Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Leif Hartmann

Entwicklung eines auf Webservices basierenden Systems zur Administration von Serverpools

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Leif Hartmann

Entwicklung eines auf Webservices basierenden Systems zur Administration von Serverpools

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Kai von Luck
Zweitgutachter : Prof. Dr. rer. nat. Gunter Klemke

Abgegeben am 5. Juli 2006

Leif Hartmann

Thema der Bachelorarbeit

Entwicklung eines auf Webservices basierenden Systems zur Administration von Serverpools

Stichworte

Administration, Verteilte Systeme, Webservices, Webservices-Sicherheit, SOAP

Kurzzusammenfassung

Kleine Webhosting- und Emailprovider müssen, um mit großen Anbietern konkurrieren zu können, ihr Angebot möglichst individuell an Kundenwünsche anpassen. Diese Individualität hat eine vielfältige Serverlandschaft zur Folge, die nur schwer zu administrieren ist. In dieser Arbeit soll der Prototyp für ein Administrationssystem entwickelt werden, das speziell auf die Anforderungen einer solchen vielfältigen Serverumgebung angepasst ist. Dabei soll untersucht werden, ob Webservices als Middleware für ein derartiges System geeignet sind.

Leif Hartmann

Title of the paper

Development of a webservices based administration system for server pools

Keywords

administration, distributed systems, webservices, webservice security, SOAP

Abstract

Large web hosting and email providers offer their services at very low prices, by utilising automated administration systems. Smaller providers can compete against them, by responding to special customer preferences. Doing so results in a heterogenous server environment. The intention of this paper is to develop a prototype of a webservices based administration system, which is especially designed to interact with heterogenous server pools.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Motivation

Große Webhosting-Anbieter – und Anbieter ähnlicher Dienstleistungen, wie Email-Provider – dominieren ihren Markt durch sehr niedrige Preise. Um diese Preise gewährleisten zu können, setzen sie unter anderem auf einen hohen Grad an Automatisierung, damit sie sehr viele Aufträge bearbeiten können. Bestellt ein Kunde bei einem solchen Anbieter ein Webhosting-Paket, muss der Anbieter wenig oder gar nichts manuell erledigen, um dem Kunden die bestellten Leistungen zur Verfügung zu stellen. Die notwendigen Mail-, DNS- und Webserver werden komplett automatisch konfiguriert. Der Nachteil an diesem hohen Automatisierungsgrad liegt darin, dass solche Massenanbieter in der Regel auf individuelle Kundenwünsche – wie z. B. eine besondere Software auf dem Webserver – nicht eingehen können.

Kleinere Anbieter dieser Branche müssen sich also etwas einfallen lassen, um konkurrenzfähig zu großen Anbietern zu sein. Eine Möglichkeit die sich ihnen bietet ist, stark auf spezielle Kundenwünsche einzugehen. Für kleine Anbieter lohnt es sich Wünsche einzelner Kunden manuell umzusetzen und gegebenenfalls zusätzliche Hard- und/oder Software dafür bereitzustellen. Diese Individualität hat allerdings eine vielfältige Serverlandschaft zur Folge, die nach Möglichkeit ebenfalls nicht komplett manuell administriert werden soll, da ansonsten der Aufwand und damit letztendlich die Kosten dafür zu hoch werden.

1.2 Zielsetzung und Problemstellung

Im Rahmen dieser Arbeit soll auf die oben beschriebene Problematik eingegangen werden. Es existieren bereits diverse Administrationssysteme für Server, die jedoch in der Regel für gleichartige Serverumgebungen oder nur für einzelne Server konzipiert sind (siehe Abschnitt 2.5).

In dieser Arbeit sollen Möglichkeiten vorgestellt werden, die administrative Aufgaben in einem heterogenen Serverumfeld vereinfachen. Um dieses Ziel zu erreichen, soll ein Prototyp für ein Systems entwickelt werden, das einem Administrator ermöglicht, mehrere unterschiedliche Server zentral zu administrieren. Häufig auftretende Aufgaben sollen damit schnell und

einfach erledigt werden können. Das System soll es einem Administrator außerdem erlauben, möglichst einfach weitere Funktionalität hinzuzufügen, um es an die beteiligten Server anzupassen.

Die Anpassung an die jeweiligen Server stellt dabei eine besondere Herausforderung dar. Der Aufwand für eine solche Anpassung muss so gering sein, dass es sich lohnt sie vorzunehmen. Würde sie zu viel Zeit in Anspruch nehmen, könnte die durch die Anpassung geschaffene Funktionalität, ebenso manuell erledigt werden. Dafür muss das System einem Administrator einerseits ausreichend Hilfestellung geben, um Anpassungen schnell durchführen zu können und andererseits mächtig genug sein, um die von ihm gewünschte Funktionalität zu gewährleisten. Diese Anpassungen sollen es außerdem ermöglichen, ein und dieselbe Aufgabe auf unterschiedlichen Servern gleichzeitig auszuführen (siehe Abschnitt 2.2.2).

Da mehrere Server zentral administriert werden sollen, muss es eine Stelle geben, die mit allen Servern in Verbindung steht. Um die dafür nötige Kommunikation umzusetzen, sollen in diesem System Webservices verwendet werden. Dabei soll untersucht werden, ob diese Technologie für den Einsatz in einem solchen, sicherheitsrelevanten Kontext geeignet ist.

1.3 Aufbau der Arbeit

Diese Arbeit ist in fünf Kapitel unterteilt. Kapitel 2 erläutert die Anforderungen an das zu entwickelnde System und stellt die Abläufe aus Benutzersicht dar. In Kapitel 3 wird ein Konzept für die Entwicklung des Systems erläutert. Kapitel 4 zeigt, ob und inwiefern das vorgestellte Konzept umgesetzt wurde. Die Abschlussbetrachtung in Kapitel 5 wirft einen kritischen Blick auf das Gesamtsystem und bietet einen Ausblick auf mögliche Erweiterungen und Verbesserungen.

2 Analyse

Im folgenden Kapitel wird untersucht, welche Anforderungen an das zu entwickelnde System gestellt werden. Dazu wird auf einige beispielhafte Anwendungsfälle eingegangen, die stellvertretend für andere Abläufe genauer erläutert werden. Aus den Anwendungsfällen werden grundlegende Abläufe gebildet, die in dem System häufig ausgeführt werden.

Abschließend wird ein kurzer Vergleich zwischen Webservices und anderen Middleware-Lösungen vorgenommen und auf vorhandene Alternativen zu dem zu entwicklenden System eingegangen.

2.1 Rollen

In den folgenden Kapiteln wird von verschiedenen Rollen gesprochen, die im System vorkommen. Eine Rolle wird nicht nur von Personen angenommen, sondern auch von Komponenten des Systems. In der Regel nimmt eine Person mehrere Rollen an. Die Rollen sind im Einzelnen:

Benutzer Ein Benutzer ist ein Endanwender, der das Webinterface bedient. Da das Webinterface für die Administration von Servern entwickelt wird, sind Benutzer in der Regel Administratoren.

Administrator Administratoren sind für die Konfiguration von Servern zuständig. Sie haben Verständnis für den Aufbau ihrer Serversysteme und verfügen in der Regel über grundlegende Programmierkentnisse, die es ihnen erlauben einfache Skripte zu entwickeln.

Aufgabenentwickler Aufgabenentwickler sind Programmierer, die Aufgaben für Server erstellen und anpassen. Eine Aufgabe ist eine administrative Tätigkeit, die mit Hilfe des Administrationssystems durchgeführt wird. Beispiele für Aufgaben sind im nachfolgenden Abschnitt 2.2.2 beschrieben.

Server An dem System sind mehrere Server beteiligt, die zum Teil gleichzeitig Client sind. Wenn im Folgenden von einem Server gesprochen wird, handelt es sich dabei, soweit nicht anders erwähnt, um einen zu administrierenden Server, also um einen Rechner auf dem ein TaskServer läuft (siehe hierzu auch Abschnitt 2.4.2 und Abbildung 2.2).

Client Bei dem Client handelt es sich, wenn nicht anders erwähnt, um den Rechner, auf dem das Admintool mit dem Webinterface läuft. Dieser Rechner ist zwar gleichzeitig ein Server, der über den Browser des Benutzers angesprochen wird, nimmt in der Beschreibung jedoch nicht die Rolle eines Servers an.

Außerdem wird in den folgenden Kapiteln immer wieder von System-Benutzern gesprochen. Dabei handelt es sich um Benutzerkonten des Betriebssystems auf einem Server. Es nimmt jedoch keine Person die Rolle eines System-Benutzers an, daher sind sie nicht als Rollen, sondern als zu bearbeitende Objekte anzusehen.

2.2 Anwendungsszenario

2.2.1 Serververmietung

Eine Firma betreibt Serverhousing[1] und vermietet außerdem vorkonfigurierte Server an ihre Kunden. Neben dem Serverhousing bietet die Firma den Verkauf von Domains an. Mit einem vorhandenen Webinterface haben Kunden die Möglichkeit, Domains zu bestellen und die Nameserver der Firma für die bestellten Domains zu konfigurieren.

Als weiterer Schritt soll es nun möglich sein, neben den Nameservern, auch gleich die Web- und Mailserver, die auf den Kundenservern laufen, über das Webinterface für die bestellten Domains einzurichten.

2.2.2 Anwendungsfälle

Die nachfolgenden Anwendungsfälle stehen jeweils exemplarisch für einen Aufgabentyp.

- An der Ausführung einer *Einzelaufgabe* ist immer nur ein Server beteiligt.
- *Massenaufgaben* führen gleichwertige Operationen auf mehreren Servern aus.
- Der Anwendungsfall “System-Benutzer umziehen” steht stellvertretend für spezielle Aufgaben, die besondere Anpassungen des Systems erfordern. Näheres dazu folgt in Abschnitt 3.6.

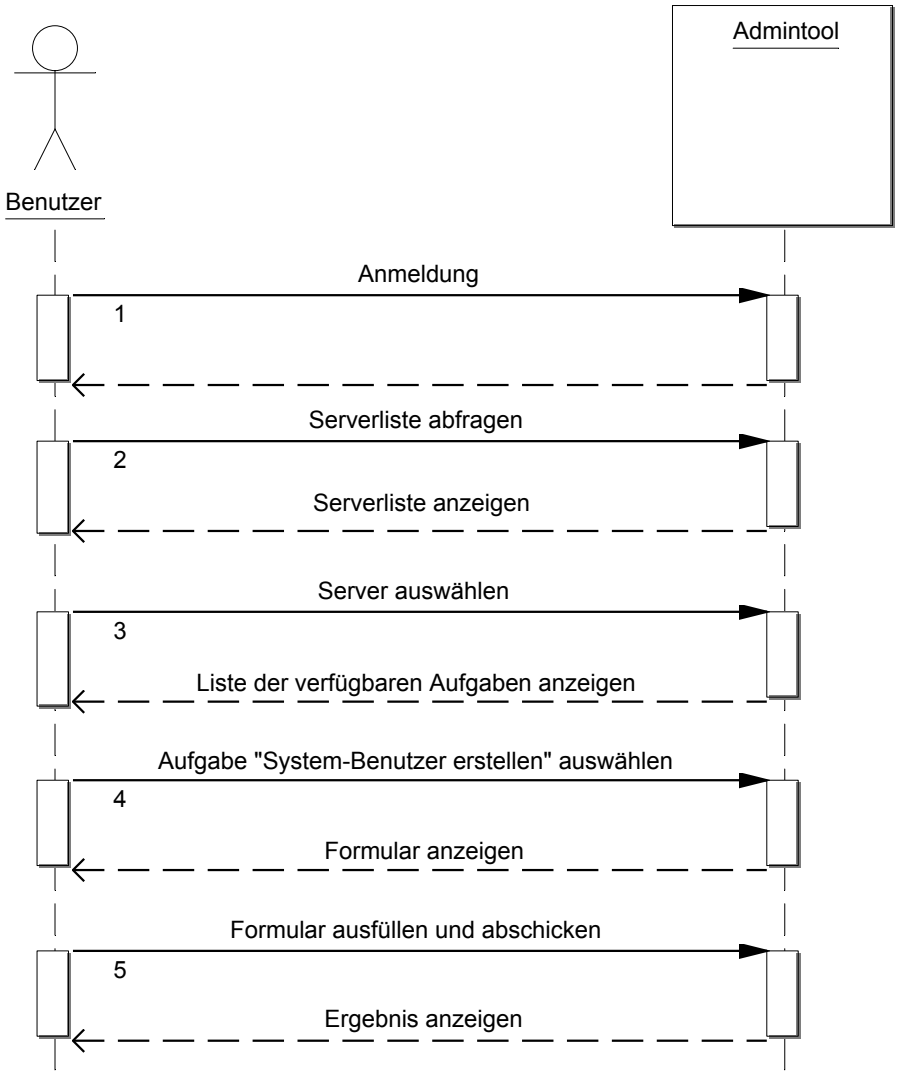

Abbildung 2.1: Ausführung der Aufgabe ’System-Benutzer erstellen’ aus Benutzersicht

Einzelaufgabe

Ein Benutzer hat eine neue Domain bestellt und möchte nun einen seiner Server als Webserver für die neue Domain konfigurieren. Dazu soll zunächst ein neuer System-Benutzer erstellt werden, in dessen Homeverzeichnis das WWW-Root-Verzeichnis[2] für den neuen virtuellen Host liegt.

Der Ablauf zum Anlegen des System-Benutzers wird in Abbildung 2.1 dargestellt. Auf eine Darstellung für das Erstellen des virtuellen Hosts wird hier verzichtet, da der Ablauf grundsätzlich identisch ist.

Der Benutzer meldet sich am Admintool an und bekommt zunächst eine Liste seiner im System registrierten Server zur Auswahl. Nachdem er den gewünschten Webserver ausgewählt hat, zeigt das Admintool an, welche Aufgaben dieser anbietet. Der Benutzer wählt nun die Aufgabe “System-Benutzer erstellen” aus und bekommt ein Formular angezeigt, in das er die benötigten Daten, wie Benutzername, Homeverzeichnis und Passwort einträgt.

Danach weist er das Admintool an die Aufgabe durchzuführen. Das Admintool ruft den angebotenen Service des ausgewählten Servers mit den Daten aus dem Formular auf und zeigt das Ergebnis der Operation an. Bei Fehlern, wenn z. B. der Benutzer bereits existiert, wird das Formular mit einer entsprechenden Fehlermeldung erneut angezeigt.

Wurde der System-Benutzer erstellt, wählt der Benutzer die Aufgabe “Virtuellen Host erstellen” aus, trägt die nötigen Daten, wie Besitzer (der eben erstellte System-Benutzer), WWW-Root-Verzeichnis (ein Unterverzeichnis des Homeverzeichnisses des System-Benutzers) und IP-Adresse (eine dem Server zugewiesene IP-Adresse) in das entsprechende Formular ein und bestätigt den Vorgang. Das Admintool ruft einen anderen Service des Servers auf und zeigt das Ergebnis an.

Massenaufgabe

Auf mehreren Servern sollen Updates auf die jeweils aktuellen Versionen der installierten Softwarepakete durchgeführt werden.

Der Benutzer meldet sich am Admintool an und wählt die gewünschten Server aus, um daraufhin die Massenaufgabe “Softwarepakete aktualisieren” auszuwählen. Das Admintool zeigt eine Liste aller Softwarepakete der jeweiligen Server an, für die neue Versionen vorliegen. Der Benutzer wählt die Pakete aus, die aktualisiert werden sollen und bestätigt den Vorgang.

[1] Der Begriff Serverhousing beschreibt das Vermieten von Serverplatz in einem Rechenzentrum. Es werden keine Server vermietet, sondern lediglich Platz in einem Rack, eine Anbindung ans Internet, Strom, etc. Die Server müssen die Kunden selbst mitbringen.

[2] Ein Verzeichnis in dem der Webserver die von ihm anzubietenden Dateien erwartet.

Das Admintool weist daraufhin alle ausgewählten Server an, die ausgewählten Softwarepakete zu aktualisierungen. Während die Aktualisierungen laufen, zeigt das Admintool deren Status an und meldet, wenn alle Aktualisierungen beendet wurden.

System-Benutzer umziehen

Ein Kunde nimmt auf einem Server sehr viel Ressourcen in Anspruch, was den Server über kurz oder lang überlasten wird. Deshalb soll der System-Benutzer des Kunden mitsamt seinen Daten auf einen anderen, performanteren Server umziehen.

Der Benutzer meldet sich dafür wie gehabt am Admintool an. Danach wählt er die Aufgabe “System-Benutzer umziehen” aus und gibt die benötigten Daten an: Zunächst wird der Quellserver, der System-Benutzer auf dem Quellserver und der Zielserver auswählt. Danach werden die Daten für den System-Benutzer auf dem Zielserver angegeben (Benutzername, Passwort, etc.).

Sind alle nötigen Daten eingegeben, startet das Admintool den Transfer des System-Benutzers, sowie dessen Dateien. Dieser Vorgang kann viel Zeit in Anspruch nehmen, deshalb zeigt das Admintool, wie auch beim Aktualisieren der Softwarepakete, in periodischen Abständen den Status des Vorgangs an.

2.3 Anforderungen

Aus den Anwendungsfällen lassen sich folgende Anforderungen ableiten:

Ausführen von Einzelaufgaben Es soll möglich sein administrative Aufgaben über das Webinterface durchzuführen.

Ausführen von Massenaufgaben Über das Webinterface sollen gleiche Aufgaben auf mehreren Servern ausgeführt werden können, ohne diese für jeden Server einzeln aufzurufen.

Ausführen von länger währenden Aufgaben Das Admintool soll Aufgaben starten können, die längere Zeit in Anspruch nehmen, ohne das Webinterface in dieser Zeit zu blockieren. Für diese Aufgaben soll eine Abfrage den aktuellen Status liefern.

Anpassbare Aufgaben Serverspezifische Daten – wie z. B. Verzeichnisse oder Pfade zu externen Programmen, die von den Aufgaben benötigt werden – sollen an den jeweiligen Server anpassbar sein. So soll erreicht werden, dass an der Ausführung einer Massenaufgabe auch unterschiedliche Server teilnehmen können.

Heterogenität Da das System für vielfältige Serverumgebungen ausgelegt ist, sollen über das Webinterface Server mit unterschiedlichen Betriebssystemen und unterschiedlicher installierter Software administrierbar sein. Gleiche Aufgaben sollen auf unterschiedlichen Systemen unterschiedlich ausgeführt werden können.

Einbinden in vorhandene Oberflächen Die Architektur soll ermöglichen, das Admintool in eine bestehende Oberfläche zu integrieren, um es beispielsweise in vorhandenen Kundenbereichen zur Verfügung zu stellen.

Modularer Aufbau Das System soll derart modular aufgebaut sein, dass ohne großen Aufwand neue Aufgaben definiert werden können. Ein Administrator eines Servers soll selbst dazu in der Lage sein, neue Aufgaben für seinen Server in das System einzubinden. Das Erstellen neuer Aufgaben soll dabei möglichst schnell und einfach sein.

Rechtevergabe Benutzer sollen nur die Server administrieren können, auf die sie Zugriff haben.

2.4 Grundlegende Abläufe

Aus den Anwendungsfällen und Anforderungen lassen sich folgende allgemeine Schritte generalisieren:

Benutzer-Authentifizierung Vor jeder Ausführung einer Aufgabe muss sichergestellt werden, dass es sich bei dem Benutzer um denjenigen handelt, der er vorgibt zu sein.

System-Authentifizierung Bei jeder Anfrage, die das Admintool an einen Server stellt, muss auf Serverseite geprüft werden, ob das Admintool berechtigt ist Aufgaben auszuführen.

Auswahl einer Aufgabe Nach erfolgreicher Authentifizierung wird entweder ein Server ausgewählt, auf dem eine Einzelaufgabe ausgeführt werden soll oder eine Massenaufgabe, die auf mehreren Servern ausgeführt werden soll.

Abfrage der unterstützten Aufgaben Um eine Aufgabe ausführen zu können, wird vorher an den ausgewählten Server eine Anfrage gestellt, welche Aufgaben dieser anbietet.

Ausführung einer Aufgabe Die Ausführung einer Aufgabe ist nur selten ein einfacher, linear ablaufender Prozess, sondern meistens eine Abfolge von Anfragen und Antworten zwischen dem Benutzer und dem Server. Das Admintool bildet dabei die Schnittstelle zwischen Benutzer und Server. Es zeigt die Antworten des Servers an und bietet dem Benutzer die Möglichkeit, die für die Aufgaben benötigten Details anzugeben.

Die minimale Aufgabe besteht genau aus einer Anfrage und einer Antwort. Um beispielsweise eine Liste der auf einem Server vorhandenen System-Benutzer anzuzeigen, wird eine Anfrage für diese Aufgabe an den Server geschickt. Als Antwort erhält das Admintool die Liste der System-Benutzer und zeigt diese dem Benutzer an.

2.4.1 Massenaufgaben

Der Aufruf einer Massenaufgabe gestaltet sich für den Benutzer etwas anders, als der Aufruf einer Einzelaufgabe:

1. Auswählen der Aufgabe: Zunächst wird eine Liste aller als Massenaufgabe registrierten Aufgaben angezeigt, aus der der Benutzer eine auswählt.
2. Abfrage der unterstützten Server: Das Admintool weist jeden Server des Benutzers an, die unterstützten Aufgaben zu übermitteln und zeigt diejenigen an, die die ausgewählte Aufgabe unterstützen.
3. Auswählen der Server: Nachdem das Admintool nun Informationen darüber hat, welche Server die gewünschte Aufgabe ausführen können, zeigt es eine Liste dieser Server an. Der Benutzer wählt dann die Server aus, auf denen die Massenaufgabe ausgeführt werden soll.
4. Ausführen der Aufgabe: Die eigentliche Ausführung der Aufgabe ist in Abschnitt 3.4 genauer beschrieben.

2.4.2 Grundlegender Aufbau

Wie in Abbildung 2.2 dargestellt, wird das System aus mindestens zwei grundlegenden Komponenten bestehen:

Admintool Das Admintool bildet eine Schnittstelle zwischen dem Benutzer und den auszuführenden Aufgaben auf den einzelnen Servern. Über das Webinterface des Admintools kann der Benutzer auf die Aufgaben zugreifen und sie ausführen lassen.

TaskServer Auf jedem im System registrierten Server läuft ein TaskServer, der verschiedene Dienste anbietet, um administrative Aufgaben zu erledigen.

