von JNDI abhängige Methoden können fehlerhaft bezeichnet worden sein und so weiter.

## 5.2 Sicherheitsdeskriptoren

In diesem Kapitel wird betrachtet, wie Sicherheit für den Simplorer Service konfiguriert wird. Die vier Ebenen (Container, Service, Resource, Client), auf denen dies erfolgen kann, zeigt die Abbildung 33. Alle Konfigurationen erfolgen über die in Kapitel 4.2.3 erwähnten Sicherheitsdeskriptoren. Auf einigen Ebenen kann die Sicherheit auch programmiert werden. Da dies allerdings bei Änderungen ein erneutes Kompilieren nach sich zieht, wird die Variante mit den Deskriptoren bevorzugt.

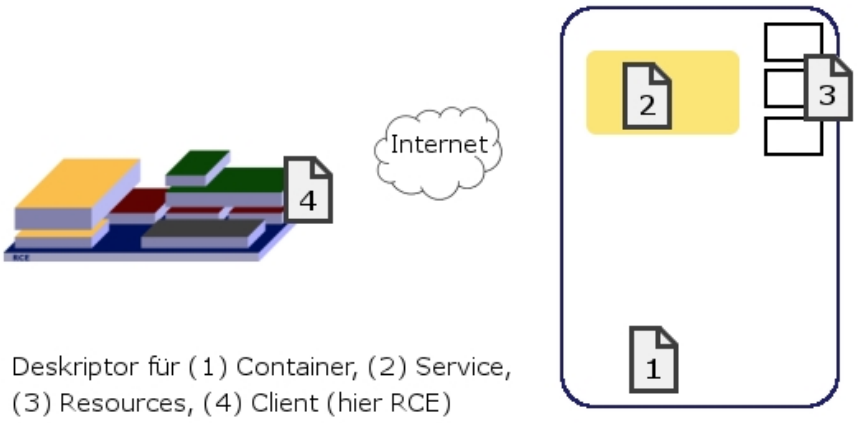


Abbildung 33: Sicherheitsebenen

Die Spezifizierung der Sicherheitsdeskriptoren erfolgt mit XML. Das Wurzelelement ist `securityConfig`. Neben gemeinsamen Unterelementen (wie z.B. das `credential`-Element) gibt es auch ebenenspezifische Unterelemente.

```
<securityConfig xmlns="http://www.globus.org">

<credential>
<key-file value="..."/>
<cert-file value="..."/>
</credential>

</securityConfig>
```

Nachfolgend wird kurz auf die Sicherheitsdeskriptoren der einzelnen Ebenen eingegangen.

- **Container**
  Es werden die vom Container zu nutzenden Credentials angegeben. Weiterhin steht im Deskriptor der Pfad zur globalen Gridmap-Datei. In der *Gridmap*-Datei stehen alle ausgezeichneten Namen (*distinguished names*) der Personen, die auf den Container zugreifen dürfen. Es sind auch andere Autorisierungsmethoden wie SAML oder deren Erweiterung Shibboleth möglich. Die Methode der Autorisierung ist letztendlich eine Frage der Konfiguration von Globus. Das Konzept der Gridmap ist im Globus-Kontext weit verbreitet und findet auch bei der hier verwendeten Globus-Installation Anwendung. Falls auf anderen, unteren Ebenen das Konzept der Gridmap genutzt, aber keine extra Datei verwendet wird, wird die Gridmap der nächsthöheren Ebene und demnach zuletzt die globale Gridmap verwendet.

- **Service**
  Die Anwendung Simplorer erfordert keine speziellen Sicherheitsanforderungen. Aus diesem Grund wird der Standard bei Globus, GSITransport, als Sicherheitsmodell verwendet. Mit GSITransport wird Sicherheit auf Transportebene gewährleistet. Damit ist keine Konfiguration der Sicherheit für einzelne Schnittstellenmethoden möglich, was die Methoden von Simplorer nicht erfordern. Weiterhin ist mit GSITransport keine Delegation von Credentials möglich (siehe Tabelle 8). Beim Plugin Simplorer handelt es sich um ein gekapseltes Plugin. Das heißt, dass innerhalb des Plugin keine weiteren Interaktionen zu anderen Plugins existieren. Das hat zur Folge, dass es nicht im Namen des Clients agieren muss. Somit ist ausgeschlossen, dass eine Delegation von Credentials erforderlich ist. Es wird der Einfachheit halber bei dieser Implementierung für die Autorisierung von Clients auf die globale Gridmap-Datei zurückgegriffen. Damit werden auf den einzelnen Ebenen (Service, Resource, ...) innerhalb der Deskriptoren keine weiteren Gridmap-Dateien referenziert.

- **Resource**
  Jeder Aufruf des Simplorer Service ist zustandsbehaftet, da jede Schnittstellenmethode des Service die Plugin-Instanz benötigt und somit auf die Resource Properties zugreift. Die Konfiguration auf Ebene des Service umfasst alle Schnittstellenmethoden des Service. Damit ist mit der Sicherheit auf Serviceebene gleichzeitig auch die Sicherheit auf Ebene der Resource abgedeckt. Folglich wird kein Sicherheitsdeskriptor auf Resourceebene definiert.

- **Client**
  Die Konfiguration des Sicherheitsmodells GSITransport ist nur im Quelltext des Clients möglich. Es wird dort definiert, dass sowohl Datenschutz als auch Datenintegrität gegeben ist. Weiterhin werden für die serviceseitige Authentifizierung und Autorisierung explizit die zu nutzenden Credentials angegeben. Wie in Kapitel 4.2.3 erklärt wurde, handelt es sich dabei um die gleichen, die für RCE verwendet werden.

Im Anhang B ist beispielhaft die vollständige Deskriptordatei für den Simplorer Service zu finden.

## 5.3 UDDI-Verzeichnis

Plugin Services sollen nach Kapitel 4.2.4 in einem UDDI-Verzeichnis registriert werden. Hierfür wurde exemplarisch ein UDDI-Verzeichis mittels jUDDI erstellt. Daraufhin wurde ein Client auf Basis der Java-Klassenbibliothek UDDI4J implementiert, welcher auf das UDDI-Verzeichnis zugreifen kann, so dass Services registriert und gesucht werden können.

jUDDI ist vollständig in Java implementiert. Zum Ausführen der Webapplikation wird ein Applikationsserver benötigt. Einer Anleitung für jUDDI folgend wurde hierfür Tomcat ausgewählt. Weiterhin erfordert jUDDI eine externe Datenspeicherung. Wiederum wurde der Anleitung gefolgt und die relationale Datenbank MySQL ausgewählt. Sowohl Tomcat als auch MySQL wurden installiert und für jUDDI entsprechend konfiguriert. Daraufhin wurde jUDDI installiert und eingerichtet, so dass auf das UDDI-Verzeichnis zugegriffen werden kann.

