UNIVERSITÄT LEIPZIG

ASV Automatische Sprachverarbeitung

DIPLOMARBEIT

Performanceanalyse von SOAP- und REST- basierten Services in einer Linguistic Resources Umgebung

ausgeführt an der
Abteilung für Automatische Sprachverarbeitung
der Universität Leipzig

Betreuer:

Prof. Dr. Gerhard Heyer

bearbeitet durch

Sebastian Sander

Harnackstraße 3
04317 Leipzig

Leipzig, 8. Oktober 2010

Zusammenfassung

Im linguistsichen Umfeld ist die Verarbeitung von größeren Datenmengen mehrerer 1.000 MB keine Seltenheit, da hier sehr viele Textdokumente in elektronischer Form zur Verfügung stehen. Aufgrund dieses hohen Speicheraufkommens ist es nicht möglich, dass jeder Rechner die nötige Hardware bereitstellen kann, um Schriften oder Texte abzuspeichern. Außerdem benötigen diese meist nicht den gesamten Text, sondern nur einen gewissen Teil bzw. lediglich Metainformationen daraus. Daher ist es wichtig, solche Texte zentral auf einen Server zu hinterlegen und für bestimmte Client-Rechner zugänglich zu machen. Um dies zu ermöglichen, ist das Entwickeln von Service- bzw. Ressourcen-orientierten Architekturen notwendig. Diese müssen sowohl effektiv als auch effizient die durch den Client angefragten Daten zur Verfügung stellen. Daher ist eine genaue Kenntnis über die Performance von den existierenden Web Service Frameworks, der Techniken zur Verarbeitung von XML-Dokumenten und die Protokolle zur Übertragung dieser nötig um solche Architekturen zu entwickeln. In vielen Artikeln wird die Verarbeitung von XML-Dokumenten als Flaschenhals bezeichnet, da dieses sehr viel Zeit und Speicher in Anspruch nimmt. Zahlreiche Quellen, wie zum Beispiel (Sirinivas, 2006), zeigen die Performanceunterschiede zwischen SOAP und REST oder den verschiedenen Web Service Frameworks und XML-Schnittstellen. Diese untersuchen jedoch meist nur die Performance auf die Bewältigung von homogenen Elementen in einem Dokument und legen dabei nicht den Fokus auf die Verarbeitung von textbasierten Dokumenten.
Im Rahmen dieser Arbeit werden zunächst die Grundlagen, die Unterschiede und die Features der beiden Service Technologien, SOAP und REST, erarbeitet. Anschließend werden die Web Service Frameworks Apache Axis 1, Apache Axis 2 und Metro 2.0 genauer betrachtet. Der letzte Teil dieser Arbeit beschäftigt sich mit der Verarbeitung von XML-Dokumenten. Hier spielen die verschiedenen Techniken, XML-Dokumente mittels DOM und StAX zu erstellen eine primäre Rolle. Dabei werden verschiedene Referenzimplementierungen der Standards sowie Frameworks betrachtet. Zusätzlich wird untersucht wie sich die Generierung von XML im Inline- und Stand Off-Markup auf die Performance auswirkt. Dazu werden die Standards der Text Encoding Initiative zur Textkodierung und -formatierung hinzugezogen.
Am Schluß jedes Kapitels werden die Details zu den Durchführungen der Tests, deren Resultate und die daraus gewonnenen Schlussfolgerungen präsentiert. Dabei wird der Fokus auf die Linguistik gelegt.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AAR Axis Archive
ACK Acknowledgement
ADB Axis Data Binding
API Application Programming Interface
AVI Audio Video Interleave
Apache Axis Apache eXtensible Interaction System
AXIOM Axis Object Model
CLARIN Common Language Resources and Technology Infrastructure
CPU Central Processing Unit
DIME Direct Internet Message Encapsulation
DNS Domain Name System
DOM Document Object Model
DOM4J Document Object Model for Java
D-SPIN Deutsche Sprachressourcen-Infrastruktur
EJB Enterprise JavaBeans
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS HyperText Transfer Protocol Secure
IDL Interface Definition Language
IP Internetprotokoll
JAR Java Archive
Java EE 5 Java Platform Enterprise Edition 5
JAX-RPC Java API for XML-based RPC
JAXB Java Architecture for XML Binding
JAXP Java API for XML Processing
JAX-RS Java API for RESTful Web Services
JAX-WS Java API for XML Web Services
JDOM Java Document Object Model
JWS Java Web Service
MIME Multipurpose Internet Mail Extensions
MP3 MPEG-1 Audio Layer 3
MTOM Message Transmission Optimization Mechanism
NACK Negative Acknowledgement
OASIS Organization for the Advancement of Structured Information Standards

OMG Object Management Group
REST REpresantional State Transfer
ROA Resource Oriented Architecture
SAAJ SOAP with Attachments API for Java
SAX Simple API for XML
SGML Standard Generalized Markup Language
SJSXP Sun Java Streaming XML Parser
SOAP4J SOAP for Java
SSL Secure Socket Layers
StAX Streaming API for XML
StAX RI Streaming API for XML Reference Implementation
SVG Scalable Vector Graphics
SwA SOAP with Attachments
TEI Text Encoding Initiative
UDDI Universal Description, Discovery and Integration
URI Uniform Resource Identifier
URL Uniform Resource Locator
W3C World Wide Web Consortium
WADL Web Application Description Language
WAR Web Application Archive
WCF Windows Communication Foundation
WSDD Web Service Deployment Descriptor
WSDL Web Service Description Language
WSIT Web Services Interoperability Technology
XML Extensible Markup Language
XOM XML Object Model
XOP XML-binary Optimized Packaging
XSD XML Schema
XSLT Extensible Stylesheet Language Transformation

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Das Projekt eAQUA[1] entwickelte vor ca. sechs Jahren eine Architektur für öffentlich zugängliche, linguistische SOAP-Services[2]. Einige Services hatten dabei die Funktion, bestimmte Teile eines in einer Datenbank hinterlegten Textes bzw. Informationen darüber zurückzuliefern. Andere führten bspsw. komplizierte Berechnungen über verschiedene Wörter oder Sätze dieses Textes aus. Wiederum andere waren für das Hoch- bzw. Herunterladen von Schriften und Dokumenten verantwortlich.
Dabei zeichnet sich der bisherige Erstellungsprozess des SOAP-Kernels folgender maßen ab: ein Drittel bis die Hälfte aller Dateien werden automatisch generiert. Dazu existieren fertige Klassen auf dem Server, um die Services zu generieren. Für den Prozess der Erstellung der Services ist eine Datenbank auf dem Server installiert die Informationen, wie z.B. Name und Typ, zu den zu generierenden Services enthält. Der Typ des Service definiert sich durch seine Arbeitsweise. Ein Service vom Typ *Select* würde folglich eine einfache Abfrage auf der Datenbank ausführen und dem Client zurückliefern. Der Vorteil dieses Erstellungsprozesses ist, dass jeder Web Service nur ein einziger Eintrag in der Datenbank ist. Sind die Services schließlich generiert, werden die Quellcode-Dateien, die die Implementierung und die Logik der Services beinhalten, schließlich an die richtige Position kopiert und auf dem Server veröffentlicht. Für das Veröffentlichen der Services wird der Server derzeitig immer wieder neu gestartet, was ineffektiv ist. Darüber hinaus wird zusätzich ein Client für die erzeugten Services und ein Frontend generiert. Das Frontend bietet dann die Möglichkeit, über eine vorherige Anmeldung die generierten Services zu verwenden und zu testen. Da nicht jeder Client fähig ist SOAP-Nachrichten zu verarbeiten, soll der Kernel um die Funktionalität erweitert werden, sowohl SOAP basierte- als auch RESTful Services zur Verfügung zu stellen.
Die Funktionalität von RESTful Services soll verwendet werden, um einfache Datenbank-Lookups zu realisieren, bei denen keine weiteren Sicherheitskriterien erfüllt werden müssen. Des Weiteren basiert der SOAP-Kernel auf Apache Axis 1. Im Rahmen dieser Arbeit soll unter anderem analysiert werden, inwiefern sich durch die Neurerungen und Weiterentwicklungen der Web Service Frameworks beim vorliegenden System die Performance in der Verarbeitung von textbasierten Dokumenten steigern lässt.
Weiterhin enthalten linguistische Texte einen hohen Grad an Strukturinformationen. Die Untersuchung auf eine effiziente Einbettung dieser Informationen in den zu generieren Dokumenten sowie die verschiedenen Möglichkeiten zur Erstellung wird ebenfalls Gegenstand dieser Arbeit sein.

[1] Siehe http://www.eaqua.net/.

[2] Im weiteren Verlauf dieser Arbeit wird diese Architektur als SOAP-Kernel bezeichnet.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, Grundlagen für die Erweiterung um die Basisfunktionalitäten von REST-basierten Anfragen des bereits existierenden Kernel zu schaffen. Daraus leiten sich zwei weitere Absichten ab. Die theoretischen Untersuchungen von SOAP- und REST-basierten Services in einer linguistischen Umgebung sowie der Vergleich des Performancegewinns von einfachen SOAP- und REST-basierten Suchanfragen auf Texten in einer Datenbank.
Da der SOAP-Kernel auf Grundlage von Apache Axis 1 entwickelt wurde, soll außerdem evaluiert werden, ob der Kernel auf Basis anderer Frameworks, wie Apache Axis 2 und Metro 2.0, eine Leistungsteigerung erreichen kann. Hieraus ergeben sich weitere Ziele. Zum einen das Aufzeigen möglicher Performanceunterschiede der drei oben genannten Frameworks auf Basis eines linguistischen Referenz-Services. Zum anderen soll anhand dieser Unterschiede anschließend eine Entscheidungsgrundlage für eine etwaige Neuentwicklung des SOAP-Kernels, getroffen werden. Bei der Neuentwicklung des Kernels soll nicht auf zusätzliche REST-Frameworks wie z.B. Jersey oder Restlet zurückegriffen werden, da kein zusätzlicher Aufwand in die Einarbeitung eines weiteres Framework investiert werden soll. Deswegen ist ein Auswahlkriterium des neuen Frameworks die Unterstützung von RESTful Services. Daher wird zusätzlich die Performance von REST- und SOAP-basierten Services innerhalb des jeweiligen Frameworks untersucht.
Ein weiteres Ziel ist die Analyse, wie mit Hilfe von bestehenden Implementierungen von XML-Programmierschnittstellenspezifikationen Dokumente in Extensible Markup Language (XML) serverseitig möglichst effektiv generiert werden können. Zu diesem Zweck werden die am häufigsten genutzten Imlementierungen von DOM und StAX getestet. Linguistische Texte werden meist mit Zusatzinformationen wie z.B. Frequenz einzelner Wörter, Kookurenzen, Wortstamminformation und Strukturdaten der Texte übertragen. Daher ist es wichtig XML-Dokumente effizient zu formatieren. Aus dieser Tatsache leitet sich ein letztes Ziel dieser Arbeit ab. Die Untersuchung der Performance zweier Markup-Arten, wie Inline- und Standoff-Markup. Zum Schluss werden diese Markup-Arten auf die XML-Syntax TEI angewandt, um herauszufinden, welche die effektivere Generierungsmöglicheit ist linguistische Texte zu kodieren.

1.3 Arbeitsaufbau

Diese Diplomarbeit unterteilt sich in vier Hauptabschnitte:

- Untersuchung von SOAP und REST (Kapitel 2)
- Untersuchung ausgewählter Web Service Frameworks (Kapitel 3)
- Untersuchung ausgewählter XML-Programmierschnittstellen (Kapitel 4)
- Untersuchung von XML-Repräsentationen (Kapitel 5)

Im 2. Kapitel wird die Grundlage für die darauffolgenden Kapitel erarbeitet. Hier werden die Web Service Technologien SOAP und REST erläutert. Am Ende dieses Kapitels sind die Performancetests und die dazugehörige Auswertung der Tests der beiden Technologien, SOAP und REST, ersichtlich.
Das 3. Kapitel beschreibt die drei ausgewählten Web Service Frameworks in ihren Arbeitsweisen, Features sowie deren Vor- und Nachteile. Ebenso werden hier Tests zur Performance und eine Auswertung am Ende dieses Kapitels präsentiert.

Das 4. Kapitel beinhaltet die Programmierschnittstellen zu XML. Hier wird zuerst DOM und anschließend StAX in ihren Grundlagen erläutert und auf ihre Performance getestet. Dabei werden verschiedene Implementierungen der beiden Spezifikationen analysiert.
Im 5. Kapitel sollen die beiden Markup-Arten Inline und Stand-Off, genauer betrachtet und mittels Tests in Relation zu einander gesetzt werden. Den Schluss dieses Kapitels bildet eine Einleitung zu TEI und dem dazugehörigen Test, der die Performance zwischen Inline- und Stand-Off- Markup in TEI-Syntax vergleicht.
Das Ende dieser Arbeit ist ein Fazit mit einer kritische Betrachtung dieser.

1.4 Allgemeine und technische Rahmenbedingungen

Dieser Abschnitt gibt einen Überblick über die technischen Rahmenbdingungen wie Server, Netzwerk und Client. Die nachfolgende Abbildung verdeutlicht die Testumgebung.

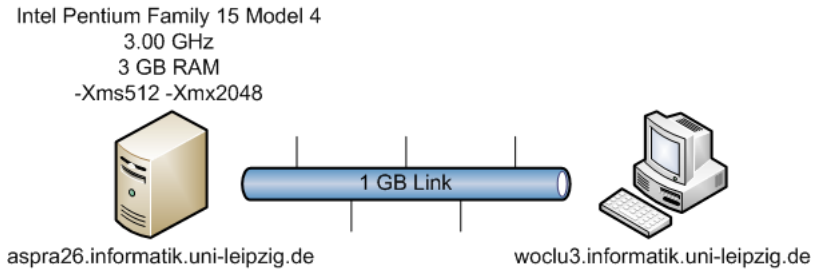

Abbildung 1.1: Die Testumgebung

Der Server, auf denen die Services veröffentlicht wurden, hat eine Central Processing Unit (CPU) von Intel Pentium mit 3 GHz und einen Arbeitsspeicher von 3GB. Auf diesem war ein Tomcat 6.0.26 als Webcontainer und Java 1.6.0.15 installiert. Als Datenbank wurde MySQL 5.1.45 verwendet. Der Tomcat Server wurde mit einem Minimalspeicher von 512 MB und einem Maximalspeicher von 2048 MB initialisiert. Den Client und den Server verband eine 1 GBit-Leitung und sie befanden sich im gleichen Netzwerk. Zum Zeitpunkt der Testdurchführung wurde darauf geachtet, dass keine anderen unnötigen Prozesse, die die Testergebnisse verfälschen könnten, starteten.
Für das Monitoring des Servers wurde JavaMelody[3] verwendet. Dies ist ein Tool zur Überwachung von Serveraktivitäten, das unter anderem die Auslastung der CPU, den Speicherverbrauch, die Anzahl der Anfragen pro Minute oder die Systemfehler misst. Die Integration von JavaMelody in die einzelnen Webanwendungen verlief ohne Probleme. Es mussten lediglich zwei Java Archive (JAR)-Dateien in die jeweilige Webanwendung eingebunden und über die Datei *web.xml* registriert werden. Dabei war es möglich, das Interval der Erzeugung der Messpunkte zu konfigurieren. Es wurden einige Versuche mit verschiedenen Intervallen durchgeführt. Das Resultat daraus war, dass ein minütliches Interval die Testergebnisse lediglich um 0.1% verfälscht. Dies schien geeignet und vernachlässigbar zu sein.
Zur Generierung der Testanfragen wurde JMeter 2.3.4[4] verwendet. JMeter ist ein Tool zur Generierung von Testplänen jeglicher Art. Für diese Arbeit wurde es verwendet, um eine gewisse Anzahl von Anfragen auf dem Server auszuführen. Es bietet außerdem die Möglich-

[3] Siehe (JavaMelody, 2010).
[4] Siehe (Project, 2010).

keit, mehrere Clients parallel Anfragen ausführen zu lassen und die Ergebnisse in Dateien zu speichern.
Anzumerken ist, dass alle Tests mehrfach und, wo dies möglich war, auf verschiedene Art und Weise durchgeführt wurden, um die Testergebnisse zu verifizieren.

Arten der Testdurchführung:

- Komplett:
  - Alle zusammengehörigen Tests wurden ohne Unterbrechung und Neustart des Servers durchgeführt.
  - Beispiel: Der SOAP-Kernel auf Basis von Apache Axis 1 wurde zuerst mit fünf verschiedenen Nachrichtengrößen(1-, 100- 1.000-, 10.000- und 100.000 Elementen), durchgeführt. Anschließend Apache Axis 2 mit den verschiednen Nachrichtengrößen und danach Metro 2.0. Dieser Testdurchlauf geschah nacheinander und ohne Neustart des Servers.
- in sich abgeschlossen:
  - Alle zusammengehörigen Tests wurden ohne Unterbrechung und Neustart des Servers durchgeführt.
  - Beispiel: Der SOAP-Kernel auf Basis von Apache Axis 1 wurde zuerst mit fünf verschiedenen Nachrichtengrößen durchgeführt. Nach Beendigung der Tests wurde der Server neu gestartet und der Test für den SOAP-Kernel auf Basis von Apache Axis 2 mit fünf verschiedenen Nachrichtengrößen ausgeführt.
- einzeln:
  - Die Test wurden für jede Nachrichtengröße einzeln durchgeführt.
  - Beispiel: Der SOAP-Kernel auf Basis von Apache Axis 1 wurde zuerst mit der Nachrichtengröße von einem Element durchgeführt. Nach Beendigung des Tests wurde der Server neu gestartet und der SOAP-Kernel auf Basis von Apache Axis 1 mit der Nachrichtengröße von 100 Elementen, usw., durchgeführt.

2 Web Service Techniken

2.1 Grundlagen

Mithilfe von Web Services ist es heute sehr leicht möglich, vorhandene Daten und Funktionen aus bestehenden Anwendungen zur Verfügung zu stellen und von anderen Anbietern abzurufen. Web Services werden als eine "Maschine zu Maschine"-Kommunikation gesehen, die über Standards und Protokolle Nachrichten austauschen. Der Nutzer eines Web Services braucht nicht zu wissen von woher er die Daten erhält und mit hilfe welcher Sprachen und Techniken die Funktionen realisiert wurden. Der Client benötigt lediglich eine Programmiersprache die Standards zur Verwendung von Web Services unterstützt.[1]
Zahlreiche Unternehmen und Organisationen haben versucht den Begriff Web Service zu definieren. Nachfolgend werden drei Definitionen präsentiert.