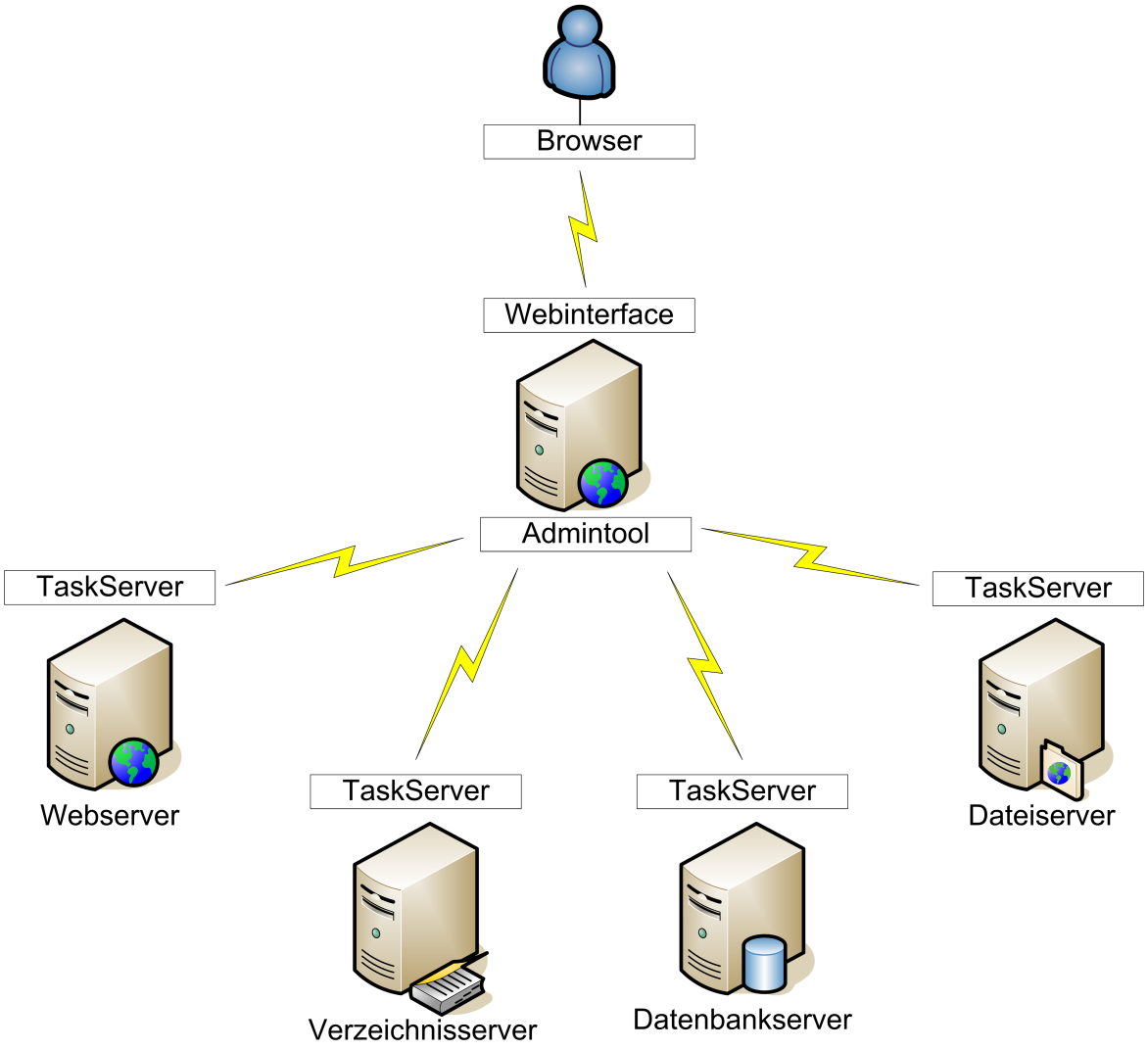

Abbildung 2.2: Grundlegender Aufbau

2.5 Vergleich mit vorhandenen Lösungen

Es existieren bereits einige (open source) Tools, die ein Webinterface für die Administration von Servern bereitstellen.

In der Regel läuft dafür auf dem zu administrierendem Server ein eigener Webserver, der mit entsprechenden Rechten ausgestattet ist, um Änderungen am System vorzunehmen. Ein populärer Vertreter ist Webmin [Webmin (2006)], das für viele Unixderivate und mittlerweile auch für Windows verfügbar ist. Webmin ist modular aufgebaut, sodass es um eigene Funktionalität erweitert werden kann.

Diese Module werden über den von Webmin mitgelieferten Webserver angeboten. Es ist theoretisch möglich Webmin-Module in jeder beliebigen Programmiersprache zu schreiben. Um Module jedoch in die Webmin-Oberfläche einzubinden, werden Perl-Funktionen verwendet, weshalb die Verwendung von Perl von den Entwicklern angeraten wird.

Während Webmin viele Module für die Administration von verschiedenen Systemkomponenten anbietet, existieren andere Tools, die speziell für die Administration von einzelnen Systemkomponenten, wie Webserver, Datenbank oder Mailserver entwickelt wurden. Diese Tools sind zwar in der Regel mächtiger, als komponentenübergreifende Lösungen, jedoch müssen für einen Server mehrere solcher Tools installiert und gewartet werden.

Das in dieser Arbeit zu entwickelnde System ist eher mit einer allgemeinen Lösung, wie Webmin, zu vergleichen, als mit spezialisierten Tools. Es muss dabei aber nicht zwangsläufig ein Ersatz dafür bilden. Während Webmin viele Möglichkeiten bietet, um Serversoftware sehr individuell zu konfigurieren, geht der Ansatz dieses Systems in eine andere Richtung. Es wird weniger Wert darauf gelegt, die auf den Servern laufenden Softwarekomponenten bis ins kleinste Detail konfigurieren zu können, sondern häufig auftretende Aufgaben schnell zu erledigen.

Außerdem soll das System speziell für die Administration von vielen Servern entwickelt werden, sodass ein Administrator sich nicht auf allen Servern separat anmelden muss, um die selbe Aufgabe durchzuführen. Es soll vielmehr die Möglichkeit geben eine Verwaltungsoberfläche für alle Server zur Verfügung zu stellen, die alle Aufgaben in einem Look-and-Feel vereint.

Die gemeinsame Oberfläche soll außerdem die Möglichkeit bieten, Aufgaben auf mehreren Servern gleichzeitig auszuführen und spezielle Aufgaben zu erledigen, die mehrere Server betreffen, wie das Umziehen von System-Benutzern.

2.6 Wahl der Middleware

Aus den vorherigen Abschnitten wird deutlich, dass das Admintool und die TaskServer in der Regel sowohl logisch, als auch physisch getrennt sind. Zwischen ihnen liegt im optimalen Fall ein abgetrenntes, lokales Netz. Allerdings kann auch das (unsichere) Internet dazwischen liegen, beispielsweise wenn mit dem Admintool Server aus unterschiedlichen Rechenzentren administriert werden. Im Folgenden werden in Kürze zwei populäre Technologien vorgestellt, die als Alternative für die Kommunikation zwischen Admintool und TaskServer in Frage kommen, sowie die Vor- und Nachteile im Vergleich mit Webservices erläutert. Ausführlichere Informationen über CORBA, Java RMI und Middlewares im Allgemeinen sind den Quellen [Tanenbaum und van Steen (2002)] und [Coulouris u. a. (2002)] zu entnehmen.

2.6.1 CORBA und Java RMI

CORBA (Common Object Request Broker Architecture) [CORBA (2006)] und Java RMI [JavaRMI (2003)] basieren auf dem Prinzip verteilter Objekte. Dadurch ist es Entwicklern möglich, auf entfernte Objekte wie auf lokale Objekte zuzugreifen.

CORBA ist nicht nur sprach- und plattformunabhängig, sondern auch sehr ausgereift. Allerdings ist CORBA recht kompliziert einzurichten und es bedarf nach einer Änderung der Schnittstellendefinition viel Aufwand auf Client- und auf Serverseite, um die Änderungen nutzen zu können.

Java RMI ist zwar soweit plattformunabhängig, wie Java selbst, allerdings nicht sprachunabhängig. Sprachunabhängigkeit ist zwar keine explizite Anforderung an das System, es wäre allerdings wünschenswert. So könnte ein Administrator, einen unter .NET laufenden TaskServer für Windows verwenden, während ein anderer auf Java basierender TaskServer unter Linux verwendet wird.

Ein weiterer Punkt, der gegen CORBA und Java RMI spricht ist, dass beide dynamisch Ports für die Kommunikation öffnen. Da die meisten TaskServer durch eine Firewall geschützt sein werden, die eingehenden Verbindungen bis auf einige ausgewählte Ports blockiert, ist zusätzlicher Aufwand nötig, um dieses Problem zu umgehen. Java RMI bietet dafür die Möglichkeit Anfragen und Antworten über HTTP zu tunneln und auch für CORBA gibt es Lösungen, die allerdings ebenfalls hohen Einarbeitungsaufwand erfordern. Weitere Informationen über das Firewallproblem sind den Quellen [Brose und Staamann (2002)] und [RMIfw (2003)] zu entnehmen.

2.6.2 Webservices

Es existiert eine Vielzahl von Webservices-Definitionen, die zum Teil sehr allgemein und zum Teil sehr speziell gehalten sind. In der Webservices-Sektion des IBM-Entwicklerportals alphaWorks, wird der Begriff Webservice wie folgt definiert:

> Web services are technologies that allow applications to communicate with each other in a platform- and programming language-independent manner. A Web service is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging. It uses protocols based on the XML language to describe an operation to execute or data to exchange with another Web service. [alphaWorks (2005)]

Webservices kommunizieren mittels Austausch von XML-Nachrichten, die über das SOAP-Protokoll [SOAP (2003)] ausgetauscht werden können. Weiterhin sind sie, ebenso wie CORBA, sprachunabhängig.

Ein wesentlicher Unterschied zwischen den bisher genannten Lösungen und Webservices ist, dass CORBA und Java RMI entfernte Objekte ermöglichen, während Webservices nachrichtenbasiert arbeiten. Enfernte Objektreferenzen gibt es also nicht. Es ist jedoch möglich komplette Objekte zu serialisieren und so auszutauschen.

Das bereits angesprochene Firewallproblem besteht bei Webservices nicht, da sie in der Regel über HTTP kommunizieren, wofür lediglich die Öffnung eines eingehenden Ports nötig ist.

Es gibt allerdings zwei wesentliche Gründe, die gegen Webservices sprechen:

- Sicherheit: Der SOAP-Standard beschreibt keine Mechanismen für Verschlüsselung, Authentifizierung und Autorisierung. Dieses Problem ist jedoch lösbar. Abschnitt 3.7 beschäftigt sich näher mit diesem Thema.
- Performance: Auf Grund des Nachrichtenformats XML muss von der Middleware viel Aufwand betrieben werden, um Nachrichten zu parsen. Außerdem sind diese im Verhältnis zu binären Nachrichten, wie sie etwa von CORBA verschickt werden, relativ groß.

  Da die Service-Methoden, die für das zu entwickelnde System benötigt werden, nur relativ selten aufgerufen werden, ist hier die Geschwindigkeit allerdings von geringer Bedeutung. Ein ausführlicher Performance-Vergleich von CORBA und Webservices ist der Quelle [Gray (2004)] zu entnehmen.

Für weitere Informationen über die Funktionsweise von Webservices, sei auf die Quelle [Knuth (2003)] verwiesen.

2.6.3 Entscheidung

Da die Nachteile von CORBA und Java RMI die von Webservices überwiegen, werden für die Entwicklung des Administrationssystems Webservices eingesetzt. Außerdem soll im Rahmen der Arbeit geprüft werden, ob diese Technologie für diesen Anwendungsbereich geeignet ist.

3 Design

Die in der Analyse erstellten Anforderungen und grundlegenden Abläufe werden in diesem Kapitel in ein Konzept umgesetzt. Dabei ist des öfteren von Formularen die Rede. Formulare werden als Träger von Informationen verwendet. Durch sie werden Informationen vom Benutzer abgefragt und dem TaskServer übermittelt.

Die Protokollbeschreibung bezieht sich auf Formulare, die zwischen Admintool und TaskServer ausgetauscht werden. Das Admintool übernimmt das Umwandeln dieser Formulare in HTML-Formulare, die den Benutzer angezeigt werden. Der Aufbau von Formularen ist im Protokoll (3.2.3) genauer beschrieben.

3.1 Architektur

Das System verwendet eine Client/Server-Architektur, wobei es hierbei einen Client gibt, der auf mehrere Server zugreift (siehe Abbildung 3.1). Das Admintool (in Verbindung mit dem Browser des Benutzers) bildet im Wesentlichen die Präsentationsschicht, während der TaskServer die Logik ausführt. Das Admintool ist gleichzeitig ein Server, der über den Browser des Benutzers angesprochen wird. Auf die Kommunikation zwischen Browser und Admintool wird hier nicht weiter eingegangen, da es sich dabei um eine “klassische” Webanwendung handelt. Daher ist im Folgenden stets das Admintool gemeint, wenn von dem Client gesprochen wird.

Präsentation Die Präsentationsschicht bildet die Schnittstelle zum Benutzer. In dieser Arbeit wird dafür ein Webinterface verwendet, es spricht allerdings grundsätzlich nichts dagegen eine andere Oberfläche zu verwenden.

Datenanpassung Die Datenanpassungsschicht hat zwei Aufgaben. Sie wandelt die Daten, die der Server liefert in ein Format um, das für die Präsentation geeignet ist. Insbesondere werden hier Formulare (siehe Abschnitt 3.2.3), die der TaskServer als Teil einer Antwort liefert, konvertiert.

Außerdem werden die Daten, die der Benutzer eingibt, in ein für den TaskServer verständliches Format umgewandelt.

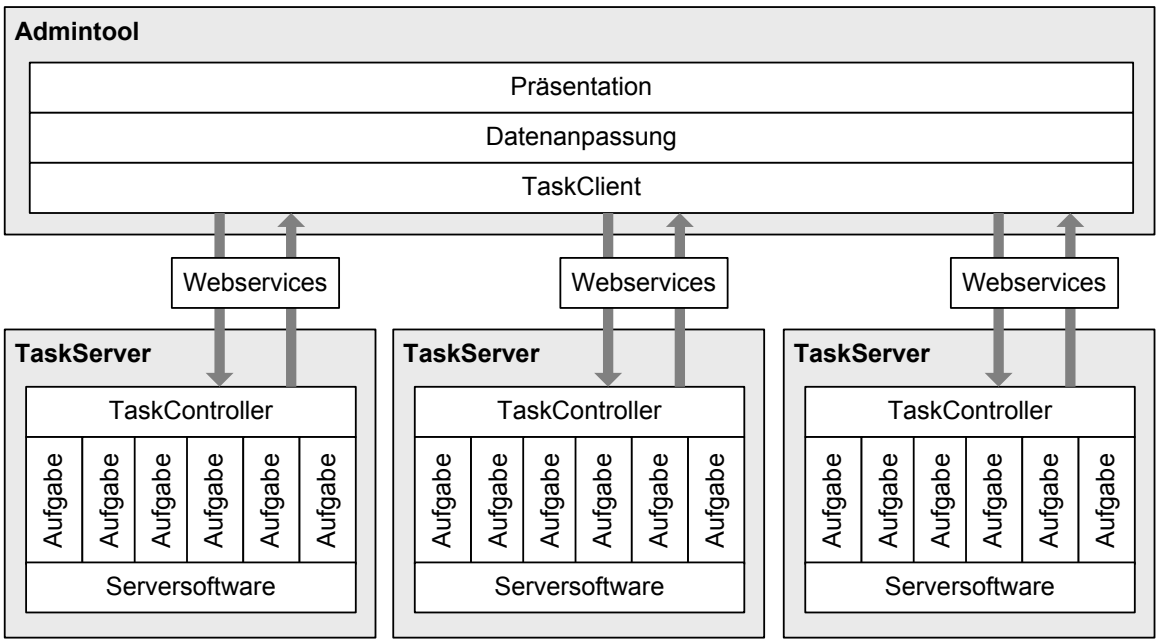

Abbildung 3.1: Architektur

TaskClient Der TaskClient übernimmt die Kommunikation mit dem TaskServer, indem er die vom TaskController angebotenen Webservice-Methoden aufruft.

TaskController Der TaskController übernimmt die Kommunikation mit dem TaskClient auf Serverseite und führt die Aufgaben aus.

Aufgabe Ein TaskController kann beliebig viele Aufgaben ausführen. Die Aufgaben sind dafür zuständig die Systemkonfiguration des Servers zu ändern. Die Aufgaben können Software installieren, deinstallieren und umkonfigurieren. Was genau die Aufgaben an der Serversoftware verändert, hängt von der Implementierung der Aufgaben ab. Eine genauere Beschreibung der Aufgaben folgt in Abschnitt 3.5.

Serversoftware Die Serversoftware beinhaltet das Betriebsystem und die unter dem Betriebssystem installierte Software.

Um das Webinterface gut zu strukturieren, wird es nach den Vorgaben des Model View Controller (MVC) Design Patterns konzipiert. Dadurch wird eine saubere Trennung der Daten (Model), der Logik (Controller) und der Benutzerschnittstelle (View) erreicht. Eine genauere Beschreibung des MVC-Patterns ist der Quelle [MVC (2002)] zu entnehmen.

3.1.1 Client/Server-Funktionalität

Eine Frage, die sich stets bei der Verwendung einer Client/Server-Architektur stellt ist, wieviel Funktionalität der Server und wieviel der Client übernimmt. Alternative Administrationstools, wie das in der Analyse vorgestellte Webmin (siehe 2.5), verwenden einen Thin-Client. Der Client ist in dem Fall der Browser, welcher lediglich für die Bedienung des Servers zuständig ist.

Ein Thin-Client reicht in diesem System nicht aus, da der Client zusätzliche Aufgaben hat:

- Koordination von Aufgaben: Insbesondere bei Massenaufgaben und “speziellen” Aufgaben, wie dem Umziehen von System-Benutzern, muss das Admintool die Ausführung koordinieren.
- Rechteverwaltung: Das Admintool muss sicherstellen, dass ein Benutzer nur Zugriff auf seine Server hat. Auch eine Thin-Client-Lösung könnte dies sicherstellen, indem die Rechte auf Serverseite geprüft werden. Allerdings können mit dem Admintool mehrere Server von einem Benutzer administriert werden und ein einzelner Server hat keine Verbindung zu anderen Servern des selben Benutzers. Daher fällt die Aufgabe der Rechteüberprüfung dem Client zu, der als einziger Informationen darüber hat, auf welche Server ein Benutzer Zugriff hat.

Aus den genannten Gründen scheidet ein Thin-Client also aus. Aber auch ein Fat-Client, also ein Client, der fast die gesamte Funktionalität beinhaltet, kommt nicht in Frage. Da die Serverlandschaft vielfältig sein kann und somit die Aufgaben an den Server anpassbar sein müssen, muss der Server viel Funktionalität selbst übernehmen. Außerdem soll es möglich sein, neue Aufgaben für einen Server zu definieren, ohne das Admintool verändern zu müssen.

Daher wird das Admintool als Rich-Client konzipiert, wobei sowohl Server, als auch Client gleichermaßen Funktionalität übernehmen.

3.2 Protokoll

Um sowohl TaskServer als auch Admintool austauschbar zu gestalten, muss zunächst eine Protokolldefinition vorgenommen werden. Dadurch wird erreicht, dass alternative TaskServer-Implenentierungen sowieso alternative Admintool-Implementierungen verwendet werden können, wenn sich diese an das Protokoll halten.

In diesem Abschnitt wird der Ablauf der Kommunikation zwischen TaskServer und Admintool genauer beschrieben.

3.2.1 Informationstypen

In der Protokollbeschreibung wird zwischen zwei Typen von Informationen unterschieden:

Benutzerinformationen Dies sind direkte Informationen für den Benutzer. Dabei handelt es sich um Informationen, die von Menschen interpretiert werden, wie z. B. die Bezeichnung eines Formularfeldes oder Fehlermeldungen.

Unter diesen Informationstyp fallen auch lokalisierte Meldungen. Um das Interface mehrsprachlich zu halten, muss also auch in der Aufgabenbeschreibung und im Protokoll auf Mehrsprachlichkeit geachtet werden.

Systeminformationen Hierbei handelt es sich um Informationen für die Kommunikation zwischen dem Admintool und dem TaskServer. Dies sind Daten, die vom System interpretiert werden, wie z. B. Steuernamen von Formularfeldern oder auszuführende Befehle.

Außerdem gibt es Informationen, die von Mensch und System interpretiert werden, beispielsweise der Wert von Formularfeldern.

3.2.2 Aufgabeliste abfragen

Die Liste der verfügbaren Aufgaben kann vom TaskController über folgendene Service-Methode abgefragt werden:

```
TaskDescriptor[] getAvailibleTasks()
```

Der Rückgabewert ist ein Array von *TaskDescriptor*-Objekten. Diese Objekte enthalten zwei Attribute. Das Attribut *name* enthält einen eindeutigen Steuernamen, über den die Aufgabe identifiziert und ausgeführt werden kann (siehe Parameter *taskName* in Abschnitt 3.2.3). Das Attribut *description* enthält eine Bezeichnung unter der der Benutzer die Aufgabe identifizieren kann.

3.2.3 Ausführung von Einzelaufgaben

Die Ausführung einer Aufgabe geschieht auf Serverseite durch das (in der Regel mehrfache) Aufrufen einiger Service-Methoden, die der TaskServer anbietet. Die Signaturen der Methoden sehen wie folgt aus:

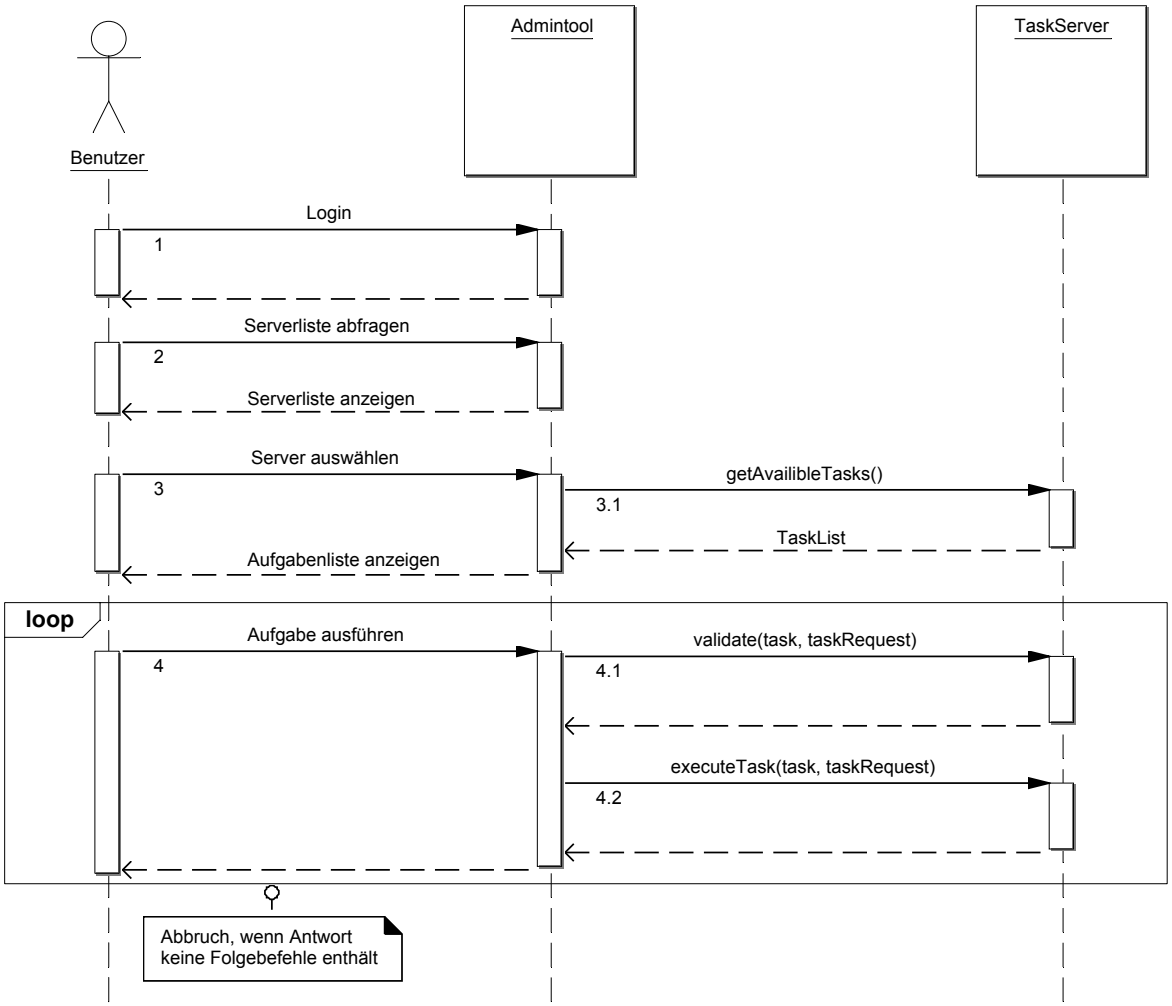

Abbildung 3.2: Ausführung einer Einzelaufgabe

```
TaskResponse executeTask(String taskName,
                         TaskRequest request);

TaskResponse validate(String taskName,
                         TaskRequest request);

TaskResponse rollBack(String taskName,
                         TaskRequest request);
```

Die Methode *executeTask* führt die eigentliche Logik aus, während *validate* den übergebenen *TaskRequest* auf korrekte Werte überprüft. Wenn *validate* keine Fehler zurückgibt, wird *executeTask* mit den selben Parametern ausgeführt. Werden bei dem Aufruf der *validate*-Methode Fehler festgestellt, muss das Admintool den *TaskResponse*, den die *validate*-Methode zurückgibt anzeigen. Dieses *TaskResponse*-Objekt muss das Formular selbst erneut enthalten, mit entsprechenden Fehlermeldungen, die den Formularfeldern zugeordnet werden können.