Der Client wurde mit Hilfe von UDDI4J implementiert. Dabei wurde auch auf Beispieldateien zurückgegriffen, die vom UDDI4J-Projekt zur Verfügung gestellt werden. Es wurde schließlich die Möglichkeit verifiziert, Services in einem UDDI-Verzeichnis registrieren und suchen zu können.
Neben der Option einen Client selbst zu implementieren, kann man auch einen ebenfalls frei verfügbaren UDDI-Browser verwenden, der selbst UDDI4J nutzt. Die Verwendung des Browser bietet sich bei der Registrierung von Plugin Services an, da dieser intuitiv bedient werden kann. Soll innerhalb von RCE automatisiert nach Plugin Services gesucht werden, ist eine eigene Implementierung erforderlich. Wenn im Gegensatz dazu andere Systeme oder Benutzer einen Plugin Service suchen, kann wiederum der Einsatz des UDDI-Browser nützlich sein, da die Suche nicht zwangsläufig automatisiert geschehen muss.

## 5.4 Stub für RCE

Im Kapitel 4.3 wurde erläutert, wie die clientseitige Realisierung bezüglich RCE aussieht. Es wurde entschieden, dass bei der Implementierung ein Stub realisiert wird, der als Plugin in RCE integriert ist. Dabei soll der Stub das Plugin Simplorer so ersetzen, dass dessen Verteilung transparent ist.

Die Implementierung des Stub basiert auf dem in Abbildung 27 dargestellten Sequenzdiagramm. Daraus resultiert das in Abbildung 34 gezeigte Klassendiagramm. Der Stub selbst besteht aus der Klasse *SimplorerServiceStub*, welche die Schnittstelle von Simplorer realisiert. Weiterhin existiert eine Klasse *SimplorerServiceStubExtension*, welche den Lebenszyklus des Stub (des Plugin) steuert.

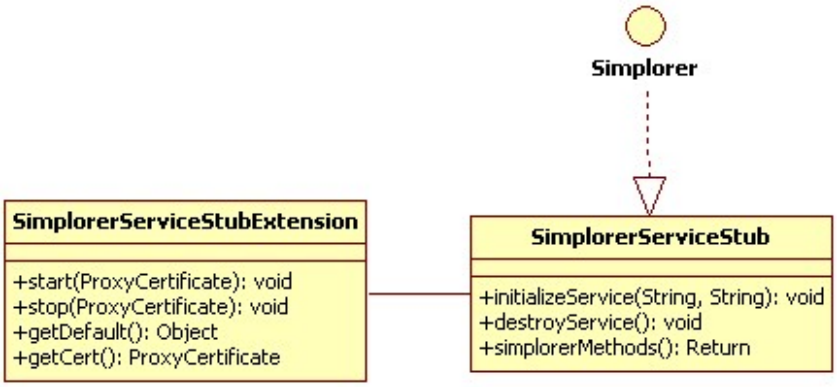


Abbildung 34: Klassendiagramm für den Stub

Der Lebenszyklus des Stub beginnt mit dem Aufruf der Methode *start* der Klasse *SimplorerServiceStubExtension*. Daraufhin erfolgt die Initialisierung des Simplorer Service durch die entsprechende Methode der Klasse *SimplorerServiceStub*. Damit können im Folgenden die Servicemethoden beliebig aufgerufen werden. Wird der Stub über den Aufruf der Methode *stop* wieder gestoppt, wird serviceseitig die verwendete Resource durch die Methode *destroyService* der Klasse *SimplorerServiceStub* zerstört.

Während der Implementierung traten unerwartete Probleme bei der Verwendung von Klassenbibliotheken auf. Grundsätzlich hat jedes Plugin seinen eigenen Classpath, so dass für den Stub verwendete Klassenbibliotheken von denen anderer Plugins unbeeinflusst sind. In diesem Fall wurde allerdings eine Systemvariable innerhalb eines RCE-Plugin so gesetzt, dass die Verwendung einer Bibliothek innerhalb des Stub zu Problemen geführt hat. Der Grund war, dass dadurch systemweit eine andere Version dieser Bibliothek existierte. Letztendlich konnte das Problem umgangen werden, indem explizit die benötigte Klassenbibliothek vom entsprechenden RCE-Bundle importiert wurde.

## 5.5 Test

Neben den in Kapitel 5.6.1 beschriebenen Modultests wurde die Implementierung an Hand des Einsatzszenarios auf dessen Verhalten getestet. Zunächst wurde hierfür das Java CoG Kit benutzt, um über verschiedene Aufrufe zu testen, ob der Simplorer Service zum Beispiel erfolgreich deployed wurde. Eine Aussage darüber ist auf Grund mangelnder Fehlermeldungen während des Deployen meist erst beim Aufruf möglich. Hierfür wurden mittels des CoG Kit beispielsweise die notwendigen Proxy-Zertifikate generiert. Später wurde wie vorgesehen auf die Credentials von RCE zurückgegriffen und das Proxy-Zertifikat innerhalb des Stub generiert. Damit erfolgte das Testen direkt an Hand der GUI vom Plugin Simplorer, welche in Abbildung 28 gezeigt wurde.

## 5.6 Entwicklung

Es wurden bei der Implementierung verschiedene Elemente der Softwareentwicklung eingesetzt. Dazu gehörte im Rahmen der RCE-Entwicklung eine Versionsverwaltung auf Basis eines Subversion-Repository. Damit verbunden wurden so genannte Nightly Builds durchgeführt. Weiterhin unterlag der geschriebene Quelltext Codierrichtlinien ohne deren Einhaltung ein Einchecken des Code in die Versionsverwaltung nicht möglich war. Darüber hinaus wurden, wie bereits in Kapitel 1.3 erwähnt wurde, Modultests geschrieben.

Dieses Kapitel beschränkt sich der Relevanz wegen im Folgenden darauf, genauer auf die Modultests einzugehen und einen Überblick über die verwendeten Entwicklungswerkzeuge zu geben.