Gartner Group

> "A Web service is a custom end-to-end application that interoperates with other commercial and custom software through a family of XML interfaces (like SOAP, UDDI and WSDL) to perform useful business functions."[2]

Frankfurter Allgemeine Zeitung

> "Web Services sind Softwarebausteine, die Programme, die auf unterschiedlichen Netzwerkrechnern laufen, über das Internet zu einer Anwendung miteinander verknüpfen."[3]

IBM

> "Web services are self-describing, self-contained, modular applications that can be published, locatet, and invoked across the Web."[4]

Zusammenfassend kann gesagt werden, dass Web Services Programme oder Funktionen sind, die auf entfernten Rechnern liegen und öffentlich, über Intranet bzw. Internet, zugänglich sind. Dabei ist anzumerken, dass SOAP für die Realisierung von Web Services nicht zwingend notwendig ist. Eine weitere weit verbreitete Möglichkeit Services zu entwickeln

[1] Die Grundlage für diese Einleitung bildete (Gustavo Alonso, 2004) S. 123 - 148.
[2] Siehe http://www.gartner.com/DisplayDocument?doc_cd=116069.
[3] F.A.Z. vom 14. Oktober 2003, Seite 18.
[4] Siehe http://www.ibm.com/developerworks/wikis/.../WebServicesOverview.pdf?version=1, Seite 2.

ist REST. Diese beiden Technologien werden in den nachfolgenden Abschnitten dieses Kapitels ausführlich erläutert. Die grundlegende technische Architektur von Web Services ist ähnlich dem ISO/OSI-Referenzmodell[5] und besteht aus aufeinander aufbauenden Schichten:

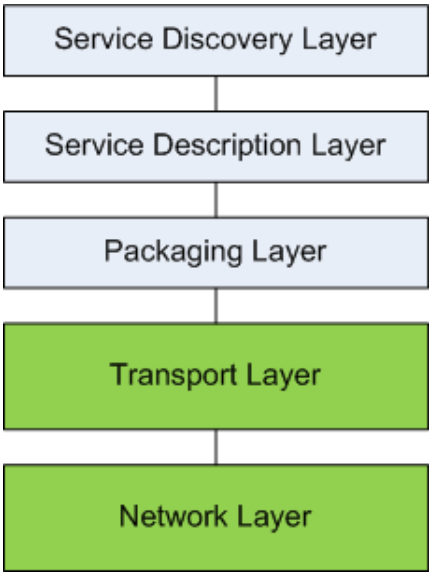

Abbildung 2.1: Web Service Stack

Wie aus Abbildung 2.1 hervorgeht, bildet die *Service Discovery Layer* die oberste Ebene des Web Service Stack. Sie ist ein Verzeichnisdienst, der mit den zur Verfügung stehenden Services vom Service Provider veröffentlicht wird. Somit können potentielle Anwender Informationen über Services einholen. Ein Beispiel für einen öffentlichen Verzeichnisdienst ist *Universal Description, Discovery and Integration (UDDI)*. Provider haben hier die Möglichkeit, ihre Web Services registrieren zu lassen. Anwender hingegen können sich über den Verzeichnisdienst Informationen zu den benötigten Web Services einholen. UDDI wurde von der Organization for the Advancement of Structured Information Standards (OASIS) [6] standardisiert und liegt zur zeit in der Version 3.0 vor.[7] Die Verwendung von UDDI bzw. eines Verzeichnisdienstes ist nicht zwingend notwendig. Jedoch reduziert solch ein Dienst den Verwaltungsaufwand einer hohen Anzahl von Services und sollte bei der Konzeption eines verteilten Systems mit in Betracht gezogen werden.
Im *Description Layer* werden die in den Web Services enthaltenen Methoden vom Serviceprovider beschrieben, damit der Konsument diese dann aufrufen kann. Für die Beschreibung eines Services werden die Sprachenstandards *Web Service Description Language (WSDL)* oder *Web Application Description Language (WADL)* genutzt.
Im *Packaging Layer* wird das Protokoll definiert, das verwendet wird, um die Daten auszutauschen[8].
Die beiden untersten Ebenen, *Transport-* und *Network- Layer*, sind elementar und setzen sich aus den bekannten Schichten des ISO/OSI Referenzmodells zusammen, das in der nachfolgenden Abbildung aufgezeigt wird.

[5] Vgl. http://www.cs.berkeley.edu/ istoica/classes/cs194/05/notes/2-NetRPC.pdf.
[6] Siehe http://www.oasis-open.org/committees/ciq/ciq.html
[7] Vgl. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm.
[8] Vgl. (Dörnemann, 2008).

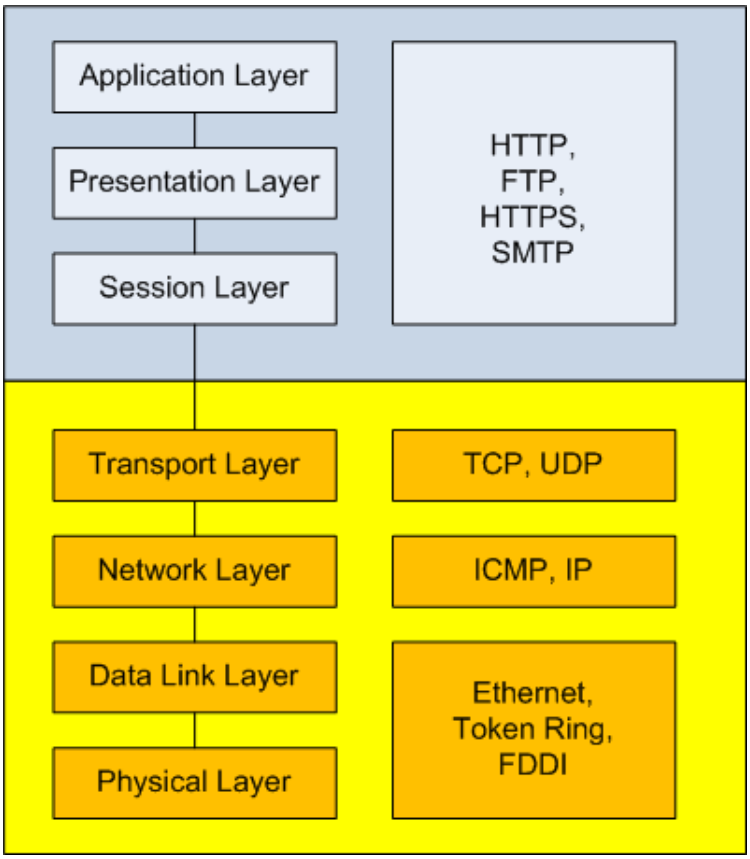

Abbildung 2.2: OSI/ISO Referenzmodell

Der Transport der Daten kann dabei über verschiedene Protokolle, die von den jeweiligen Web Service Technologien unterstützt werden, erfolgen. Welche Protokolle dabei genau genutzt werden können, wird bei SOAP in der WSDL angegeben, wohingegen bei REST zum Großteil Hypertext Transfer Protocol (HTTP) zum Einsatz kommt. Die Transportebene in Abbildung 2.1 entspricht hier den ersten drei Schichten des ISO/OSI Referenzmodells. Somit wird deutlich, dass sowohl REST als auch SOAP in der Anwendungsebene bzw. in der Präsentationsebene des Referenzmodells anzusiedeln sind. Die beiden Web Service Technologien benötigen die Protokolle der anwendungsorientierten Schichten des Modells, um zu kommunizieren.
Die Netzwerkschicht in Abbildung 2.1 bildet dabei den physikalischen Übertragungsweg zwischen Servicekonsument und Servicesprovider. Sie ist eine Zusammenfassung der vier unteren Schichten des Referenzmodells. Die Kommunikation erfolgt in der Regel über TCP/IP.

2.2 Einführung SOAP

Das standardisierte Kommunikationsprotokoll SOAP findet seine Anwendung, um Daten zwischen Systemen auszutauschen und Remote Procedure Calls durchzuführen[9]. Die Spezifikation beschreibt vor allem den Aufbau und das Format der Nachrichten, die bei einer Kommunikation zwischen Servicekonsument und Serviceproduzent verschickt werden. Der Entwurf von SOAP wurde im Jahr 2000 dem World Wide Web Consortium (W3C) von einer

[9] Vgl. (Gustavo Alonso, 2004) S. 152.

Arbeitsgruppe der Firmen DevelopMentor, IBM, Lotus, Microsoft und Userland angeboten. Im Jahr 2003 hat das W3C in vier Teilen eine Spezifikation von SOAP in der Version 1.2 veröffentlicht. In diesem Abschnitt werden zunächst die wichtigsten Aspekte der SOAP-Spezifkation herausgearbeitet. [10]

2.2.1 Spezifikation von SOAP

Die Spezifikation von SOAP unterteilt sich in vier Dokumente, die in der nachfolgenden Tabelle aufgezeigt werden. Diese bietet eine Übersicht zu den einzelnen Dokumenten der Spezifikation, die anschließend kurz erläutert werden[11]. Die Spezifikation definiert eine XML-Grammatik, in der Informationen in einer Nachricht zusammengefasst werden.

Dokument	Inhalt
SOAP Version 1.2 Part 0: Primer	Tutorial zu SOAP
SOAP Version 1.2 Part 1: Messaging Framework	Kernspezfikation von SOAP
SOAP Version 1.2 Part 2: Adjuncts	HTML-Dokument, MPEG-1 Audio Layer 3 (MP3)
Specification Assertions and Test Collection	Media Type, Last-Modified Time

Tabelle 2.1: SOAP-Dokumente für die Spezifikation

Das erste Dokument ist *SOAP Version 1.2 Part 0: Primer*. Dies ist nicht normiert, sondern stellt ein Tutorial dar, das die Features von SOAP und seine Arbeitsweise beschreibt. Hier werden unter anderem Themen besprochen, die aufzeigen, wie der Nachrichtenaustausch zwischen zwei Parteien verläuft oder wie verschiedene Protokollbindings realisiert werden. Die einzelnen Punkte werden hier immer wieder anhand von Szenarios erklärt.
SOAP Version 1.2 Part 1: Messaging Framework ist die Hauptspezifikation zu SOAP. Sie besteht aus vier weiteren Teilen. Das *SOAP-Processing Model* definiert die Regeln, wie SOAP-Nachrichten zwischen Client und Server ausgetauscht und verarbeitet werden. *SOAP Extensible Model* beschreibt, wie Erweiterungen des Frameworks vorgenommern werden können und legt zusätzlich fest, welche Anforderungen zu stellen sind. Ein weiteres Framework der Spezifikation Messaging Framework ist *SOAP Protocol Binding Framework*. Darin werden die zu verwendenden Protokolle für den Nachrichtenaustausch über SOAP definiert. Das letzte Dokument des zweiten Teils der SOAP Spezifikation ist *SOAP Message Construct*, dass die Informationseinheiten einer SOAP-Nachricht festlegt.
SOAP Version 1.2 Part 2: Adjuncts baut auf dem Messaging Framework auf und definiert Details des Datenmodells für SOAP und die Kodierung.
Das letzte, der hier vorgestellten Dokumente, ist *Specifikation Assertions and Test Collection*, welches als Hauptziel die Förderung der Interoperabilität zwischen verschiedenen SOAP 1.2 Implementierungen hat.
In den nachfolgenden Kapiteln werden die kurz erläuterten Spezifikationen im Detail behandelt.

[10] Die Grundlage für dieses Kapitel bildete (Gustavo Alonso, 2004) S. 151 - 230.
[11] Vgl. (W3C, 2007)

2.2.2 Aufbau von SOAP-Nachrichten

Eine SOAP Nachricht ist ein XML-Dokument mit einem *SOAP-Envelope* als Wurzelelement. In Abbildung 2.3 ist eine vereinfachte SOAP-Nachricht dargestellt. Dieser sogenannte „Umschlag“ beinhaltet wiederum zwei weitere Elemente, den *Header* und genau ein *Body*-Element und bildet somit das Wurzelelement. Der Header ist optional und kann weitere Elemente mit Metainformationen über Verschlüsselung, Vorschriften für Authentisierung, Caching oder Routing enthalten. Sobald ein Header-Element existiert, ist es das erste Kindelement des Umschlags. Jedes Header-Element wird auch als *Header Block* bezeichnet, die mit den Attributen *mustUnderstand* und *actor* versehen werden können. Wenn SOAP-Nachrichten ein Header-Element mit dem Attribut *mustUnderstand* beinhalten, muss der jeweilige Client, für den diese Nachricht bestimmt ist, dieses Element verarbeiten können. Ist dies nicht der Fall, wird ein Fehler zurückgeliefert.
Actor enthält den Namen eines SOAP-Intermediärs. Der SOAP-Intermediär sitzt innerhalb des Nachrichtenpfades und kann die SOAP-Nachricht teilweise bearbeiten. Das SOAP-Body Element enthält die eigentliche Nachricht, die verarbeitet werden muss. Alles, was in XML Syntax dargestellt werden kann, kann auch in das SOAP-Body Element geschrieben werden.[12]

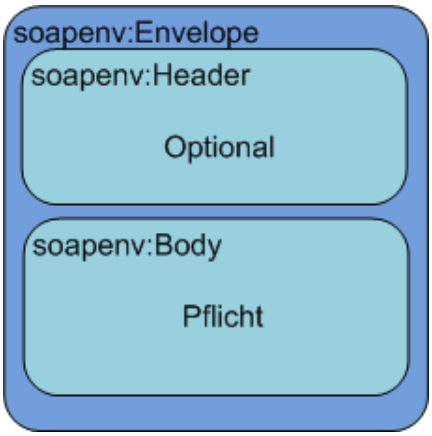

Abbildung 2.3: Struktur von SOAP-Nachrichten

Im Anhang B sind Beispiele einer Komminikation zwischen einem Client und einem Server gegeben. Darin stellt der Serviceclient eine Suchanfrage an den Provider. Er soll den Satz mit der Id 1 zurückliefern. Der Service-Provider erstellt eine Datenbankabfrage über den gewünschten Satz. Die Schnittstelle des Services bietet eine Methode an, die es erlaubt, anhand einer Start- und einer End-Id den aktuellen Datenbestand nach der Id der Sätze zu durchsuchen. Als Ergebnis erhält der Client alle Sätze, die zwischen der Start- und der End-Id liegen. Sie befinden sich im Body- Element des Antwortdokumentes.

2.2.3 SOAP Faults

Ein *SOAP-Fault* ist eine spezielle Art einer SOAP-Nachricht, die Informationen über Fehler beinhaltet, die während der Verarbeitung aufgetreten sind. Sie kann genau einmal im SOAP-Body enthalten sein und gilt als ein spezielles SOAP-Body Element und muss, falls

[12] Vgl. (James Snell, 2002) Seite 11 - 19.

vorhanden, das einzige Element im SOAP-Body sowie das direkte Kindelement vom SOAP-Body-Element sein. In dem nachfolgenden Listing ist ein Beispiel einer Fehlernachricht von SOAP aufgezeigt.
Der *Fault Code* ist ein algorithmisch generierter Wert, um den Fehler zu identifizieren. Dieser enthält einen der Werte:

VersionMismatch: Der SOAP-Envelope verwendet einen ungültigen Namespace.

MustUnderstand: Ein Element im SOAP-Header mustUnderstand hatte den Wert *true*, doch der Nachrichtenempfänger konnte diese Element nicht verarbeiten.

Client/Receiver: Bei der Formatierung der Nachricht trat ein Fehler auf oder die enthaltenen Daten waren fehlerhaft.

Server/Sender: Ein Fehler trat auf der Seite des Serviceproduzenten auf.

DataEncodingUnknown: Das angegebene SOAP-Encoding wird nicht unterstützt.

Der *Fault String* ist dabei eine für den Menschen lesbare Nachricht, die den Fehler beschreibt. Der *Fault Actor* gibt den Knoten an, wo der Fehler auftrat. Im Element *Fault Details* stehen anwendungsspezifische Details zum Fehlerhergang. [13]

```
<?xml version='1.0' encoding='UTF-8'?>
  <soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/
     soap-envelope">
    <soapenv:Body>
      <soapenv:Fault>
        <soapenv:Code>
          <soapenv:Value>soapenv:Receiver</soapenv:Value>
        </soapenv:Code>
        <soapenv:Reason>
          <soapenv:Text xml:lang="en-US">
            Exception occurred while trying to invoke service
                method getSentences
          </soapenv:Text>
        </soapenv:Reason>
        <soapenv:Detail />
      </soapenv:Fault>
    </soapenv:Body>
  </soapenv:Envelope>
```

Listing 2.1: Beispiel einer SOAP Fehlernachricht

2.2.4 SOAP-Datentypen und -Encoding

Das SOAP-Encoding beschreibt, unter Beachtung einiger Regeln, wie Werte in XML kodiert werden sollen. Die Konvertierung eines Wertes in XML wird dabei *Serialisierung* genannt.

[13] Vgl. (James Snell, 2002) Seite 19 - 22.

Die Kodierung, der im Body-Element einer SOAP-Nachricht übertragenen Daten, kann mit dem Attribut *encodingStyle* für jedes Body-Element festgelegt werden. EncodingStyle bezeichnet dabei die Standardkodierung, die verwendet wird, falls die einzelnen Bodyelemente keine eigene Kodierung haben. Es drückt außerdem aus, wie die Datentypen in einem SOAP-Envelope dargestellt werden.
Das Typsystem von SOAP unterscheidet zwischen einfachen und komplexen Datentypen. Dabei entsprechen in der SOAP-Spezifikation die einzelnen Datentypen denen des XML-Schemas. In der nachfolgenden Tabelle ist eine Übersicht der einfachen Datentypen von XML-Schema und Java ersichtlich.