Die Methode *rollBack* macht die Auswirkungen von Aufgaben rückgängig. Näheres hierzu folgt im Abschnitt “Massenaufgaben” (3.4).

Der Parameter *taskName* muss den Steuernamen einer im TaskServer registrierten Aufgabe enthalten. Das Objekt *TaskRequest* enthält alle für die Ausführung der Aufgabe relevanten Daten. Die Objekte *TaskRequest* und *TaskResponse* werden im folgenden Abschnitt genauer beschrieben.

Aufgaben können in mehrere Teilausführungen unterteilt werden, um komplexere Abläufe zu strukturieren. Diese Teilausführungen werden im Folgenden als Aufgabenbefehle bezeichnet. So kann beispielsweise ein Aufgabenbefehl für das Anzeigen eines Formulars zuständig sein, während ein weiterer die Daten aus dem Formular verarbeitet. Als abgeschlossen gilt eine Aufgabe, wenn die Aufgabenantwort keine weiteren Aufgabenbefehle (im Folgenden als Folgebefehle bezeichnet) enthält (siehe Abbildung 3.2). Gestartet wird eine Aufgabe üblicherweise mit dem Aufgabenbefehl *initCommand*, das heißt, wenn dem Admintool nicht explizit ein Aufgabenbefehl mitgeliefert wird, ruft es die Aufgabe mit diesem Befehl auf.

Klassenbeschreibung

Die Objekte *TaskRequest* und *TaskResponse* sind reine Datenobjekte und enthalten keinerlei Logik. Für die Logik sind die Aufgaben auf dem TaskServer zuständig. Ein konzeptionelles Schema ist in Abbildung 3.3 zu sehen.

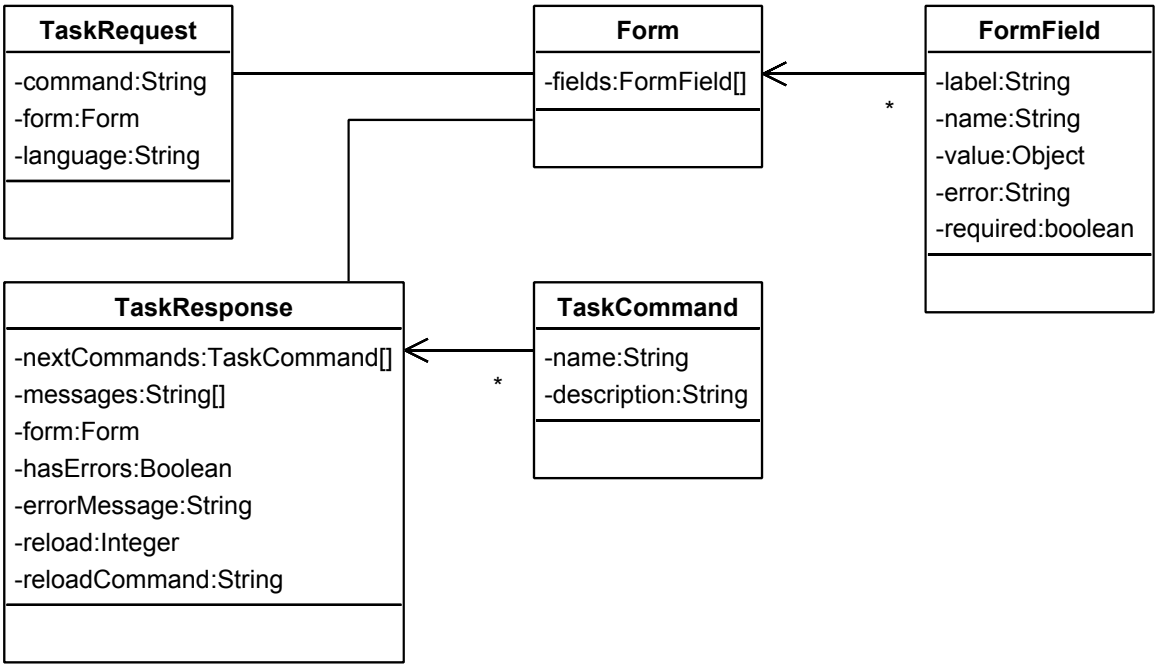

Abbildung 3.3: Klassenübersicht: *TaskRequest* und *TaskResponse*

TaskRequest Ein *TaskRequest* besteht immer aus einem Aufgabenbefehl (*command*) und optional einem Formular (*form*). Der Befehl ist ein pro Aufgabe eindeutiger Name, der von der Aufgabe interpretiert wird (Systeminformation).

Das Attribut *language* gibt an, in welcher Sprache die Benutzerinformationen der Aufgabenantwort vom TaskServer geliefert werden sollen.

TaskResponse Ein *TaskResponse* muss ein Formular und/oder eine oder mehrere Nachrichten (*messages*) enthalten. Sind ein oder beide Attribute gesetzt, werden das Formular bzw. die Nachrichten angezeigt (Benutzerinformationen). Ist keines der beiden Attribute gesetzt, ist die Antwort ungültig, da sie keine Informationen enthält.

Das Attribut *nextCommands* enthält die möglichen Folgebefehle, die wieder über die Methode *executeTask* ausgeführt werden können. Sind diese leer oder nicht gesetzt, ist die Ausführung der Aufgabe beendet.

Fehlermeldungen, die sich nicht auf ein spezielles Formularfeld beziehen, werden in *errorMessage* gespeichert. Das Attribut *hasErrors* gibt an, ob ein allgemeiner Fehler oder ein Formularfeld-Fehler im *TaskResponse* vorhanden ist.

Das Attribut *reloadCommand* (String) kann den Namen eines gültigen Aufgabenbefehls enthalten, wie z. B. “status”. Dieser Aufgabenbefehl wird nach einer bestimmten

Zeit, die mit dem Attribut *reload* (Integer) festgelegt wird, automatisch ausgeführt. Näheres dazu folgt in Abschnitt 3.3.

Form Das Formular dient als Träger für Informationen, die von dem Benutzer angegeben werden und ist optional. Es beinhaltet eine Liste von Feldern (*fields*), die die relevanten Daten für das Ausführen der Aufgabe enthalten.

FormField Ein Formularfeld wird durch folgende Attribute beschrieben:

- *name* (Systeminformation) enthält einen formularweit eindeutigen Namen zur Identifizierung des Feldes.
- *label* (Benutzerinformation) enthält den Namen des Feldes, der dem Benutzer angezeigt wird.
- *value* (System- und Benutzerinformation) enthält den Wert eines Feldes. Wenn das Formular zum ersten Mal angezeigt wird, kann *value* als Initialwert verwendet werden. In diesem Fall ist das Formular Bestandteil eines *TaskResponse*-Objektes. Hauptsächlich wird dieses Attribut allerdings für die Übertragung von Benutzereingaben (via *TaskRequest*) verwendet.
- *error* (Benutzerinformation) enthält eine Fehlermeldung, falls ein ungültiger Wert angegeben wurde.
- *required* (Systeminformation) gibt an, ob die Angabe eines Wertes in dieses Feld erforderlich ist.

TaskCommand Die Klasse *TaskCommand* enthält nichts weiter, als einen eindeutigen Namen, der als Steuerbefehl für *TaskRequest*-Objekte dient und eine Bezeichnung unter der der Benutzer den Befehl identifizieren kann.

3.2.4 Protokoll für Massenaufgaben

Der bereits vorgestellte Ablauf der Aufgabenausführung ändert sich für den TaskServer auch bei Massenaufgaben nicht. Um eine Massenaufgabe auszuführen, werden die “normalen” Einzelaufgaben verwendet. Das Admintool vereinigt die Antworten, also die Formulare und Nachrichten, für mehrere Server. Daher ist das Protokoll für die Ausführung von Massenaufgaben nicht weiter anzupassen.

Es sind allerdings zusätzliche Mechanismen nötig, um mehr Kontrolle über die Ausführung des Massenaufgaben einzubringen. Dazu gehören die Methoden *validate* und *rollBack*. Näheres dazu wird im folgenden Abschnitt 3.4 beschrieben.

3.3 Lange laufende Aufgaben

Die Ausführung einiger Aufgaben nimmt unter Umständen sehr viel Zeit in Anspruch. Wenn ein Benutzer eine solche Aufgabe per Browser ausführt, ist es ungünstig, wenn der Browser über die gesamte Zeit auf eine Antwort vom Admintool wartet. In diesem Fall würde der Browser nach einer gewissen Zeit einen Fehler wegen Zeitüberschreitung melden und abbrechen. Deshalb müssen solche Aufgaben im Hintergrund ablaufen und der Benutzer muss nach Ablauf einer Aufgabe informiert werden.

Grundsätzlich gibt es für solche Fälle zwei Möglichkeiten. Entweder der TaskServer teilt dem Admintool (und im Endeffekt dem Benutzer) mit, wenn die Aufgabenausführung beendet wurde oder das Admintool fragt immer wieder den Status der Ausführung ab und zeigt sie dem Benutzer an.

Um die erst genannte Variante umzusetzen, muss ein Mechanismus geschaffen werden, der es dem TaskServer erlaubt dem Admintool den Status zu übermitteln. Eine Möglichkeit dies zu bewerkstelligen, ist auf dem Admintool auch einen Webservice anzubieten, der diese Statusmeldungen entgegen nimmt und weiter verarbeitet. Dieser Ansatz ist aus zwei Gründen nachteilhaft. Zum einen muss das Admintool einen Webservice-Server zur Verfügung stellen und zum anderen bleibt das Problem, dass der Status auch noch dem Benutzer mitgeteilt werden muss. Das Admintool muss also von sich aus dem Browser des Benutzers die Statusmeldung mitteilen.

Die zweite Variante lässt sich deutlich einfacher implementieren. Sobald der TaskServer eine lange laufende Aufgabe im Hintergrund startet, fügt er der folgenden Aufgabenantwort ein Feld bei, das das Admintool anweist nach einer bestimmten Zeit den Status abzufragen. Zur Antwort der Statusabfrage selbst, wird ebenfalls ein solches hinzugefügt, es sei denn die Aufgabe wurde bereits beendet.

Diese Variante lässt sich gut auf die Kommunikation zwischen Admintool und Browser übertragen. Aufgabenantworten werden dem Browser immer als serverseitig generierte HTML-Seite zugeschickt, die der Browser dem Benutzer anzeigt. In die Antwortseite einer Statusabfrage wird nun ebenfalls eine Anweisung eingefügt, die den Browser dazu auffordert nach einer bestimmten Zeit den Status erneut abzufragen. Auch hier geschieht dies nur, wenn die Aufgabe noch nicht beendet ist.

Im Beispiel “Softwarepakete aktualisieren” werden die Attribute zum Steuern der automatischen Aktualisierung gesetzt. Das Attribut *reloadCommand* erhält den Wert *status* und *reload* den Wert 3, wenn der Systemprozess für die Aktualisierung noch läuft. Dies hat zur Folge, dass der Browser vom Admintool angewiesen wird, nach drei Sekunden eine URL

aufzurufen, die den Aufgabenbefehl *status* für die aktuelle Aufgabe ausführt. Ist die Aktualisierung beendet, werden die Attribute auf *null* gesetzt, sodass der Browser nicht mehr angewiesen wird den Status automatisch abzufragen.

3.4 Ausführung von Massenaufgaben

Es ist grundsätzlich möglich jede Aufgabe als Massenaufgabe auszuführen, allerdings macht es nur für wenige wirklich Sinn. Die einzelnen Aufgabenimplementierungen müssen auf jedem Server dem selben Befehlsprotokoll folgen. Das heißt, sie müssen die selben Aufgabenbefehle verstehen können und für jeden Aufgabenbefehl die selben Folgebefehle definieren. Die Implementierung der einzelnen Befehle kann durchaus unterschiedlich sein, wobei diese Unterschiede durch die unterschiedlichen Server zustande kommen. So ist es möglich eine Massenaufgabe auszuführen, an der unterschiedliche Systeme beteiligt sind.

Dabei sind zwei Arten zu unterscheiden, wie Aufgabenbefehle, bzw. deren Antworten zu verwenden sind:

1. Ein Formular pro beteiligtem Server: In diesem Fall wird für jeden Server ein eigenes Formular angezeigt, welche in einer Ansicht vereint werden. Lediglich die möglichen Folgebefehle werden nur einfach angezeigt.

   Auf diese Weise kann beispielsweise ein Update der Softwarepakete durchgeführt werden. Um dem Benutzer mehr Kontrolle darüber zu geben, welche Pakete aktualisiert werden sollen, kann er für jeden Server die Pakete separat auswählen.

2. Ein Formular für alle beteiligten Server: Der Benutzer kann so über ein einziges Formular die Daten für eine Aufgabe angeben, die auf mehreren Servern ausgeführt wird. Auch hier werden die Folgebefehle nur einfach angezeigt.

   So könnte beispielsweise ein neuer System-Benutzer für einen Backupzugang auf allen ausgewählen Servern erstellt werden.

Die Unterschiede dieser beiden Ausführungsarten sind nicht besonders groß und spielen in der anschließenden Erläuterung keine Rolle. Um ein Formular für alle beteiligten Server zu verwenden, muss lediglich statt mehreren, ein Formular angezeigt werden. Es erfordert nur geringfügige Anpassungen auf Seiten des Admintools. Daher wird in der folgenden Erläuterung nicht weiter auf den Fall “Ein Formular für alle beteiligten Server” eingegangen.

Abbildung 3.4 zeigt den schematischen Ablauf einer Massenaufgabe.

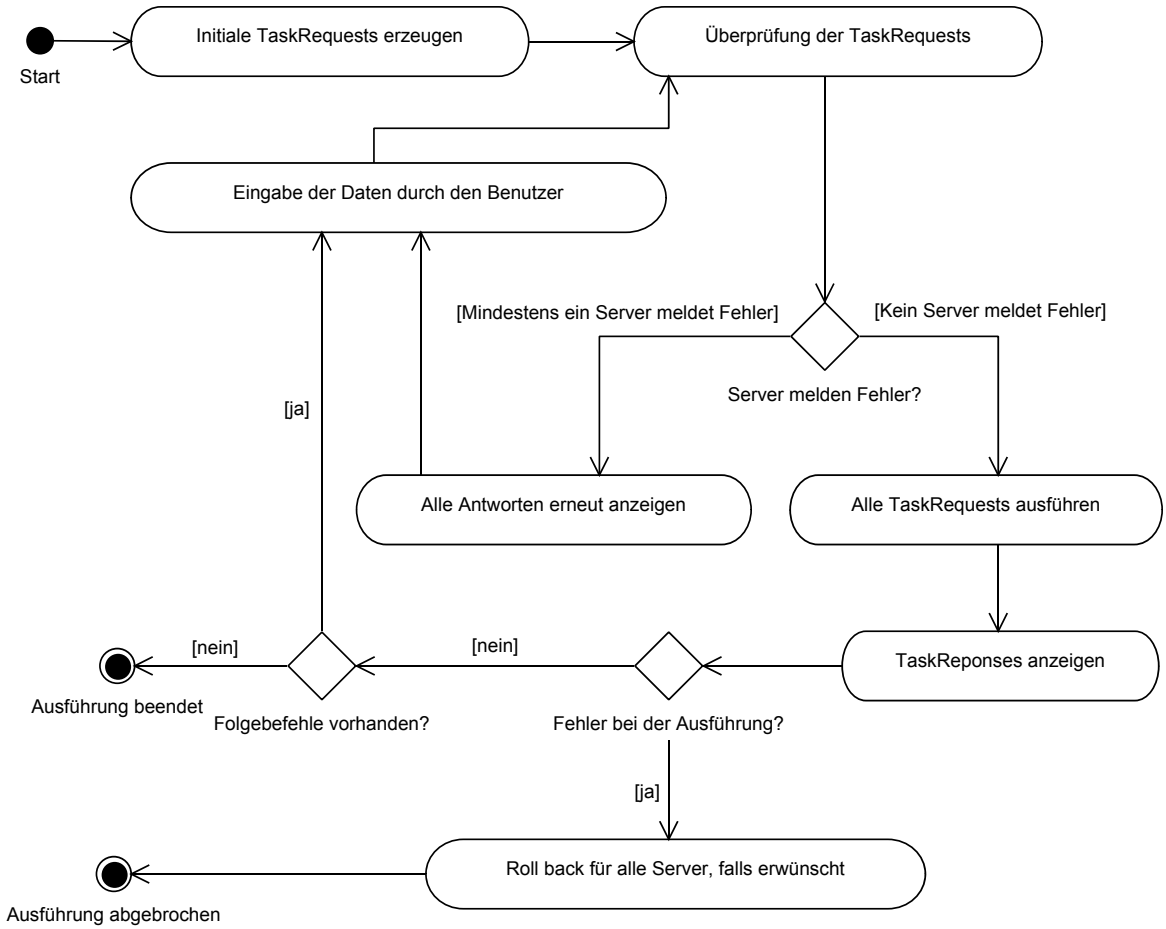

Abbildung 3.4: Ausführung von Massenaufgaben

3.4.1 Trennung von Werteüberprüfung und Ausführung

Statt nur eine Aufgabe auszuführen, wird für jeden Server eine Aufgabe ausgeführt. Allerdings sind zusätzliche Mechanismen nötig, um die Ausführung zu koordinieren. Die im Protokoll beschriebene *validate*-Methode ist für die Ausführung von Einzelaufgaben nicht nötig. Ein *TaskRequest* könnte direkt vor der Ausführung der eigentlichen Logik überprüft werden. Im Fehlerfall würde ein *TaskResponse* mit entsprechenden Fehlermeldungen zurückgegeben, die Logik aber nicht ausgeführt.

Ohne die *validate*-Methode, kann die Ausführung einer Massenaufgabe allerdings nicht koordiniert werden kann. Die Problematik ist am besten an einem Beispiel zu erläutern: Bei der Ausführung der Massenaufgabe “Softwarepakete aktualisieren” (siehe 2.2.2) sind drei Server beteiligt. Der Benutzer vergisst für den zweiten Server Pakete auszuwählen, was laut Beschreibung der Aufgabe nicht zulässig ist. Er weist das Admintool an die Aktualisierung durchzuführen, wodurch nacheinander die Service-Methode *executeTask* auf jedem TaskServer aufgerufen wird. Die Werteüberprüfung bei dem Aufruf der Service-Methode des ersten TaskServers meldet keine Fehler und der Aktualisierungsvorgang wird im Hintergrund gestartet. Der zweite TaskServer jedoch meldet, dass kein Paket ausgewählt wurde und antwortet erneut mit dem Formular und einer entsprechenden Fehlermeldung. Die Ausführung der Service-Methode auf dem dritten TaskServer kann zwar jetzt vermieden werden, die auf dem ersten TaskServer läuft allerdings bereits.

Dieses fehlerhafte Szenario kann durch die Aufteilung von *validate* und *executeTask* vermieden werden. Nachdem der Benutzer das Admintool angewiesen hat die Aktualisierung durchzuführen, werden zunächst die *validate*-Methoden der drei TaskServer ausgeführt. Enthält auch nur eine der Antworten einen Fehler, wird für keinen TaskServer die Methode *executeTask* ausgeführt. Stattdessen werden alle Formulare mit den bisher eingegeben Daten erneut angezeigt. Das Formular für den zweiten Server enthält außerdem eine Fehlermeldung, die den Benutzer dazu auffordert, mindestens ein Paket auszuwählen.

3.4.2 Atomicy-Eigenschaft

Durch die oben beschriebene Trennung von Werteüberprüfung und Ausführung der Logik, können also einige Fehler verhindert werden. Es sind allerdings noch weitere Fälle denkbar, in denen diese beiden Methoden nicht ausreichen. Angenommen ein Benutzer möchte auf mehreren Servern eine Serversoftware aktualisieren, die Versionen dieser Software sollen aber auf allen Servern aus Kompatibilitätsgründen gleich sein. Er führt die Aufgabe “Softwarepakete aktualisieren” aus und wählt die entsprechende Serversoftware aus. Die *validate*-Methoden melden keinen Fehler, da die Formulare korrekt ausgefüllt wurden. Die eigentliche Aktualisierung der Software schlägt jedoch auf einem Server fehl, beispielsweise, weil nicht

genug Speicher verfügbar ist. Nun ist der Fall eingetreten, der vermieden werden sollte. Die Softwareversionen sind unterschiedlich.

Das beschriebene Szenario stellt eine neue Anforderung an die Ausführung der Aufgabe, die man überlicherweise an Datenbanken stellt. Datenbanken sollten die ACID-Eigenschaften erfüllen. Eine dieser Eigenschaften ist die Atomicy-Eigenschaft, die besagt, dass eine Transaktion entweder ganz oder gar nicht ausgeführt werden soll. Schlägt eine Teiloperation der Transaktion fehl, müssen alle bisher durchgeführten Änderungen rückgängig gemacht werden.

Um die Atomicy-Eigenschaft einer Massenaufgabe sicherzustellen, wird auch hier ein Machanismus benötigt, der Änderungen rückgängig macht. Diese Funktion übernimmt die Methode *rollBack*, die wie auch *executeTask* und *validate* von jeder Aufgabe selbst übernommen werden muss. Im oben beschriebenen Szenario müssten also die beiden Server auf denen die Softwareaktualisierung erfolgreich verlief, die alte Version der Software wiederherstellen.

Während in Datenbanksystemen Änderungen protokolliert und so rückgängig gemacht werden können, hat es der TaskServer je nach Aufgabe unterschiedlich schwer. Tritt beispielsweise bei der Softwareaktualisierung ein Fehler auf, ist es nicht immer ohne weiteres möglich die alte Version wiederherzustellen. Soll aber nur ein System-Benutzer erstellt werden, dürfte es unproblematisch sein diesen gegebenenfalls wieder zu löschen.