### 5.6.1 Modultests

Der Test der Implementierung erfolgte zum einen durch Modultests (engl. Unit Tests), wie es in dem in Kapitel 1.3 vorgestellten Vorgehensmodell beschrieben ist. Zum anderen fanden Verhaltenstests auf Basis des Einsatzszenarios statt, welches in Kapitel 4 in der Abbildung 15 erläutert wurde. Auf das jeweilige Vorgehen wird nachfolgend kurz eingegangen.

Mit den Modultests wurde bereits während der Implementierung die Funktion der einzelnen Klassen überprüft. Für jede Klasse wurde daher ein Testfall implementiert, die innerhalb eines Packages zu einer Testfolge zusammengefasst wurden. Damit können Module sowohl auf Ebene der Packages als auch auf Ebene der Klassen getestet werden.
Um einen Grid Service nutzen zu können, muss er im Globus-Container deployed werden. Das Deployen übernimmt bei Globus ein Kommandozeilenwerkzeug. Dieses Werkzeug sorgt dafür, dass beim Neustart des Containers beispielsweise bestimmte Erzeugungs- und Initialisierungsprozesse durchgeführt werden, die für das Ausführen der Serviceimplementierung notwendig sind. Der Quellcode des Grid Service ist demnach nur eingebettet in einem Container ausführbar. Daraus folgt, dass der Simplorer Service nur getestet werden kann, wenn er im Container deployed ist. Theoretisch besteht die Möglichkeit, die Implementierung der Testfälle so zu abstrahieren, dass ein Umfeld für den Grid Service geschaffen wird, indem er ausgeführt und damit auch getestet werden kann. Dieses Vorgehen wird bei den Modultests von RCE-Komponenten verfolgt. Es sind dafür von den RCE-Entwicklern abstrakte Klassen implementiert worden, von denen die Testklassen abgeleitet werden können. Damit ist es möglich, einzelne Bundles/Plugins zu testen, ohne dass die RCE-Plattform gestartet werden muss. Derzeit ist nichts bekannt, was Modultests für Grid Services so unterstützt, dass der Aufwand für das Erstellen der Modultests akzeptabel ist. Aus diesem Grund wird die Serviceimplementierung nicht serverseitig auf Modulebene getestet. Der Test des Simplorer Service erfolgte durch die clientseitigen Modultestimplementierungen innerhalb des Stub, welcher als RCE-Plugin realisiert ist (siehe Kapitel 4.3).

Die Klassen auf Stubseite werden sowohl auf Erfolg als auch auf Fehlschlag getestet. Die dritte Möglichkeit, Klassen auf ihre Sinnhaftigkeit zu testen, ist in diesem Fall nicht möglich. Simplorer basiert auf Visual Basic Script. Da der Globus-Container nicht unter dem Betriebssystem Windows läuft, können die Simplorer-Skriptdateien dort nicht unmittelbar ausgeführt werden. Wie in Kapitel 2.3 geschildert wurde, werden externe Anwendungen mittels Wrapper-Technologie in RCE als Plugins integriert. Die Entwicklung von Wrapper ist unbequem und aufwändig. Zurzeit ist die Anwendung Simplorer die einzige, zu der ein Wrapper existiert, so dass diese als Plugin in RCE installiert werden kann. Auf Grund mangelnder Alternativen hinsichtlich zu verteilender Plugins wurde entschieden, dass es bei dieser Implementierung ausreichend ist, wenn das Prinzip der Skriptaufrufe nachgeahmt wird. Es werden an Stelle der VBS-Skripte Shell-Skripte serverseitig aufgerufen, die allerdings keine Simplorer-Funktionalität enthalten.
Würde die Skriptproblematik nicht existieren, wären Tests auf Sinnhaftigkeit möglich. In diesem Fall könnten die Daten, welche die Schnittstellenmethoden zurückliefern, wenn Simplorer lokal als Plugin installiert ist, mit den Daten verglichen werden, welche die Methoden des Simplorer Service zurückgeben.

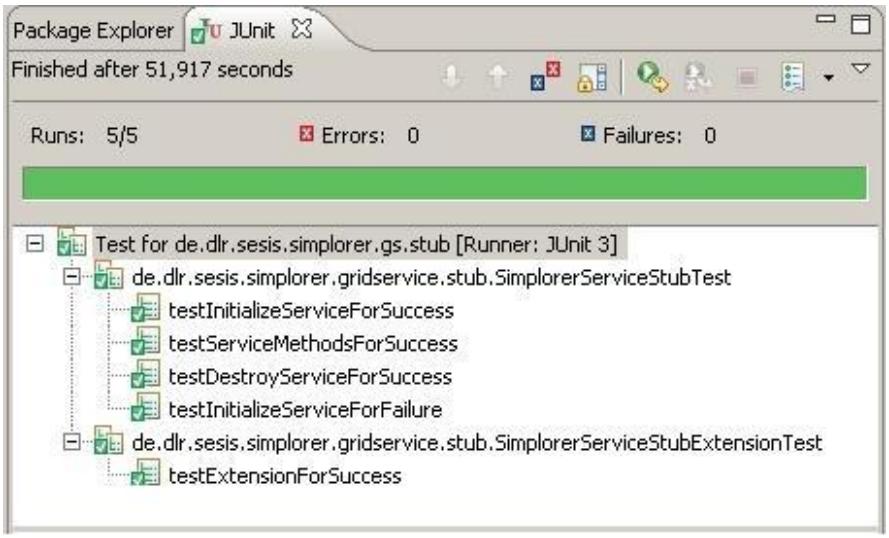


Abbildung 35: Ergebnisse der Testfälle

Bei den Modultests auf Erfolg wurde die Initialisierung des Simplorer Service, der Aufruf aller Servicemethoden und die Lebenszyklusmethode *destroy* berücksichtigt. Auf Misserfolg wurde die Initialisierung überprüft, indem falsche Parameter (URI, Pfade, ...) verwendet wurden. Die Servicemethoden auf Misserfolg zu testen, war wegen der fehlenden serverseitigen Implementierung von Simplorer nicht möglich. Auch die *destroy*-Methode bietet wegen fehlender Parameter keine sinnvolle Möglichkeit für einen Test auf Misserfolg. Die Abbildung 35 zeigt die Ergebnisse der Testdurchläufe.