Datentyp	XSD	Java
Zeichenketten	xsd:string	java.lang.String
Ganzzahlwerte	xsd:integer	java.math.BigInteger
Ganzzahlwerte	xsd:int	int
Ganzzahlwerte	xsd.long	long
Ganzzahlwerte	xsd:short	short
Dezimalzahlwerte	xsd:decimal	java.math.BigDecimal
Gleitkommazahlwerte	xsd:float	float
Gleitkommazahlwerte	xsd:double	double
Wahrheitswerte	xsd:boolean	boolean
Ganzzahlwerte	xsd:byte	byte
XML qualifizierte Namen	xsd:QName	javax.xml.namespace.QName
Zeitverwaltung	xsd:dateTime	javax.xml.datatype .XMLGregorianCalendar
Array aus Bytes	xsd:base64Binary	byte[]
Array aus Bytes	xsd:hexBinary	byte[]
Ganzzahlwerte	xsd:unsignedInt	long
Ganzzahlwerte	xsd:unsignedShort	int
Ganzzahlwerte	xsd:unsignedByte	short
Zeitverwaltung	xsd:time	javax.xml.datatype .XMLGregorianCalendar
Zeitverwaltung	xsd:date	javax.xml.datatype .XMLGregorianCalendar
Zeichenkette	xsd:anySimpleType	java.lang.String
Zeitverwaltung	xsd:duration	javax.xml.datatype.Duration
Namespace	xsd:NOTATION	javax.xml.namespace.QName

Tabelle 2.2: einfache Datentypen in SOAP

Diese Datentypen und von ihnen abgeleitete Typen dürfen direkt im Element verwendet werden. Der Typ eines Datenelements kann in einer SOAP-Nachricht auch explizit mit *xsi:type* angegeben werden.
Neben den einfachen Datentypen unterstützt SOAP auch die zusammengesetzten. Zu diesen gehören *structs*. Sie entsprechen den aus den Programmiersprachen bekannten Strukturen oder Records. Dabei dienen die Namen der Zugriffselemente als einzige Unterscheidung zwischen den Elementen. Ein weiterer zusammengesetzter Datentyp in SOAP sind die Felder. Ihre Entsprechung zu den Programmiersprachen werden in den Arrays und Listen gefunden. Hierbei erfolgt die einzige Unterscheidung der Elemente über die Position des

einzelnen Elements, da sie alle den gleichen Namen besitzen. Ein dritter, komplexer Typ in SOAP sind die Mischformen aus den Strukturen und Feldern. Dabei können Elemente, wie auch in Strukturen über den Namen referenziert werden, aber auch mehrfach auftauchen, wie in Feldern. Die zusammengesetzten Datentypen können wiederrum ineinander verschachtelt werden. [14]

2.2.5 Nachrichtenaustausch in SOAP

Wie im ersten Teil dieses Kapitels bereits erwähnt, befindet sich SOAP über der Transport- und Netzwerk-Ebene im ISO/OSI-Referenzmodell. Bei der Übertragung einer Nachricht kann sie mehrere *intermediaries* passieren. Somit enwickelt sich ein Nachrichtenpfad. Dabei wird ein SOAP-Intermediär als ein Knoten bezeichnet, der sich zwischen Servicekonsument und -produzent befindet und die Nachricht weiter bearbeitet. Jeder dieser Intermediäre auf einem Nachrichtenpfad wird als *actor* bezeichnet. Informationen, die nicht für den endgültigen Empfänger bestimmt sind, finden sich im SOAP-Header, gekennzeichnet durch das Attribut *actor*, wieder. Die SOAP-Spezifikation sieht nicht vor, welchen Pfad eine Nachricht zu nehmen hat. Es kann lediglich festgelegt werden, welche Informationen für welche Empfänger bestimmt sind. Um dieses Problem zu umgehen, schlägt unter anderem Microsoft mit dem "Web Service Routing Protocol" eine Lösung vor. Dieses Protokoll wird in dem Web Service Standard *WS-Routing* [15] definiert. Dies ist ein SOAP-basiertes zustandsloses Protokoll für den Austausch von SOAP-Nachrichten von einem anfänglichen Sender zu einem endgültigen Empfänger. Dazu generiert der Sender einen Routingpfad im Header, der den Pfad anzeigt. Der Pfad kann einen oder mehrere SOAP-Intermediäre beinhalten. Diese werden im *via*-Element als Subelement des *fwd*-Elements angegeben. Der entgültige Sender ist dann das letzte *via*-Element. [16]

2.2.6 WSDL

Da Web Services vom Service Provider beschrieben werden müssen, wird dazu WSDL verwendet. Es basiert auf XML und ist plattformunabhängig. Mit hilfe von WSDL werden die veröffentlichten Funktionen, Daten, Datentypen und Austauschprotokolle eines Web Services beschrieben. Im Anhang A sind automatisch erzeugte, komplette WSDL-Dateien der im nächsten Kapitel untersuchten Web Service Frameworks aufgezeigt. Im Folgenden werden die Elemente einer Dienstbeschreibung kurz erläutert.
Das Element *message* beinhaltet abstrakte Definitionen der Daten, die zwischen Konsument und Provider des Services ausgetauscht werden. Hier wird eine Auflistung der Methoden, die der Web Service veröffentlicht, die dazugehörigen Eingangsparameter sowie das Format der zurückgegebenen Ergebnismenge angegeben.

```
<wsdl:message name="getSentencesResponse">
    <wsdl:part name="parameters"
               element="ns:getSentencesResponse" />
</wsdl:message>
```

Listing 2.2: Beispiel eines message Elementes

[14] Vgl. (James Snell, 2002) Seite 16 - 33 und (Vonhoegen, 2007) Seite 122 - 136.
[15] Siehe http://docs.openlinksw.com/virtuoso/ws-routing.html/.
[16] Vgl. (James Snell, 2002) Seite 22 - 24.

Im WSDL-Element *types* werden die Datentypen definiert, die in den Nachrichten ausgetauscht werden. Das Typsystem kann verwendet werden, um in einer Nachricht Typen zu definieren. Ob das Format tatsächlich XML ist bzw. ob das XML Schema (XSD) das spezielle Format validiert braucht nicht berücksichtigt werden.. Dies erhält seine Wichtigkeit, wenn für die gleiche Nachricht mehrere Bindings definiert werden oder wenn nur ein Binding existiert, aber der Bindingtyp kein eigenes Typsystem hat.
Im Element *portType* wird die Serviceschnittstelle und die Serviceoperationen festgelegt. Somit wird hier gesagt, über welchen Port bestimmte Methoden des Services vom Client aufgerufen und die Antworten des Providers gesendet werden. Wie im folgenden Listing zu sehen ist, referenziert jede in portType enthaltene Operation auf eine Eingangs- und eine Ausgangsnachricht.

```
<wsdl:portType name="SentencesPortType">
    <wsdl:operation name="getSentences">
      <wsdl:input message="ns:getSentencesRequest"
                  wsaw:Action="urn:getSentences"/>
      <wsdl:output message="ns:getSentencesResponse"
                   wsaw:Action="urn:getSentencesResponse"/>
      <wsdl:fault message="ns:RemoteException"
                  name="RemoteException"
                  wsaw:Action=
                   "urn:getSentencesRemoteException"/>
    </wsdl:operation>
</wsdl:portType>
```

Listing 2.3: Beispiel eines portType Elementes

Die Elemente des WSDL-Dokumentes werden in zwei Gruppen untergliedert. Die Elemente *messages*, *types* und *portTypes* gehören zu der Gruppe der abstrakten Definitionen.
Zu der Gruppe der konktreten Definitionen gehören die im folgenden erläuterten Elemente: Das Element *binding* spezifiziert das konkret eingesetzte Protokoll und das Datenformat für die Operationen, die im Element*portType* definiert wurden.

```
<wsdl:binding name="SentencesSoap11Binding"
              type="ns:SentencesPortType">
    <soap:binding transport=
                "http://schemas.xmlsoap.org/soap/http"
                 style="document"/>
    <wsdl:operation name="getSentences">
      <soap:operation soapAction="urn:getSentences"
                      style="document"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
      <wsdl:fault name="RemoteException">
        <soap:fault use="literal"
```

```
                    name="RemoteException"/>
      </wsdl:fault>
      </wsdl:operation>
</wsdl:binding>
```

Listing 2.4: Beispiel eines portType Elementes

Das Element *service* fast die Menge der dazugehörigen Ports zusammen. Dieses Element beinhaltet Elemente vom Typ *port*, in welchem die Adresse des Serviceproviders angegeben wird, unter der der Web Service erreichbar ist. Diese Adresse wird für ein Binding angegeben. Dabei ist zu beachten, dass das Element *port* nicht mehr als eine Adresse beinhalten darf.

```
<wsdl:service name="Sentences">
    <wsdl:port name="SentencesHttpSoap11Endpoint"
               binding="ns:SentencesSoap11Binding">
      <soap:address location="http://aspra26.informatik.uni-
          leipzig.de:8080/WebServicesKernelAxis2/services/
          Sentences.SentencesHttpSoap11Endpoint/" />
    </wsdl:port>
    <wsdl:port name="SentencesHttpSoap12Endpoint"
               binding="ns:SentencesSoap12Binding">
      <soap12:address location="http://aspra26.informatik.uni
          -leipzig.de:8080/WebServicesKernelAxis2/services/
          Sentences.SentencesHttpSoap12Endpoint/" />
    </wsdl:port>
    <wsdl:port name="SentencesHttpEndpoint"
               binding="ns:SentencesHttpBinding">
    <http:address location="http://aspra26.informatik.uni-
        leipzig.de:8080/WebServicesKernelAxis2/services/
        Sentences.SentencesHttpEndpoint/" />
    </wsdl:port>
  </wsdl:service>
```

Listing 2.5: Beispiel eines portType Elementes

In diesem Beispiel werden drei Ports beschrieben. Der Erste veröffentlicht einen SOAP-Version 1.1-Endpunkt, der Zweite einen SOAP-Version 1.2-Endpunkt und der Dritte einen HTTP-Endpunkt. [17]

2.2.7 Erweiterungen von SOAP Services

Sicherheit

Für die Einhaltung der Identität, Integrität und Vertraulichkeit von Nachrichten kommt bei SOAP der Standard *WS-Security*[18] zum Einsatz. Er wurde ursprünglich durch IBM, Microsoft und VeriSign entwickelt. Heute wird er durch OASIS erweitert.
WS-Security verwendet für die Realisierung von Sicherheit bereits bestehende Standards. Dabei kommen *Kerberos* und *X.509* für die Authentifizierung, *XML Encryption* und *XML*

[17] Vgl. (Erik Christensen, 2001).
[18] Vgl. (IBM, 2004) und (Tobias Hauser, 2004).

Signature[19] für die Verschlüsselung und das Signieren von Inhalt von XML Nachrichten sowie *XML Canonicalization* für den Prozess der Verschlüsselung zum Einsatz. Deshalb muss keine vollständig neue Sicherheitslösung definiert werden.
Der Standard selbst definiert ein SOAP-Header-Element, in dem sich sicherheitsrelevante Daten befinden. Somit gibt WS-Security lediglich an, wie die von den anderen Spezifkationen festgelegten Sicherheitsinformationen in einer SOAP Nachricht eingebettet werden können.

Zuverlässige Nachrichtenübertragung

Für die korrekte Übertragung und die richtige Reihenfolge von übertragenen Nachrichten kommt in SOAP der Standard *WS-ReliableMessaging* zum Einsatz. Er wurde von Microsoft, IBM, TIBCO und BEA entwickelt. Dabei werden Nachrichten zu Sequenzen zusammengefasst. Nach der Erstellung einer solchen Sequenz werden diese versendet. Die Sequenznummer wird bei der Übertagung im SOAP-Header mitgeschickt. Zusätzlich wird jeder Nachricht intern eine aufsteigende Nummer durch den Sender zugewiesen. Der Empfänger bestätigt nun den Erhalt der Nachricht, durch das Zurückschicken der Nummern aller empfangenen Nachrichten. WS-ReliableMessaging definiert vier Arten der Zusicherung der Nachrichtenübertragung:

AtMostOnce: Nachrichten werden höchstens einmal ohne Duplikate ausgeliefert. Werden dennoch Duplikate geschickt, wird der Endpunkt, welcher die doppelte Nachricht erhalten hat, einen Fehler ausgeben. Es ist jedoch möglich, dass Nachrichten innerhalb einer Sequenz nicht ausgeliefert werden.

AtLeastOnce: jede Nachricht, die gesendet wird, wird auch ausgeliefert, sonst wird ein Fehler erzeugt. Duplikate werden nicht weiter behandelt, da hier die korrekte Auslieferung mindestens einer Nachricht zu gesichert wird.

ExactlyOnce: Jede Nachricht wird genau einmal gesendet.

InOrder: Bestimmt die Auslieferung in korrekter Reihenfolge. Diese Einstellung kann mit den drei vorherigen kombiniert werden, da sie nichts über das Verhalten eines Endpunktes bei Verlust oder bei Duplizierung von Nachrichten aussagt.

Die Kontrolle über die Strategiecodes AtMostOnce, AtLeastOnce, ExactlyOnce oder InOrder liegt bei dem Sender. Dieser idenfiziert Erfolge und Misserfolge in der Übertragung durch die vom Empfänger geschickten *Acknowledgement (ACK)*, welche die Sequenznummer der gerade empfangenen Nachricht beinhaltet.
Ein weiterer Standard, der sich mit der sicheren Nachrichtenübertragung befasst, ist *WS-Reliability*[20]. Er wurde von Oracle und Sun entwickelt. Beide Standards haben die gleichen Ziele. Allerdings sind sie nicht miteinander kompatibel. Der Unterschied zu diesen beiden Standards ist, dass WS-Reliability nicht, wie WS-ReliableMessaging auf *WS-Adressing*[21] aufbaut.
WS-Adressing erlaubt Services den Austausch von Adressinformationen. Dabei definiert es mit *Endpointreferenzen* und *speziellen Headerinformationen* zwei Konstrukte. Endpunktreferenzen definieren dabei die Informationen, die für die Adressierung einer Nachricht notwendig sind. Die speziellen Headerinformationen erweitern den ursprünglichen Nachrichten

[19] Siehe (Tobias Hauser, 2004) S. 137 - 146.
[20] Vgl. (Davis, 2005).
[21] Vgl. (Francisco Curbera, 2004).

Header um weitere Informationen. Als Beispiel ist hier zu nennen, dass die Adresse des Empfängers angegeben oder eine Nachrichten-Identifikationsnummer übermittelt werden kann.
Ein weiterer Unterschied ist, dass WS-Reliability die Möglichkeit bietet Regeln für die Zusicherung der Nachrichtenauslieferung zu definieren. Somit kann zum Beispiel angegeben werden, dass Nachrichten *in-order* versendet werden und Duplikate ignoriert werden sollen. In WS-ReliableMessaging ist dies zwar auch möglich, jedoch nur durch Hinzunahme von WS-Policy[22] als einen weiteren Web Service Standard. Außerdem kann das Ziel für die *ACKs gepollt* werden. Das ermöglicht dem Sender, der sich hinter einer Firewall befindet, trotzdem ACKs zu empfangen. In WS-ReliableMessaging ist es nicht möglich, asynchrone ACKs zu empfangen. Hier muss der Sender das ACK zurück zu dem HTTP-Response-Flow oder zu einem Endpunkt außerhalb der Firewall senden. In WS-ReliableMessaging ist es wiederum möglich, *Negative Acknowledgement (NACK)* zu verschicken. Diese erlauben es dem Client dem Server mitzuteilen, welche spezifische Nachricht erneut gesendet werden muss.
Die letzte Unterscheidung dieser beiden Standards für sichere Nachrichtenübertragung ist, dass WS-Reliability ein OASIS-Standard ist und WS-RelaibleMessaging von keiner Organisation standardisiert wurde. [23]

Transaktionen

Die Spezifikation *WS-Transaction* beschreibt einen Mechanismus, der es erlaubt, Operationen entweder vollständig oder gar nicht auszuführen. Sie besteht aus drei Unterspezifikationen.

WS-Coordination stellt Protokolle zur Verfügung, die es erlauben Aktionen von Anwendungen zu koordinieren. Außerdem schreibt WS-Coordination vor, dass ein Koordinator existiert, bei dem sich die Web Services registrieren.

WS-Atomic Transaction ist eine Erweiterung von WS-Coordination und definiert drei Vereinbarungsprotokolle, *completion, volatile two-phase commit* und *durable two-phase commit*, für atomare Transaktionen. Er ist für kurze Prozesse vorgesehen.

WS-BusinessActivity definiert zwei Protokolle: *BusinessAgreementWithParticipantCompletion* und *BusinessAgreementWithCoordinatorCompletion*. Diese Spezifikation ist für länger andauernde Geschäftsprozesse vorgesehen. [24]

Nachrichtenoptimierung

Wenn Anwendungen über das Internet Nachrichten austauschen, können diese Daten aus den verschiedensten Formaten bestehen und große Mengen an binären Nutzdaten beinhalten. Wenn große binäre Objekte in XML kodiert werden, um diese dann in einer SOAP-Nachricht zu verpacken, werden die zu übertragenden Daten immer größer. Dies hat negative Auswirkungen auf die Performance der Web Service Anwendungen und der Netzwerklast.
Nachrichtenoptimierung garantiert eine effiziente Übertragung von Nachrichten über das Internet und versetzt einen Serviceendpunkt in die Lage große binäre Nutzdaten zu erkennen,

[22] Siehe (Schlimmer, 2006).
[23] Vgl. (Iwasa, 2004) und (Davis, 2005).
[24] Vgl. (Feingold, 2005) und (IBM, 20004)

diese aus dem SOAP-Body zu entfernen, die Nutzdaten mit einem effektiveren Kodierungsmechanismus zu kodieren und wieder in die SOAP-Nachricht als Attachment hinzuzufügen. Zwei dieser Mechanismen sind:

SOAP with Attachments (SwA): ist ein Vorchlag der W3C für den Transport von SOAP-Nachrichten mit Attachments innerhalb von Multipurpose Internet Mail Extensions (MIME) bzw. Direct Internet Message Encapsulation (DIME). In SwA besteht eine SOAP-Nachricht aus zwei Teilen. Dem primären Part SOAP-Envelope und dem sekundären Part Attachment. Der Attachment Part kann über eine eindeutige *Content-ID* von der SOAP Nachricht referenziert werden.[25]

Message Transmission Optimization Mechanism (MTOM): ist ein Vertrag zwischen zwei direkt benachbarten SOAP-Knoten in einem Nachrichtenkanal. Das Ziel von MTOM ist es, die Übertragung von in *Base64*-kodierten Binärdaten zu optimieren. Die Übertragung einer Nachricht wird nun optimiert, indem Teile der Nachricht, die vom Typ *xsd:base64Binary* sind, während der Übertragung effizienter kodiert werden. Als Serialisierungsformat wird *XML-binary Optimized Packaging (XOP)* verwendet. XOP stellt dabei ein Verfahren dar, was es ermöglicht, den Inhalt bestimmter Datentypen effizient zu verpacken. Bei der XOP-Verarbeitung wird ein XML-Infoset zuerst in ein erweiterbares Format verpackt und anschließend werden die Teile, die in Base64-Kodierung vorliegen, extrahiert, wieder kodiert und im Paket separat abgelegt. Die Teile, die extrahiert wurden, werden mit einem speziellen Element markiert, dass über einen Uniform Resource Identifier (URI) auf den ausgelagerten Teil im Paket verweist[26].