Es ist also dem Gesamtsystem nicht möglich die Atomicy-Eigenschaft zu gewährleisten. Es kann lediglich dafür sorgen, dass die beteiligten Aufgaben dazu aufgefordert werden die nötigen Änderungen durchzuführen. Ob die Aufgaben dieser Aufforderung korrekt nachkommen, hängt von ihrer jeweiligen Implementierung ab. Außerdem ist das beschriebene “ganz oder gar nicht” Verhalten auch nicht unbedingt gewollt. Gerade in dem Szenario “Softwarepakete aktualisieren” ist es je nach Situation sogar sinnvoller auf allen Servern die Änderungen durchzuführen auf denen es geht. Auf den problematischen Servern muss dann wohl oder übel das Problem “von Hand” gelöst werden. Aus diesem Grund macht es Sinn vor der Ausführung einer Massenaufgabe dem Benutzer die Wahl zu lassen, ob versucht werden soll die Atomicy-Eigenschaft zu gewährleisten oder nicht.

Weitere ACID-Eigenschaften

Consistency Die Konsistenzeigenschaft definiert, dass eine Transaktion das System von einem konsistenten Zustand in einen konsistenten Zustand überführt. Diese Eigenschaft ist praktisch nicht umsetzbar. Ein Server ist mitsamt der installierten Software ein komplexes System, das anders als ein Datenbanksystem, keine Mechanismem bereitstellt, um die Konsistenz zu wahren.

Isolation Diese Isolationseigenschaft definiert, dass Transaktionen sich nicht gegenseitig beeinflussen können. Diese Eigenschaft trifft bei Massenaufgaben immer zu. Die beteiligten Einzelaufgaben laufen zwar auf mehreren Servern parallel ab, sie kommunizieren aber nicht untereinander. Daher können sie sich nicht beeinflussen.

Durability Das Ergebnis einer abgeschlossenen Transaktion bleibt erhalten, auch wenn ein Systemabsturz erfolgt ist. Diese Eigenschaft kann von einer Aufgabe im Regelfall nicht gewährleistet werden. Die Aufgaben verändern meistens Dateien der Serversoftware und starten gegebenfalls Prozesse neu. Sobald diese Dateien gespeichert sind, obliegt es dem Dateisystem und dem Betriebssystem, ob bei einem Absturz die Daten erhalten bleiben oder nicht.

Die ACID-Eigenschaften zu gewährleisten ist also nur teilweise möglich. Die möglichen Umsetzungen erfordern ihrerseits wiederum viel Aufwand, der sich für Aufgabenentwickler in der Regel nicht rechnet. Schon die *rollBack*-Methode für die Aufgabe “Softwarepakete aktualisieren” ist nur sehr schwer umsetzbar. Daher wird auf die Implementierung dieser Transaktionskonzepte im Prototyp verzichtet. Weitere Informationen über lange laufende, verteilte Transaktionen sind der Quelle [Gerlach (2005)] zu entnehmen.

3.4.3 Vereinigung der Folgebefehle

Die Vereinigungsmenge der Folgebefehle aller beteiligten Einzelaufgaben, bildet die möglichen Folgebefehle, die dem Benutzer nach der Ausführung eines Aufgabenbefehls zur Verfügung stehen. Da das Befehlsprotokoll aller teilnehmenden Einzelaufgaben gleich definiert sein muss, sollten auf diese Weise die gleichen Folgebefehle angezeigt werden, wie auch bei der Ausführung der entsprechenden Einzelaufgabe.

Es gibt allerdings einen Fall, in dem es gewollt ist, unterschiedliche Folgebefehle zu erhalten. Dieser Fall ist am einfachsten an dem bekannten Szenario “Softwarepakete aktualisieren” zu erläutern (siehe 2.2.2). Dabei wird ein lange laufender Systemprozess ausgeführt, um die eigentliche Aktualisierung vorzunehmen. Dieser Prozess wird nicht durch einen einzigen Aufgabenbefehl komplett ausgeführt, sondern durch einen gestartet und einen weiteren regelmäßig auf seinen Status abgefragt (siehe 3.3). Die Aufgabe prüft beim Ausführen des Aufgabenbefehls “Status”, ob der Systemprozess bereits beendet ist. Ist dies der Fall, werden der Aufgabenantwort keine Folgebefehle hinzugefügt. Läuft der Prozess noch, wird enthält die Aufgabenantwort den Folgebefehl “Status”.

Nun ist es sehr wahrscheinlich, dass die Aktualisierung auf verschiedenen Servern unterschiedlich lange dauert. In diesem Fall wird also, wie oben erwähnt, die Vereinigungsmenge der Folgebefehle verwendet, die nun entweder leer ist (wenn alle Aktualisierungen beendet

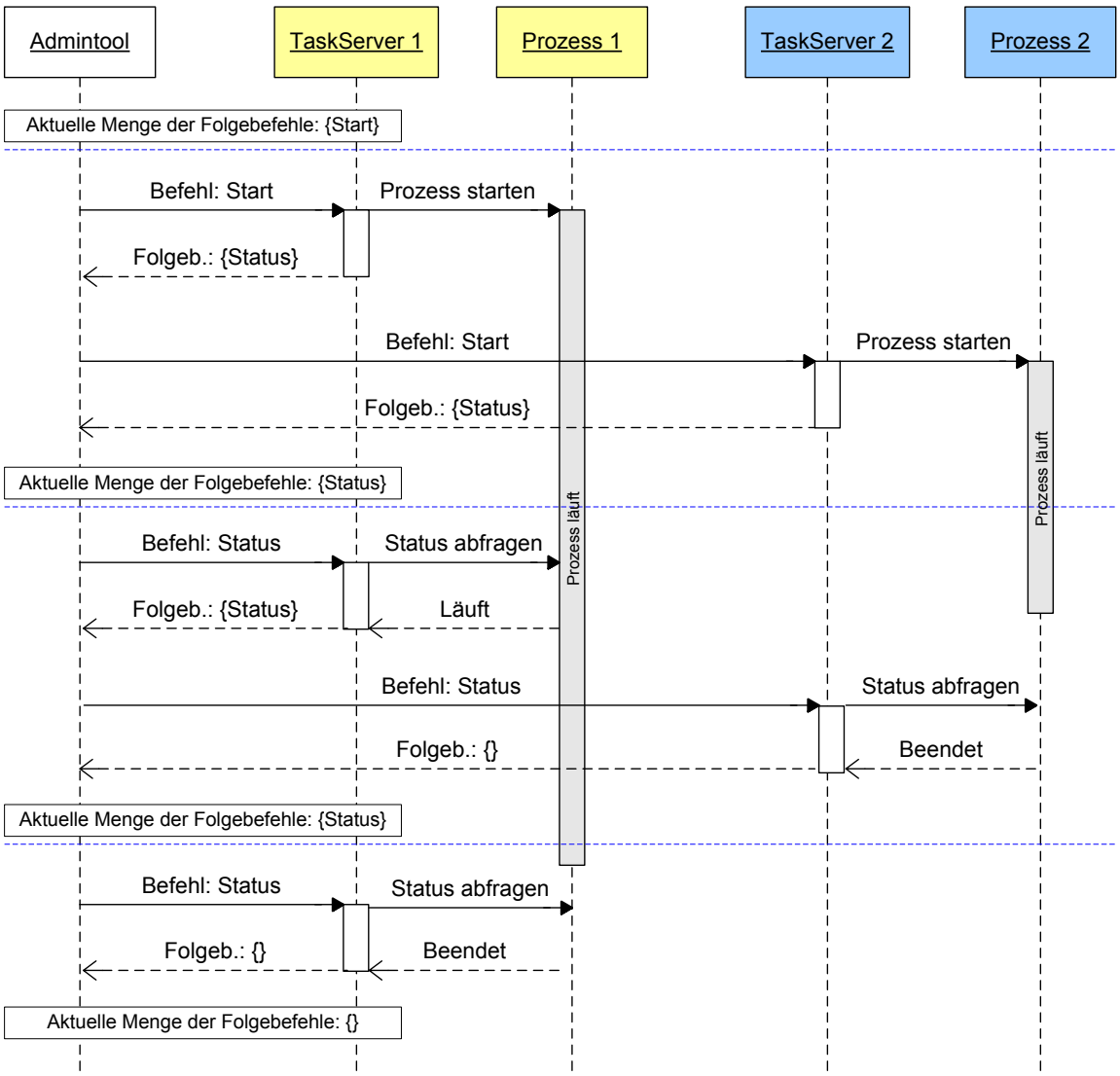

Abbildung 3.5: Beispielhafter Ablauf der Aufgabe ’Softwarepakete aktualisieren’

wurden) oder sie enthält den Befehl “Status”. So wird immer wieder der Befehl “Status” angezeigt, bis alle Server mit der Aktualisierung fertig sind, also die Vereinigungsmenge der Folgebefehle leer ist.

Abbildung 3.5 zeigt den Ablauf der Aufgabe “Softwarepakete aktualisieren”, nachdem der Benutzer Softwarepakete für zwei Server ausgewählt hat. Die eigentliche Aktualisierung dauert auf TaskServer 1 deutlich länger, als auf TaskServer 2.

3.5 Beschreibung von Aufgaben

Eine genaue Protokolldefinition reicht grundsätzlich aus, um einen TaskServer zu schreiben, der mit dem Admintool kommunizieren kann. Üblicherweise wird aber nicht auf jedem Server ein eigens geschriebener TaskServer laufen, sondern der selbe TaskServer, der lediglich unterschiedliche Aufgaben anbietet.

Die Aufgaben wiederum können auch zu einem großen Teil identisch sein. Um beispielsweise einen System-Benutzer auf einem Linux-System zu erstellen, wird üblicherweise der Systembefehl *useradd* ausgeführt und das Home-Verzeichnis erstellt. Es gibt allerdings Einstellungen, die an den jeweiligen Server angepasst werden müssen. So liegen z. B. auf einem Server alle Home-Verzeichnisse unter */home/kunden/* und auf einem anderen direkt unter */home/*. Einstellungen dieser Art werden bei den meisten denkbaren Aufgaben zu tätigen sein.

Jede Aufgabe sollte für die oben genannten Einstellungen eine Konfigurationsdatei zur Verfügung stellen, damit dafür der Quellcode der Aufgabe nicht verändert werden muss. Weiterhin ist es sinnvoll, diese Konfigurationsdateien einheitlich zu gestalten und zentral auf dem TaskServer abzulegen, um die Konfiguration des TaskServern und dessen Aufgaben zu erleichtern.

Zusätzlich zu der Aufgabenkonfiguration, sollte der TaskServer selbst konfigurierbar sein, insbesondere um festzulegen, welche Aufgaben unter welchen Namen er zur Verfügung stellt.

3.5.1 Mögliche Methoden der Aufgabenbeschreibung

Um eine Aufgabe zu beschreiben, gibt es mehrere Möglichkeiten. Einige Ansätze werden im Folgenden näher erläutert.

Als eigene Klasse

Jede Aufgabe wird durch eine eigene Klasse beschrieben, die im TaskServer als ausführbar registriert wird. Dazu bietet sich das Command Design Pattern an (siehe [Gamma u. a. (1994)]). Jede Aufgabenklasse implementiert ein Command-Interface (oder erbt von einer abstrakten Oberklasse), das folgende Methode enthält:

```
TaskResponse executeCommand(TaskRequest request)
```

Diese Methode wird vom TaskController aufgerufen, sobald dessen *executeTask*-Methode aufgerufen wird. In der Methode wird entschieden, wie der Befehl aus dem *TaskRequest* ausgeführt werden soll.

Der Nachteil dieser Aufgabenbeschreibung besteht darin, dass für jede Aufgabe eine eigene Klasse geschrieben und kompiliert werden muss.

Vorteilhaft ist, dass jede Aufgabe die gesamte Mächtigkeit der verwendeten Programmiersprache zur Verfügung hat und dass diese Variante sehr einfach zu realisieren ist.

Mittels Interpreter

Um Aufgaben einfacher erstellen zu können, also ohne Schreiben und Kompilieren von Klassen, ist eine Skriptsprache denkbar, die von dem TaskServer interpretiert wird. Eine solches Skript kann stark an den Ablauf der Aufgabenbehandlung angepasst sein und wird dadurch kürzer und damit übersichtlicher. Möglich wäre eine Unterteilung in zwei Bereiche. In einem Bereich werden die benötigten Formulare beschrieben und in dem anderen die möglichen ausführbaren Aufgabenbefehle.

Um diese Variante effektiv nutzbar zu gestalten, muss der Interpreter möglichst viele grundlegende Konzepte von Programmiersprachen verstehen können, was mit viel Aufwand verbunden ist.

Der Nachteil besteht darin, dass man auf die Mächtigkeit der Skriptsprache angewiesen ist, die unter der einer ausgewachsenen Programmiersprache zurückliegen dürfte. Außerdem ist der Realisierungsaufwand für einen eigenen Interpreter recht hoch.

Codegenerierung

Ein Kompromiss aus den beiden bisher erwähnten Varianten ist Codegenerierung. Wie in der vorigen Variante wird ein Skript geschrieben, das jedoch nicht vom TaskServer interpretiert wird, sondern von einem Code-Generator in Programmiersprachencode umgewandelt wird. Auf diese Weise wird das Erstellen der Aufgabe beschleunigt und man kann den generierten Code im Nachhinein erweitern. Im optimalen Fall, muss der Code gar nicht mehr angefasst werden und kann gleich kompiliert und im TaskServer registriert werden.

Skriptaufgaben

Auch die Variante der Codegenerierung erfordert, dass Aufgabenentwickler Klassen kompilieren und auf den Server kopieren müssen, was recht umständlich ist. Einfacher ist es, ein Skript in einer Sprache zu schreiben, die der Aufgabenentwickler gut beherrscht und die auf dem Server verfügbar oder leicht zu installieren ist. Dann muss dem TaskServer lediglich mitgeteilt werden, welches Skript von welchem Interpreter aufgerufen werden soll, um eine Aufgabe auszuführen. Ein weiterer Punkt, der für Skriptaufgaben spricht ist, dass auf vielen Servern bereits Skripte für alltägliche Administrationsaufgaben vorhanden sind. Diese müssten dann lediglich angepasst werden.

Um dies zu ermöglichen wird eine Skriptaufgabenklasse erstellt. Diese verhält sich wie eine normale Aufgabenklasse, leitet jedoch die Anfragen an ein beliebiges Skript weiter und wandelt die Ergebnisse des Skripts wieder in Aufgabenantworten um. Es können mehrere Skriptaufgaben in einem TaskServer registriert werden, da eine Aufgabenklasse mehrfach im TaskServer unter unterschiedlichen Namen registriert werden kann. In der Konfigurationsdatei einer Skriptaufgabe wird der Dateiname des Skriptes, sowie der zu verwendende Interpreter definiert. Dadurch ist es ebenso denkbar eine Aufgabe durch ein anderes externes Programm ausführen zu lassen, das nicht in Skriptform vorliegt.

Die Skriptaufgabenklasse bildet eine zusätzliche Schicht zwischen dem TaskServer und der eigentlichen Aufgabenlogik, die im Skript definiert ist. Für das Admintool erscheinen Skriptaufgaben allerdings wie “normale” Einzelaufgaben. Es ist also keine weitere Anpassung auf Seiten des Admintools nötig, weshalb diese Variante der Aufgabenbeschreibung mit den anderen bisher vorgestellten kombinierbar ist.

3.5.2 Darstellung

Der wesentliche Teil der Aufgabenbeschreibung ist die Logik. Auf den ersten Blick ist bei einem Werkzeug für Serveradministration die Darstellung relativ unwichtig. Wenn es allerdings

eingesetzt werden soll, um vermietete Server von deren Mietern administrieren zu lassen, spielt die Darstellung eine größere Rolle.

Der grundlegende Aufbau geschieht hierbei auf Seiten des Webinterfaces. Es ist ohne großen Aufwand möglich ein einheitliches Grundlayout (Seitenköpfe, Seitenfüße, Navigationsleisten, etc.) zu erstellen. Die Vorgehensweise unterscheidet sich nicht von anderen Webanwendungen.

Die Darstellung der Aufgabenantworten jedoch, also Nachrichten, Listen und Formulare, obliegt den Aufgaben zu einem gewissen Grad selbst, da nur sie wissen, was sie darstellen. Deshalb ist für ein komplett anpassbares Layout dafür Sorge zu tragen, dass die Aufgabenantworten Layoutinformationen enthalten.

Da die Darstellung für diese Arbeit allerdings von wenig Relevanz ist, wird auf die Anpassung von Aufgabenantworten verzichtet.

3.6 System-Benutzer umziehen

Eine Besonderheit unter den Aufgaben stellt die Aufgabe “System-Benutzer umziehen” (siehe 2.2.2) dar. Diese verwendet nicht auf mehreren Servern die gleichen Einzelaufgaben, wie die Massenaufgaben, sondern führt verschiedene Aufgaben auf verschiedenen Servern aus. Diese Aufgabe kann als Stellvertreter für einen weiteren Aufgabentyp angesehen werden, bei dem das Admintool allerdings selbst angepasst werden muss, um eine Aufgabe dieses Typs zu erstellen.

Zusätzlich zum Erstellen des System-Benutzers auf dem Zielserver und dem Löschen des System-Benutzers auf dem Quellserver, werden bei dieser Aufgabe die Dateien des System-Benutzers kopiert. Um die Dateien vom Quell- auf den Zielserver zu kopieren, gibt es zwei grundsätzliche Möglichkeiten:

Direkter Transfer Der Zielserver baut zu dem Quellserver eine Verbindung auf und lädt die Dateien direkt herunter.

Vorteile: Der Transfer geht schneller, da er direkt vorgenommen wird und es wird doppelter Datenverkehr (und die dadurch entstehenden Kosten) vermieden.

Nachteile: Quell- und Zielserver müssen sich kennen und der Quellserver muss dem Zielserver eine Möglichkeit bieten an die Daten zu kommen.

Indirekter Transfer Das Admintool dient als Vermittler zwischen den Servern. Die Dateien werden vom Quellserver auf den Rechner des Admintools und danach auf den Zielserver kopiert.

Vorteile: Quell- und Zielserver müssen sich nicht kennen und keine weiteren Zugänge von außen erlauben. Das Admintool und die TaskServer haben sowieso Zugang zueinander. Es muss also kein weiterer Aufwand betrieben werden, um zusätzlichen Zugang zu gewährleisten.

Nachteile: Transferdauer und Datenverkehr verdoppeln sich.

Unabhängig davon welche der beiden Methoden verwendet wird, muss entschieden werden, welche Methode zum Transfer der Daten zum Einsatz kommt. Denkbar sind diverse bekannte Protokolle, wie FTP, HTTP, SCP oder andere. Da sowohl Admintool als auch TaskServer bereits HTTP-Server bereitstellen, bietet sich HTTP an. So muss kein zusätzlicher Server auf den teilnehmenden Servern installiert sein bzw. gestartet werden.

Da das Umziehen eines Benutzers keine alltägliche Aufgabe ist, sind die Nachteile der doppelten Zeit und des doppelten Datenverkehrs durch den indirekten Transfer vernachlässigbar.

Um eine direkte Verbindung vom Quellserver zum Zielserver zu ermöglichen, muss unter Umständen eine vorhandene Firewall, die nur dem Admintool Zugriff auf den TaskServer gibt, für eine gewisse Zeit auch für den Zielserver geöffnet werden. Dies erfordert zusätzlichen Verwaltungsaufwand, insbesondere wenn die Firewall nicht direkt auf dem TaskServer läuft, und die Öffnung der Firewall bildet ein zusätzliches Sicherheitsrisiko.

Aus diesen Gründen, wird hier der indirekte Transfer verwendet.

3.6.1 Ablauf

Die Ausführung dieser Aufgabe läuft weitestgehend linear ab. Lediglich bei fehlerhaften Benutzereingaben wird das entsprechende Formular mehrfach angezeigt. Da das Kopieren der Daten viel Zeit in Anspruch nehmen kann, werden zunächst alle benötigten Daten vom Benutzer abgefragt und danach der Transfer gestartet, der im fehlerfreien Fall ohne weitere Benutzerinteraktion abläuft.

1. Auswählen des Quellservers: Zunächst muss der Benutzer den Server auswählen, auf dem sich der System-Benutzer befindet, der auf einen anderen Server ziehen soll. Dieser Schritt bildet keine Einzelaufgabe, da er komplett auf Seiten des Admintools abläuft. Dafür wird eine Liste aller dem Benutzer zugänglichen Server angezeigt.
2. System-Benutzer auswählen: Als nächstes muss der System-Benutzer ausgewählt werden, der umziehen soll. Dafür wird die Einzelaufgabe “System-Benutzer Info” verwendet. Diese besteht aus zwei Aufgabenbefehlen. Der Befehl *initCommand* zeigt eine Liste der auf dem TaskServer verfügbaren System-Benutzer an. Nachdem der Be-

nutzer einen System-Benutzer ausgewählt hat, wird der Aufgabenbefehl *getInfo* ausgeführt, der ein Formular mit den Benutzerdaten (Name, Gruppe, Home-Verzeichnis, etc.) liefert.

Die Daten aus dem Antwortformular werden im Rahmen der “System-Benutzer umziehen”-Aufgabe jedoch nicht angezeigt, sondern für den späteren Verlauf gespeichert.

3. Auswählen des Zielservers: Die Auswahl des Zielserver verläuft genauso wie Schritt 1.

4. Zielbenutzer anlegen: Ein System-Benutzer, der umziehen soll, muss auf dem Zielserver nicht zwangsläufig den gleichen Namen haben. Deshalb wird die Einzelaufgabe “System-Benutzer erstellen” verwendet, um einen neuen Benutzer auf dem Zielserver zu erstellen. Als Antwort liefert diese Einzelaufgabe die Benutzerdaten des angelegt System-Benutzers. Diese werden später benötigt, um die Zugriffsrechte zu setzen.

5. Archiv der Benutzerdateien erstellen: Nachdem der Zielbenutzer erstellt wurde, ist keine weitere Benutzereingabe erforderlich. Da der folgendene Vorgang allerdings viel Zeit in Anspruch nehmen kann, informiert das Admintool den Benutzer in regelmäßigen Abständen über den Status des Transfers. Dies geschieht, in dem der Browser des Benutzers immer wieder zu einem Aktualisieren der Seite aufgefordert wird.

   Als nächster Schritt wird die Einzelaufgabe “Download-Archiv erzeugen” auf dem Quellserver ausgeführt. Diese Einzelaufgabe erstellt ein Archiv eines Verzeichnisses und macht es per HTTP (bzw. HTTPS) zugänglich. Dieses Archiv wird nicht über einen Webservice zugänglich gemacht, sondern direkt über HTTP. Um sicherzustellen, dass nur das Admintool bzw. der Besitzer des Servers Zugriff auf diese Datei hat, muss bei der entsprechenden HTTP-Anfrage ein Passwort übergeben werden. Der Dateiname wird zufällig generiert und in der Aufgabenantwort dem Admintool mitgeteilt.

6. Download der Benutzerdateien auf das Admintool: Das Admintool baut nun eine HTTP-Verbindung zum Quellserver auf, um das in Schritt 5 erstellte Archiv herunterzuladen. Ist der Download beendet, wird das Archiv wieder per HTTP vom Admintool zur Verfügung gestellt und erneut mit einem Passwort gesichert.