Es wurde darauf geachtet, dass die Modultests mindestens 80% des Code abdecken. Im anderen Fall können die Modultests die Komponente nicht sinnvoll testen. Die Abbildung 36 zeigt die erzielte Abdeckung. Eine 100-prozentige Codeabdeckung kann in den meisten Fällen auf Grund von privaten Methoden und Konstruktoren nicht erreicht werden.

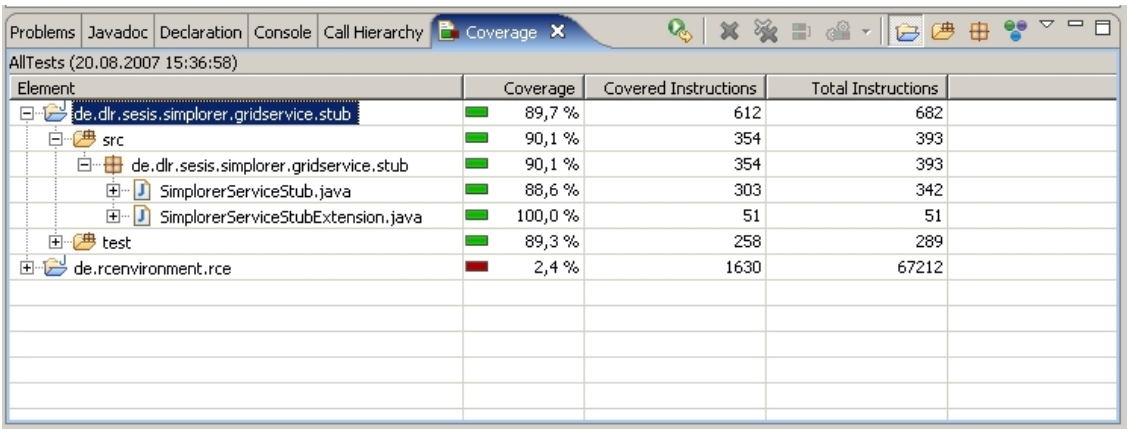


Abbildung 36: Anteil des getesteten Code (Codeabdeckung)

### 5.6.2 Werkzeuge

Dieses Kapitel gibt abschließend in Tabellenform einen Überblick über die während der Entwicklung eingesetzten Werkzeuge.

| Werkzeug | Verwendung |
|---|---|
| Eclipse Java IDE | Implementierung |
| Globus Toolkit 4 | Grid Services |
| Java CoG Kit | Zugriff auf Grid Services |
| Jupiter | Code Review |
| ArgoUML | Modellierung |
| Checkstyle | Einhaltung von Codierrichtlinien |
| Subversion | Versionkontrolle |

Tabelle 9: Überblick eingesetzter Entwicklungswerkzeuge

# 6 Schlussbetrachtung

In diesem Kapitel wird abschließend die Masterarbeit zusammengefasst. Dabei wird deutlich, dass die erzielten Ergebnisse denen entsprechen, die in der Aufgabenstellung gefordert wurden. Weiterhin gibt dieses Kapitel einen Ausblick auf zukünftige Arbeiten.

## 6.1 Zusammenfassung

Das Ziel der Masterarbeit war es, ein Plugin-basiertes System innerhalb einer Grid-Umgebung zu verteilen. Dafür sollte untersucht werden, wie Plugins als Grid Services realisiert werden können. Eine der erarbeiteten Möglichkeiten sollte daraufhin umgesetzt werden. Als Fallbeispiel diente das im Projekt SESIS entwickelte Plugin-basierte System RCE.

Nachdem die Anforderungen spezifiziert waren, wurden vier Möglichkeiten für die Realisierung eines Grid Service erarbeitet, an Hand verschiedener Aspekte bewertet und verglichen. Es wurde daraufhin die Möglichkeit, den Grid Service als Kommunikationsschnittstelle zu RCE und Plugin zu verwenden, als praktikabelste erkannt. Im Weiteren erfolgte der Entwurf dieser Variante des Plugin Service. Dabei wurden damit zusammenhängende Gesichtspunkte erschlossen und betrachtet. Diese betrafen die Abbildung von RCE-Funktionen auf Grid-Technologien, die Sicherheit, die Service Discovery und den Verteilprozess allgemein, der intuitiv und wiederverwendbar sein soll.
Darüber hinaus wurden Möglichkeiten untersucht, wie clientseitig aus RCE auf verteilte Plugins zugegriffen werden kann. Einerseits kann der Zugriff für den RCE-Anwender transparent erfolgen. Andererseits kann der Anwender mit der Verteilung des Plugin direkt konfrontiert werden, so dass er selbst den Plugin Service aussuchen kann, den er nutzen möchte.

Am Beispiel der Anwendung Simplorer, welche als Plugin in RCE integriert ist, erfolgte im Anschluss an den Entwurf die Implementierung. Dabei wurde clientseitig in RCE ein für den Anwender transparenter Zugriff realisiert.

## 6.2 Ausblick

Der Umfang der Implementierung wurde bereits bei der Anforderungsanalyse begrenzt. Es wurde demnach bei der Implementierung nicht alles berücksichtigt, was zuvor entworfen worden ist. Es ist erstrebenswert, die Implementierung zukünftig so zu erweitern, dass der gesamte Entwurf umgesetzt wird. Dies betrifft zum einen das UDDI-Verzeichnis und zum anderen das Abbilden von RCE-API-Funktionalitäten auf Grid-Technologien.

Weiterhin ist es auf Basis der Realisierung des UDDI-Verzeichnisses denkbar, die zweite Möglichkeit für die clientseitige Anbindung von RCE zu implementieren, da diese, wie in der Arbeit beschrieben wurde, mehr dem Konzept eines Grid entspricht.

Innerhalb der Arbeit wurde erwähnt, dass die automatisierte Erzeugung von Quelltext sowohl serverseitig als auch clientseitig an bestimmten Stellen möglich ist. Die Generierung von Code ist auf Grund von dauerhafter Zeitersparnis und Fehlervermeidung immer erstrebenswert. Folglich ist es vorstellbar, dass sich zukünftige Arbeiten mit der Umsetzung eines Codegenerators beschäftigen.

Darüber hinaus kann eine Performanzmessung durchgeführt werden. Auf deren Basis ist eine Evaluierung möglich, inwieweit eine Verteilung für bestimmte Plugins sinnvoll ist. Werden beim Aufruf von Schnittstellenmethoden eines Plugin sehr viele Daten übertragen, so können die Antwortzeiten bei einem entsprechenden Plugin Service auf Grund der notwendigen Kommunikation inakzeptabel groß werden.