2.3 Einführung REST

REpresantional State Transfer (REST) wurde von Roy Thomas Fielding im Rahmen seiner Dissertation (Fielding, 2000) im Jahr 2000 veröffentlicht. REST wird als ein netzwerkorientierter Architekturstil eingeführt, mit dem Ziel die Latenz und die Kommunikation über das Netzwerk zu minimieren und gleichzeitig die Unabhängigkeit und Skalierbarkeit der Komponenten zu erhöhen.[27] Die Features von REST sind Caching und die Wiederverwendbarkeit von Interaktionen, die Ersetzbarkeit von Komponenten und die Verarbeitung von Aktionen durch Zwischenstationen wie Gateways, Firewalls und Proxyserver. Dadurch werden die Anforderung an ein verteiltes Hypermedia-System erfüllt.
Fielding betrachtet das moderne Web als eine Instanz des REST Architekturstils. Dabei ist jedoch zu beachten, dass REST keine spezifische Technologie darstellt und nicht an HTTP gebunden ist. Es beschreibt lediglich abstrakte Eigenschaften, die ein REST-konformes System erfüllen muss.
In den folgenden Abschnitten werden die Elemente sowie die Prinzipien von REST auf Basis von (Roy T. Fielding, 2000) und (Fielding, 2000) beschrieben.

2.3.1 Einschränkungen des REST-Architekturstils

Der REST-Architekturstil ist aus mehreren Einschränkungen zusammengesetzt, denen einer Architektur genügen muss. In den nachfolgenden Unterabschnitten werden diese anhand von

[25] Vgl. (Thilo Frotscher, 2007) Seite 411 - 414.
[26] Vgl. (Martin Gudgin, 2005a,b; Unbekannt, 2010) und (Thilo Frotscher, 2007) Seite 414 - 419.
[27] Vgl. (Roy T. Fielding, 2000) Kapitel 1.

(Fielding, 2000) S. 76ff beschrieben.

Client-Server

Der Client-Server-Architekturstil ist das erste Constraint, das sich in REST wiederfindet. Darin wird das sogenannte *Separation of Concerns*-Prinzip [28], welches die Trennung von Zuständigkeiten vorschreibt, verwendet. Somit verbessert sich die Skalierbarkeit des Servers, da die Komponenten einfach gehalten werden können.

Zustandlosigkeit

Die nächste Einschränkung ist die zustandslose Kommunikation in einer Client-Server-Interaktion. Dies hat zur Folge, dass eine erfolgreiche Anfrage alle benötigten Informationen enthalten muss, die zu ihrer Bearbeitung nötig sind. Somit liegt die Verwaltung von Sitzungen (Sessions) im Aufgabenbereich des Clients. Aus dieser Beschränkung ergibt sich der Vorteil, dass sich die Skalierbarkeit und die Zuverlässigkeit auf der Seite des Servers erhöht. Ein Nachteil ist jedoch die Steigerung der Netzwerklast aufgrund des erneuten Sendens von Informationen.

Cachefähigkeit

Um die Netzwerklast zu minimieren, wurde der Cache in dem REST-Architekturstil hinzu genommen. Die Einschränkung der Cachefähigkeit besagt, dass Antworten eines Servers implizit oder explizit als cachefähig bzw. nicht-cachefähig markiert werden können. Sobald eine Antwort cachfähig ist, liefert der Client-Cache zu einer equivalenten Anfrage die richtige Antwort.

Einheitliche Schnittstelle

Das zentrale Feature des REST-Architekturstils ist, dass zwischen Softwarekomponenten eine einheitliche, allgemeine Schnittstelle besteht. Dadurch erhöht sich die Überwachbarkeit der Transaktionen und die Implementierungen werden von deren Services losgelöst. Der Nachteil daraus ist, dass sich aufgrund einer standardisierten Schnittstelle die Effizienz vermindert. Um solch eine einheitliche Schnittstelle realisieren zu können, wurden mit:

- Bestimmung von Ressourcen,
- Manipulation von Ressourcen über Repräsentationen,
- selbstbeschreibende Nachrichten und
- Hypermedia als Engine des Application State

vier weitere Einschränkungen definiert. Diese werden im nachfolgenden Abschnitt genauer erläutert.

[28] Siehe http://www.aspiringcraftsman.com/2008/01/art-of-separation-of-concerns/.

Geschichtetes System

Ein System, bestehend aus verschiedenen Schichten, erlaubt eine klare Trennung von Zuständigkeiten. Somit ist es möglich Zwischenstationen, wie Proxies oder Gateways, zwischen Client und Server einzuführen. Dies hat zwar zur Folge, dass eine gewisse Verzögerung in der Verarbeitung der Daten und ein Overhead existiert, sich jedoch gleichzeitig der Grad der Wiederverwendbarkeit der einzelnen Komponenten erhöht sowie deren Komplexität verringert.

Code on Demand

Die letzte Einschränkung im REST-Architekturstil ist *Code on Demand*. Damit ist gemeint, dass REST es dem Client ermöglicht den Quellcode herunterzuladen sowie auszuführen, welcher in Applets oder Scripten zur Verfügung gestellt wird. Somit wird die Clientfunktionalität zur Laufzeit dynamisch erweitert, was ihn leichtgewichtiger macht.

2.3.2 Grundelemente von REST

Datenelement

Bei REST stehen die Datenelemente und somit die Informationen selbst im Vordergrund. Dabei kommunizieren die Komponenten, indem sie Repräsentationen von Informationen übertragen. Das Format der Informationen wird dabei dynamisch durch die Standarddatentypen oder durch den Empfänger und den Eigenschaften der Informationen bestimmt. In der nachfolgenden Tabelle 2.3[29] werden die Datenelement aufgezeigt und im weiteren Verlauf dieses Abschnittes näher erläutert.

Datenelement	Beispiele
resource	Ziel einer Hypertext Ressource
resource identifier	URL, URN
representation	HTML-Dokument, MP3 Datei
representation metadata	Media Type, Last-Modified Time
resource metadata	Kopfzeile eines HTTP-Responses (alternates, vary)
control data	Kopfzeile eines HTTP-Responses (if-modified-since)

Tabelle 2.3: Datenelemente in REST

Eine *Ressource* ist die Kernabstraktion einer Information in REST. Jede Information und jeder Datensatz kann eine Ressource sein. Fielding definiert in seiner Dissertation eine Ressource als eine Abbildungsfunktion, die zu einer bestimmten Zeit eine Menge von Werten abbildet. Die Werte in dieser Menge können Repräsentationen von Ressourcen sowie *resource identifier* (= Ressourcenbezeichner) sein. Sie werden in REST verwendet, um Resourcen zu identifizieren bzw. zu adressieren.
Die Komponenten von REST führen Aktionen auf einer Ressource aus. Dies wird realisiert, indem sie eine *Repräsentation* dieser Ressource verwenden. Somit wird der tatsächliche oder der zukünftige Zustand einer Ressource festgehalten. Die Repräsentationen werden dann an andere Komponenten übertragen und bestehen aus Information aus der Ressource, den Metadaten, die diese Daten beschreiben sowie, auf Veranlassung, aus Metadaten, die

[29] Aus (Fielding, 2000)Table 5-1. REST Data Elements.

die Metadaten beschreiben. Metadaten sind dabei sogenannte Namen-Werte-Paare, wobei der Name einem Standard entspricht, der die Struktur der Werte und die Semantik definiert. Antwortdokumente können dabei sowohl *resources metadata* als auch *represantation metadata* beinhalten. Dabei sind resources metadata Informationen über Ressourcen, die nicht einer bestimmten Ressource zugeordnet werden können.
Control data definiert den Zweck einer Nachricht, die zwischen den Komponenten ausgetauscht wird. Es wird auch verwendet, um Requests zu parametrisieren. Anhand der control data ist es möglich über eine Repräsentation den tatsächlichen oder den gewünschten Status einer angefragten Ressource zu bestimmen.

Konnektoren

Um den Zugriff auf Ressourcen und die Übertragung der Repräsentationen der Ressourcen zu verkapseln, werden in REST verschiedene Typen von Konnektoren verwendet. Konnektoren sind abstrakte Schnittstellen für die Kommunikation zwischen Komponenten. Sie tragen dazu bei, dass der Kommunikationsmechanismus und dessen Implementierung getrennt werden kann.
Alle auf REST basierenden Interaktionen sind zustandslos, was bedeutet, dass der Server lediglich den Zustand seiner eigenen Ressourcen verwaltet und nicht den des Clients und der Anwendung. Der Ansatz der Zustandslosigkeit hat den Nachteil, dass jeder Request alle benötigten Informationen beinhalten muss, damit er erfolgreich abgeschlossen werden kann. Dies kann eine Erhöhung der Netzwerklast mit sich bringen. Die Vorteile, die aus der Zustandslosigkeit resultieren, sind dabei von größerer Bedeutung. Die Konnektoren müssen nicht den Zustand der Anwendung zwischen den Requests speichern. Interaktionen können parallel abgearbeitet werden, obwohl zwischen den Anfragen keinerlei Zusammenhang besteht.
Mit Client-, Server-, Cache-, Resolver- und Tunnel-Konnektor existieren in REST fünf verschiedene Typen von Konnektoren.
Die beiden Hauptkonnektoren sind der Client- und der Serverkonnektor. Der Unterschied zwischen beiden liegt darin, dass ein Clientkonnektor eine Kommunikation beginnt, indem er ein Request sendet. Der Serverkonnektor hingegen wartet auf Requests und antwortet auf diese.
Der Cachekonnektor befindet sich auf der Schnittstelle zum Client- bzw. Serverkonnektor. Seine Aufgabe ist es wiederverwendbare Antworten zu speichern. Dabei kann der Cache auf zwei Arten verwendet werden. Zum einen auf der Clientseite, um die Netzwerklast zu senken und zum anderen auf der Serverseite, um den Prozess der Erstellung der Antwortnachricht nicht wiederholt ausführen zu müssen. Der Cache selbst besitzt die Fähigkeit zu entscheiden, ob eine Nachricht *cacheable* ist oder nicht.
Der Resolverkonnektor übersetzt, komplett oder teilweise, den Resource Identifier in eine Netzwerkadresse, um die für den physikalischen Netzwerkzugriff benötigten Daten zu erhalten.
Der Tunnelkonnektor ermöglicht es, Informationen über Systemgrenzen hinweg auszutauschen. Der Grund, warum dieser Konnektor nicht als Teil der Netzwerkinfrastruktur ausgelagert wurde, ist, dass sich einige REST-Komponenten dynamisch von einer aktiven Komponente zu einem Tunnel ändern.

Typen von Komponenten

In REST werden Komponenten anhand ihrer Rollen in der gesamten Anwendung typisiert. Dabei werden mit Server, Gateway, Proxy und User Agent vier Arten von verarbeitenden Elementen unterschieden.
Ein User Agent verwendet einen Clientkonnektor, um einen Request zu starten. Als Beispiel für diese Komponente wären hier Webbrowser zu nennen, die die Antworten der Server anzeigen. Der Server verwendet einen Serverkonnektor, um den Namespace für eine angefragte Ressource zu verwalten. Er ist der endgültige Empfänger einer Anfrage. Der User Agent hingegen ist der endgültige Empfänger einer Antwort.
Die Zwischenstellen stellen in Abhängigkeit zum Nachrichtenverlauf sowohl Client als auch Server dar. Dabei ist die Proxykomponente eine Zwischenstelle, die durch den Client auserwählt wurde. Die Gatewaykomponente hingegen ist eine durch das Netzwerk oder durch den Server ausgewählte Komponente. Der einzige Unterschied zwischen Gateway und Proxy ist, dass der Client bestimmt, wann er einen Proxy verwendet.

2.3.3 Prinzipien einer Ressourcen-orientierten Architektur

Die in den vorangegangenen Kapiteln dargestellten Einschränkungen und Elemente eines REST-Architekturstils werden in diesem Abschnitt exemplarisch anhand einer Linguistic Ressources Umgebung erläutert. In (Ruby, 2007) S. 79 ff wird auf Basis des REST-Architekturstils und den Web-Technologien XML, URI und HTTP eine *Resource Oriented Architecture (ROA)* beschrieben. Die Prinzipien dieser Architektur werden nachfolgend dargestellt. Die zentrale Idee dieser Architektur besteht darin, Web Services als HTTP-basierte Ressourcen mit den HTTP-Methoden als generische Schnittstellen zu realisieren.

Adressierbarkeit

Ressourcen sind in einer Ressourcen-orientierten Architektur Elemente, die Informationen zur Verfügung stellen. Dabei kann dies ein Algorithmus, eine Datenbankoperation oder Ähnliches sein, die über das Web zugänglich ist. Die Adresse einer Ressource sowie deren Name wird durch die URI festgelegt. Dadurch kann sie global eindeutig identifiziert werden.

```
Protokoll://Host:Port/Pfad?Querystring/#fragment
```

Listing 2.6: Aufbau einer URI

Protokoll gibt an welches Protokoll zur Kommunikation genutzt wird. Dies kann zum Beispiel HTTP oder HyperText Transfer Protocol Secure (HTTPS) sein. Der *Host* ist der Domain Name System (DNS)-Name oder die Internetprotokoll (IP)-Adresse einer adressierten Ressource. Der *Port* ist optional und numerisch und repräsentiert in Verbindung mit dem Host eine Ressource im Netzwerk. Der *Pfad* hat dabei die Gestalt des Pfades zu einem bestimmten Ordner auf dem Rechner. *Querystring* ist eine Menge aus Wertepaaren und einem Fragment, dass verwendet wird, um eine bestimmte Stelle in einem Dokument anzuzeigen.
(Ruby, 2007) empfiehlt, dass URIs so strukturiert werden sollten, dass die Bedeutung der referenzierten Ressource daraus hervorgeht. In Tabelle 2.4 ist ein Beispiel einer linguistischen RESTful Umgebung mit den dargestellten Ressourcen ersichtlich.
Um die spätere Verwendung von URIs einfacher zu gestalten, sollten sie in menschenlesbarer Form abgebildet werden. Dies ist allerdings keine Voraussetzung für einen RESTful Service.

URI	Bedeutung der Ressource
/sentences/getSentences/	Auflistung aller existierenden Sätze innerhalb eines Intervalls
/sentences/{sentence_id}/getReference/	Liefert Belegstelle für einen bestimmten Satz
/sentences/{sentence_id}/getCitation/	Liefert Zitate für einen bestimmten Satz
/words/{word_id}/	Liefert ein bestimmtes Wort
/words/{word_id}/getFrequencies/	Liefert die Frequenz eines Wortes aus einem Satz
/words/{word_id}/position/	Liefert die Position eines Wortes in einem Satz
/words/{word_id}/cooccurrences/	Auflistung der signifikanten Kookurrenzen für ein bestimmtes Wort
/authors/	Auflistung aller registrierten Autoren
/authors/{author_id}/	Liefert einen speziellen Autor

Tabelle 2.4: exemplarische Ressourcen einer linguistischen Umgebung

Einheitliche Schnittstelle

Es wird ein kleiner Satz an wohldefinierten Methoden, die es ermöglichen Ressourcen zu manipulieren, eingeführt. Die Idee ist, dass nur die generischen Methoden von HTTP für die RESTful Services genutzt werden. In Tabelle 2.5 ist eine Übersicht dieser HTTP-Methoden dargestellt. Es gibt noch weitere, wie TRACE und CONNECT, jedoch sind diese für RESTful Web Services nicht relevant und werden aus diesem Grund nicht näher betrachtet.

Methode	Beschreibung
GET	ist eine read-only Operation. Dient der Abfrage der Information einer Resource.
PUT	fügt einen Datensatz hinzu oder aktualisiert ihn. Der Client kennt die Identität der Ressource, die hinzufügt oder aktualisiert wird.
DELETE	löschen einer Ressource
POST	Hinzufügen von Informationen, in Form einer Repräsentation einer Resource.
HEAD	ähnelt GET mit der Ausnahme, dass nur Metainformationen einer Resource zurück geliefert werden.
OPTIONS	wird verwendet, um Informationen über die Kommunkiationsoptionen und Resourcen abzufragen.

Tabelle 2.5: HTTP-Methoden

Die Beschränkung der Methoden hat Vorteile. Es ist bei einer URI, die auf einen Web Service verweist ersichtlich, welche Methoden für diese Ressource verfügbar sind. Außerdem werden keine neuen Client Libraries benötigt, denn die meisten Programmiersprachen stellen eine solche Bibliothek standardmäßig bereit.
In der nachfolgenden Tabelle 2.6 sind mögliche Operationen auf den aus Tabelle 2.4 dargestellten Ressourcen aufgelistet.

HTTP Methode	Bedeutung
Resource /sentences/getSentences/	
GET	Abfrage aller existierenden Sätze innerhalb eines Intervalls
PUT	nicht verwendet
POST	Hinzufügen von Sätzen
DELETE	Löschen von Sätzen innerhalb eines Intervalls
Resource/authors/{author_id}/	
GET	Liefert einen speziellen Autor
PUT	Änderung von Autorendaten
POST	Einen neuen Autor hinzufügen
DELETE	Löschen eines Autors

Tabelle 2.6: exemplarische Methoden der Ressourcen einer linguistischen Umgebung

Zustandslosigkeit

Durch die zustandslose Kommunikation erfolgt eine starke Vereinfachung der Verwendung von RESTful Services. Der Server kennt nur seinen Zustand und nicht den der Clients. Dies hat zur Folge, dass falls Informationen aus früheren Abfragen benötigt werden, diese wiederholt geschickt werden müssen.