7. Download der Benutzerdateien auf den Zielserver: Auf dem Zielserver wird als nächstes die Einzelaufgabe “System-Benutzer-Dateien herunterladen” ausgeführt. Diese Aufgabe erwartet eine URL zu dem Archiv mit den Dateien, den Benutzernamen und die Gruppe des Quellbenutzers, sowie die ID des Zielbenutzers.

   Die Ausführung dieser Einzelaufgabe besteht aus drei Schritten, die allerdings alle in einem Aufgabenbefehl ablaufen. Das Archiv, das über die URL angegeben wurde

wird auf den Zielserver heruntergeladen, entpackt und die Rechte werden anhand des Benutzernamens und der Gruppe gesetzt.

8. Löschen der nicht mehr benötigten: Der System-Benutzer auf dem Quellserver einschließlich dessen Daten, sowie die Dateiarchive zur Übertragung auf Quell- und Zielserver werden gelöscht.

3.6.2 Fehlerbehandlung

Der oben beschriebene Ablauf ist vereinfacht dargestellt. Im praktischen Einsatz finden sich oftmals noch weitere Daten, die transferiert werden müssen, wie beispielsweise Datenbanken und Einstellungen von Webservern oder anderer Serveranwendungen. Das bedeutet, dass diese Aufgabe unter Umständen noch mehr Schritte erfordert, je nachdem wie viele verschiedene Daten transferiert werden sollen.

In jedem Schritt können Fehler entstehen, die dazu führen sollten, dass der letzte Schritt, also das Löschen der nicht mehr benötigten Daten, nicht ausgeführt wird. Je nach Situation, kann es wünschenswert sein, die bis zu dem Fehler vorgenommenen Schritte rückgängig zu machen. Dies sollte allerdings dem Benutzer überlassen werden, da auf Grund eines Fehlers nicht unbedingt die bereits transferierten Daten wieder gelöscht werden sollen. Der Benutzer könnte ebenso den Transfer beenden wollen, ohne die alten Benutzerdaten zu löschen, um die fehlenden Daten manuell zu übertragen.

Wie auch bei Massenaufgaben, sollte es dem Benutzer überlassen werden, alle bisherigen Änderungen rückgängig zu machen.

3.7 Sicherheit

3.7.1 Absicherung der Admintool-TaskServer-Kommunikation

Sobald der TaskServer auf einem Rechner läuft, kann mit ihm sehr tief in das System eingegriffen werden. Deshalb muss sichergestellt werden, dass nur das Admintool auf den TaskServer zugreifen kann, wofür verschiedene Mechanismen in Frage kommen.

Auf Vermittlungsebene sollte der TaskServer durch eine Firewall abgesichert sein, die nur dann eingehende Pakete auf dem Port des TaskServers akzeptiert, wenn sie von der IP-Adresse des Servers kommen, auf dem das Admintool läuft. Da der TaskServer über HTTP angesprochen wird, muss nur ein eingehender Port geöffnet werden.

Auf diese Weise wird der Zugriff auf den TaskServer zwar schon stark eingeschränkt, was aber auf Grund der Beschaffenheit des Internets nicht ausreicht. Da die Kommunikation zwischen Admintool und TaskServer im Regelfall über mehrere Stationen geht – es sei denn sie befinden sich im selben privaten Netz – ist es dringend ratsam die Anfragen und Antworten zu verschlüsseln und die Authentifizierung der beteiligten Partner durchzuführen.

Secure Socket Layer

Secure Socket Layer (SSL) ist ein Verschlüsselungsprotokoll für sichere Datenübertragung im Internet. Es bildet eine Schicht zwischen Applikationsprotokollen, wie HTTP, und Transportprotokollen, wie TCP/IP. SSL wird sehr oft verwendet, um sichere HTTP-Verbindungen anzubieten. Auch der TaskServer könnte also über HTTPS (HTTP over SSL) angesprochen werden.

Bevor die eigentlichen Daten per SSL übertragen werden, findet zunächst eine Handshake-Phase statt. In dieser Phase einigen sich die beiden Kommunikationspartner auf die zu verwendenden Verschlüsselungs- und Komprimierungsverfahren, tauschen die dafür benötigten Schlüssel aus und authentifizieren sich (optional). Bei der Verwendung von HTTPS im Web, authentifiziert sich via SSL üblicherweise nur der Server gegenüber dem Client. So wird beispielsweise sichergestellt, dass ein Homebanking-Kunde seine Kontodaten auch wirklich zu seiner Bank und nicht zu einem anderen Server überträgt.

Im Fall der Kommunikation zwischen Admintool und TaskServer reicht diese einseitige Authentifizierung nicht aus. Es ist zwar wichtig, dass sich ein Angreifer nicht als TaskServer ausgeben kann, da er so Daten, die eigentlich an den TaskServer gesendet werden sollten, ausspionieren könnte. Es ist aber ebenso wichtig, dass sich der Client am Server authentifiziert, da über die angebotenen Dienste des TaskServers, je nach vorhandenen Aufgaben, sehr viel Schaden angerichtet werden kann. Aus diesem Grund ist es wünschenswert, dass sich sowohl Server, als auch Client gegenseitig authentifizieren. Auch wenn es nicht immer eingesetzt wird, unterstützt SSL wechselseite Authentifizierung mittels Zertifikaten.

In einer produktiven Umgebung ist es darüber hinaus empfehlenswert für beide Seiten Zertifikate zu verwenden, die von einer Zertifizierungsstelle (Certificate Authority) ausgestellt wurden, die Server und Client als vertrauenswürdig einstufen. Auf diese Weise wird sichergestellt, dass die Zertifikate von dem Absender kommen, der dieser vorgibt zu sein.

Weitere Informationen über die Funktionsweise von SSL (einschließlich der Zertifizierungsstellen) sind den Quellen [SSL (2000)] und [SSL-RFC (1999)] zu entnehmen.

Secure-Shell-Tunnel

Eine weitere Möglichkeit eine Verschlüsselung zu erreichen ist, die Daten über einen SSH-Tunnel (Secure-Shell-Tunnel) auszutauschen. Bei diesem Verfahren kann der Netzwerkverkehr zu einem bestimmten Server-Port, durch einen verschlüsselten SSH-Kanal getunnelt werden. Um einen TCP-Dienst, wie den TaskServer, über einen SSH-Tunnel zu verwenden, wird auf dem Server ein freier, aber nicht zwangsläufig von außen zugänglicher, Port benötigt. Der Client benötigt ebenfalls einen freien Port, der allerdings nicht fest vorgegeben ist. Über diesen Port können Anwendungen, in diesem Fall das Admintool, den entfernten Dienst, also den TaskServer, ansprechen. Um beispielsweise einen TaskServer auf dem Port 8080 anzusprechen, wird vom Admintool ein SSH-Tunnel zum TaskServer aufgebaut, der alle Anfragen an einen beliebigen Port auf dem Client (z. B. 18080) verschlüsselt an den TaskServer auf Port 8080 weiterleitet.

Dieses Verfahren bietet eine Reihe von Vorteilen:

- Für den TaskServer selbst muss kein Port, wie oben beschrieben, für den Zugriff durch das Admintool geöffnet werden.
- SSH (insbesondere SSH-2) bietet ausgereifte Verfahren für die Authentifizierung. In diesem System bietet sich das Public-Key-Verfahren an. Die Funktionsweise des Public-Key-Verfahrens wird hier nicht weiter erläutert, siehe dazu [Maseberg (2002)].
- Auf vielen Servern (insbesondere auf Servern mit Unixderivaten als Betriebssystem) laufen bereits SSH-Server, für die sowieso ein Port offen gehalten wird. Natürlich ist dieser Vorteil zugleich ein Nachteil, wenn auf dem Server noch kein SSH-Server läuft.
- Es existieren frei verfügbare SSH-Server, die ohne großen Aufwand zu installieren sind. [OpenSSH (2006)]

Leider bringt die Verwendung von SSH-Tunneln einen nicht unerheblichen Nachteil mit sich. Für jede Verbindung zu einem TaskServer wird ein Port auf dem Rechner des Admintools benötigt. Nun kann dieser Port nach jeder Anfrage wieder freigegeben werden, indem der SSH-Tunnel beendet wird. Auf diese Weise sind nur so viele Ports nötig, wie Aufgaben tatsächlich parallel ablaufen. Zur Erinnerung: Selbst bei der Ausführung einer Massenaufgabe, laufen die Anfragen an die TaskServer nacheinander ab. So bleibt in der Praxis die Anzahl genutzter Ports zwar relativ gering, aber der Auf- und Abbau von SSH-Tunneln nimmt eine gewisse Zeit in Anspruch, wodurch die Performance leidet.

Um nicht für jede Anfrage einen neuen Tunnel auf- und abbauen zu müssen, kann man die Tunnel mit einem Zeitfenster versehen. Sie werden dann nach einer bestimmten Zeit nach dem Aufbau wieder abgebaut. Dadurch erhöht sich allerdings die Anzahl der benötigten Ports und es muss zusätzlicher Verwaltungsaufwand betrieben werden, um die Tunnel wieder abzubauen.

WS-Security

Die oben vorgestellten Mechanismen bieten Sicherheit auf Transportebene an und sind damit unabhängig davon, welche Software auf Anwendungsebene verwendet wird. Da der TaskServer ausschließlich mit HTTP angesprochen wird, ist weiterhin ein anwendungsspezifischer Ansatz möglich.

Die SOAP-Spezifikationen [SOAP (2003)] enthalten keine Informationen über Verschlüsselung und Authentifizierung. Es existieren allerdings weitere Spezifikation, die sich mit der Sicherheit in Webservices beschäftigen.

Die WS-Security-Spezifikationen [WS-Security (2005)] wurden von einer Reihe von großen Firmen, wie IBM, Microsoft und Verisign ins Leben gerufen und werden nun von der Organization for the Advancement of Structured Information Standards (OASIS) gepflegt. Am 1. Februar 2006 wurden die aktuellen WS-Security Spezifikationen in der Version 1.1 von der OASIS als Standard freigegeben. Sie beschreiben Erweiterungen zum SOAP-Standard, die Nachrichtenintegrität und Vertraulichkeit gewährleisten sollen. Außerdem beschreiben sie generelle Mechanismen, um Security-Tokens, wie Benutzernamen und Passwörter, innerhalb SOAP-Nachrichten zu versenden.

Ein wesentlicher Vorteil von WS-Security gegenüber Sicherheitsmechanismen auf Transportebene liegt darin, dass mit WS-Security auch Webservices abgesichert werden können, die über Vermittler kommunizieren. Da es in diesem System allerdings keine Webservice-Vermittler gibt, ist dieser Vorteil hinfällig.

Zusätzlich zu WS-Security existieren weitere Spezifikationen, wie WS-Trust [WS-Trust (2005)] und WS-Policy [WS-Policy (2006)]. Diese bauen auf die Funktionen von WS-Security auf. Zum Zeitpunkt der Erstellung dieser Arbeit waren sie jedoch noch nicht als Standard freigegeben.

Weitere Details über WS-Security und den darauf aufbauenden Standards sind der Quelle [MSWS] und den Spezifikationen [WS-Security] zu entnehmen.

Zusammenfassung

Die WS-Security-Spezifikationen decken einen großen Teil der Sicherheitsanforderungen ab, die an Webservices gestellt werden. Da die auf WS-Security aufbauenden Standards noch nicht freigegeben wurden, ist grundsätzlich von ihrer Benutzung abzuraten. Es soll allerdings gezeigt werden, ob und wie es möglich ist, WS-Security-Mechanismen zu implementieren. Näheres hierzu folgt in Abschnitt 4.6.1.

Da die Sicherheitsmechanismen von SSL und WS-Security auf unterschiedlichen Ebenen greifen, ist es problemlos möglich beide zu kombinieren. Um eine sichere Punkt-zu-Punkt-Kommunikation zwischen TaskServer und Admintool zu gewährleisten, wird daher SSL mit wechselseitiger Authentifizierung verwendet.

Zusätzlich authentifiziert sich das Admintool bei dem TaskServer, indem jeder SOAP-Nachricht Benutzername und Passwort hinzugefügt werden. Wie diese Daten in der SOAP-Nachricht angegeben und interpretiert werden, legen die UsernameToken-Spezifikationen fest, welche ein Teil des WS-Security-Standards sind. Die UsernameToken-Spezifikationen werden in Quelle [UsernameToken (2006)] beschrieben.

Unabhängig davon welche Mechanismen zur Gewährleistung der Sicherheit verwendet werden, ist es wünschenswert diese so in das System zu integrieren, dass sie leicht austauschbar sind. Wie dies geschieht, hängt von den verwendeten Komponenten ab, auf denen der TaskServer und das Admintool aufbauen. Diese werden im Kapitel Realisierung näher beschrieben (4.6).

3.7.2 Absicherung der Admintool-Benutzer-Kommunikation

Neben der Webservice-Kommunikation, muss der Zugriff zum Admintool durch den Benutzer abgesichert werden. Es muss sichergestellt werden, dass nur autorisierte Benutzer Zugriff auf das Webinterface haben. Dies ist mindestens ebenso wichtig wie die Absicherung des TaskServer-Zugriffs, da ein Benutzer über das Admintool Zugriff auf mehrere Server haben kann.

SSL

Auch für diese Absicherung ist die Verwendung von HTTPS ratsam, da die meisten aktuellen Browser HTTPS unterstützen. Clientseitige Authentifizierung ist hier allerdings ein Problem. Zwar ist auch dafür Unterstützung durch aktuelle Browser vorhanden – unter anderem durch Opera, Mozilla Firefox und den Internet Explorer – jedoch muss auch hier der Client (in diesem Fall der Browser) ein Zertifikat an den Server (das Admintool) senden und der Server muss dieses Zertifikat akzeptieren.

Wenn also ein Benutzer dem Admintool hinzugefügt wird, muss für den Benutzer zunächst ein Zertifikat erstellt werden. Wenn der Benutzer sich am Admintool anmelden möchte, muss er dieses Zertifikat mitsenden. Dafür muss er seinem Browser das Zertifikat übergeben, was beudeutet, dass er für die Bedienung des Admintools nur Browser verwenden kann, die Zugriff auf das Zertifikat haben. Das Zertifikat könnte beispielsweise auf einem anderen gesicherten Server abgelegt werden, auf den der Benutzer Zugriff hat. Dadurch wird jedoch

der Vorteil des einfachen Zugangs von nahezu jedem Rechner mit Internetzugang eingeschränkt.

Da die serverseitige Authentifizierung kein Problem darstellt, sollte diese auf jeden Fall verwendet werden. Ob eine clientseitige Authentifizierung sinnvoll ist, müssen die Serveradministratoren selbst entscheiden. Es wäre auch denkbar die Entscheidung, ob clientseitige Authentifizierung über HTTPS verwendet werden soll, jedem Administrator selbst zu überlassen.

Benutzername und Passwort

Unabhängig von der SSL-Verschlüsselung, sollte zur Authentifizierung Benutzername und Passwort verwendet werden. Um diese abzufragen, bieten sich zwei Mechanismen an:

HTTP-Authentifizierung Es gibt zwei Mechanismen für eine Authentifizierung auf HTTP-Ebene: Basic Authentification sendet Benutzername und Passwort im Klartext, während Digest Authentification das Passwort verschlüsselt (siehe hierzu [RFC2617 (1999)]). Da HTTP ein zustandsloses Protokoll ist, wird in beiden Fällen, bei jeder HTTP-Anfrage Benutzername und Passwort mitgesendet.

Formularbasierte Authentifizierung Bei der formularbasierten Authentifierung werden Benutzername und Passwort einmalig unverschlüsselt als HTTP-Parameter übertragen und die Benutzerdaten in der HTTP-Session gespeichert. Bei jeder weiteren HTTP-Anfrage sendet der Browser, per Cookie oder als HTTP-Parameter, die vom Server zugewiesene Session-ID mit, sodass der Server prüfen kann, ob sich der Benutzer authentifiziert hat. Die Vorteile dieser Methode liegen darin, dass Benutzername und Passwort nur ein mal übertragen werden und dass der Server den Benutzer automatisch nach einer gewissen Zeit ohne Rückmeldung abmelden kann, indem er die Sessiondaten löscht.

Der Nachteil, dass Benutzername und Passwort unverschlüsselt übertragen werden, ist vernachlässigbar, da SSL die Verschlüsselung auf Transportebene übernimmt. Da ein Benutzer bei der Verwendung der HTTP-Authentifizierung nicht nach einem Timeout vom Server abgemeldet werden kann, wird hier formularbasierte Authentifizierung verwendet.

4 Realisierung

Dieses Kapitel erläutert die Umsetzung des in Kapitel 3 vorgestellten Designs und gibt einen Überblick über dessen Machbarkeit.

4.1 Verwendete Komponenten

4.1.1 TaskServer

Der TaskServer wird in Java geschrieben. Diese Sprache bietet sich an, da im Rahmen dieser Arbeit der TaskServer unter einem Linux-System läuft. Aus diesem Grund fällt eine mächtige Alternative .NET weg, da .NET nur für Windows-Systeme erhältlich ist. Es existiert zwar eine freie, auch unter Linux laufende .NET-Alternative names Mono, die allerdings weniger ausgereift ist, insbesondere im Hinblick auf Webservices. Dank der Sprachunabhängigkeit von Webservices, ist es jedoch theoretisch möglich einen TaskServer in einer anderen Sprache zu schreiben. Ob dies tatsächlich gelingt, hängt davon ab, ob die beteiligten Kommunikationskomponenten die vorgegebenen Standards auf die gleiche Weise implementieren.

Die vom TaskServer angebotenen Webservice-Methoden, werden mit Hilfe von Apache Axis [Axis (2006)] bereitgestellt. Axis ist eine frei verfügbare open source Implementierung des SOAP Protokolls in Java.

Axis wiederum benötigt für den Betrieb einen Application Server oder eine Servlet Engine. In diesem System wird dafür das ebenfalls frei verfügbare open source Produkt Apache Tomcat [Tomcat (2006)] eingesetzt. Tomcat ist die offizielle Referenzimplementierung für Java Servlets und JavaServer Pages.

4.1.2 Admintool

Das Admintool wird als Webapplikation realisiert. Es verwendet, ebenso wie der TaskServer, Tomcat als Servlet Engine und Axis für die Webservice-Kommunikation, in diesem Fall auf Clientseite.

WebWork

Zusätzlich wird WebWork [WebWork (2006)] verwendet, ein Framework für die Entwicklung Java-basierter Webanwendungen. WebWork erleichtert unter anderem die Einhaltung der Vorgaben des Model View Controller Patterns. Alternativen zu WebWork, wie Apache Struts oder JavaServer Faces, wurden nicht weiter betrachtet, da der Autor bereits einige (positive) Erfahrungen mit WebWork gesammelt hat.

HTTP-Anfragen, die über den Browser gestellt werden, werden im Wesentlichen durch Actions ausgeführt, im Folgenden als WebWork-Aktionen bezeichnet, die den Controller bilden. Die Benutzerschnittstelle (View) wird durch Templates beschrieben, in diesem System werden dafür JavaServer Pages verwendet, die die Darstellungsschicht bilden. Das Modell wird durch Objekte repräsentiert, die in einer Datenbank gespeichert werden können.

Bevor eine Benutzeranfrage an den zuständigen Controller weitergeleitet wird, empfängt ein Front Controller die Anfrage und entscheidet, in der Regel anhand der URL, an welchen Controller die Anfrage bearbeiten soll. Der Front Controller wird von WebWork als Servlet-Filter bereitgestellt. Eine genauere Beschreibung des Front Controller Patterns, sowie der Architektur von WebWork sind den Quellen [FrontController (2003)] und [WWArchitektur (2006)] zu entnehmen.

Datenbankanbindung

Die vom Admintool benötigten Daten, wie Benutzerdaten, Serverdaten, Massenaufgaben, werden in einer PostgreSQL-Datenbank [PostgreSQL (2006)] hinterlegt. Die Datenbankanbindung übernimmt Hibernate Annotations [Hibernate (2006)]. Hibernate ist ein sehr ausgereiftes Framework für objektrelationales Mapping. Mit diesem Framework ist es möglich Java-Objekte in einer relationalen Datenbank zu speichern. Gleichzeitig wird dadurch Unabhängigkeit von einer spezifischen Datenbank erreicht, da Hibernate eine Vielzahl von Datenbanken unterstützt und diese sehr einfach in einer Konfigurationsdatei eingestellt werden können. Bei Hibernate Annotations werden die Persistenzinformationen als Java Annotations direkt im Quellcode der Modell-Klassen definiert und ermöglichen so ein einfaches Strukturieren des Datenbankschemas.

4.2 Beschreibung von Aufgaben

In Kapitel 3.5 wurden verschiedene Methoden zum Beschreibung von Aufgaben vorgestellt. Da die Entwicklung eines eigenen Interpreters zu aufwendig wäre, um ihn im Rahmen dieser Arbeit auf einen einsetzbaren Stand zu bringen, wurde sich hier für die Variante der

Codegenerierung entschieden. Außerdem wurde eine Aufgabenklasse für Skriptaufgaben entwickelt, die in Abschnitt 4.5.3 genauer beschrieben wird.

Die Aufgabenbeschreibung findet in Form von XML-Dateien statt, die die für die Aufgabe nötigen Formulare enthalten, sowie ein grundlegendes Gerüst für ausführbare Aufgabenbefehle. Die Dateien bestehen aus zwei grundlegenden Teilen: Den Formularen und den Aufgabenbehlen.

4.2.1 XML-Dateien

Formulare

Die Formulare bestehen im aus einer Menge von Feldern und werden ähnlich wie HTML-Formulare definiert. Ein Beispiel:

```
<form name="default">
  <textfield label="Benutzername" name="username" />
  <textfield label="Passwort" name="password"
   type="password" minLength="4" />
  <textfield label="Passwort (wiederholen)"
   name="password_verify" type="password" minLength="4" />
  <textfield label="Home-Verzeichnis" name="homedir" />
  <selectfield label="Gruppe" name="group"
   optionsMethod="getGroups" />
</form>
```

Einige der XML-Attribute werden direkt in Attribute gleichen Namens der FormField-Klasse umgewandelt (siehe 3.2.3). So enthält ein Feld immer die Attribute *label* und *name*.

Neben den bereits im Design beschriebenen FormField-Attributen, kommen hier Attribute hinzu, die die Wertüberprüfung vereinfachen. In diesem Beispiel ist dies das Attribut *minLength*, das die minimale Wert-Länge für die Felder *password* und *password_verify* festlegt.

Das Attribut *optionsMethod* der Auswahlliste *group* enthält den Namen einer Methode, die nach der Codegenerierung manuell angelegt werden muss. Eine mittels *optionsMethod* definierte Methode muss ein Array von Option-Objekten zurückgeben, das die Werte für die Auswahlliste enthält.