# A WSDL-Datei

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions name="SimplorerService"
targetNamespace="http://www.dlr.de/namespaces/SimplorerService"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://www.dlr.de/namespaces/SimplorerService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.xsd"
xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-01.wsdl"
location="../wsrf/properties/WS-ResourceProperties.wsdl"/>
<wsdl:import
namespace="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-01.wsdl"
location="../wsrf/lifetime/WS-ResourceLifetime.wsdl"/>

<!-- TYPES -->

<types>
<xsd:schema targetNamespace="http://www.dlr.de/namespaces/SimplorerService"
xmlns:tns="http://www.dlr.de/namespaces/SimplorerService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<!-- REQUESTS AND RESPONSES -->

<xsd:element name="startRequest">
<xsd:complexType/>
</xsd:element>
<xsd:element name="startResponse">
<xsd:complexType/>
</xsd:element>
<xsd:element name="stopRequest">
<xsd:complexType/>
</xsd:element>
<xsd:element name="stopResponse">
<xsd:complexType/>
</xsd:element>
<xsd:element name="getValueRequest">
<xsd:complexType/>
</xsd:element>
<xsd:element name="getValueResponse" type="xsd:double"/>
<xsd:element name="setValueRequest" type="xsd:double"/>
<xsd:element name="setValueResponse">
<xsd:complexType/>
</xsd:element>

<!-- RESOURCE PROPERTIES -->

<xsd:element name="PluginKey" type="xsd:string"/>

<xsd:element name="SimplorerResourceProperties">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tns:PluginKey" minOccurs="1" maxOccurs="1"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

</xsd:schema>

</types>


<!-- MESSAGES -->

<message name="startInputMessage">
```

```
<part name="parameters" element="tns:startRequest"/>
</message>
<message name="startOutputMessage">
<part name="parameters" element="tns:startResponse"/>
</message>
<message name="stopInputMessage">
<part name="parameters" element="tns:stopRequest"/>
</message>
<message name="stopOutputMessage">
<part name="parameters" element="tns:stopResponse"/>
</message>
<message name="getValueInputMessage">
<part name="parameters" element="tns:getValueRequest"/>
</message>
<message name="getValueOutputMessage">
<part name="parameters" element="tns:getValueResponse"/>
</message>
<message name="setValueInputMessage">
<part name="parameters" element="tns:setValueRequest"/>
</message>
<message name="setValueOutputMessage">
<part name="parameters" element="tns:setValueResponse"/>
</message>


<!-- PortType -->

<portType name="SimplorerPortType"
wsdlpp:extends="wsrpw:GetResourceProperty
wsrpw:GetMultipleResourceProperties
wsrpw:SetResourceProperties
wsrpw:QueryResourceProperties
wsrlw:ImmediateResourceTermination
wsrp:ResourceProperties="tns:SimplorerResourceProperties">

<operation name="start">
<input message="tns:startInputMessage"/>
<output message="tns:startOutputMessage"/>"
</operation>
<operation name="stop">
<input message="tns:stopInputMessage"/>
<output message="tns:stopOutputMessage"/>"
</operation>
<operation name="getValue">
<input message="tns:getValueInputMessage"/>
<output message="tns:getValueOutputMessage"/>"
</operation>
<operation name="setValue">
<input message="tns:setValueInputMessage"/>
<output message="tns:setValueOutputMessage"/>"
</operation>

</portType>

</definitions>
```

# B Service-Sicherheitsdeskriptor

```
<securityConfig xmlns="http://www.globus.org">

   <method name="start">
      <auth-method>
         <GSISecureConversation>
            <protection-level>
               <privacy>
            </protection-level>
         </GSISecureConversation>
      </auth-method>
   </method>

   <method name="stop">
      <auth-method>
         <GSISecureMessage/>
      </auth-method>
   </method>

   <auth-method>
      <none/>
   <auth-method>

   <authz value="gridmap"/>
   <authz value="etc/gridmapfile"/>

</securityConfig>
```

# C WSDD-Datei

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment name="defaultServerConfig"
xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<service name="SimplorerService" provider="Handler" use="literal" style="document">
<parameter name="className" value="de.dlr.sesis.simplorer.gridservice.impl.SimplorerService"/>
<wsdlFile>share/schema/simplorer/Simplorer_service.wsdl</wsdlFile>
<parameter name="allowedMethods" value="*"/>
<parameter name="handlerClass" value ="org.globus.axis.providers.RPCProvider"/>
<parameter name="scope" value="Application"/>
<parameter name="providers" value="
GetRPProvider GetMRPProvider SetRPProvider QueryRPProvider
DestroyProvider SetTerminationTimeProvider
SubscribeProvider GetCurrentMessageProvider PauseSubscriptionProvider ResumeSubscriptionProvider"/>
<parameter name="loadOnStartUp" value="true"/>
</service>

<service name="SimplorerFactoryService" provider="Handler" use="literal" style="document">
<parameter name="className" value="de.dlr.sesis.simplorer.gridservice.impl.SimplorerFactoryService"/>
<wsdlFile>share/schema/simplorer/SimplorerFactory_service.wsdl</wsdlFile>
<parameter name="allowedMethods" value="*"/>
<parameter name="handlerClass" value="org.globus.axis.providers.RPCProvider"/>
<parameter name="scope" value="Application"/>
</service>

</deployment>
```

# D JNDI-Datei

```
<?xml version="1.0" encoding="UTF-8"?>

<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

<service name="SimplorerService">
<resource name="home" type="de.dlr.sesis.simplorer.gridservice.impl.SimplorerResourceHome">
<resourceParams>
<parameter>
<name>resourceClass</name>
<value>de.dlr.sesis.simplorer.gridservice.impl.SimplorerResource</value>
</parameter>
<parameter>
<name>resourceKeyType</name>
<value>java.lang.String</value>
</parameter>
<parameter>
<name>resourceKeyName</name>
<value>http://www.dlr.de/namespaces/SimplorerServiceResourceKey</value>
</parameter>
<parameter>
<name>pluginServicePath</name>
<value>SimplorerService</value>
</parameter>
<parameter>
<name>factory</name>
<value>org.globus.wsrf.jndi.BeanFactory</value>
</parameter>
<parameter>
<name>sweeperDelay</name>
<value>1000</value>
</parameter>
</resourceParams>
</resource>
</service>

<service name="SimplorerFactoryService">
<resourceLink name ="pluginServiceHome" target="java:comp/env/services/SimplorerService/home"/>
</service>

</jndiConfig>
```