Repräsentationen

Eine Ressource ist eine Menge von Repräsentationen, die lediglich den derzeitigen Zustand einer Ressource wiederspiegeln. Jede Ressource ist über eine spezifische URI adressierbar, über die der Client und der Server die Repräsentationen der Ressource austauschen. Mit einer GET-Operation wird eine aktuelle Repräsentation einer Ressource empfangen. Durch ein POST oder PUT wird eine Repräsentation einer Ressource aktualisiert oder hinzugefügt. Die Repräsentation ist der Nachrichten-Body eines Requests oder Responses und kann ein beliebiges Nachrichtenformat, wie zum Beispiel *XML*, *Hypertext Markup Language (HTML)*, *Scalable Vector Graphics (SVG)* oder *Audio Video Interleave (AVI)* beinhalten. Mit dem Content-Type Header wird dem Client oder dem Server mitgeteilt welches Format benutzt wird.

Verbund von Ressourcen

Eine Vielzahl von RESTful Services sind Repräsentationen von Hypermedia. Sie können auch Links zu anderen Ressourcen beinhalten. Wenn nun Ressourcen intelligent miteinander verbunden werden, ist es möglich mit nur einem einzigen Link den Einstiegspunkt für weitere Services bereitzustellen.[30]
Um noch ein Beispiel zum Thema Verbund von Ressourcen aus der linguistischen Umgebung zu nennen, könnte ein Aufruf der Ressource */sentences/getSentences/* die Links zu den Wörtern innerhalb des Satzes, also zu der Ressource */words/{word_id}/*, sowie zu der Ressource */authors/{author_id}/*, die einen bestimmten Autor zurückliefert, beinhalten.

[30] Siehe (Ruby, 2007) S. 95.

2.3.4 Erweiterungen von RESTful Services

In (Ruby, 2007) werden Möglichkeiten besprochen, wie RESTful Services gesichert werden können. Diese werden in diesem Abschnitt genannt. Dabei wird ein Vergleich zum Abschnitt 2.2.7 gezogen, um aufzuzeigen, wie ein Äquivalent zu den für die SOAP Services existierenden WS-* Spezifikationen geschaffen werden kann.

Sicherheit

Für die Sicherung von Ressourcen gibt es bereits Mechanismen für HTTP die nicht neu erfunden werden müssen. Bei HTTP und somit auch bei RESTful Services findet Verschlüsselung von Nachrichten typischerweise auf Transportebene auf Basis von *Secure Socket Layers (SSL)* statt. Authentifizierung und Autorisation kann mit den sogenannten Standardfeatures von HTTP, *Basic-Authentication*, *Digest-Authentication* und *Username Token* erreicht werden. Des Weiteren ist es möglich, Ressourcen per Rechtevergabe, ähnlich wie bei einem Dateisystem, zu sichern. So kann eine Ressource *read-only* gestaltet werden, indem sie nur für GET und HEAD implementiert wurde.

Zuverlässige Nachrichtenübertragung

Ein Standard zur Sicherung der Nachrichtenübertragung wie bei SOAP ist in REST nicht nötig, da alle Methoden in HTTP idempotent[31] sind. Sobald ein GET, HEAD, PUT oder DELETE nicht ankommt, kann der Request problemlos wiederholt werden. Um *in-order* zu realisieren, muss lediglich gewährleistet werden, dass Nachrichten in der richtigen Reihenfolge gesendet werden. Die Ausnahme bildet hier die POST-Methode, die auch in SOAP verwendet wird. SOAP verwendet für die Gewährleistung der zuverlässigen Nachrichtenübertragung Felder mit Attributen im Header. In REST hat man mit der Implementierung eines sogenannten *POST Once Exactly*[32] eine Möglichkeit HTTP POST idempotent, wie es zum Beispiel bei GET oder DELETE ist, zu gestalten. Jedoch wird am häufigsten empfohlen POST für die sichere Nachrichtenübertragung ausser Acht zu lassen.

Transaktionen

Es ist in REST nicht sinnvoll, langandauernde und aufeinander aufbauende Request-Response-Zyklen zu erstellen, da HTTP und REST zustandslos sind. Es wird jedoch in (Ruby, 2007) Seite 231 ff. eine Möglichkeit beschrieben, wie Transkationen in RESTful Services realisiert werden. Darin wird ein Mechanismus dargestellt, der Transaktionen ebenfalls als Ressourcen ansieht. Ihnen werden Operationen durch Anfragen hinzugefügt.
Zu Beginn wird ein POST an eine Transaktionen-Factory-Ressource gesendet. Die Antwort ist dann der URI zu der gerade erstellten Transaktionsressource. Weitere Requests mit PUT fügen dann die Operationen zu der neuerstellten Ressource hinzu. Das Löschen der Ressource mit DELETE führt dann zu einem Rollback der Transaktion. Dies kann zu jedem Zeitpunkt in diesem Szenario geschehen. Ein Commit wird durch einen weiteren PUT Request mit *committed=true* erzeugt. Wurde nun das letzte Request durch den Server empfangen, können abschließende Überprüfungen auf Inkonsistenzen stattfinden. Die Implementierung

[31] Eine lineare Abbildung von $p : V \rightarrow V mit p^2 = p$ wird eine *Projektion* oder *idempotent* genannt. Siehe (Pareigis, 2000) S. 176

[32] Wurde von Mark Nottingham 2005 in http://www.mnot.net/drafts/draft-nottingham-http- poe-00.txt definiert.

dieses Szenarios kann auf Serverseite realisiert werden, indem eine Warteschlange für die zu den Transaktionen gehörigen Aktionen erstellt wird. Sobald die Transaktion bestätigt wird, kann der Server zum Beispiel eine Datenbanktransaktion starten und diese committen.
Da Transaktionen dem Konzept von RESTful Services widersprechen, sollte die Verwendung vermieden werden.

2.4 Performanceevaluation von SOAP- und RESTful-Services

In diesem Abschnitt werden die Details zur Serviceimplementierung sowie die Auswertung der Performance präsentiert. Zuvor sind die aus den vorangegangenen Abschnitten erarbeiteten Unterschiede der beiden Technologien, SOAP und REST, in der Tabelle 2.7 zusammengefasst[33]. Vorallem die Unterschiede in der Verarbeitung der Nachrichten, lassen vermuten, dass Services in einer Ressourcen-orientierten Architektur eine bessere Performance aufweisen, da der Overhead an zusätzlichen Daten sehr viel geringer ist, als bei Nachrichten, die mit SOAP verarbeitet werden. Diese Behauptung gilt es anhand von Tests zu beweisen.
Der für die Durchführung der Tests verwendete Referenz-Service gilt dabei als ein Synonym für zahlreiche in einer linguistischen Umgebung stattfindende Prozesse. Dieser Service[34] liefert anhand eines Intervalls von zwei Zahlwerten einen Bereich von Sätzen zurück, die in einer Datenbank gespeichert waren. Bei der Entwicklung des Services, war es primär wichtig, mit möglichst wenig Aufwand viele Daten zu generieren. Auch die angesprochenen Prozesse einer linguistischen Umgebung müssen teilweise große Mengen an Daten verarbeiten. Als Beispiel wäre hier die Abfrage von Informationen ganzer Bücher durch einen Client zu nennen. Wenn dieser einen bestimmten Absatz oder einen Satz mit den dazu gehörigen Meta-Informationen wie die Position im Buch, die Anzahl der darin befindlichen Wörter usw. benötigt. Solche Informationen müssen vorher aufbereitet bzw. wenn sie schnell genug ermittelt werden können, zur Laufzeit berechnet und anschließend an den Client geschickt werden. Dieses Szenario wurde verallgemeinert und bei der Implementierung der Tests aufgegriffen und simuliert.

2.4.1 Service-Implementierung

Für die Durchführung der Tests wurde eine Web-Anwendungen entwickelt. Diese veröffentlicht zwei Schnittstellen und wurde in dem Web Service Container Tomcat gehostet. Eine der beiden Schnittstellen wurde auf Basis von Java API for XML Web Services (JAX-WS) 2.2[35] erstellt und kann ausschließlich für die Verarbeitung SOAP-basierten Anfragen verwendet werden. Die zweite Schnittstelle wurde für den Empfang von REST-basierten Anfragen auf Basis von Java API for RESTful Web Services (JAX-RS) 1.1[36] entwickelt. Beide dieser Schnittstellen verarbeiteten die Anfragen nach der gleichen Logik, lieferten die gleichen

[33] Diese Tabelle enthält eine Zusammenfassung der erarbeiteten Unterschiede zwischen SOAP und REST. Aus diesem Grund wird von einer erneuten Angabe der Quellen abgesehen.

[34] Im weiteren Verlauf dieser Arbeit wird dieser Service als SentencesService bezeichnet.

[35] Details zu JAX-WS 2.2, Siehe Kapitel 3.3. Es wird hier nicht näher darauf eingegangen, da lediglich der Vergleich zwischen SOAP und REST im Vordergrund steht.

[36] JAX-RS ist eine Programmierschnittstelle für Java, die es ermöglicht den im vorangegangenen Abschnitt erläuterten, REST Architekturstil zu verwenden. Die Grundlagen für die Entwicklung RESTful Services wurden aus (Burke, 2010) entnommen.

Nutzdaten im Antwortdokument zurück und mussten mit zwei Parametern aufgerufen werden. Diese beiden Parameter waren dabei zwei Zahlwerte, die ein Intervall repräsentierten. Anhand dieses Intervalls wurde eine Datenbankabfrage gestartet, welche alle Sätze zwischen den beiden Werten zurücklieferte. Die Sätze waren dabei in einer Tabelle namens *sentences* in der Datenbank *TLG*[37] gespeichert. Jedem Satz war dazu noch eine global eindeutige Id zugeordnet, anhand derer der Service bzw. die Ressource die Sätze identifizieren konnte. Nachdem die Datenbankabfrage erfolgreich durchgeführt wurde, wurde ein *ResultSet*-Objekt mit den Sätzen zurückgeliefert und in einer Schleife durchlaufen, um das XML-Dokument zu generieren. Eine Zeile dieses ResultSet-Objektes enthält dann die Id des Satzes sowie den dazugehörigen Satz. Das resultierende Dokument wurde schließlich an JAX-WS bzw. JAX-RS für die endgültige Verarbeitung in Form einer Zeichenkette übermittelt und an den Client geschickt.

[37] Zum Zeitpunkt der Durchführung der Tests, beinhaltete die Datenbank Datensätze der Bibel in Altgriechischer Sprache.

	SOAP	REST
Technologieart	Protokollframework	Architekturvorschlag
Sichtweise des Web	Möglichkeit zum Transport von Nachrichten	Möglichkeit zum Zugriff auf Informationen
Fokus	auf Design von integrierten verteilten Systemen	auf Performance und Skalierbarkeit von verteilten Hypermedia-Systemen.
Kommunikation	indirekt (über SOAP Intermediär), direkt (Client-Server)	direkt (Client-Server)
Transport	Protokoll unabhängig	HTTP bzw. HTTPS
Session Management	Server kann Konversationen über mehrere Request-Response-Zyklen aufrecht erhalten	Server ist zustandslos
Design	1. Serviceoperationen identifizieren 2. Datenmodell für den Inhalt der Nachrichten definieren 3. Transportprotokoll wählen 4. Implementierung und deployen des Web Services auf Web Service Container	1. Ressourcen identifizieren 2. URLs definieren 3. HTTP-Methoden für Ressource definieren 4. Beziehungen zwischen einzelnen Ressourcen identifizieren 5. Implementierung und deployen auf Web Server
Naming	kein standardisierter Namingmechanismus vorhanden	konsistenter Namingmechanismus für Ressourcen
Schnittstellen	Jede Operation bzw. Funktion muss eigenen Service darstellen	global (über URIs)
Schnittstellenbeschreibung	über WSDL	WADL/WSDL
Verzeichnisdienst	UDDI	Suchmachine (z. B.: http://ecosia.org/)
zuverlässige Nachrichtenübertragung	WS-ReliableMessaging, WS-Reliability	POST Exactly Once, ansonsten nicht weiter nötig, da PUT, DELETE, GET idempotent
Transaktionen	WS-Transaction	über Transaktionsmanager

Tabelle 2.7: Zusammenfassung der Unterschiede zwischen SOAP und REST

Der Client ruft den SOAP- bzw. RESTful-Service über eine Uniform Resource Locator (URL), wie in dem Beispiel 2.7 bzw. in 2.8 gezeigt ist, auf.

```
http://aspra26.informatik.uni-leipzig.de:8080/
    SoapRestComparator/SentencesService
```

Listing 2.7: Aufruf des SOAP-Services

```
http://aspra26.informatik.uni-leipzig.de:8080/
```

```
SoapRestComparator/Resources/Sentences/
getSentences?StartId=1&EndId=1
```

Listing 2.8: Aufruf desRESTful-Services

Bei Aufruf des SOAP-Service schickt der Client dem Server ein SOAP-Datei, die die gewünschte Start- und End-Id beinhaltet. Die Angabe der beiden Ids erfolgt beim RESTful-Service direkt in der URL.
Der Client wurde mit JMeter erzeugt. Dieser generierte pro Nachrichtengröße, 1.000 Requests. Die Größe der Nachricht wurde dabei nicht direkt an der Größe in MB festgelegt, sondern anhand der Anzahl der Sätze, die angefragt wurden und sich schließlich im Antwortdokument befanden. Dabei existierten mit 1-, 100-, 1.000-, 10.000- und 100.000 Sätzen im Antwortdokument fünf verschiedene Nachrichtengrößen. Es wurde immer nur ein Request an den Server gestellt. Der Client erzeugte erst eine neue Anfrage, sobald er eine Antwort empfangen hat.
Zur Verifizierung der Testergebnisse wurden die Tests mehrfach und auf verschiedene Art und Weise durchgeführt. Dies wurde zu Beginn der Arbeit erwähnt.

2.4.2 Versuchsergebnisse

In diesem Abschnitt sind die Ergebnisse des im vorangegangenen Abschnittes beschriebenen Tests dargestellt. Diese sind in den Tabellen 2.8 und 2.9 sichtbar. Dabei ist anzumerken, dass die Gesamtverarbeitungszeit die Dauer vom Senden des Requests bis hin zum Empfang des Response vom Server beinhaltet. Die Serververarbeitungszeit hingegen ist lediglich die Zeit, die der Server benötigt, um einen Request zu verarbeiten und ein entsprechendes Response zu generieren. Beim Durchsatz handelt es sich um die Requests, die pro Sekunde gestellt werden konnten und die Nachrichtengröße repräsentiert die Größe des vom Server generierten Antwortdokumentes.

Anzahl der Elemente	Gesamt-verarbeitungs-zeit (in ms)	Server-verarbeitungs-zeit (in ms)	Durchsatz (in Requests/Sekunde)	Nachrichtengröße (in KB)
1	8	4	85,9	0,8
100	16	8	51,5	29,8
1.000	38	27	24,5	269
10.000	467	264	2,1	3.347
100.000	7.364	2.186	0,1	27.961

Tabelle 2.8: Versuchsergebnisse von SOAP

Aus diesen Testresultaten wird deutlich, dass die Dauer eines Request-Response-Zyklus bei SOAP bis zu 60% größer ist als bei REST. Dies ist vor allem dem Overhead an Daten zu zuschreiben, die im SOAP Request- und Responsedokument enthalten sind. Denn das Responsedokument ist bis zu 18% größer als das von REST. Das Requestdokument ist bei SOAP zwar lediglich nur maximal 303 Bytes groß, jedoch existiert für REST kein solches Dokument. Beide dieser Tatsachen haben zur Folge, dass sich die Netzwerklast bei SOAP erhöht und mehr Daten übertragen werden müssen. Dies wiederum erhöht die Dauer der Request-Response-Zyklen.

Anzahl der Elemente	Gesamt-verarbeitungs-zeit (in ms)	Server-verarbeitungs-zeit (in ms)	Durchsatz (in Requests/Sekunde)	Nachrichtengröße (in KB)
1	5	3	177,0	0,4
100	7	6	120,8	24,2
1.000	28	25	34.2	217
10.000	240	228	4,2	2.863
100.000	2.203	1.985	0,5	23.129

Tabelle 2.9: Versuchsergebnisse von REST

Auch ein Vergleich der Serververarbeitungszeiten zeigt, dass der SOAP-Service für größere Dokumente bis zu 15% mehr Zeit für die Verarbeitung einer Anfrage benötigt als der RESTful-Service. Dieser Mehraufwand kommt zu stande, da der SOAP-Service das SOAP-Anfragedokument auswerten und die Antwort in ein neues SOAP-Dokument verpacken muss. Weiterhin wurden die Zeiten der Datenbankabfragen gemessen. Diese waren bis auf wenige Millisekunden nahezu gleich, da die gleichen Abfragen für beide Services verwendet wurden.
In Abbildung 2.4 wurden die durchschnittlichen Gesamtverarbeitungszeiten normalisiert und grafisch dargestellt.

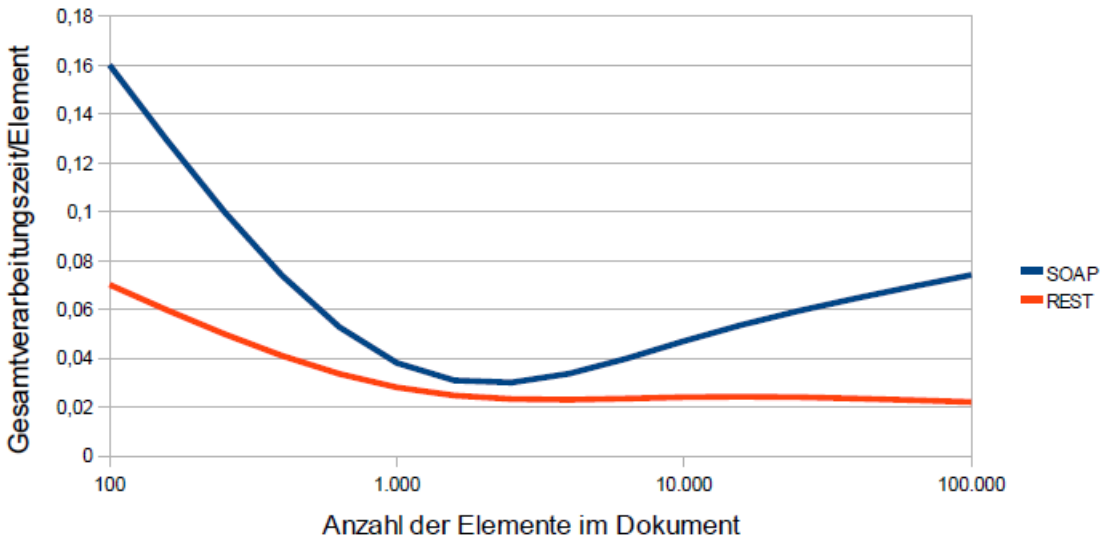

Abbildung 2.4: Durchschnittliche Gesamtverarbeitungszeit pro Element

Aus dieser Grafik geht hervor, dass ab einer Größe von 1.000 Elementen in einem Dokument die Verarbeitungszeiten der REST-basierten Anfragen mit ca. 0.02 Millisekunden relativ konstant sind. Bei SOAP-basierten Anfragen hingegen steigen die Verarbeitungszeiten ab 1.000 Elementen in einem Dokument auf ca. 0,08 Millisekunden pro Element an. Dies bestätigt erneut die Behauptungen, dass die Anfragen in REST schneller verarbeitet werden als in SOAP.