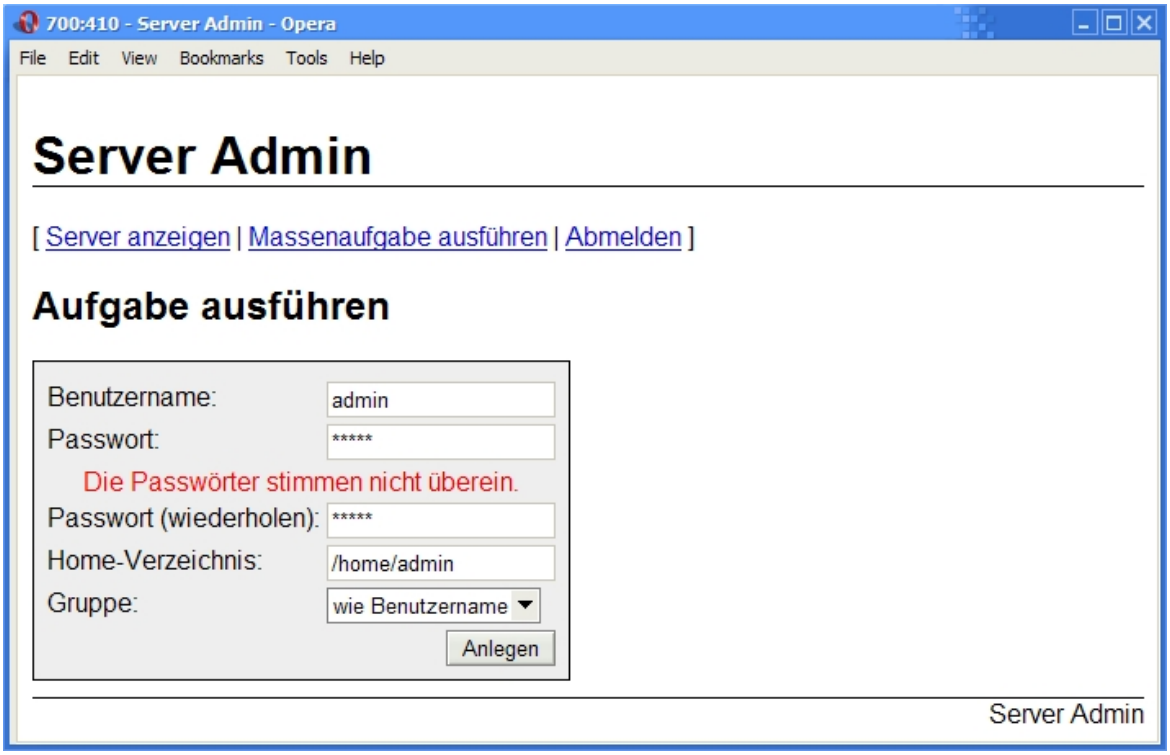

Abbildung 4.1: Bildschirmfoto eines Formulars der Aufgabe ’System-Benutzer erstellen’

Aufgabenbefehle

Die Aufgabenbefehle bilden lediglich ein Grundgerüst für das Ausführen der Logik. Die wesentliche Arbeit wird dafür erst nach der Codegenerierung fällig.

Der Teil für die Aufgabenbefehle für das obige Beispiel sieht wie folgt aus:

```
<command name="initCommand">
  <messages>
    <message>Benutzer anlegen</message>
  </messages>
  <form>default</form>
  <nextcommands>
    <nextcommand name="initCommand">Abbrechen</nextcommand>
    <nextcommand name="create">Anlegen</nextcommand>
  </nextcommands>
</command>

<command name="create">
  <messages>
    <message>Benutzer wurde angelegt</message>
  </messages>
</command>
```

In diesem Beispiel werden zwei Aufgabenbefehle definiert: Der obligatorische Befehl *initCommand*, der per Konvention immer als erstes ausgeführt wird, und der Befehl *create*. Letzterer übernimmt das eigentliche Erstellen des System-Benutzers.

Wie bereits in 3.2.3 erwähnt, muss eine gültige Antwort immer mindestens eine Nachricht oder mindestens einen Folgebefehl enthalten. Diese können in den *command*-Tags vordefiniert werden. Der Tag *messages* enthält die Nachrichten und *nextCommands* die möglichen Folgebefehle.

Falls die Antwort ein Formular enthalten soll, kann dies mit dem *form*-Tag definiert werden. Dieser Tag enthält eine Referenz auf ein vorher definiertes Formular (siehe oben).

4.2.2 Generierte Klassen

Aus den XML-Dateien erzeugt der *TaskGenerator* zwei Java-Klassen, sowie für jedes Formular eine eigene XML-Datei, die von der generierten Aufgabenklasse gelesen wird.

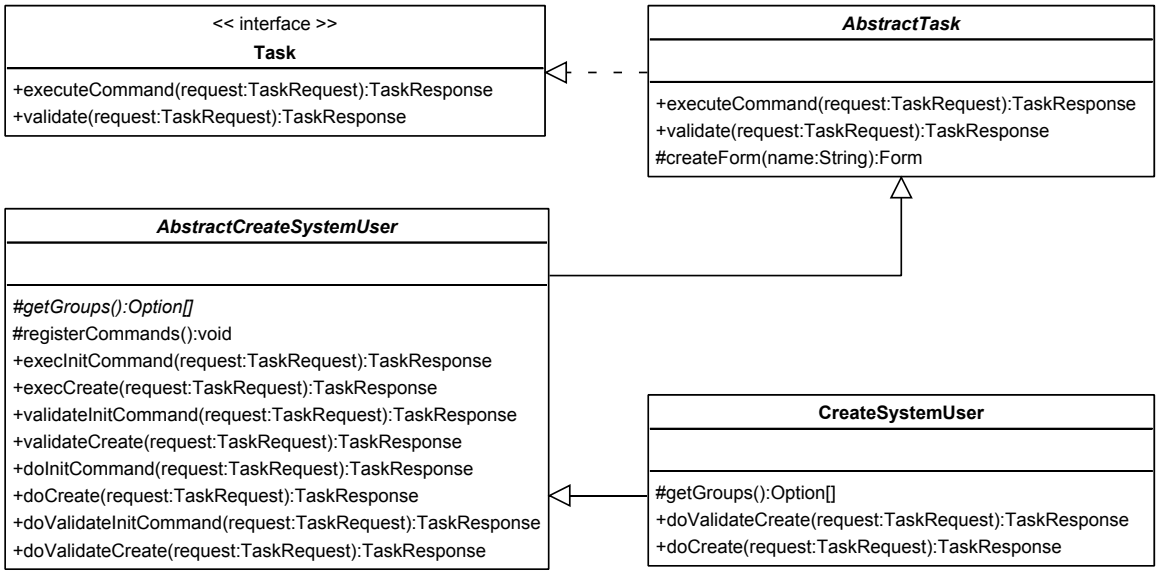

Abbildung 4.2: Generierte Klassen

In Abbildung 4.2 sind die generierten Klassen, sowie die von ihnen verwendeten Klassen für die Beispielaufgabe “System-Benutzer erstellen” zu sehen. Nur *AbstractCreateSystemUser* und *CreateSystemUser* sind generierte Klassen.

Task Um als Aufgabe vom TaskServer aufgerufen werden zu können, muss eine Aufgaben-Klasse lediglich das Task-Interface implementieren. Es definiert folgende Methoden, die für das Aufrufen von Aufgabenbefehlen und der Wertüberprüfung zuständig sind:

```
TaskResponse executeCommand(TaskRequest request)
TaskResponse validate(TaskRequest request)
```

AbstractTask Diese abstrakte Oberklasse enthält einige Methoden, die häufig von Aufgaben verwendet werden. Die für die generierten Klassen wichtigen Methoden sind:

`Form createForm(String name)`
Erstellt ein Form-Objekt anhand einer Form-XML-Datei.

`TaskResponse executeCommand(TaskRequest request)`
Dies ist die Implementierung der gleichnamigen Methode aus dem Task-Interface. Sie liest den Namen des Aufgabenbefehls aus dem *TaskRequest* aus und führt eine entsprechend benannte Methode aus. Heißt der Aufgabenbefehl beispielsweise *create*, wird die Methode *execCreate* ausgeführt. Diese *exec*-Methoden sind in der generierten Klasse *AbstractCreateSystemUser* definiert.

`TaskResponse validate(TaskRequest request)`
Hierbei handelt es sich ebenfalls um die Implementierung einer Methode aus dem Task-Interface. Sie verhält sich genauso, wie *executeCommand*, bis auf den Unterschied, dass hier eine *validate*-Methode ausgeführt wird. Für den Aufgabenbefehl *create*, wird also die Methode *validateCreate* ausgeführt. Auch die *validate*-Methoden sind in *AbstractCreateSystemUser* definiert.

AbstractCreateSystemUser Diese Klasse enthält die gesamte vom TaskGenerator erzeugte Logik. Jede *exec*-Methode behandelt die Ausführung eines Aufgabenbefehls. Waren in der XML-Quelldatei Nachrichten und/oder Formulare für einen Aufgabenbefehl definiert, werden in diesen Methoden entsprechende Java-Objekte erzeugt und dem *TaskResponse*-Objekt hinzugefügt.

In den *validate*-Methoden werden Wertüberprüfungen vorgenommen. In diesem Beispiel werden die Werte der Passwortfelder auf ihre Mindestlänge geprüft.

Die *do*-Methoden sind leer (siehe dazu *CreateSystemUser*). Diese Klasse sollte nach Möglichkeit nicht vom Aufgabenprogrammierer verändert werden.

Die abstrakte Methode *getGroups*, muss in *CreateSystemUser* überschrieben werden. Sie ist dafür zuständig das Auswahlfeld *groups* mit Auswahloptionen zu füllen.

registerCommands speichert alle Aufgabenbefehle mit ihren zugehörigen Folgebefehlen in einem assoziativen Array. Darüber können Aufgabenentwickler einfach auf die Folgebefehle von bestimmten Befehlen zugreifen. Dies ist in der Regel sinnvoll, wenn ein Formular fehlerhafte Werte enthält, denn dann müssen die Folgebefehle des vorherigen Befehls angezeigt werden.

CreateSystemUser Diese Klasse wird nur generiert, wenn sie nicht bereits existiert und ist standardmäßig leer. Auf diese Weise kann ein Aufgabenprogrammierer diese Klasse erweitern und auch im Nachhinein noch die XML-Datei der Aufgabenbeschreibung verändern und die Klassen neu generieren lassen, ohne die angepasste Klasse zu überschreiben.

Hier sollten die *do*-Methoden aus der abstrakten Oberklasse (in diesem Beispiel *AbstractCreateSystemUser*) überschrieben werden, um zusätzliche Logik einzubringen, die der TaskGenerator nicht bietet. In den *doValidate*-Methoden kann zusätzliche Wertüberprüfung untergebracht werden. In diesem Beispiel wird in der Methode *doValidateCreate* geprüft, ob die Werte der Felder *password* und *password_verify* übereinstimmen.

Die abstrakte Methode *getGroups* aus der Oberklasse wird hier implementiert. Der vom TaskGenerator erzeugte Methodenrumpf ist allerdings leer. Hier muss der Auf-

gabenprogrammierer selbst Hand anlegen, um die Logik zu implementieren. In dem Beispiel werden hier die vorhandenen System-Gruppen ausgelesen.

4.3 Fehlerbehandlung

Insbesondere in verteilten System ist die Fehlerbehandlung schwierig, da die Fehler oftmals nicht da zu sehen sind, wo sie eigentlich auftreten.

Entsteht im TaskServer, beim Ausführen einer Aufgabe, ein Fehler, muss dieser also über das Admintool bis hin zum Benutzer weitergereicht werden und nach Möglichkeit so aussagekräftig sein, dass der Benutzer weiß, wo der Fehler aufgetreten ist.

Alle Exceptions, die im TaskServer auftreten, werden geloggt und als RemoteException an das Admintool weitergegeben.

Das Admintool wiederum setzt alle Exceptions, bis auf RemoteExceptions, in UserMessageExceptions um. Eine UserMessageException enthält neben der normalen ExceptionMessage eine UserMessage, die eine für den Benutzer verständlichere Fehlermeldung enthält. Auch die UserMessageExceptions werden geloggt, allerdings auf Seiten des Admintools.

Das WebWork-Framework erleichtert die Behandlung von Exceptions. Zu jeder WebWork-Aktion können sogenannte Interceptoren definiert werden. Diese Interceptoren verwenden das Interceptor Pattern [Interceptor (2002)], um die Ausführung von WebWork-Aktionen zu modifizieren. Um die Exceptions auf Seiten des Admintools zu behandeln, werden zwei Interceptoren verwendet. Der LogInterceptor fängt jede Exception ab und loggt sie, während der von WebWork bereitgestellte Exception Interceptor die UserMessageExceptions an ein entsprechendes Fehler-Template weiterleitet, das die UserMessage bzw. die RemoteException dem Benutzer zeigt.

Bei der Behandlung von Exceptions muss zwischen erwarteten und unerwarteten Fehlern unterschieden werden.

Erwartete Fehler können abgefangen werden und in für den Benutzer verständliche Meldungen umgesetzt werden. Wenn beispielsweise die Webservice-URL für einen Server ausgelesen werden soll und diese in der entsprechenden Konfigurationsdatei fehlerhaft ist, so entsteht eine MalformedURLException. Dieser erwartete Fehler kann also aufgefangen werden und als UserMessageException mit der UserMessage “Ungültige Adresse für Server x in serveradmin.properties” weitergereicht werden.

Unerwartete Fehler hingegen machen es für den Benutzer meist schwer zu verstehen, was für ein Fehler eigentlich entstanden ist. Besonders bei Webanwendungen ist es allerdings

ratsam nicht alle Informationen, die über einen Fehler verfügbar sind (wie z. B. den Stack-Trace) anzuzeigen, da darin unter Umständen ein Sicherheitsrisiko besteht und sie den Benutzer in der Regel nur verwirren. Aus diesem Grund wird von dem Admintool eine wenig sagende Fehlermeldung, wie “Unbekannter Fehler aufgetreten” angezeigt, allerdings zusätzlich die Serverzeit und ein Verweis auf die Logdatei. So kann ein Administrator bzw. ein Entwickler, der die Meldung von einem Benutzer erhält, direkt im Log unter der richtigen Uhrzeit nachsehen, um den Fehler zu finden.

Eine Ausnahme bilden in diesem Fall die RemoteExceptions. Da der Benutzer in der Regel auch der Administrator des Servers ist, bei dem die RemoteException aufgetreten ist, hat er natürlich ein Interesse daran zu erfahren, was für ein Fehler aufgetreten ist. Deshalb werden die RemoteExceptions direkt dem Benutzer angezeigt, mit einem Hinweis, dass der Fehler auf Seiten des TaskServers entstanden ist.

4.4 Realisierung des Admintool

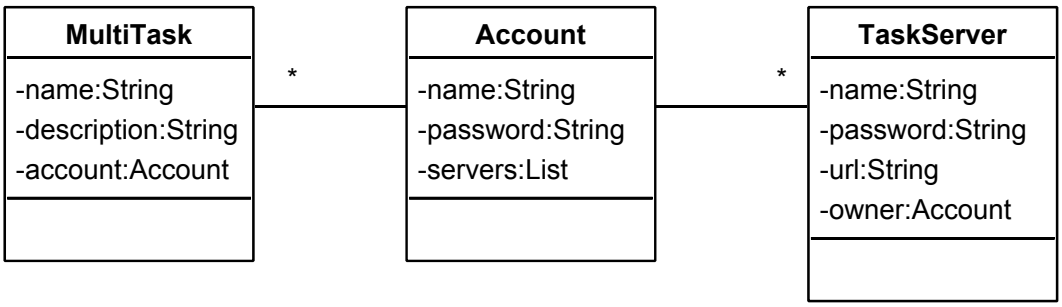

Abbildung 4.3: Datenbank-Schema des Admintools

Wie bereits in 4.1.2 erwähnt, werden die Daten des Admintools in einer Datenbank gespeichert. Das verwendete Datenmodell ist in Abbildung 4.3 dargestellt.

Die Klasse *Account* beschreibt ein Benutzerkonto des Admintools. Jeder Account kann mehrere *TaskServer* besitzen. Ein Account hat keinen Zugriff auf die *TaskServer* anderer Accounts.

Die Klasse *MultiTask* beschreibt eine Massenaufgabe. Eine Massenaufgabe wird immer einem Account zugeordnet. So kann ein Benutzer aus den von ihm definierten Massenaufgaben wählen.

4.4.1 Einzelaufgaben

Die WebWork-Aktion *ExecuteTaskAction* behandelt die Auführung von Einzelaufgaben. Sie erwartet mindestens zwei HTTP-Paramter. Der Parameter *server*, muss den Namen des Servers enthalten, auf dem die gewünschte Aufgabe ausgeführt werden soll, während *task* den Namen der auszuführenden Aufgabe (so wie im entsprechenden TaskServer registriert) enthalten muss. Zusätzlich kann der Parameter *command* übergeben werden, der den Namen des Aufgabenbefehls enthält. Ist dieser nicht angegeben, wird der Befehl *initCommand* ausgeführt.

Die Ausführung der Aufgabe geschieht wie bereits im Design erläutert (siehe 3.2.3). Vorher wird anhand des Modells geprüft, ob der angemeldete Benutzer Zugriff auf den entsprechenden Server hat.

Zusätzlich zur eigentlichen Ausführung der Webservice-Methode, führt die *ExecuteTaskAction* die Datenanpassung durch. Die in den Aufgabenantworten enthaltenen Formulare werden ausgelesen und anhand von Templates in HTML-Formulare umgesetzt. Zwar ist es dank der Templates möglich das Layout von Formularfeldern anzupassen, aber ein individuelles Layout pro Aufgabe ist nicht in dieser Umsetzung nicht möglich (siehe auch 3.5.2). In einer Umgebung mit einer anderen Oberfläche, z. B. Swing, müssten die Formularfelder an dieser Stelle statt in HTML in entsprechende Swing-Komponenten umgesetzt werden.

4.4.2 Massenaufgaben

Für die Ausführung einer Massenaufgabe ist die Klasse *ExecuteMultiTaskAction* zuständig. Diese WebWork-Aktion steuert den Ablauf der Massenaufgaben, wie im Design (2.4.1) beschrieben. Da das gewährleisten der Atomicy-Eigenschaft einen hohen Aufwand erfordert und die Anzahl der Anwendungsfälle vergleichsweise gering ist, wurde in der Realisierung auf die Rollback-Funktionalität verzichtet.

Da Massenaufgaben durch Einzelaufgaben ausgeführt werden, teilen sich *ExecuteTaskAction* und *ExecuteMultiTaskAction* eine gemeinesam Oberklasse, in der eine Großteil der genutzen Logik implementiert ist. Die *ExecuteMultiTaskAction* erwartet zunächst nur den HTTP-Parameter *task*, der den Namen der auszuführenden Massenaufgabe enthält. Nach dem ersten Aufruf wird eine Liste der unterstützten Server angezeigt, die bei jedem weiteren Aufruf mit übergeben wird.

Abbildung 4.4 zeigt ein Bildschirmfoto der Massenaufgabe “Softwarepakete aktualisieren”. An der Ausführung sind die beiden Server “Xarfai” und “Karmoth” beteiligt.

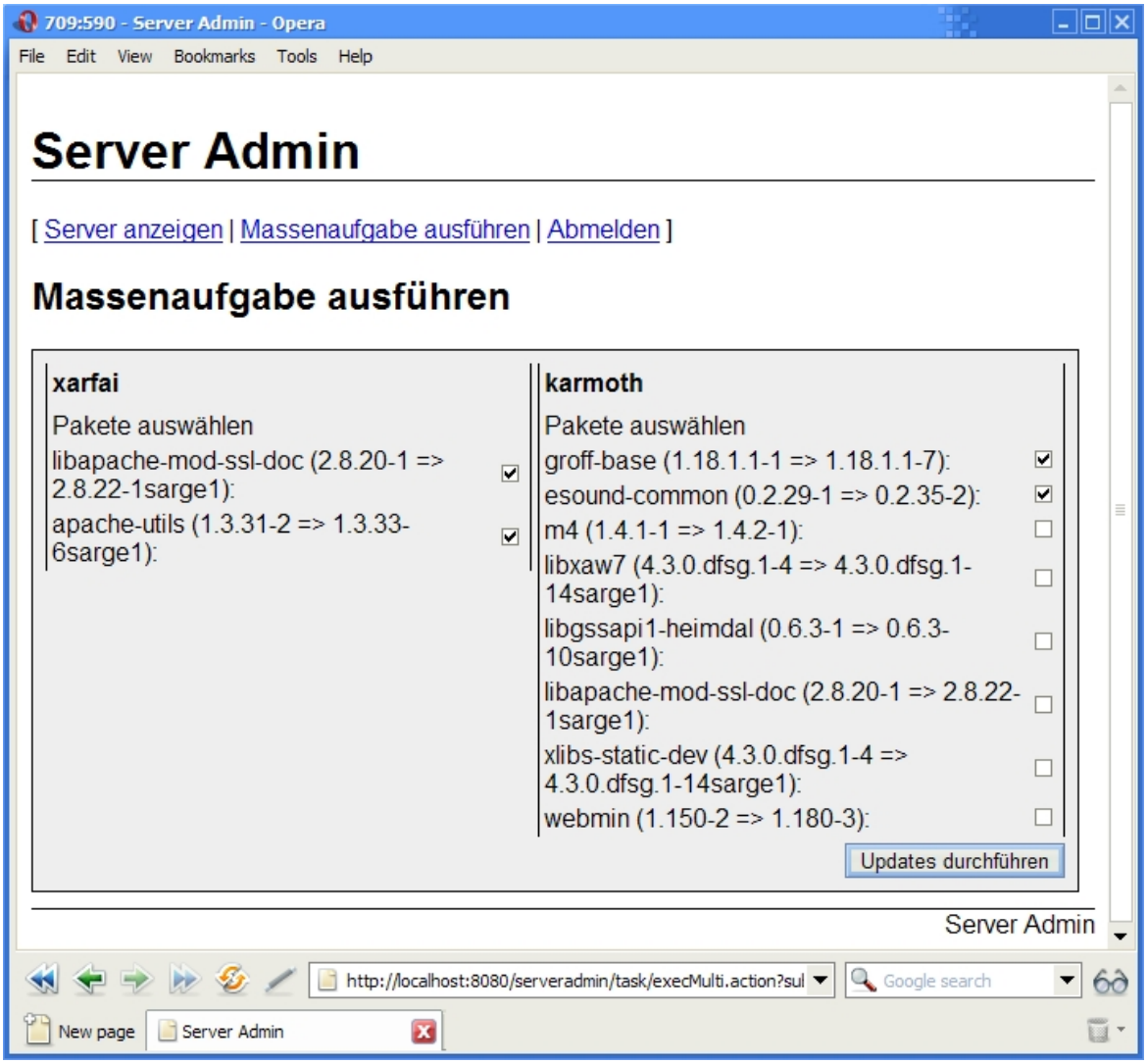

Abbildung 4.4: Bildschirmfoto der Massenaufgabe ’Softwarepakete aktualisieren’

4.4.3 System-Benutzer umziehen

In 3.6 wird erklärt, dass für die Aufgabe “System-Benutzer umziehen” am Admintool selbst Hand angelegt werden muss. Auf Seiten des Admintools ist dafür die WebWork-Aktion *MoveUserAction* zuständig. Sie erbt von der Klasse *ExecuteTaskAction* und kann so deren Mechanismen zur Ausführung von Einzelaufgaben verwenden.

Um eine Einzelaufgabe auszuführen, erwartet die *ExecuteTaskAction* eine Reihe von Parametern, die die Ausführung steuern. Diese können entweder als HTTP-Parameter übergeben werden oder durch andere Methoden vorher gesetzt werden. Die Methode *execute* führt dann die eigentliche Einzelaufgabe aus.