# E Glossar

**Bundle** Kleinste funktionstragende Einheit im Kontext der OSGi-Technologie.

**Codeabdeckung** Prozentualer Anteil des Code, welcher bei dem Durchlauf eines Modultests ausgeführt wird.

**CoG Kit** Commodity Grid Kit. Werkzeug, welches einen abstrahierten Zugriff auf ein Grid ermöglicht.

**Container** Serverseitige Software, in der Grid Services eingebettet sind.

**Credential** Nachweis zur Authentifizierung einer Person.

**Eclipse** Plugin-basiertes System. Entwicklungsumgebung.

**Equinox** Eine Implementierung des OSGi-Framework.

**Endpoint Reference** Adresse einer WS-Resource (URL (Service) + ID (Resource)).

**Extension** Funktionelle Erweiterung eines Plugin durch ein anderes Plugin innerhalb eines Plugin-basierten Systems.

**Extension Point** Definierter Punkt eines Plugin, an dem es funktionell durch eine Extension erweitert werden kann.

**Fabrikmethode** Erzeugungsmuster, bei dem ein Objekt über eine definierte Schnittstelle erzeugt wird.

**FQN** Full Qualified Name. Eindeutiger Bezeichner nach der XML-Spezifikation.

**GAR** Grid-Archiv. Wird zum Deployen verwendet. Fasst eine Menge an Dateien zusammen, die zum Deployen notwendig sind.

**Globus Toolkit** Softwarekomponenten der Globus Alliance, die Entwicklung gridbasierter Anwendungen unterstützten.

**Gridmap** Datei, in der ausgezeichnete Namen von berechtigten Personen stehen.

**Grid Service** Zustandsbehafteter Web Service.

**GSI** Grid Security Infrastructure. Konzepte und Technologien für die Gewährleistung von Sicherheit im Grid.

**Index Service** Komponente des MDS zur Registrierung von Resource Properties.

**MDS** Monitoring and Discovery System. GT4-Komponenten zum Suchen und Überwachen von Ressourcen im Grid.

**OGSA** Open Grid Services Architecture. Spezifikation, welche die Anforderungen an eine Grid-Middleware beschreibt.

**OSGi** Technologie zur Realisierung einer integrierten Serviceplattform (OSGi-Framework).

**Proxy-Zertifikat** Stellvertreter-Zertifikat eines X.509-Zertifikats zur Realisierung des Konzepts Single Sign-On.

**RCE** Reconfigurable Computing Environment. Plugin-basiertes Basissystem von SESIS.

**RCE*** Bezüglich des Funktionsumfangs reduziertes RCE.

**RCE-API** Programmierschnittstelle, über die Plugins auf Basisdienste von RCE zugreifen können.

**RCE-Basisdienste** Plugins des Basissystems RCE (Kommunikation, Update, Privilegien, Service Broker, Datenmanagement).

**Resource** Komponente, die für einen Grid Service notwendige Zustandsinformationen speichert.

**Resource Home** Komponente innerhalb des WSRFP zur Erzeugung und Verwaltung von Resources.

**Resource Properties** Zustandsinformationen eines Grid Service.

**Plugin** Kleinste funktionstragende Einheit im Kontext Plugin-basierter Systeme.

**Plugin-basiertes System** System, dessen Logik ausschließlich durch integrierbare Plugins bereitgestellt wird.

**Plugin Service** Grid Service für ein im Grid verteiltes Plugin.

**SESIS** Schiffsentwurfs- und Simulationssystem.

**Sicherheitsdeskriptor** XML-Datei zur Beschreibung der Sicherheitskonfiguration einer Ebene (Container, Service, Resource, Client).

**Single-Sign-On** Konzept, bei dem ein Benutzer nach einer einmaligen Authentifizierung für eine bestimmte Zeit innerhalb eines bestimmten Systems als authentifiziert gilt.

**SOAP** Standardmäßig für Web Services verwendetes Kommunikationsprotokoll.

**Stub** Lokaler Anknüpfungspunkt für das Ansprechen entfernter Komponenten.

**UDDI** Universal Description, Discovery and Integration. Verzeichnisdienst zur Registrierung von Web Services.

**Verteilprozess** Vorgehen bei der Bereitstellung eines Plugin als Service im Grid.

**Verzeichnisdienst** Sammlung von Informationen über Web Services.

**Virtuelle Organisation** Dynamischer Zusammenschluss von Individuen und/oder Institutionen zur Verfolgung eines gemeinsamen Ziels bei der Nutzung des Grid.

**Web Service** Anwendung, die über ein Netzwerk von anderen Systemen über eine (meist mit WSDL) definierte Schnittstelle angesprochen werden kann.

**Wrapper** Implementierung, welche eine externe Anwendung so umhüllt, dass sie als Plugin in RCE integriert werden kann.

**WSDL** Web Service Description Language. Unabhängige Beschreibungssprache für Web Services.

**WS-Resource** Web Service + Resource.

**WSRFP** Web Services Resource Factory Pattern. Muster für die Erzeugung von Resources nach WSRF)

**WSRF** Web Services Resource Framework. Spezifikation, die beschreibt, wie Web Services zustandsbehaftet werden.

# Abbildungsverzeichnis


1 Schichten des Grid . . . 9
2 Vorgehen bei der Entwicklung . . . 12
3 Vision des Grid . . . 14
4 Zusammenhang zwischen OSGA, WSRF und Web Services . . . 18
5 Web-Service-Aufruf durch Client . . . 18
6 Serverseitige Software mit eingebetteten Web Services . . . 19
7 WS-Resource [SC05] . . . 20
8 Komponenten von Globus Toolkit 4 [Fos06] . . . 21
9 Schichtenaufbau einer OSGi-Architektur . . . 22
10 Lebenszyklus eines Bundle [OSG05] . . . 23
11 Architektur von Eclipse . . . 25
12 Ausprägungen zweier RCE-Installationen . . . 26
13 Architektur von RCE [KKFM+07] . . . 27
14 Anwendungsfalldiagramm . . . 28
15 Einsatzszenario . . . 32
16 Plugin als Grid Service . . . 35
17 RCE als Grid Service . . . 36
18 Reduziertes RCE als Grid Service . . . 38
19 Grid Service als Kommunikationsschnittstelle zu RCE und Plugin . . . 39
20 WS-Resource Factory Pattern nach [SC05] . . . 44
21 Klassendiagramm für die Implementierung des WS-Resource Factory Pattern . . . 45
22 Abbilden von RCE-Notifications auf Globus-Notifications . . . 48
23 Abbilden von RCE-Funktionen auf Grid-Technologien . . . 49
24 Beziehung zwischen Web-Service-Konsument, UDDI-Verzeichnis und Web Service . . . 52
25 MDS-Registrierung von Plugin Service und RFT Service (Reliable File Transfer) . . . 53