2.5 Zusammenfassung

Die Aufgabe in diesem Kapitel war es, SOAP und REST miteinander in Relation zu setzen. Dabei wurde neben den Grundlagen und Features der beiden Techniken auch die Performance getestet.
Der Vergleich zeigte, dass REST nicht nur für einfache Datenbank-Lookups ein gleichwertiges Mittel ist, um Services zu entwickeln und bereitzustellen. Fast jede SOAP-Erweiterung konnte ohne Verwendung von irgendwelchen zusätzlichen Spezifkationen in REST gleichwertig nachgestellt werden. Die Auswertung der Tests ergab zusätzlich, dass REST sowohl auf dem Server als auch in der Gesamtheit für größere Dokumente eine wesentlich bessere Performance aufwies als SOAP.
Wenn es lediglich um die Verwendung einfacher Datenbank-Lookups oder Datenmanipulationen geht, ist REST, aufgrund der Resultate, die bessere Alternative. REST verwendet ausschießlich ausgereifte und weitverbreitete Standards, wie zum Beispiel HTTP und URIs. Außerdem sind HTTP-Bibliotheken in verschiedenen Programmiersprachen eher zu finden als SOAP Frameworks. Ein weiterer Punkt, der für REST spricht, ist die geringere Lernkurve. SOAP sollte dann eingesetzt werden, wenn der Service komplex wird, was zum Beispiel bei langandauernden Transaktionen der Fall ist. Obwohl SOAP sehr komplex und mächtig ist, wird es von einer Vielzahl von Tools unterstützt, was wiederum für SOAP spricht. Jedoch benötigt man für REST keine weiteren Tools, als eine HTTP-Bibliothek für die verwendete Programmiersprache. Die Für- und Widersprüche von SOAP und REST verdeutlichen, dass die Schwierigkeit darin besteht, sich auf eine Technologie festzulegen. Anstatt sich somit für eine der beiden Technologien zu entscheiden, ist das zur Verfügung stellen beider Technologien wohl die beste Lösung.

3 Web Service Frameworks

In diesem Kapitel werden drei Web Service Frameworks, in ihren Grundlagen, Features und Performance genauer betrachtet. Die Motivation bildet dabei ein existierender SOAP-Kernel, der vor sechs Jahren auf Basis von Apache Axis 1 [1] entwickelt wurde. Dieser besteht dabei aus einer Ansammlung der verschiedensten Services. Aufgrund der Annahme, dass Apache Axis 1 veraltet sei und womöglich eine wesentlich schlechtere Performance als andere, neuere Frameworks besitzt werden hier mit Apache Axis 2 und Metro 2.0 zwei weitere Web Service Frameworks vorgestellt. Hier soll durch Analysen der Arbeitsweisen, Features und der Performance dieser Frameworks eine Entscheidungsgrundlage für eine etwaige Neuentwicklung auf Basis eines der anderen beiden Frameworks erfolgen.
Zu Beginn dieses Kapitels werden die oben genannten Frameworks auf ihre Architektur und Features untersucht. Im Anschluss daran werden sie in Relation zueinander gesetzt. Den Abschluss bildet dann die Analyse der Performance.

3.1 Apache Axis 1

Apache eXtensible Interaction System (Apache Axis) ist eine Open Source-Implementierung des Web Sevice Standards SOAP. Es ist ein Framework, um Web Service Komponenten, wie zum Beispiel Client, Server oder Gateways zu erstellen. Zusätzlich beinhaltet Apache Axis 1 unter anderem einen Standalone Server der Unterstützung für WSDL bietet und liefert Überwachungstools für TCP/IP-Packete. Es läuft unter der Lizenz der Apache Software Foundation.
Apache Axis 1 gilt als die dritte Generation von Apache SOAP. Anfänglich war das Projekt noch unter dem Namen SOAP for Java (SOAP4J) bekannt. Die letzte Version 1.4 wurde im April 2006 veröffentlicht. Die Architektur wurde dabei von Grund auf neu entwickelt. Eine der wesentlichsten Veränderungen war der Umstieg von DOM auf Simple API for XML (SAX).[2]

3.1.1 Architektur

Apache Axis 1 besteht aus verschiedenen Komponenten. Die Hauptkomponente bildet dabei die *AxisEngine*. Sie ist für die Verarbeitung von SOAP-Nachrichten verantwortlich und koordiniert weitere Komponenten. Um eine SOAP-Nachricht zu verarbeiten, durchläuft sie mehrere *Handler*, die von der Engine aufgerufen werden. Die Auswahl der verschiedenen Handler sowie die Reihenfolge der Aufrufe dieser ist von der Konfiguration abhängig. Auch die Bestimmung, ob es sich bei dem Host um einen Server oder einen Client handelt wird

[1] Siehe (Foundation, 2005).
[2] Vgl. http://ws.apache.org/axis/index.html.

in der Konfiguration angegeben. Beim Ansteuern der Handler wird jedem ein *Message Context* übergeben, der wiederum den Request, falls vorhanden, den Response sowie weitere Informationen beinhaltet. Zu den Informationen können zum Beispiel der Name des Web Services, Session Informationen oder die aufgerufene Operation zählen. Jeder Handler ist in der Lage den Message Context mit all seinen enthaltenen Informationen zu verarbeiten. Eine Menge von Handlern werden somit zu einer so genannten *Chain* zusammengefasst. Chains repräsentieren dabei einen speziellen Handler, der eine Menge von anderen Handlern nacheinander aufruft.
Die Erzeugung des Message Contexts geschieht auf Clientseite nicht durch die AxisEngine, sondern durch die Klasse *Call*. Nachdem ein Call-Objekt erzeugt und alle nötigen Informationen angegeben wurden, wird das Message-Context-Objekt erstellt und an die Engine übergeben. Auf dem Server wiederum empfangen die *Transport Listener* die SOAP-Nachricht und erzeugen schließlich den Message-Context mit allen benötigten Informationen aus der Nachricht.

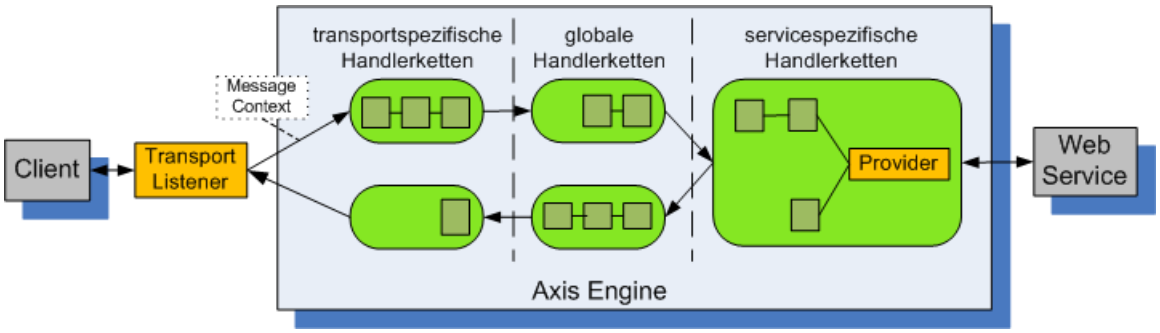

Abbildung 3.1: Verarbeitung von SOAP-Nachrichten in Apache Axis 1 auf der Serverseite

Abbildung 3.1 verdeutlicht den Weg des Message Contexts durch die Handler. Jedes kleine Rechteck repräsentiert dabei einen Handler. Die hellen Rechtecke mit den abgerundeten Kanten stellen die Chains dar.
Der Weg des Message Context zum aufgerufenen Web Service wird als *Request Flow* bezeichnet, der Rückweg als *Response Flow*. Der Request Flow beinhaltet drei verschiedene Handlerketten. Diese sind die *transportspezifische*, die *globale* und die *servicespezifische* Kette. Eine Kette wird nur dann durchlaufen, wenn Handler für diese konfiguriert wurden. Die transportspezifische Handlerkette wird nur beim Eingang einer SOAP-Nachricht über ein bestimmtes Transportprotokoll aufgerufen. Danach wird die globale Kette, die die aufgerufenen Handler beinhaltet, ausgeführt. Zum Schluss kommt die servicespezifische Kette zum Einsatz, wenn die Nachricht an einen ganz bestimmten Service gerichtet ist. Ein besonderer Handler in dieser Kette ist der *Provider*, der schließlich den Web Service aufruft, an den die Nachricht gerichtet ist. Dieser Handler füllt den Message-Context, der auch den Request enthält, mit dem Ergebnis des aufgerufenen Services. Ab diesem Zeitpunkt befindet sich die Nachricht im Response Flow. Hier durchläuft der Message Context wieder die drei Handlerketten, jedoch in umgekehrter Reihenfolge. Anzumerken ist, dass die Art der beinhalteten Handler sowie die Konfiguration vom Request Flow abweichen kann.
Auf der Seite des Clients ist der Nachrichtenfluss ähnlich wie auf der Serverseite. Wie in Abbildung 3.2 verdeutlich wird, ist lediglich die Reihenfolge der Chains umgekehrt. Auf der Clientseite wird der Message-Context im Request Flow durch die Anwendung an die Engine übergeben. Im Gegensatz zur Serverseite ist hier die erste Anlaufstation die servicespezifische Kette, die jedoch keinen Provider beinhaltet. Danach werden die Handler der globalen-

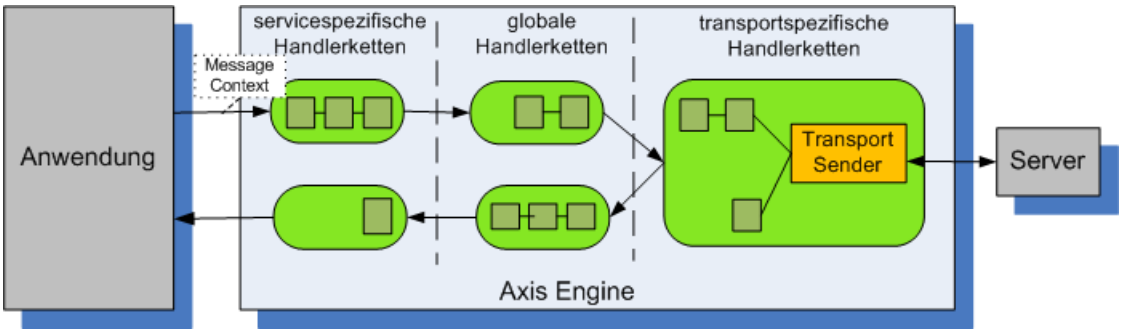

Abbildung 3.2: Verarbeitung von SOAP-Nachrichten in Apache Axis 1 auf der Clientseite

und der transportspezifischen Kette durchlaufen. Die transportspezifische Kette auf der Clientseite beinhaltet den *Transport Sender* als besonderen Handler. Er hat als Aufgabe ausgehende SOAP-Nachrichten mit hilfe des zu verwendenden Transportprotokolls zu versenden und eingehende SOAP-Nachrichten zu empfangen. Die empfangene Antwort wird dann vom Transport Sender in das Message Context Objekt gepackt und über den Response Flow an die Anwendung geschickt. [3]

Subsysteme

Um Apache Axis 1 modular aufzubauen, wurden verschiedene Subsysteme eingeführt, die nach ihren Zuständigkeiten unterteilt wurden und zusammenarbeiten können. Jedes dieser Systeme ist unabhängig voneinander, sie können dennoch gemeinsam verwendet werden. Diese Modularität wurde allerdings bei der Entwicklung nicht konsequent durchgeführt. So ist im Quellcode ersichtlich, dass einige Systeme über verschiedene Pakete verteilt sind und einige in anderen Systemen mehrfach verwendet werden. In den nachfolgenden Abschnitten werden einige der wichtigsten Subsysteme vorgestellt.[4]

Message Flow Subsystem

Der zentrale Punkt des Message Flow Subsystems ist die AxisEngine, die den gesamten Ablauf der Nachrichtenverarbeitung kontrolliert. Sie ruft Handler der Reihe nach auf, damit diese Nachrichten verarbeiten. Die AxisEngine ist als abstrakte Klasse mit zwei konkreten Unterklassen implementiert. Diese sind *AxisClient*, die die Handler auf der Clientseite kontrolliert und *AxisServer*, die die Handler der Serverseite kontrolliert.
Weitere Komponenten des Message Flow Systems sind die Handler und Chains. Die transportspezifischen, servicespezifischen und globalen Handler werden zu Chains zusammengefasst. Somit umfasst die komplette Folge der Handler drei Chains. Eine vom Typ Transport, eine vom Typ Service und eine vom Typ Global. An einem gewissen Punkt in der Aufrufreihenfolge kann es Handler geben, die sowohl Requests verarbeiten als auch Responses erzeugen. Solche Handler werden *Pivot Point Handler* der Sequenz genannt. Eine Nachricht wird verarbeitet, indem sie durch die drei Handlerketten gereicht wird. Eine besondere Handlerkette ist die sogenannte *Target Chain*. Das sind spezielle Handlerketten, die einen Request Handler, einen Pivot Point Handler und einen Response Handler besitzen.[5]

[3] Vgl. (Dapeng Wang, 2004) S. 277 - 282 und (Project, 2005).
[4] Vgl. (Project, 2005).
[5] Vgl. (Dapeng Wang, 2004) S. 282 - 287 und (Project, 2005).

Message Model Subsystem

In Apache Axis 1 existieren Klassen, die den Elementen einer SOAP-Nachricht entsprechen. Somit gibt es zum Beispiel eine Klasse *SOAPHeaderElement*, die die Attribute actor und mustUnderstand beinhaltet oder eine Klasee *SOAPBody*, die die Nutzdaten einer SOAP-Nachricht enthalten.
Während der Deserialisierung wird ein Baum aus SOAP-Klassen erstellt. Dieser wird in Abbildung 3.3 präsentiert.

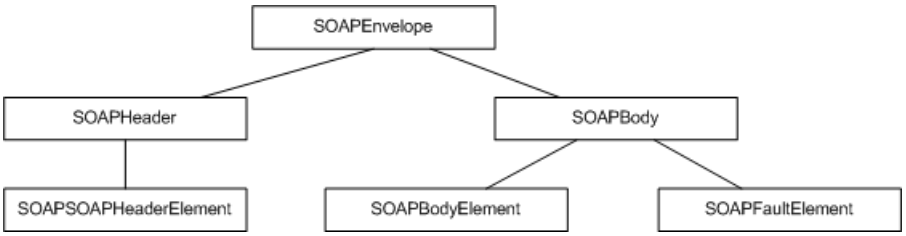

Abbildung 3.3: SOAP Elemente in Apache Axis 1

Die Klasse, die hauptsächlich für das Parsen von XML verantwortlich ist, ist *DeserializationContext*. Sie koordiniert den Erstellungsprozess des Baumes und enthält unter anderem einen Stack von SAX-Handlern und eine Referenz auf die Klasse, die die Ergebnisse der Nachrichtenverarbeitung enthält. [6]

3.1.2 Data Binding

Apache Axis 1 liefert ein eigenes Type-Mapping-Framework, mit dem die Umwandlung der Datentypen der XML-basierten SOAP-Nachrichten in Java-Typen und umgekehrt erfolgt, welches in diesem Abschnitt erläutert wird.
Das grundlegende Mapping zwischen Java-Typen und SOAP-Datentypen ist durch die Java API for XML-based RPC (JAX-RPC)-Spezifikation bestimmt. Das Subsystem, was in Apache Axis 1 für die Serialisierung und Deserialisierung von Daten verantwortlich ist, ist das Encoding Subsystem. Darin existieren zwei Komponenten: *Serializer* und *Deserializer*. Diese sind für die Serialisierung bzw. Deserialisierung von verschiedensten Java- und XML-Datentypen zuständig. Sie wurden so entwickelt, dass sie sowohl DOM als auch SAX[7], als XML-Verarbeitungsmechanismus unterstützen. Im Encoding Subsystem wird für jedes Paar, von Java- und XML-Datentyp, ein spezieller Serializer und Deserializer angeboten. Zusätzlich unterscheiden sich diese noch in DOM- bzw. SAX- Serializer und Deserializer. Für die Identifizierung des richtigen Serializers bzw. Deserializers wurde mit dem Type-Mapping ein spezielles Mapping in Apache Axis 1 eingeführt. Dieses identifiziert die Datentypen anhand des *QName*. Jedoch gibt es auch hier, für jeden Datentyp verschiedene Type-Mapping-Implementierungen. Die Wahl, welches Type-Mapping für eine spezielle Nachricht zu verwenden ist, wir anhand des Encodings bestimmt, was in der Nachricht angegeben ist. Eine sogenannte *Type-Mapping-Registry* liefert dann anhand des Namens des Encodings ein bestimmtes Type-Mapping.[8]

[6] Vgl. (Project, 2005).
[7] Vgl. http://www.saxproject.org/.
[8] Vgl. (Dapeng Wang, 2004) Seite 161 - 214 und Seite 399 - 424 und (Project, 2005).

3.1.3 Features und Erweiterungen von Apache Axis 1

Realisierung von Web Service Erweiterungen

Um WS-* Spezifikationen einsetzen zu können, müssen Handler, die sich um die Sicherheit, Integretität und Vertraulichkeit der Web Services kümmern, von Hand implementiert werden. Jedoch werden einige Frameworks von Apache Web Services Project angeboten, die diese Arbeit abnehmen, elche nachfolgend kurz aufgelistet und erläutert werden.