Die *MoveUserAction* setzt vor dem Aufrufen der *execute*-Methode die genannten Parameter, um die Einzelaufgaben korrekt auszuführen. So werden der Aufgabenname, der Aufgabenbefehl und der initiale *TaskRequest* angegeben. Bei der Ausführung der Einzelaufgabe “Download-Archiv erzeugen” beispielsweise, wird der Aufgabenname auf *createDownload* und der Aufgabenbefehl auf *createArchive* gesetzt. Hier wird also nicht wie üblich zuerst *initCommand* ausgeführt, da dies zur Folge hätte, dass ein Formular angezeigt wird, in der der Benutzer aufgefordert wird das Verzeichnis von Hand einzugeben. Da das Verzeichnis jedoch bekannt ist – es ist das Home-Verzeichnis des Quellbenutzers – kann gleich der Folgebefehl *createArchive* mit einem angepassten *TaskRequest* aufgerufen werden.

4.4.4 Test-Client in PHP

Eine in Abschnitt 2.3 definierte Anforderung besagt, dass es möglich sein soll, das Admintool in vorhandene Oberflächen zu integrieren. Da viele Webanwendungen in PHP geschrieben sind, wurde ein minimaler PHP-Client entwickelt, der anstelle des Java-basierten Admintools, eine Aufgabenanfrage an einen TaskServer schickt und die Antwort anzeigt.

Da PHP eine dynamisch typisierte Sprache ist, lassen sich die Aufgabenanfragen leicht erstellen. Es müssen weder Interfaces noch Klassen definiert werden, die sich mit den Datentypen auseinander setzen. Allerdings muss beim Zuweisen der Attribute darauf geachtet werden, dass die Werte den richtigen Typ haben.

Mit der SOAP-Extension von PHP 5 [Stogov (2004)] existiert eine Implementierung des SOAP-Protokolls in PHP, die einfachen Zugriff auf Webservice-Methoden bietet. WS-Security wird von dieser Extension jedoch noch nicht unterstützt, weshalb für diesen Test die Authentifizierung deaktiviert wurde.

Der Test-Client ruft lediglich die Webservice-Methode *executeTask* für die Aufgabe “System-Benutzer erstellen” und dem Aufgabenbefehl *initCommand* auf. Der Inhalt Aufgabenantwort wird als Text angezeigt.

Der Test zeigt, dass die PHP-SOAP-Extension die Anfragen korrekt auf SOAP-Anfragen abbilden kann und empfangene SOAP-Antworten korrekt in PHP-Objekte umsetzt. Es ist also grundsätzlich möglich den TaskServer mit einem PHP-Client anzusprechen.

Es existieren noch weitere SOAP-Implementierungen für PHP, wie z. B. PEAR::SOAP und nuPHP, die hier nicht weiter vorgestellt werden. Da auf den offiziellen Webseiten dieser beiden Projekte keine Informationen über WS-Security zu finden waren, ist davon auszugehen, dass sie diese Erweiterungen ebenfalls nicht unterstützen. Weitere Informationen zu diesen Projekten sind den Quellen [PEAR-SOAP (2006)] und [nuSOAP (2004)] zu entnehmen.

4.5 Realisierung des TaskServers

4.5.1 Veränderungen gegenüber dem Design

Die grundlegende Struktur des TaskServers, insbesondere die Webservice-Methoden, wurden wie im Design beschrieben realisiert. Die Objekte, die als Parameter und als Rückgabewerte der Webservice-Methoden dienen, halten sich an das JavaBean Design Pattern für Getter und Setter [JavaBeans (2003)]. Axis besitzt für diesen Fall einen vorgefertigten BeanSerializer, der Objekte, die sich an den genannten Standard halten in das XML-Format serialisieren und wieder deserialisieren kann. Dadurch ist es möglich komplexe Datenobjekte ohne zusätzliche Codeanpassungen in SOAP-Nachrichten abzubilden.

Die Klassenstruktur ist identisch mit der im Design. Es wurden allerdings ein paar zusätzlich Attribute hinzugefügt, die zu Gunsten der Übersicht im Design nicht erwähnt wurden. Die Klasse *FormField* enthält weitere Attribute, die eine grundlegende Wertüberprüfung auf Clientseite erlauben. Außerdem können diese Werte auch auf Serverseite benutzt werden, um einen Teil der Überprüfung vorzunehmen. Diese Attribute sind im Einzelnen:

Attribut	**Datentyp**	**Beschreibung**
minLength	Integer	Definiert die minimale Wertlänge des Feldes
maxLength	Integer	Definiert die maximale Wertlänge des Feldes
required	Boolean	Gibt an, ob ein Feld ausgefüllt werden muss

An dieser Stelle sind natürlich noch viele weitere Überprüfungen denkbar, wie z. B. ein Attribut für einen Regulären Ausdruck, der auf den Wert des Feldes angewendet wird. Ist dieser Ausdruck nicht auf den Wert anwendbar, schlägt die Überprüfung fehl.

4.5.2 Objekt-Lebenszyklen

Für einen Aufgabenentwickler ist es wichtig zu wissen, wann Aufgabeninstanzen erstellt werden und wie viele es im System geben kann.

Die Klasse *TaskController* enthält die Methoden, die als Webservice aufgerufen werden können. Im TaskServer wird die Standardeinstellung von Axis verwendet, durch die bei jeder Anfrage ein *TaskController*-Objekt erzeugt wird. Der TaskController erzeugt wiederum für jede Anfrage ein neues Aufgabenobjekt, dessen *executeTask*-Methode mit den Parametern der Webservice-Methode aufgerufen wird.

Der Vorteil dieser Konfiguration liegt darin, dass Aufgabenentwickler nicht auf Threadsicherheit achten müssen. Zwar wird dadurch unter Umständen etwas mehr Speicher verwendet und die Erzeugung der Objekte nimmt etwas CPU-Zeit in Anspruch, aber da ein TaskServer in der Regel nur sehr wenige Anfragen parallel (wenn überhaupt) beantworten muss, ist dieser Nachteil vernachlässigbar.

4.5.3 Skriptaufgaben

Die in 3.5 vorgestellte Skriptaufgabenklasse bildet eine zusätzliche Schicht zwischen dem TaskServer und den Skripten. Diese Schicht muss die Daten des *TaskRequest*-Objekts dem Skript zur Verfügung stellen und aus den Daten, die das Skript liefert, ein *TaskResponse*-Objekt erstellen.

Anfragen umwandeln

Um die *TaskRequest*-Objekte dem Skript zur Verfügung zu stellen, werden die Formulardaten vor der Ausführung des Skripts als Umgebungsvariablen gesetzt. Innerhalb des Skripts kann dann über diese Variablen, deren Namen aus einem vordefinierten Präfix und den Namen der Felder bestehen, auf die Feldwerte zugegriffen werden.

Neben den Formulardaten enthält ein *TaskRequest*-Objekt noch den Aufgabenbefehl, der ebenfalls dem Skript über eine Umgebungsvariable zur Verfügung gestellt wird.

Antworten erstellen

Da die Antworten, insbesondere die Formulare, aus verhältnismäßig komplexen Datenstrukturen bestehen, wird für deren Erzeugung zum Teil auf vorhandene Komponenten zurückgegriffen. In 4.2 wurde bereits erwähnt, dass die Formulare mittels XML-Dateien definiert werden. Es ist also ratsam, die Formulare für Skriptaufgaben auf die gleiche Weise zu definieren und zu interpretieren. Die Skriptaufgabenklasse greift genauso auf die Formular-Dateien zu, wie eine “normale” Aufgabenklasse. Die Skripte selbst können die Skriptaufgabenklasse dazu anweisen, ein Formular anzuzeigen (siehe nachfolgende Beispielbefehle).

Aufgabenantworten bestehen aber noch aus weiteren Daten, wie Nachrichten, Fehlermeldungen und Aktualisierungsbefehlen, die nicht in der Formular-Datei definiert werden können. Außerdem müssen die Feldwerte der Formulare dymanisch gesetzt werden, was nicht über die XML-Datei geschehen kann. Daher wird eine Möglichkeit benötigt, diese Werte der Skriptaufgabenklasse mitzuteilen, damit sie diese der Aufgabenantwort hinzufügen kann.

Eine Möglichkeit dies zu realisieren wäre, diese Daten in temporären Dateien abzulegen, die nach der Ausführung des Skripts von der Skriptaufgabenklasse ausgelesen, interpretiert und gelöscht werden. Diese Methode würde allerdings einen zusätzlichen Umweg über Dateien bedeuten, die von dem Skript selbst erstellt werden müssten. Da es bei den Skriptaufgaben jedoch darum geht, die Erstellung von Aufgaben zu vereinfachen, wurde eine andere Methode gewählt:

Die Skripte geben die Daten, die zur Aufgabenantwort hinzugefügt werden sollen, auf der Standardausgabe aus. Die Skriptaufgabenklasse liest die Standardausgabe des Skripts und interpretiert sie. Um der Skriptaufgabenklasse mitzuteilen, dass Daten zur Aufgabenantwort hinzugefügt werden sollen, muss das Skript eine Zeile ausgeben, die mit einem definierten Präfix beginnt.

Ein paar Beispiele:

`#ts: message: System-Benutzer wurde erstellt:: testuser`
Diese Zeile weist die Skript-Aufgabe an, die Nachricht “System-Benutzer wurde erstellt: testuser” der Aufgabenantwort hinzuzufügen. Doppelpunkte sind Steuerzeichen. Zwei aufeinander folgende Doppelpunkte geben einen Doppelpunkt aus.

`#ts: exception: Ungültiger Aufgabenbefehl`
Veranlasst die Skript-Aufgabe dazu, eine RemoteException mit der Nachricht “Ungültiger Aufgabenbefehl” zu generieren.

`#ts: next: create: Erstellen`
Fügt der Aufgabenantwort einen Folgebefehl mit dem Namen *create* und der Beschreibung “Erstellen"hinzu.

`#ts: form: createSystemUser.xml`
Fügt das in der Datei *createSystemUser.xml* definierte Formular hinzu.

Zusammenfassung

Die Erstellung einer Skriptaufgabe ist in der Regel deutlich weniger umständlich, als die Erstellung einer eigenen Aufgabenklasse. Die notwendigen Schritte können, beispielsweise über eine Remote Shell (z. B. per SSH), erledigt werden:

- Erstellen des Skripts, sowie der Formulardateien.
- Erstellen der Konfigurationsdatei, in der Interpreter und Dateiname des Skripts angegeben werden.
- Registrieren der Aufgabe in der TaskServer-Konfigurationsdatei.

Im Gegensatz dazu sind zum Erstellen einer Aufgabe mittels TaskGenerator zusätzliche und aufwendigere Schritte notwendig:

- Erstellen der XML-Datei für das Grundgerüst der Aufgabe.
- Erweitern der generierten Klassen um die eigentliche Logik.
- Kompilieren der Klassen. Für diesen Schritt wird eine Umgebung benötigt, die das Kompilieren der Klassen erlaubt. Es wird also mindestens ein Java Development Kit benötigt, sowie die vom TaskServer verwendeten Bibliotheken. Das Testen ist ohne eine lokale Version des TaskServers, der ebenfalls im Admintool registriert ist, relativ umständlich, da bei jeder Änderung die Klassen wieder auf den TaskServer kopiert werden müssen.
- Kopieren der Klassen auf den TaskServer.
- Registrieren der Aufgabe in der TaskServer-Konfigurationsdatei.

Die Verwendung von Skriptaufgaben bringt allerdings auch Nachteile mit sich:

- Da Aufgabenanfragen und -antworten nun eine zusätzliche Konvertierung erfahren und die Logik an anderer Stelle ausgeführt wird, ist das Auffinden von Fehlern unter Umständen problematischer.
- Die Skripte sind nicht so gut in den TaskServer eingebunden, wie normale Aufgabenklassen. Dadurch ist es beispielsweise nicht möglich von einer Aufgabe andere Aufgaben aufzurufen, es sei denn, es handelt sich bei diesen ebenfalls um Skriptaufgaben. Aber selbst dann ist es umständlicher die Antworten der zusätzlich aufgerufenen Aufgabe abzufangen und gegebenenfalls zu verändern.

4.6 Umsetzung der Sicherheitsmechanismen

In Abschnitt 3.7 wird beschrieben, dass zwei Mechanismen für die Gewährleistung der Sicherheit zuständig sind, die im Folgenden näher beschrieben werden.

4.6.1 WS-Security

Da die auf WS-Security aufbauenden Spezifikationen noch nicht als Standard freigegeben wurden, existieren bisher nur wenige Implementierungen. Mit WSS4J [WSS4J (2006)] existiert allerdings bereits eine Implementierung des WS-Security 1.0 Standards für Java, mit der es möglich ist, eine zusätzliche Sicherheitsschicht in die Abarbeitungskette von Webservice-Nachrichten einzuklinken. Aber auch diese Implementierung basiert auf noch nicht freigegebenen Spezifikationen.

Die Dokumentation von WSS4J ist außerdem noch nicht besonders ausführlich. Daher wird sich im Rahmen dieser Arbeit auf die Verwendung des UsernameTokens beschränkt, um zu demonstrieren, wie WS-Security-Mechanismen in das System integriert werden können.

Um die Einbindung des UsernameTokens genauer erläutern zu können, ist es sinnvoll sich die Mechanismen genauer anzusehen, die Axis für das Versenden und Empfangen von SOAP-Nachrichten zur Verfügung stellt.

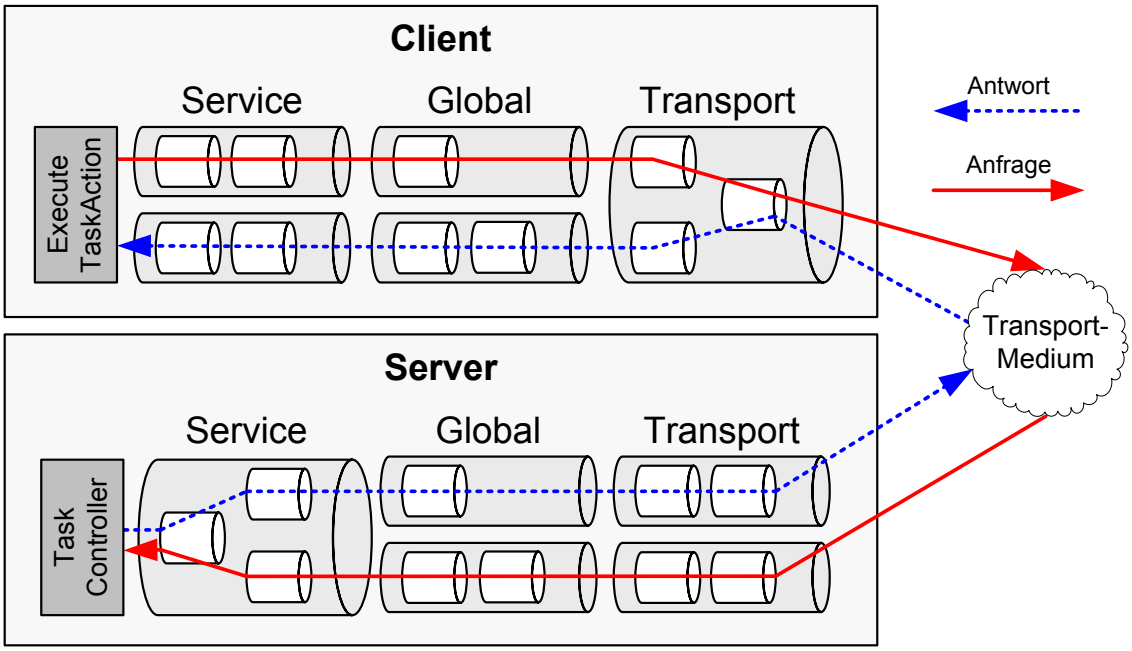

Abbildung 4.5: Axis Nachrichtenverarbeitung

Die Verarbeitung von Anfrage- und Antwortnachrichten wird durch sogenannte Handler durchgeführt (siehe Abbildung 4.5). Bei jeder Verarbeitung durchläuft eine Nachricht mehrere Handler (in der Abbildung durch Zylinder dargestellt), die wiederum weitere Handler enthalten können und damit eine Handler-Kette bilden. Jedem Handler wird beim Aufruf seiner *handle*-Methode ein *MessageContext*-Objekt übergeben, das wichtige Daten über die zu behandelnde Nachricht enthält. Dies sind unter anderem eine Anfragenachricht, eine Antwortnachricht und eine Reihe von Eigenschaften, die von der AxisEngine gesetzt werden. Um zusätzliche Handler in den Verarbeitungsstrom einzuklinken, werden sowohl auf Client-, wie auch auf Serverseite, zusätzliche Einstellungen im Deployment Descriptor[1] vorgenommen.

Von Axis werden drei Handler vorgegeben:

- Der *Transport*-Handler übernimmt die Verarbeitung des Transport-Protokolls, in diesem Fall also HTTP. Er bildet die Schnittstelle zwischen den protokollspezifischen Daten und dem MessageContext.
- Der *Global*-Handler kann verschiedene Aufgaben übernehmen und ist standardmäßig leer.
- Der *Service*-Handler bildet die Schnittstelle zum eigentlichen Service, also in diesem Fall der Java-Methode, die aufgerufen werden soll.

In diesem konkreten Fall werden die WSS4J-Handler *WSDoAllSender* (auf Clientseite) und *WSDoAllReceiver* (auf Serverseite) im Global-Handler registriert. Beide Handler werden durch Parameter im Deployment Descriptor konfiguriert. Durch sie wird festgelegt, welche Klassen die Überprüfung beziehungsweise die Generierung des UsernameTokens vornehmen.

Da das Admintool mehrere TaskServer mit unterschiedlichen Benutzernamen und Passwörtern ansprechen muss, wird im Deployment Descriptor des Clients nicht festgelegt, welcher Benutzername und welches Passwort bei jeder Webservice-Anfrage mitgesendet werden sollen. Stattdessen wird bei der ersten Webservice-Anfrage an einen Server, dem SoapBindingStub[2] der für diesen Server zu verwendende Benutzername mitgeteilt. Die im Deployment Descriptor festgelegte Klasse *ClientPWCallback*, liest das Passwort anhand des Benutzernamens aus der Datenbank und fügt die Authentifizierungsdaten gemäß den UsernameToken-Spezifikationen der SOAP-Nachricht hinzu.

Die Überprüfung des UsernameTokens auf Serverseite übernimmt die im Deployment Descriptor des Servers definierte Klasse *ServerPWCallback*. Dort werden Benutzername und

[1] Ein Deployment Descriptor ist eine XML-Datei, durch die definiert wird, wie ein Webservice in Axis eingebunden wird

[2] Ein Stub ermöglicht, die Funktionalität eines entfernten Systems so anzusprechen, als wäre es ein Bestandteil des lokalen Systems.

Passwort ausgelesen und mit den Daten in der TaskServer-Konfigurationsdatei verglichen. Stimmen Benutzername oder Passwort nicht überein oder fehlt der UsernameToken komplett, wird die Ausführung der Service-Methode abgebrochen.

Um weitere Sicherheitsmechanismen der WS-Security-Spezifikationen zu verwenden, könnten weitere Klassen registriert werden, die beispielsweise Nachrichten signieren beziehungsweise die Signatur überprüfen. Die Handler *WSDoAllSender* und *WSDoAllReceiver* sind in der Lage auch mehrere Klassen anzusprechen. Sie müssen dafür ebenfalls in den Deployment Descriptoren definiert werden.

4.6.2 SSL

Die Verwendung von SSL für alle Webservice-Anfragen wird auf Seiten des TaskServers von Tomcat übernommen. Auf die von Tomcat bereitgestellten Webanwendungn kann über verschiedene Konnektoren zugegriffen werden. Einer dieser Konnektoren bietet die Möglichkeit HTTPS mit clientseitiger Authentifizierung zu verwenden.

Auf Seiten des Admintools (also auf Clientseite) übernimmt die von Java bereitgestellte Klasse URLConnection beziehungsweise deren Unterklassen die HTTPS-Kommunikation.

Da Server und Client sich gegenseitig authentifizieren, müssen beide Zertifikate bereitstellen. Außerdem müssen beide Seiten das Zertifikat des Partners akzeptieren. Auf Serverseite wird dem Tomcatkonnektor per Konfigurationsdatei mitgeteilt, welche Zertifikate akzeptiert werden. Auf Clientseite geschieht dies über die System-Properties der Java Virtual Machine.

4.6.3 Zusammenfassung

Auch wenn die WS-Security-Spezifikationen noch nicht vollständig ausgereift sind und auch die vorhandenen Implementierungen noch zu Wünschen übrig lassen, ist es dennoch möglich sie einzusetzen. Was im Rahmen dieser Arbeit allerdings nicht geprüft wird ist, ob WSS4J auch mit anderen WS-Security-Implementationen – wie beispielsweise mit Microsofts “Web Services Enhancements” (WSE) für .NET – zusammenarbeiten kann. Eine Übersicht über die Funktionen der WSE ist der Quelle [Parys und Mauerer (2004)] zu entnehmen.

Da SSL und WS-Security auf unterschiedlichen Ebenen im System implementiert werden, ist es kein Problem beides zu kombinieren. Außerdem ist es dank der modularen Handler-Architektur von Axis ohne Weiteres möglich WS-Security-Funktionalität nachzurüsten, ohne dass die eigentlichen Webservice-Methoden angepasst werden müssen.

4.7 Hinzufügen neuer Server

Das Einbinden neuer Server in das Administrationssystem ist relativ einfach:

1. Installation des TaskServers auf dem gewünschten Server: Die Installation besteht im Normalfall nur aus dem Kopieren des TaskServers, einschließlich der Servlet-Engine (Tomcat). Ist auf dem Server bereits eine Servlet-Engine installiert, kann der TaskServer unter dieser zum Laufen gebracht werden. Falls nicht vorhanden, muss außerdem auf dem Server noch eine Java Runtime Environment installiert werden.
2. Anpassen der Firewall: Ist der Server durch eine Firewall gesichert, muss ein beliebiger Port für eingehende TCP-Anfragen, die von der IP-Adresse des Admintools kommen, geöffnet werden. Außerdem muss Tomcat so konfiguriert werden, dass dieser auf dem gewählten Port SSL-Anfragen beantwortet.
3. Zertifikate austauschen: Das SSL-Zertifikat des Admintools muss im TaskServer als vertrauenswürdig registriert werden. Dies geschieht sehr einfach über das Java-Keytool. Für den neuen Server muss, sofern noch nicht vorhanden, ein Zertifikat generiert werden, welches wiederum im Admintool registiert werden muss.
4. TaskServer im Admintool registrieren: Abschließend muss der neue TaskServer mitsamt seiner Webservice-URL im Admintool registriert werden. Außerdem muss mindestens ein Admintool-Benutzer angelegt werden, der Zugriff auf diesen Server hat.

4.8 Fazit

Während der Realisierung hat sich gezeigt, dass das Design im Großen und Ganzen gut umzusetzen ist. Allerdings wurden an einigen Stellen Abstriche an der Funktionalität gemacht, da der Aufwand für den Rahmen dieser Arbeit zu groß geworden wäre.

So wurde beispielsweise die im Design vorgestellte Webservice-Methode *rollBack* nicht implementiert. Die Bereitstellung der Methode wäre zwar kein Problem und auch nicht die Implementierung auf Seiten des Admintools, das die Methode im Fehlerfall ausführen müsste. Die Funktion der *rollBack*-Methode jedoch, ist je nach Aufgabe sehr umständlich. Sie erfordert vom Aufgabenentwickler doppelten Aufwand, da bei der Anpassung einer Aufgabe nicht nur deren Ausführung geändert werden muss, sondern auch die Funktionalität für das rückgängig machen dieser Ausführung.