26 Aktivitätsdiagramm des Verteilprozesses . . . . . 54
27 Sequenzdiagramm für den Lebenszyklus eines Stub . . . . . 57
28 GUI des Plugin Simplorer in RCE . . . . . 58
29 Details der Properties einer Resource des Simplorer Service . . . . . 59
30 Aufbau einer WSDL-Datei . . . . . 60
31 Porttype des Simplorer Service . . . . . 61
32 Generierung einer GAR-Datei . . . . . 63
33 Sicherheitsebenen . . . . . 64
34 Klassendiagramm für den Stub . . . . . 67
35 Ergebnisse der Testfälle . . . . . 69
36 Anteil des getesteten Code (Codeabdeckung) . . . . . 70

# Tabellenverzeichnis

1 Liste der Anforderungen . . . . . 30
2 Abgrenzung der Implementierung . . . . . 31
3 Zusammenfassung der Realisierungsaspekte für: Plugin als Grid Service . . . . . 36
4 Zusammenfassung der Realisierungsaspekte für: RCE als Grid Service 37
5 Zusammenfassung der Realisierungsaspekte für: Reduziertes RCE als Grid Service . . . . . 39
6 Zusammenfassung der Realisierungsaspekte für: Grid Service als Kommunikationsschnittstelle zu RCE und Plugin . . . . . 40
7 Vergleich der Realisierungsmöglichkeiten hinsichtlich der Aspekte . . 41
8 Vergleich von GSI-Sicherheitsmodellen nach [SC05] . . . . . 51
9 Überblick eingesetzter Entwicklungswerkzeuge . . . . . 70

# Literatur

[AHLZ04] AMIN, Kaizar ; HATEGAN, Mihael ; VON LASZEWSKI, Gregor ; ZALUZEC, Nestor J.: Abstracting the Grid. In: *Proceedings of the 12 Euromicro Conference on Parallel, Distributed and Network-Based Processing*. Los Almitos, CA, USA : IEEE Computer Society Press, 2004, S. 250–257

[BBG+05] BANERJEE, Sujata ; BASU, Sujoy ; GARG, Shishir ; GARG, Sukesh ; LEE, Sung-Ju ; MULLAN, Pramila ; SHARMA, Puneet: Scalable Grid Service Discovery Based on UDDI. In: *Proceedings of the 3rd international workshop on Middleware for grid computing*. New York, NY, USA : ACM Press, 2005, S. 1–6

[BCE+07] BRANDES, Thomas ; CHRISTIANSEN, Katja ; ESINS, Eric ; KLEIN, Jürgen ; KRÄMER-FUHRMANN, Ottmar ; METSCH, Thijs ; ROSSOW, Dirk ; SCHREIBER, Andreas ; SCHRÖDTER-KLEIN, Sandra. *System Design für ein schiffbauliches Entwurfs- und Simulationssystem*. Dokument im Rahmen des Projekts SESIS. Mai 2007

[BHM+04] BOOTH, David ; HAAS, Hugo ; MCCABE, Francis ; NEWCOMER, Eric ; CHAMPION, Michael ; FERRIS, Chris ; ORCHARD, David. *Web Services Architecture*. W3C Working Group Note. Februar 2004

[BS06] BARTH, Thomas ; SCHÜLL, Anke: *Grid Computing. Konzepte, Technologien, Anwendungen*. 1. Auflage. Wiesbaden, Germany : Vieweg, März 2006. – 216 S. – ISBN 3–83480–033–3

[CER] CERN. *Grid Café*. http://gridcafe.web.cern.ch/gridcafe/index.html (Oktober 2007)

[CGP] C3-GRID-PROJEKT. *Collaborative Climate Community Data and Processing Grid*. http://c3grid.de (Oktober 2007)

[DFP+96] DEFANTI, Thomas ; FOSTER, Ian ; PAPKA, Michael ; STEVENS, Rick ; KUHFUSS, Tim: Overview of the I-WAY: Wide Area Visual Supercomputing. In: *International Journal of Supercomputer Applications and High Performance Computing* 10 (1996), Nr. 2, S. 123–131

[DRBJS03] DE ROURE, David ; BAKER, Mark A. ; JENNINGS, Nicholas R. ; SHADBOLT, Nigel R.: The Evolution of the Grid. In: BERMAN, Fran (Hrsg.) ; FOX, Geoffrey (Hrsg.) ; HEY, Tony (Hrsg.): *Grid Computing: Making the Global Infrastructure a Reality*. Chichester, West Sussex, UK : John Wiley & Sons, 2003, S. 65–100

[EGE] EGEE-PROJEKT. *Enabling Grids for E-sciencE*. http://egee.org (Oktober 2007)

[ES01] ERWIN, Dietmar ; SNELLIND, David: UNICORE - a Grid computing environment. In: SAKELLARIOU, Rizos (Hrsg.) ; KEANE, John (Hrsg.) ;

GURD, John R. (Hrsg.) ; FREEMAN, Len (Hrsg.): *Proceedings of 7th International Conference Euro-Par*. Manchester, UK : Springer, 2001, S. 825–834

[FK97] FOSTER, Ian ; KESSELMAN, Carl: Globus: A Metacomputing Infrastructure Toolkit. In: *The International Journal of Supercomputer Applications and High Performance Computing* 11 (1997), Nr. 2, S. 115–128

[FK98] FOSTER, Ian ; KESSELMAN, Carl: *The Grid: Blueprint for a new computing infrastructure*. 1. Auflage. Chicago, IL, USA : Morgan Kaufmann Publishers, November 1998. – 701 S. – ISBN 1–55860–475–8

[FKNT02] FOSTER, Ian ; KESSELMAN, Carl ; NICK, Jeffrey M. ; TUECKE, Steven: Grid Services for Distributed System Integration. In: *IEEE Computer* 35 (2002), Nr. 6, S. 37–46