WS-Coordination, WS-AtomicTransaction und WS-BusinessAcitivity:

- Wird durch Kandula1 implementiert.[9]
- Bietet Interoperabilität mit anderen WS-Coordination, WS-AtomicTransaction und WS-BusinessAcitivity Implementierungen, speziell derer von .NET und IBM.

WS-ReliableMessaging:

- Wird durch Sandesha1 implementiert. [10]
- Veröffentlich durch IBM, Microsoft, BEA und TIBCO.
- Unterstützt WS-Policy und WS-Addressing.
- Interoperabilität mit anderen WS-ReliableMessaging Implementierungen.
- Stellt *inOrder*-Nachrichtenübermittlung zur Verfügung.

WS-Security:

- Wird durch WSS4J implementiert.[11]
- Ist eine Implementierung der OASIS WS-Security.
- Bietet Features wie XML-Security, mit XML-Signature und XML-Encryption und Tokens, mit Username-Tokens, Timestamps und SAML-Tokens.

Nachrichtenoptimierung

Für die Realisierung von *SOAP with Attachments* müssen in Apache Axis 1 lediglich mit *mail.jar* und *activation.jar* zwei Bibliotheken in die Webanwendung aufgenommen werden. Um das Attachment einer SOAP-Nachricht auszulesen bzw. in eine Nachricht zu verpacken, existieren die beiden Klassen *JAFDataHandlerSerializerFactory* und *JAFDataHandlerDeserializerFactory*. Wenn ein Service Attachments erzeugt, erstellt der Serializer eine externe Datei im DIME bzw. MIME Format. Über die Klasse *DataHandler* wird schließlich die Attachmenterzeugung bzw. der Empfang gesteuert. [12]

[9] Sieht http://ws.apache.org/kandula/1/index.html/.
[10] Siehe http://ws.apache.org/sandesha/sandesha1.html/.
[11] Siehe http://ws.apache.org/wss4j/.
[12] Vgl. (Dapeng Wang, 2004) Seite 473 - 481

Deployment

Apache Axis 1 beherrscht mit dem sogenannten instant- und dem benutzerdefinierten-Deployment zwei Deploymentmechanismen. Das automatische Deployment ist der einfachste Weg für einen Entwickler, Web Services bereitzustellen. Bei diesem Mechanismus wird lediglich die Dateinamenserweiterung *.java*, die den Quelltext des Web Services enthält in *.jws*[13] umbenannt. Es werden alle Methoden als Web Service veröffentlicht, die innerhalb der Klasse als *public* deklariert wurden. Der Nachteil bei dieser Deploymentmethode ist, dass die Web Services nur in der Lage sind, einfache Datentypen wie *String*, *Integer*, *Boolean* usw. zu verarbeiten. Des Weiteren ist es nicht möglich, Handler zu konfigurieren. Die JWS-Datei muss nun in das Axis Verzeichnis kopiert werden. Das Kompilieren passiert im Hintergund automatisch, sobald der Service aufgerufen wird.
Die Basis für das benutzerdefinierten Deployment bildet die kompilierte Version des Services und ein *Web Service Deployment Descriptor (WSDD)*. Ein WSDD bündelt die Methoden, die es in Axis zu veröffentlichen gilt. Im folgenden Listing 3.1 ist ein Beispiel einer WSDD-Datei dargstellt.

```
<ns1:deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
  xmlns:ns1="http://xml.apache.org/axis/wsdd/">
 <ns1:globalConfiguration>
  <ns1:parameter name="disablePrettyXML" value="true"/>
  <ns1:parameter name="adminPassword" value="admin"/>
      ...
  <ns1:requestFlow> ...</ns1:requestFlow>
 </ns1:globalConfiguration>
    ...
 <ns1:service name="Sentences" provider="java:RPC"
       style="wrapped" use="literal">
  <ns1:operation name="getSentences" qname="ns3:getSentences"
       returnQName="ns3:getSentencesReturn"
       returnType="xsd:string" soapAction=""
       xmlns:ns3="http://ServiceTypes"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <ns1:parameter qname="ns3:startId" type="xsd:int"/>
   <ns1:parameter qname="ns3:endId" type="xsd:int"/>
  </ns1:operation>
  <ns1:parameter name="allowedMethods" value="getSentences"/>
  ...
  <ns1:parameter name="className"
        value="ServiceTypes.Sentences"/>
 </ns1:service>
 <ns1:transport name="http">
 ...
</ns1:deployment>
```

Listing 3.1: Ausschnitt eines Web Service Deployment Descriptor

[13] Java Web Service (JWS).

Das äußerste Element zeigt der AxisEngine an, dass es sich um einen Deployment-Descriptor handelt. Zusätzlich werden hier Namensräume für Java und WSDD definiert. Das Element *globalConfiguration* beinhaltet die globalen Definitionen. Dies ist dann für alle Services gültig, die in dieser Axis-Instanz veröffentlicht werden. Ein weiteres wichtiges Element ist *service*, welches den eigentlichen Service definiert. In diesem Beispiel wird als Provider *java:RPC* definiert. Dieser Provider ist in Apache Axis 1 integriert und deutet an, dass es sich hier um einen *Remote Procedure Call*-Service handelt. Dem Provider muss nun noch bekannt gegeben werden, welche Klasse er aufrufen muss. Dies geschieht im Element *parameter*, mit dem Attribut *className*. Der Parameter mit dem Attribut *allowedMethods* definiert, welche Methoden als Service veröffentlicht werden. Hier ist es möglich weitere Methoden zu definieren. Wenn jedoch definiert werden soll, dass alle als *public* deklarierten Methoden dieses Services veröffentlicht werden, dann muss als Wert lediglich * gesetzt werden.[14] [15]

3.2 Apache Axis 2

Apache Axis 2 ist der Nachfolger von Apache Axis 1. Es unterstützt sowohl SOAP als auch RESTful Services. Apache Axis 2 wurde von Grund auf neu entwickelt. Die Vorteile, die daraus resultieren, sind unter anderem eine höhere Performance, einfachere Erweiterbarkeit und leichtere Konfiguration. Des Weiteren unterstützt Axis 2 HTTP, TCP, JMS und SMTP als Transportprotokolle. [16]

3.2.1 Architektur von Apache Axis 2

Apache Axis 2 wurde zwar von Grund auf neu entwickelt, jedoch wurden einige bewährte Konzepte von Apache Axis 1, wie zum Beispiel das der Handler und das der Module, beibehalten. Die Hauptkomponente ist das Objektmodell Axis Object Model (AXIOM) [17] Wie auch schon bei Apache Axis 1 ist das Zentrum des Frameworks die *AxisEngine*. Dieses ist für die Verarbeitung einer SOAP- bzw. REST-basierten Nachricht verantwortlich.
An den Schnittstellen nach außen sitzen die Transport- und die XML Data Binding-Komponenten. Die Aufgabe der Transportkomponenten besteht darin eingehende Nachrichten zu empfangen. Aufgabe der Data Binding Komponenten ist es, die Daten aus den Nachrichten von XML in Java-Objekte umzuwandeln und umgekehrt.
Die Architektur von Apache Axis 2 ist modular aufgebaut und unterteilt sich in *Kernmodule* und *Zusatzmodule*. Die Kernmodule bilden die Kernarchitektur und die Grundlage für die Zusatzmodule. Die folgenden Tabellen 3.1 und 3.2 zeigen einen Überblick über diese.
Das Information Model hat mit *Description-* und *Contexthierarchien* zwei Haupthierachien. Die Descriptionhierachie beschreibt die statischen Informationen. Dies können zum Beispiel die Konfiguration von Apache Axis 2 oder die Operationen der veröffentlichten Web Services sein. Typischerweise sind solche Informationen in den Konfigurationsdateien wie *services.xml* oder *axis2.xml* eingebettet. Diese werden ausgelesen und in der Description-Hierarchie gespeichert. Die Contexthierachien beinhalten die eher dynamischen Informationen. Dies sind zum Beispiel Informationen über Objekte, Kontext einer Nachricht oder Sitzungen. In Abbildung 3.4 ist ein Überblick über die Beziehungen zwischen Context- und

[14] Vgl. (Foundation, 2005).

[15] Es existieren noch zahlreiche weitere Möglichkeiten, Services über WSDD zu konfigurieren. Da diese alle zu nennen nicht Teil dieser Arbeit ist, sei an dieser Stelle auf (Project, 2006).

[16] Vgl. (Thilo Frotscher, 2007) S. 21 und (Grimm, 2007).

[17] Siehe Kapitel 3.2.1.

Module	Beschreibung
Information Model	Alle Informationen und der Status werden in diesem Modul gehalten. Es besteht aus einer Hierarchie von Informationen und verwaltet den Lebenszyklus der Objekte in den Hierarchien.
XML Processing Model	Es handelt sich hierbei um die wichtigste und komplexeste Aufgabe: die Verarbeitung der SOAP-Nachrichten. Es wurde in ein separates Projekt ausgelagert, namens AXIOM.
SOAP Processing Model	Diese Komponente kontrolliert die Ausführung der Verarbeitung. Es werden hier verschiedene Phasen der Ausführung definiert, die durchlaufen werden müssen. Der Benutzer hat hier die Möglichkeit das Modul zu erweitern.
Deployment Model	Es ermöglicht es dem Benutzer Services zu verteilen, die Beförderung der Nachrichten zu konfigurieren und das SOAP Processing Model zu erweitern.
Client API	Dies ist die API, die es ermöglicht mit Web Services über Apache Axis 2 zu kommunizieren.
Transports	In Apache Axis 2 existiert ein Transport Framework was dem Benutzer ermöglicht, verschiedene Transportmöglichkeiten zu konfigurieren. Das Framework fügt sich in das SOAP Processing Model ein.

Tabelle 3.1: Kernmodule von Apache Axis 2

Module	Beschreibung
Code Generation	Dies ist ein Tool, womit es möglich ist, clienstseitigen- sowie serverseitigen Code generieren zu lassen.
Data Binding	Serialiserung und Deserialisierung

Tabelle 3.2: Zusatzmodule von Apache Axis 2

Descriptionhierarchie zu sehen. In der anschließenden Tabelle 3.3 werden die Hierarchien, des Information Models kurz erläutert. Dabei ist anzumerken, dass eine untergeordnete Konfiguration eine übergeordnete Konfiguration überschreiben kann.

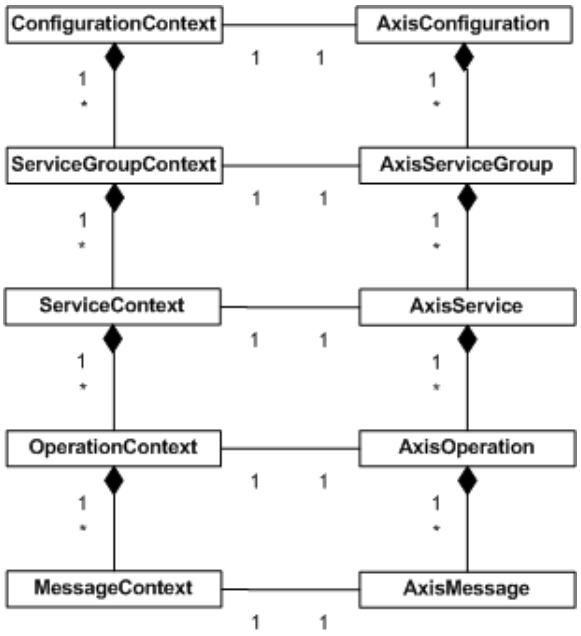

Abbildung 3.4: Beziehung zwischen Context- und Description-Hierarchie

Kontext	Beschreibung	Konfiguration	Beschreibung
Configuration-Context	Ist die Wurzel der Kontexthierachien und ein Container für globale Laufzeitinformationen	Axis-Configuration	Ist die Wurzel der Description-Hierarchie und dient als Container für Konfigurationsinformationen.
ServiceGroup-Context	Beinhaltet Informationen über die jeweiligen Servicegruppen	AxisService-Group	Diese Instanz repräsentiert eine Servicegruppe
Service-Context	Beinhaltet den Kontext eines Services und ist in dem jeweiligen Service verwendbar	AxisService	Beinhaltet Operationen und die Konfiguration des Servicelevels
Operation-Context	Beinhaltet Information über das Nachrichtenaustauschformat	AxisOperation	Beinhaltet Konfiguration des Operationlevels
Message-Context	Beinhaltet alle Informationen über die derzeitig ausgeführte Nachricht	AxisMessage	Beinhaltet statische Informationen, wie das Schema der Nachricht

Tabelle 3.3: Information Model Hierarchien in Apache Axis 2

In den nachfolgenden Abschnitten werden einige der bereits angesprochenen Module genauer betrachtet.

XML Processing Model

AXIOM war anfänglich ein Bestandteil von Apache Axis 2. Jedoch hatte sich herausgestellt, dass es auch als eigenständige Komponente in anderen Bereichen eingesetzt werden kann. AXIOM gilt als der Kern von Axis 2 und ist ein sogenanntes *On-Demand-Objektmodell*, um XML-Dateien in Java zu verarbeiten. Es basiert auf den *Pull-Parsern* und der StAX-API.
Beim Pull-Parsing liegt die Kontrolle des Parsens von XML-Infosets bei der Anwendung. Ein Vorteil, der daraus resultiert, ist, dass das Einlesen von Dokumenten manipuliert werden kann. Das kann sich bei großen Dokumenten als sehr effizient erweisen, da nicht das komplette Objektmodell im Speicher liegt.
AXIOM vereint die Vorteile von ereignisbasierten- und baumbasierten APIs. Ursprünglich war es als ein Mechanismus zur Sammlung und Zwischenspeicherung für StAX-Ereignisse vorgesehen. Mit Hilfe dieses Mechanismus war es möglich, dass diese Ereignisse später für die Verarbeitung wieder herangezogen werden können. Aufgrund der hohen Flexibilität, die dadurch geboten wird, wurde es zu einem kompletten Objektmodell für XML-Infosets aufgebaut.
Die wesentlichen Features von AXIOM sind u.a. die Leichtgewichtigkeit. Dies wurde durch eine geringe Tiefe der Klassenhierarchie sowie einer ebenfalls geringen Anzahl der Methoden und Attribute der Klassen erreicht. Somit ist ein niedrigerer Speicherplatzverbrauch gewährleistet. Des Weiteren stellt AXIOM ein Objektmodell dar, welches das XML-Infoset vollständig unterstützt. Das wichtigste Feature ist das sogenannte *deferred building*. Dies bedeutet, dass, wenn die Anwendung ein bestimmtes Element anfordert, liest AXIOM auch nur den Stream bis zu diesem bestimmten Element aus. Der Rest des Dokuments wird weiterhin im Stream gehalten. Dieses Feature ist gleichzeitig der Hauptunterschied zu anderen Objektmodellen, wie DOM oder JDOM, welche das komplette Dokument parsen.
Die Architektur von AXIOM stellt sich wie folgt dar: Es existiert ein Interface (*Builder*), dass für den Aufbau des Objektmodells zuständig ist. Für dieses Interface werden mehrere Implementierungen geliefert, die neben der Erstellung von spezifischen SOAP Nachrichten und der Generierung von Attachments, auch Ereignisse von SAX verarbeiten können. Für die Verwaltung der Objekmodelle wurden während der Entwicklungszeit mehrere Speichermodell ausgetestet. Es wurde sich schließlich für eine Variante entschieden, die mit verketteten Listen arbeitet. Jedoch hat der Entwickler die Möglichkeit sein eigenes Speichermodell zu integrieren. [18]

SOAP Processing Model

Bei der internen Verarbeitung von Nachrichten werden im Gegensatz zu Apache Axis 1 auf Server- sowie Clientseite nicht nur eine AxisEngine verwendet, sondern zwei. In Abbildung 3.5 ist diese Verarbeitung visualisiert. Eine Engine kann in Apache Axis 2 nur eingehende oder ausgehende Nachrichten verarbeiten. Wie bei Apache Axis 1 befindet sich auf dem Weg, den eine Nachricht durch die AxisEngine nimmt, eine Menge von Handlern, die nacheinander aufgerufen werden. Eine Handlerkette wird *Flow* genannt. Von diesen existieren in Apache Axis 2 vier verschiedene. Der *InFlow* ist die Handlerkette, die für die Verarbeitung von eingehenden Nachrichten zuständig ist. Der *OutFlow* ist demnach für die ausgehenden Nachrichten verantwortlich. Die beiden anderen Flows sind der *InFaultFlow* und der *OutFaultFlow*. Diese beiden Flows sind spezielle Flows von InFlow und OutFlow, die jeweils an deren Stelle treten, sobald die Nachricht einen Fehler beinhaltet. Die Nachrichten

[18] Vgl. (Foundation, 2009) und (Thilo Frotscher, 2007) Kapitel 9 Seite 111 - 135.