Außerdem wurde auf die Implementierung eines mehrsprachlichen TaskServers verzichtet.

Das Konzept des TaskGenerators geht zwar auf, jedoch generiert dieser nicht viel mehr, als ein Grundgerüst für eine Aufgabe. Auch in dieser Form ist er zwar eine große Erleichterung, besonders was die Definition von Formularen angeht, die durch ihn erzeugte Logik ist jedoch sehr gering. Mit einigem Aufwand ist der TaskGenerator jedoch soweit erweiterbar, dass er grundlegende Konzepte von Programmiersprachen umsetzen kann. Es wäre hierbei denkbar sich an der Syntax von Taglibs zu orientieren, die es ermöglichen, einer JSP-Seite viel Logik mit XML-Syntax hinzuzufügen.

Die Skriptaufgaben bieten hierzu eine gute Alternative, bringen allerdings die genannten Probleme mit sich. Inwiefern diese Probleme die Entwicklung von Aufgaben beeinflussen, hängt von der Komplexität der Aufgaben ab. Einfache Testaufgaben waren als Skriptaufgabe schnell und einfach realisierbar.

Das Protokoll lies sich, bis auf die oben erwähnte Ausnahme, so implementieren, wie im Design vorgeschrieben. Es bleibt jedoch noch zu untersuchen, ob der TaskServer auch durch andere SOAP-Implementierungen nachgebaut werden kann bzw. wie aufwendig sich dies gestaltet. Auf Clientseite hingegen wurde ein minimaler Test-Client für PHP entwickelt, der demonstriert, dass auch mit anderen Sprachen auf den TaskServer zugegriffen werden kann.

Gleichzeitig bietet das Protokoll Anlass zum Nachbessern. Viele Funktionen, die die Benutzerführung vereinfachen könnten, sind mit dem Protokoll nicht umsetzbar. Ein Beispiel dafür sind kleine, helfende Abläufe, die in Webanwendungen überlichweise durch Javascript erledigt werden, wie z. B. das automatische Ausfüllen von Feldern anhand von Benutzereingaben in anderen Feldern.

Des Weiteren enthält das Protokoll zwar Definitionen für Werteüberprüfung, allerdings könnten auch diese durchaus allgemeiner gestaltet werden. Im Grunde sind zwar gar keine Informationen über die Werteüberprüfung im Protokoll nötig, da sie auch komplett auf Serverseite ausgeführt werden kann, eine Clientseitige Werteüberprüfung geht jedoch deutlich schneller und erhöht dadurch den Bedienungskomfort.

4.8.1 Mögliche Detailerweiterungen

Das entwickelte System kann an verschiedenen Stellen verbessert werden:

- Erweiterte Antwortobjekte: Die Antwortobjekte enthalten zu wenig Informationen über die daraus zu generierende Benutzeroberfläche. Komplexere Aufgaben, die mehr als nur “einfache” Formulare erfordern können hier schnell an ihre Grenzen stoßen. Werteüberprüfung, Layoutinformationen und clientseitige Interaktion zwischen den Formularfeldern können jedoch verbessert werden.

- Besseres Debugging: Während der Entwicklung des Admintools konnte stets Einsicht in die Ausgaben und Logdateien des TaskServers und des Admintools genommen werden. Ein Aufgabenentwickler hat allerdings im Normalfall nur Zugriff auf den TaskServer und dessen Ausgaben. Daher wäre es wünschenswert die Ausgaben des Admintools möglichst ausführlich zu gestalten. Es sollte also mehr, als die Meldung "Die Aufgabenantwort enthielt nicht alle nötigen Daten" angezeigt werden. Denkbar wäre es, einen Debug-Modus im Admintool anzubieten, der die Aufgabenantworten genauer ausgibt und überprüft.
- Caching: Das Admintool stellt derzeit sehr viele Anfragen an die TaskServer. Wenn Massenaufgaben ausgeführt werden, wird jedes Mal von jedem TaskServer die Aufgabenliste angefordert. Diese könnte auf dem Admintool zwischengespeichert werden.
- Netzwerkfehler behandeln: Während Fehler auf Anwendungsebene so weitergegeben werden, dass ein Aufgabenentwickler damit etwas anfangen kann, werden Netzwerkfehler gar nicht weiter behandelt. Insbesondere Timeouts sollten besser behandelt werden, damit die Ausführung einer Massenaufgabe nicht bei jedem Aufruf einer Webservice-Methode stark verzögert wird.
- Mehrsprachlichkeit: Das System könnte in mehreren Sprachen angeboten werden. Dafür muss das Protokoll allerdings eine Möglichkeit bieten, dem TaskServer mitzuteilen, in welcher Sprache die Benutzerinformationen der Antworten geliefert werden sollen. Der Aufwand dafür wäre für Aufgabenentwickler naturgemäß hoch, da alle Komponenten übersetzt werden müssen. Da für einen Server meist nur wenige Administratoren zuständig sind, die in der Regel die gleiche Sprache sprechen, ist Mehrsprachlichkeit auf Seiten des TaskServers eher unwichtig. Das Admintool sollte allerdings auf die Sprache des Benutzers eingestellt werden können. Der TaskServer, der auf den Servern des Benutzers läuft, könnte dann Meldungen in der Sprache des Benutzers liefern. Für diese "einseitige" Mehrsprachlichkeit, müsste das Protokoll nicht angepasst werden.

5 Abschlussbetrachtung

5.1 Zusammenfassung

Im Rahmen dieser Arbeit, sollten im Wesentlichen zwei Ziele verfolgt werden: Es sollte ein System entwickelt werden, das die Administration von heterogenen Serverumgebungen vereinfacht. Außerdem sollte die Nutzbarkeit von Webservices für dieses System untersucht werden.

Für die Entwicklung des Systems, wurde zunächst ermittelt, welche Anforderungen das System erfüllen muss, damit es ausreichende Erleichterung bietet. Dafür wurden einige, häufig autretende Administrationstätigkeiten beschrieben, anhand derer die Anforderungen generell aufgezeigt wurden.

Anhand der Erkentnisse aus der Analyse, wurde ein Konzept für die Entwicklung des Systems erstellt. Im Rahmen des Designs wurden verschiedene Möglichkeiten aufgezeigt, um Aufgaben zu entwickeln und so dem System zusätzliche Funktionalität hinzuzufügen. Das beschriebene Protokoll ermöglicht es, unterschiedliche Server mit den gleichen Befehlen anzusprechen. Dadurch wird erreicht, dass durch den TaskServer unterschiedliche Server nach außen hin, also für das Admintool, gleich erscheinen.

Aus den Erkentnissen über die möglichen Aufgabenbeschreibungen aus dem Design, wurde ein Codegenerator entwickelt, der das Erstellen von Aufgaben erleichtert. Da die vom Codegenerator erzeugten Klassen wenig Logik enthalten, wurde zusätzlich eine Möglichkeit entwickelt, Aufgaben von beliebigen Skripten ausführen zu lassen, um die Aufgabenbeschreibung zu vereinfachen.

Der Prototyp des Systems wurde auf einen Stand gebracht, der die Anforderungen grundsätzlich erfüllt. Es bedarf allerdings noch großen Aufwands, um den Prototypen auf einen Stand zu bringen, der die nötige Erleichterung für Administratoren bietet. Es sind einige Detailverbesserungen nötig oder zumindest wünschenswert, damit der Einsatz des Systems wirklich lohnt.

Die Tauglichkeit von Webservices für die Umsetzung des Protokolls hat sich als positiv herausgestellt. Die negative Performance im Gegensatz zu anderen Middleware-Lösungen fällt

in diesem System nicht besonders ins Gewicht, da es mit vergleichsweise wenig Webservice-Methodenaufrufen auskommt.

TaskServer und Admintool zusammen bilden ein lose gekoppeltes System mit einer klar definierten Schnittstelle. Wird diese von beiden Seiten eingehalten, ist es möglich, beide Komponenten auszutauschen, ohne die andere verändern zu müssen. Es ist relativ einfach möglich weitere Server zum System hinzuzufügen, sofern ein bereits entwickelter TaskServer eingesetzt wird.

Die Sicherheit ist dank SSL und WS-Security in diesem System kein besonderes Problem. In Zukunft ist außerdem damit zu rechnen, dass auch Anwendungen mit komplexeren Webservice-Strukturen mittels WS-Security an Sicherheit gewinnen, sobald die darauf aufbauenden Standards verabschiedet wurden und Implementierungen verfügbar sind.

5.2 Ausblick und Kritik

Um mögliche Erweiterungen aufzuzeigen, sei zunächst eine Anforderung erwähnt, die von dem System *nicht* erfüllt werden soll.

Es war nie vorgesehen, das System auf einen Stand zu bringen, damit jeder es bedienen kann. Administration von Servern ist eine Tätigkeit, die nicht von Laien durchgeführt werden sollte. Auch wenn sie durch eine gute Benutzeroberfläche vereinfacht wird, sollte der Benutzer die zu administrierenden Server möglichst genau kennen. Insbesondere in einem heterogenen Serverumfeld ist diese Vereinfachung praktisch nicht erreichbar.

Diese Einschränkung trifft allerdings nicht hundertprozentig zu. In der Analyse wurde der Anwendungsbereich Serververmietung vorgestellt (2.2.1). Dieser Service wird durchaus von Kunden in Anspruch genommen, die nur wenig Erfahrung im Bereich Administration haben. Um diesen Kunden eine Erleichterung zu bieten, könnten erfahrene Entwicklern, Aufgaben erstellen, die auf die von diesen Kunden gemieteten Server angepasst sind. Es gilt allerdings auch für diese Kunden, dass sie ein grundlegendes Verständnis für die Abläufe auf ihren Servern mitbringen sollten.

Ein weiterer denkbarer Einsatz für das Administrationssystem wäre, durch das Admintool eine Möglichkeit zu bieten, beurlaubte Administratoren zu vertreten. Auch in diesem Fall sollte die Vertretung Erfahrung im Bereich Administration mitbringen und das System einigermaßen kennen. So könnten vorkonfigurierte Aufgaben, wie beispielsweise das Anlegen von System-Benutzern, durch die Vertretung durchgeführt werden, auch wenn diese das System nicht bis ins Detail kennt. Allerdings ist der Einsatz in einem solchen Vertretungsszenario auch nur dann möglich, wenn die entsprechenden Server bereits mit dem Administrationssystem arbeiten. Es wäre zu viel Aufwand, für ein paar Server “mal eben” vor dem Urlaub

des Administratoren, den TaskServer zu installieren, anzupassen und im Admintool zu registrieren. Dies wäre höchstens dann denkbar, wenn es sich um Server handelt, deren Konfiguration anderen Servern gleicht, für die bereits ein TaskServer konfiguriert wurde. In dem Fall könnte der TaskServer komplett übernommen werden.

Allgemein sollte für jeden Server geprüft werden, ob es sich lohnt ihn in das System aufzunehmen. Viele Aufgaben sind auf nahezu jedem Server auszuführen, wie das Verwalten von System-Benutzern. Daher wäre es sinnvoll eine Reihe von Standardaufgaben für den TaskServer zu entwickeln, die dann (wenn überhaupt) nur noch in der entsprechenden Konfigurationsdatei der Aufgabe angepasst werden müssen. Ob sich der Aufwand für die Entwicklung von neuen Aufgaben lohnt, dürfte die Praxis zeigen.

Es bleibt also ein wichtiger Punkt im Rahmen dieser Arbeit ungeklärt: Bietet das System eine ausreichende Erleichterung, damit sich der Einsatz lohnt?

Um diese Frage zu beantworten, muss das System im praktischen Einsatz – oder zumindest in Testumgebungen – untersucht werden. Es müssen Aufgaben unterschiedlicher Komplexität entwickelt und angepasst werden. Dabei fallen sicherlich einige Probleme auf, die aufzeigen, an welchen Stellen das System einem Aufgabenentwickler zusätzliche Hilfestellung geben könnte.

Abgesehen von der Problematik des Aufwand/Nutzen-Faktors, wurde eine Anforderung an das System nur unzureichend erfüllt. Diese Anforderung ist die clientseitige Integration in vorhandene Oberflächen. Dies ist zwar grundsätzlich möglich, es muss jedoch sehr viel Aufwand für eine solche Integration betrieben werden. Es wäre daher wünschenswert, Client-Bibliotheken in verschiedenen Sprachen anzubieten, die den Zugriff auf die TaskServer erleichtern.

Wie in der Einleitung bereits erwähnt, wurde das System im Hinblick auf kleinere Anbieter entwickelt, die weniger als 100 Server verwalten. Daher bleibt zu untersuchen, wieviele Server das System verkraften kann. Bei der Entwicklung wurde nicht weiter auf eine wirklich große Anzahl von Servern geachtet. Nicht nur die Performance spielt ab einer gewissen Anzahl eine Rolle, sondern auch die Darstellung. Wenn beispielsweise Softwarepakete auf mehr, als 20 Servern gleichzeitig aktualisiert werden sollen, wird die dafür angezeigte Seite sehr groß. Zwar beeinträchtigt das nicht die Funktionalität, aber die Bedienung wird – je nach Anbindung des Benutzers – schwerfälliger und unübersichtlicher. Allerdings ist es fraglich, ob diese Aufgabe überhaupt mit so vielen Servern auf einmal ausgeführt werden soll.

Wenn das Admintool viele Benutzer hat, die das System gleichzeitig nutzen, könnte sich die Performance der Webservice-Anfragen auf Clientseite als zu gering herausstellen. Die Serverseite sollte kein Problem darstellen, da im praktischen Einsatz nicht damit zu rechnen ist, dass mehr als ein oder zwei Benutzer gleichzeitig auf einen TaskServer zugreifen.

Ingesamt betrachtet, bietet das in dieser Arbeit vorgestellte System eine solide Grundlage für die Administration von heterogenen Serverumgebungen. Es muss allerdings noch Detailerweiterungen erfahren, um den Einrichtungsaufwand für TaskServer weiter zu reduzieren.

Literaturverzeichnis

[alphaWorks 2005] ALPHAWORKS: *alphaWorks : SOA and Web services : New to SOA and Web services*. Verifiziert am 12.05.2006. 2005. – URL http://www.alphaworks.ibm.com/webservices/newto

[Axis 2006] TEAM, The Axis D.: *WebServices - Axis*. Verifiziert am 09.05.2006. 2006. – URL http://ws.apache.org/axis/

[Brose und Staamann 2002] BROSE, Dr. G. ; STAAMANN, Sebastian: *Application Security Gateways - Eine Komplettlösung für die Sicherheit von Applikationsservern*. Verifiziert am 12.05.2006. 2002. – URL http://www.xtradyne.com/documents/articles/Xtradyne-I-DBC-Objektspektrum.pdf

[CORBA 2006] GROUP, Object M.: *Catalog of CORBA/IIOP Specifications*. Verifiziert am 13.05.2006. 2006. – URL http://www.omg.org/technology/documents/corba_spec_catalog.htm

[Coulouris u. a. 2002] COULOURIS, George ; DOLLIMORE, Jean ; KINDBERG, Tim: *Verteilte Systeme: Konzepte und Design*. Pearson Studium, 2002. – ISBN 3-8273-7022-1

[FrontController 2003] MICROSYSTEMS, Sun: *Core J2EE Patterns - Front Controller*. Verifiziert am 18.05.2006. 2003. – URL http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html

[Gamma u. a. 1994] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. – ISBN 0-201-63361-2

[Gerlach 2005] GERLACH, Martin: *Service-Oriented Architectures: Transaktionsmanagement mit Services und Geschaftsprozessen*, HAW Hamburg, Ausarbeitung zum Seminar Anwendungen 1 des Master-Studiengangs Informatik an der HAW Hamburg (21.12.2005), 2005. – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2005/gerlach/abstract.pdf

[Gray 2004] GRAY, N.A.B.: *Comparison of Web Services, Java-RMI, and CORBA service implementations*. Verifiziert am 13.05.2006. 2004. – URL http://mercury.it.swin.edu.au/ctg/AWSA04/Papers/gray.pdf

[Hibernate 2006] JBOSS.COM: *hibernate.org - Hibernate*. Verifiziert am 10.05.2006. 2006. – URL http://www.hibernate.org/

[Interceptor 2002] MICROSYSTEMS, Sun: *Core J2EE Patterns - Intercepting Filter*. Verifiziert am 07.05.2006. 2002. – URL http://java.sun.com/blueprints/corej2eepatterns/Patterns/InterceptingFilter.html

[JavaBeans 2003] MICROSYSTEMS, Sun: *JavaBeans*. Verifiziert am 07.05.2006. 2003. – URL http://java.sun.com/products/javabeans

[JavaRMI 2003] MICROSYSTEMS, Sun: *Java Remote Method Invocation (Java RMI)*. Verifiziert am 13.05.2006. 2003. – URL http://java.sun.com/products/jdk/rmi/

[Kahlbrandt 2001] KAHLBRANDT, Bernd: *Software-Engineering mit der Unified Modeling Language*. Springer Berlin, 2001. – ISBN 3-540-41600-5

[Knuth 2003] KNUTH, Michael: *Web Services - Einführung und Übersicht*. Software und Support Verlag GmbH, 2003. – ISBN 3-935042-38-8

[Maseberg 2002] MASEBERG, Jan S.: *Fail-Safe-Konzept für Public-Key-Infrastrukturen*, Technische Universität Darmstadt, Doktorarbeit, 2002. – URL ftp://ftp.informatik.tu-darmstadt.de/pub/TI/reports/maseberg.diss.pdf

[MVC 2002] MICROSYSTEMS, Sun: *Design Patterns: Model-View-Controller*. Verifiziert am 13.05.2006. 2002. – URL http://java.sun.com/blueprints/patterns/MVC.html

[nuSOAP 2004] AYALA, Dietrich: *Dietrich Ayala | NuSOAP*. Verifiziert am 21.05.2006. 2004. – URL http://dietrich.ganx4.com/nusoap/

[OpenSSH 2006] OPENBSD: *OpenSSH*. Verifiziert am 10.05.2006. 2006. – URL http://www.openssh.com/

[Parys und Mauerer 2004] PARYS, Dariusz ; MAUERER, Jürgen: *Web Services Enhancements*. Verifiziert am 18.05.2006. 2004. – URL http://www.microsoft.com/germany/msdn/library/xmlwebservices/WebServicesEnhancements.mspx

[PEAR-SOAP 2006] CARAVEO, Shane ; BAKER, Al ; SCHNEIDER, Jan: *PEAR :: Package :: SOAP*. Verifiziert am 21.05.2006. 2006. – URL http://pear.php.net/package/SOAP

[PostgreSQL 2006] GROUP, PostgreSQL Global D.: *PostgreSQL: The world's most advanced open source database*. Verifiziert am 10.05.2006. 2006. – URL http://www.postgresql.org/

[RFC2617 1999] DIVERSE: *RFC2617 HTTP Authentication: Basic and Digest Access Authentication*. Verifiziert am 15.05.2006. 1999. – URL http://rfc.net/rfc2617.html

[RMIfw 2003] JGURU: *jGuru: Remote Method Invocation (RMI)*. Verifiziert am 13.05.2006. 2003. – URL http://java.sun.com/developer/onlineTraining/rmi/RMI.html#FirewallIssues

[SOAP 2003] DIVERSE: *SOAP Version 1.2 Part 1: Messaging Framework*. Verifiziert am 07.05.2006. 2003. – URL http://www.w3.org/TR/soap12/

[SSL 2000] THOMAS, Lehner: *Secure Socket Layer*. Verifiziert am 07.05.2006. 2000. – URL http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2000/Lehner/

[SSL-RFC 1999] DIERKS, T. ; ALLEN, C.: *The TLS Protocol Version 1.0*. Verifiziert am 07.05.2006. 1999. – URL http://www.rfc.net/rfc2246.html

[Stogov 2004] STOGOV, Dmitry: *Zend Technologies - PHP 5 In Depth - PHP SOAP Extension*. Verifiziert am 21.05.2006. 2004. – URL http://www.zend.com/php5/articles/php5-SOAP.php

[Tanenbaum und van Steen 2002] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002. – ISBN 0-13-088893-1

[Tomcat 2006] PMC, Apache T.: *Apache Tomcat*. Verifiziert am 07.05.2006. 2006. – URL http://tomcat.apache.org/

[UsernameToken 2006] DIVERSE: *Web Services Security UsernameToken Profile 1.1*. Verifiziert am 07.05.2006. 2006. – URL http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf

[Webmin 2006] CAMERON, Jamie: *Webmin*. Verifiziert am 07.05.2006. 2006. – URL http://www.webmin.com

[WebWork 2006] OPENSYMPHONY: *WebWork*. Verifiziert am 07.05.2006. 2006. – URL http://www.opensymphony.com/webwork/

[WS-Policy 2006] DIVERSE: *Web Services Policy Framework (WS-Policy)*. Verifiziert am 07.05.2006. 2006. – URL http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf

[WS-Security 2005] DIVERSE: *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*. Verifiziert am 07.05.2006. 2005. – URL http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-pr-SOAPMessageSecurity-01.pdf

[WS-Trust 2005] DIVERSE: *Web Services Trust Language (WS-Trust)*. Verifiziert am 07.05.2006. 2005. – URL http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf

[WSS4J 2006] FOUNDATION, Apache S.: *Apache WSS4J*. Verifiziert am 07.05.2006. 2006. – URL http://ws.apache.org/wss4j/index.html

[WWArchitektur 2006] OPENSYMPHONY: *WebWork - WebWork - Architecture*. Verifiziert am 18.05.2006. 2006. – URL http://www.opensymphony.com/webwork/wikidocs/Architecture.html

Glossar

ACID Atomicity (Atomarität), Consistency (Konsistenz), Isolation (Isolation), Durability (Dauerhaftigkeit). Wünschenswerte Eigenschaften von Transaktionen.

CORBA Common Object Request Broker Architecture. Sprachunabhängige, plattformübergreifende, objektorientierte Middleware.

Deployment Descriptor XML-Datei, durch die definiert wird, wie ein Webservice in Axis eingebunden wird

DNS Domain Name System. Hierarchisch aufgebautes System für die Vergabe von Rechnernamen.

FTP File Transfer Protocol. Protokoll zur Übertragung von Dateien über ein Netzwerk.

HTTP Hypertext Transfer Protocol. Dient zur Übertragung von Daten im World Wide Web.

Java RMI Java Remote Method Invocation. Objektorientierte Middleware für Java.

Middleware Zwischenschicht für die Kommunikation von verteilten Systemen.

MVC Model View Controller. Design Pattern für die Trennung von Daten, Benutzerschnittstelle und Logik.

SCP Secure Copy. Protokoll zur verschlüsselten Dateiübertragung über ein Netzwerk.

SOAP Simple Object Access Protocol. Auf XML basierendes Protokoll zum Austausch von Webservices-Nachrichten.

SSH Secure Shell. Protokoll zur verschlüsselten Datenübertragung über ein Netzwerk.

SSL Secure Socket Layer. Protokoll zur verschlüsselten Datenübertragung über ein Netzwerk.

WWW-Root Verzeichnis in dem ein Webserver die von ihm anzubietenden Dateien erwartet.

XML Extensible Markup Language. Metasprache zur Erstellung von Dokumenten.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Juli 2006	
Ort, Datum	Unterschrift