[FKNT03] FOSTER, Ian ; KESSELMAN, Carl ; NICK, Jeffrey M. ; TUECKE, Steven: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In: BERMAN, Fran (Hrsg.) ; FOX, Geoffrey (Hrsg.) ; HEY, Tony (Hrsg.): *Grid Computing: Making the Global Infrastructure a Reality*. Chichester, West Sussex, UK : John Wiley & Sons, 2003, S. 217–249

[FKS+05] FOSTER, Ian ; KISHIMOTO, H. ; SAVVA, A. ; BERRY, D. ; DJAOUI, A. ; GRIMSHAW, A. ; HORN, B. ; MACIEL, F. ; SIEBENLIST, F. ; SUBRAMANIAM, R. ; TREADWELL, J. ; VON RECIH, J. *The Open Grid Services Architecture, Version 1.5*. http://www.ggf.org/documents/GFD.80.pdf (Oktober 2007). Februar 2005

[FKT01] FOSTER, Ian ; KESSELMAN, Carl ; TUECKE, Steven: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In: *The International Journal of High Performance Computing Applications* 15 (2001), Nr. 3, S. 200–222

[Fos02] FOSTER, Ian: What is the Grid? A Three Point Checklist. In: *GRIDToday* 1 (2002), Nr. 6

[Fos03] FOSTER, Ian: The Grid: A New Infrastructure for 21st Century Science. In: BERMAN, Fran (Hrsg.) ; FOX, Geoffrey (Hrsg.) ; HEY, Tony (Hrsg.): *Grid Computing. Making the Global Infrastructure a Reality*. Chichester, West Sussex, UK : John Wiley & Sons, 2003, S. 51–63

[Fos06] FOSTER, Ian: Globus Toolkit 4: Software for Service-Oriented Systems. In: *Journal of Computer Science and Technology* 21 (2006), Nr. 4, S. 513–520

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1. Auflage. Amsterdam, Netherlands : Addison-Wesley, März 1995. – 395 S. – ISBN 0–20163–361–2

[GHM+05] GRUBER, Olivier ; HARGRAVE, B. J. ; MCAFFER, Jeff ; RAPICAULT, Pascal ; WATSON, Thomas: The Eclipse 3.0 platform: Adopting OSGi technology. In: *IBM Systems Journal* 44 (2005), Nr. 2, S. 289–299

[Glo] GLOBUS. *Information Services (MDS) : Key Concepts*. http://globus.org/toolkit/docs/4.0/info/key-index.html (Oktober 2007)

[Glo05] GLOBUS. *Globus Toolkit Version 4 Grid Security Infrastructure: A Standards Perspective*. http://www.globus.org/toolkit/docs/4.0/security/GT4-GSI-Overview.pdf (Oktober 2007). September 2005

[GP] GRIP-PROJEKT. *Grid Interoperability Project*. http://www.grid-interoperability.org (August 2005)

[HGS+05] HEAD, Michael R. ; GOVINDARAJU, Madhusudhan ; SLOMINSKI, Aleksander ; LIU, Pu ; ABU-GHAZALEH, Nayef ; VAN ENGELEN, Robert ; CHIU, Kenneth ; LEWIS, Michael J.: A Benchmark Suite for SOAP-based Communication in Grid Web Services. In: *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA : IEEE Computer Society Press, 2005, S. 19–32

[Ins07] INSTANT-GRID-PROJEKT. *Grundlagen des Grid Computing*. http://instant-grid.de/?q=de/node/177 (Oktober 2007). Juni 2007

[KKFM+07] KLEIN, Jürgen ; KRÄMER-FUHRMANN, Ottmar ; METSCH, Thijs ; NURZENSKI, André ; SCHREIBER, Andreas. *RCE System Design*. Dokument im Rahmen des Projekts SESIS. Januar 2007

[LFGL01] VON LASZEWSKI, Gregor ; FOSTER, Ian ; GAWOR, Jarek ; LANE, Peter: A Java commodity Grid kit. In: *Concurrency and Computation: Practice and Experience* 13 (2001), Nr. 8-9, S. 645–662

[OAS] OASIS. *UDDI (Universal Description, Discovery and Integration)*. http://www.uddi.org (August 2007)

[Obj03] OBJECT TECHNOLOGY INTERNATIONAL. *Eclipse Platform: Technical Overview*. http://www.eclipse.org/whitepapers/eclipse-overview.pdf (Oktober 2007). Februar 2003

[OSG05] OSGI ALLIANCE. *OSGi Service Platform - Core Specification*. Dokumentation zur OSGi-Spezifikation. August 2005

[RHS05] RICHTER, Jan-Peter ; HARALD, Haller ; SCHREY, Peter. *Informatiklexikon der GI - Serviceorientierte Architektur*. http://www.gi-ev.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/118/ (Oktober 2007). 2005

[SC05] SOTOMAYOR, Borja ; CHILDERS, Lisa: *Globus Toolkit 4: Programming Java Services*. 1. Auflage. Chicago, IL, USA : Morgan Kaufmann, November 2005. – 536 S. – ISBN 0–12369–404–3

[SDS] SDSS-PROJEKT. *Sloan Digital Sky Survey*. http://www.sdss.org (Oktober 2007)

# Index

3-Punkte-Checkliste, 16

Bundle, 22

CoG Kit, 21
Container, 19

Eclipse, 24
Endpoint Reference, 43
Equinox, 24
Extension Points, 24
Extensions, 24

Fabrikmethode, 43
FAFNER, 15

GAR, 62
Globus Alliance, 19
Globus Toolkit, 19
Grid, 14
Grid Computing, 15
Grid Security Infrastructure, 50
Grid Service, 17
Grid-Bibel, 15
Gridmap, 65
GT4-Komponenten, 20

I-WAY, 15
Ian Foster, 15
Index Service, 52

jUDDI, 53

MDS, 52

OASIS, 17
OGSA, 17
Open Grid Forum, 17
OSGi, 22

Plugin Service, 28
Proxy-Zertifikat, 50

RCE, 26
RCE*, 38
RCE-API, 27
RCE-Basisdienste, 27
Resource, 19
Resource Home, 44
Resource Properties, 19

SESIS, 25
Sicherheitsdeskriptor, 51
SOAP, 18
Stub, 18

UDDI, 52
UDDI4J, 53

Virtuelle Organisation, 15

Web Service, 18
WebMDS, 52
WS-Resource, 19
WS-Resource Factory Pattern, 43
WSDL, 18
WSRF, 17

X.509-Zertifikat, 50