fließen in Form einer Instanz der Klasse MessageContext durch die Flows. Bei einem HTTP-Szenario empfängt das *AxisServlet* eingehende Nachrichten auf der Serverseite. Die Nachricht wird geparst und ein MessageContext erzeugt. Nach dem Parsen wird eine neue Instanz von AxisEngine erstellt und diese an die MessageContext-Instanz übergeben. Anhand der Konfiguration bestimmt die AxisEngine die Zusammensetzung des Flows. Wenn alle Handler des InFlows durchlaufen sind, wird durch die Engine eine neue MessageReciever-Instanz erstellt. Der MessageReciever ruft dann die Web Service Operationen auf. Aus den eventuellen Rückgabewerten des Web Services wird ein neuer MessageContext erstellt und der alte verworfen. Sind Rückgabewerte vorhanden, wird eine neue AxisEngine-Instanz durch den MessageReciever erzeugt. Er übergibt die MessageContext-Instanz an die neue AxisEngine. Nun befindet sich der MessageContext im Outflow. Am Ende der Kette existiert der *Transport Sender*. Er hat die Aufgabe, die Nachricht über das richtige Transportprotokoll zu dem Client zu schicken.
Auf der Clientseite ist die Verarbeitung von Nachrichten der der Serverseite ähnlich. Zuerst wird der ServiceStub aufgerufen. Dieser kann mit hilfe der WSDL generiert werden. Bei einer Request-Response Operation erzeugt der Stub[19] eine neue Instanz von *OutInAxisOperation*. Die Information, um welche Operation es sich handelt, nimmt der Stub aus der WSDL. Der Stub erzeugt ebenfalls eine MessageContext-Instanz und übergibt diese an OutInAxisOperation, die wiederum eine neue AxisEngine-Instanz erzeugt und den MessageContext übergibt. An diesem Punkt befindet sich der MessageContext bereits im OutFlow. Die AxisEngine erstellt wieder anhand der Konfiguration die Handlerkette und schickt den MessageContext durch diese. Am Ende des Outflows ruft die AxisEngine den Handler Transport Sender auf, welcher dann die Nachricht versendet. Bei einem Response des Servers wird eine durch OutInAxisOperation neue MessageContext-Instanz und anschließend eine neue AxisEngine erstellt. Die AxisEngine ermittelt wieder den InFlow anhand der Konfiguration und schickt MessageContext durch diese Kette. OutInAxisOperation gibt dann die Kontrolle an den Stub, wo der Inhalt von MessageContext weiter verarbeitet werden kann.
Im Falle eines Fehlers, zum Beispiel auf der Serverseite, wird ein *AxisFault* geworfen und die Verarbeitung des Requests sofort abgebrochen. Die Exception wird dann bis hin zum Servlet zurückgegeben. Dann erzeugt das AxisServlet einen neuen MessageContext und eine neue Instanz der AxisEngine für den Fehlerfall. Danach wird der OutFaultFlow eines SOAP-Faults erstellt und an den Client zurückgeschicken. Auf Clientseite kommt demnach ein InFaultFlow anstelle des InFlows zum Einsatz.
Es bleibt anzumerken, dass für jede SOAP Nachricht diese Abarbeitungsreihenfolge vollzogen wird.
Eine weitere Neuerung gegenüber Apache Axis 1 ist, dass Flows in *Phasen* unterteilt sind. Dieses Konzept dient dazu, Handler relativ zueinander anzuordnen. Somit ist eine Phase ein Teilschritt eines Flows, der wiederum beliebig viele Phasen enthalten kann. Phasen werden unterschieden in Systemphasen und benutzerdefinierte Phasen. Handler einer Systemphase stehen allen Services und Operationen zur Verfügung. Für Handler von benutzerdefinierten Phasen kann konfiguriert werden, ob sie allen Services zur Verfügung stehen oder nur bestimmten. Die Konfiguration der Flows und aus welchen Phasen sie sich zusammensetzen, geschieht in der *axis2.xml*. In der Standardeinstellung bietet Apache Axis 2 drei spezielle Handler an. Der *Dispatcher* findet den Service und die dazugehörige Operation. Er befindet sich im InFlow in der *Dispatch Phase*. Der *Message Reciever* konsumiert die SOAP-Nachricht und übergibt sie der Anwendung. Er ist der letzte Handler im InFlow. Der *Transport Sender* ist für das Verschicken der Nachrichten verantwortlich und ist somit der letzte

[19] Es können auch die Klassen *ServiceClient* oder *OperationClient* der Client-API verwendet werden.

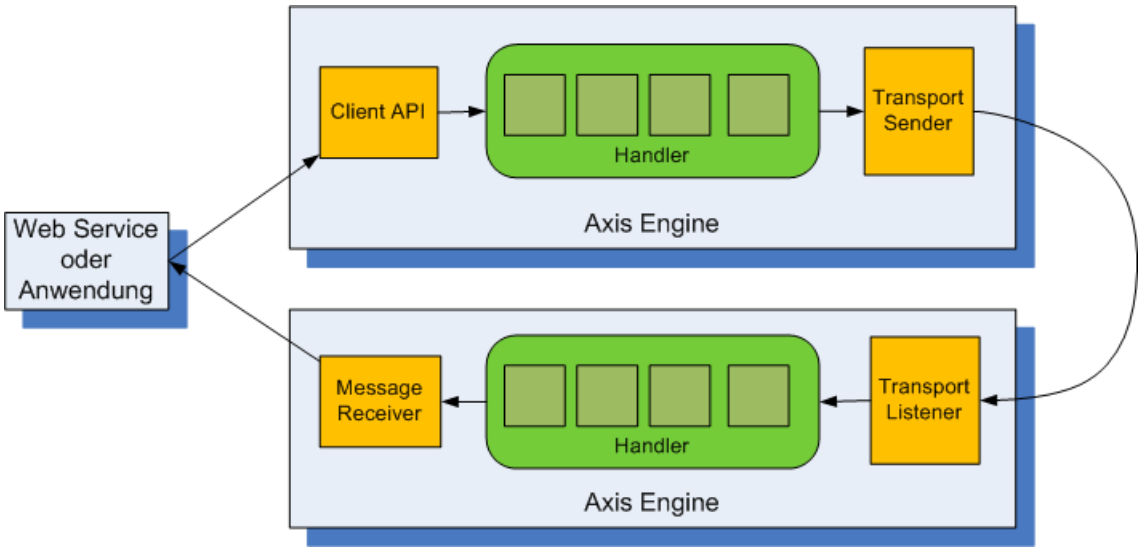

Abbildung 3.5: Verarbeitung von SOAP-Nachrichten in Apache Axis 2

Handler des OutFlows.
Eingehende und ausgehende Nachrichten werden der Reihe nach durch die in den Tabellen 3.4 und 3.5 ersichtlichen Phasen geleitet.

	Phasen	Beschreibung
1.	Transportphase	Handler verarbeiten transportspezifische Informationen, wie zum Beispiel die Validierung des Transportheaders und fügen wichtige Informationen zum MessageContect hinzu.
2.	Pre-Dispatch Phase	Befüllen des MessageContext, um Dispatching einzuleiten.
3.	Dispatch Phase	Sucht den korrekten Service und die dazugehörige Operation.
4.	benutzerdefinierte Phase	Hier werden die vom Entwickler hinzugefügten Handler durchlaufen.
5.	Nachrichten-Validierungsphase	Prüft, ob die Nachricht korrekt verarbeitet wurde.
6.	Nachrichten-Verarbeitungsphase	Ausführung der Business Logik der SOAP Nachricht.

Tabelle 3.4: Phasen einer eingehenden Nachricht in Apache Axis 2

Das SOAP Processing Model in Apache Axis 2 kann durch das Hinzufügen von Handlern und Modulen erweitert werden.[20]

3.2.2 Data Binding

Ein großer Vorteil von Apache Axis 2 ist die Erweiterbarkeit des Data Bindings. Es befindet sich nicht im Kern von Apache Axis 2. Somit ist es möglich, mit Hilfe des *Code Generator*

[20] Vgl. (Thilo Frotscher, 2007) Kapitel 9 Seite 245 - 263.

	Phasen	Beschreibung
1.	Nachrichten-Initialisierungsphase	Erste Phase des OutFlows.
2.	Benutzerphase	Führt Handler der benutzerdefinierten Phase aus.
3.	Transportphase	Senden der Nachricht zum Ziel.

Tabelle 3.5: Phasen einer ausgehenden Nachricht in Apache Axis 2

Frameworks andere Data Binding Frameworks problemlos einzubinden. In der Tabelle 3.6 sind die zur Verfügung stehenden Frameworks in Apache Axis 2 angegeben.

Frameworks	Beschreibung
ADB	Leichtgewichtiger Schema Compiler, der mit StAX arbeitet.
XMLBeans	Unterstützt vollständig XML Schema.
JAXB-RI	Ist die Referenzimplementierung der einzigen Spezifikation für XML Data Binding in Java.
JibX	Zeichnet sich durch seine hohe Performanz aus.

Tabelle 3.6: Zur Verfügung stehende Data Binding Frameworks in Apache Axis 2

Damit das Code Generator Framework diese verschiedenen Data Bindings ohne Veränderungen am Kern vorzunehmen einfügen kann, wurden sogenannte *Extensions* eingeführt. Dies sind Erweiterungen, die bestimmte Funktionalität, wie zum Beispiel Formatierung und Validierung von WSDL oder das Verarbeiten von Schema- Informationen, während der Generierung durchführen.
Extensions werden hauptsächlich eingesetzt, um Pre-Processing vor der Codegenerierung durch den Emitter durchzuführen und ist somit die zentrale Komponente des Generator-Frameworks und regelt den Generierungsprozess.
Während der Generierung werden alle Extensions der Reihe nach aufgerufen, ein Modell erzeugt und dieses dem sogenannten *Emitter* übergeben.
Damit das Generierungs-Framework arbeiten kann, wird eine Konfigurationsdatei namens *codegen-config.properties* benötigt. In ihr werden die Extensions, Emmitters und Templates, welche Anpassungen und Erweiterungen für verschiedene Programmiersprachen enthalten, konfiguriert. [21]

3.2.3 Features und Erweiterungen von Apache Axis 2

Entwicklung von RESTful Services

In Apache Axis 2 ist die Entwicklung und das Deployment von RESTful Services sehr einfach. Lediglich in der Konfigurationsdatei axis2.xml wird der Parameter *disableREST* auf *false* gestellt. Ein Ausschnitt dieser Konfigurationsdatei, mit dem REST-Abschnitt ist im folgenden Listing 3.2 dargestellt.

```
...
<parameter name="servicePath">services</parameter>
<parameter name="restPath">rest</parameter>
```

[21] Vgl. (Thilo Frotscher, 2007) Kapitel 9 Seite 309 - 354.

```
<!-- Following parameter will completely disable REST
   handling in Axis2-->
<parameter name="disableREST" locked="true">
   false
</parameter>
<parameter name="enableRESTInAxis2MainServlet"
                    locked="true">
   true
</parameter>
<parameter name="disableSeparateEndpointForREST"
                     locked="true">true</parameter>
...
```

Listing 3.2: Ausschnitt axis2.xml für REST Support

In dieser Einstellung wird der REST-Support in Apache Axis 2 freigeschalten, was jedoch bereits in der Standardeinstellung so konfiguriert ist. Dabei besagt der Wert *disableREST*, wenn er auf true gesetzt wird, dass REST-Support für beide Endpunkte deaktiviert ist. *enableRESTInAxis2MainServlet* besagt, wenn der Wert auf true steht, dass das AxisServlet unter */services/** auch REST-basierte Anfragen entgegen nimmt. *disableSeparateEndpointForREST* aktiviert bzw. deaktiviert einen separaten Endpunkt für REST-basierte Anfragen.[22]

Realisierung von Web Service Erweiterungen

Die Apache Axis 2 Engine kann im Prinzip nur einfache SOAP-Nachrichten verarbeiten. Um SOAP-Erweiterungen, wie zum Beispiel WS-Security oder WS-Addressing zu verwenden, wurde der Plugin-Mechanismus von Apache Axis 1 übernommen, der es erlaubt Erweiterungsmodule zu integrieren. Für alle anderen WS-* Spezifikationen, für die Apache Axis 2 standardmäßig kein Modul zur Verfügung stellt, können weitere Frameworks integriert werden. Diese werden im folgenden genannt:

WS-Coordination, WS-AtomicTransaction und WS-BusinessAcitivity:

- Wird durch Kandula2[23] implementiert.
- Bietet Interoperabilität mit anderen WS-Coordination, WS-AtomicTransaction und WS-BusinessAcitivity Implementierungen, speziell derer von .NET und IBM.

WS-ReliableMessaging:

- Wird durch Sandesha2[24] implementiert.
- Veröffentlich durch IBM, Microsoft, BEA und TIBCO.
- Wurde für Apache Axis 2 entwickelt.
- Bietet Interoperabilität mit anderen WS-ReliableMessaging Implementierungen.

WS-ResourceFrameworks, WS-BasNotification und WS-DistributedManagement:

- Wird durch Muse[25] implementiert.

[22] Vgl. (Thilo Frotscher, 2007) Kapitel 9 Seite 230 - 243.
[23] Siehe http://ws.apache.org/kandula/2/index.html/.
[24] Siehe http://ws.apache.org/sandesha/sandesha2/index.html/.
[25] Siehe http://ws.apache.org/muse/.

- Ist konform mit WS-Addressing und SOAP 1.2.

WS-Security:

- Wird durch Rampart[26] implementiert.
- Ist eine Erweiterung zu WSS4J.
- Unterstützt WS-Security 1.0, WS-Security 1.1, WS-Secure Conversation, WS-Security Policy 1.1, WS-Security Policy 1.2, WS-Trust, WS-Trust, WS-SX, SAML 1.2 und SAML 2.0-Issuance.
- Es wird zusätzlich das Paket *Apache XML Security* benötigt.

WS-Policy:

- Wird durch Neethi[27] implementiert.
- Ist auch integriert im Apche WS-Commons[28] Projekt.
- Auf der Clientseite werden, wenn die WSDL Policies enthält, im Code entsprechende Erweiterungen hinzugefügt.
- Auf der Serverseite wird das WSDL-Dokument um alle in der *axis.xml*-Datei befindlichen Policies erweitert.

addressing:

- Implementiert WS-Addressing.
- Ist bereits Standardmäßig im *modules*-Ordner von Axis 2 enthalten und ist mit allen anderen Web Services verfügbar.
- Auf Clientseite muss lediglich ein Repository erstellt werden, in dem das Modul hinzugefügt wird.

Nachrichtenoptimierung

Für die Nachrichtenoptimierung ist AXIOM in Apache Axis 2 verantwortlich. Dafür gibt es die Klasse *OMText*, die sowohl Text als *String* als auch Binärdaten entgegennehmen kann. Um ein OMText-Objekt für Attachment zu erzeugen, muss entweder ein Objekt vom Typ *DataHandler* oder eine Zeichenkette vom Typ *String*, der ein in Base64-kodiertes Attachment darstellt, übergeben werden. [29]

Deployment

Das Deployment von Services in Apache Axis 2 ist, ebenfalls wie in Apache Axis 1, unkompliziert. Jedoch unterscheidet sich der Mechanismus erheblich. Hier werden auch wieder mehrere Möglichkeiten geboten, Web Services zu veröffentlichen.
Eine davon ist, Services in der Web-Anwendung von Axis 2 bereitzustellen. Dies geschieht, indem im ersten Schritt die von Apache mit gelieferte Datei axis2.war im Web Service

[26] Siehe http://ws.apache.org/rampart/.
[27] Siehe http://ws.apache.org/commons/neethi/.
[28] Siehe http://ws.apache.org/commons/.
[29] Vgl. (Thilo Frotscher, 2007) Kapitel 9 Seite 409 - 444.

Container, wie z.B. Tomcat, integriert wird. Anschließend muss das Service-Archive, welches mit der Endung *Axis Archive (AAR)* versehen ist, in den Ordner *services* der Axis 2-Anwendung kopiert werden. Dabei bietet Axis 2 mit dem sogenannten *hot deployment* bzw. *hot update* eine Neuerung gegenüber Apache Axis 1. Ersteres bedeutet dabei, dass neue Service-Archive verwendet werden können, ohne dass die Web-Anwendung neu gestartet werden muss. Hot Update bezeichnet dabei die Möglichkeit Services zu aktualisieren ohne einen Serverneustart. Beide Features sind über die axis2.xml Datei ein- bzw. ausschaltbar.
Ein weitere Möglichkeit Services in Axis 2 zu veröffentlichen, wird über den mitgelieferten Standalone-Server *SimpleHTTPServer*, *TCPServer*, *SimpleMailListener* oder *SimpleAxis2Server* anwendbar. Diese können von der Kommandozeile gestartet werden und sollten jedoch nur bei der Entwicklung verwendet werden, da sie, wie der Präfix der Namen schon andeutet, simple sind und keine weiteren Sicherheitsmechanismen bieten.[30]

3.3 Metro 2.0

Metro ist ein Web Service Stack, der von Sun Microsystems und Microsoft entwickelt wurde. Der Hauptgrund für dieses Projekt war die Unterstützung der Interoperabilität zwischen Suns Java Web Services und Microsofts Windows Communication Foundation (WCF). Metro gilt als eine Kombination von Bibliotheken, die zu einem Web Service Stack zusammengefasst sind. Die Komponenten haben alle ihren eigenen Zuständigkeitsbereich und arbeiten zusammen. Die wesentlichsten sind dabei JAX-WS, Java Architecture for XML Binding (JAXB) und Web Services Interoperability Technology (WSIT). Zum Zeitpunkt der Erstellung dieses Kapitels war die aktuellste Version Metro 2.0. Von dieser Version wurden auch die Vergleiche aufgestellt.[31]

3.3.1 Architektur

In Abbildung 3.6 ist erkennbar, dass Metro 2.0 aus drei Ebenen, die sich auf zwei Komponenten verteilen, besteht. Die untersten beiden Ebenen Web Service Core und XML Processing werden durch die Referenzimplementierung von JAX-WS realisiert, Die obere durch WSIT. Durch WSIT ist es u.a. in Metro möglich, sichere Nachrichtenübertragung oder Verschlüsselung der Nachrichten zu implementieren.

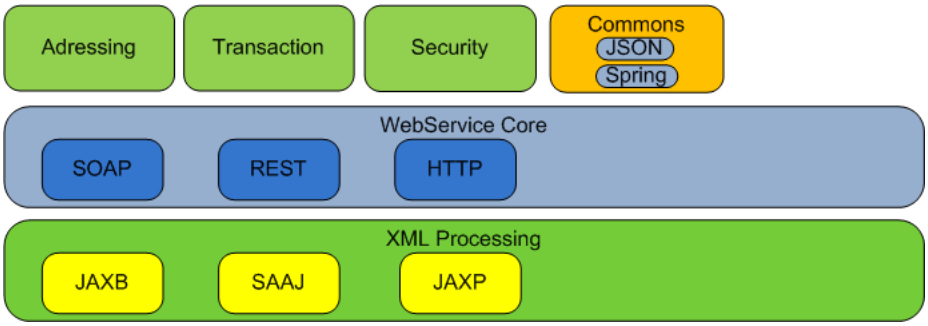

Abbildung 3.6: Metro Web Services Stack

[30] Vgl. (Thilo Frotscher, 2007) Kapitel 9 Seite 279 - 280.
[31] Vgl. (Microsystems, 2010) Kapitel 1.

Als Kern von Metro 2.0 gilt JAX-WS. Es implementiert die Basiskomponenten, wie SOAP-Nachrichtenverarbeitung, Transportmechanismen und Provider, um Services zu veröffentlichen. Es wurde in der Java Platform Enterprise Edition 5 (Java EE 5) eingeführt und ist der Nachfolger der JAX-RPC-Spezifikation. Ziel ist es den Aufwand der Entwicklung zu verringern, sowie das Deployment von Web Services zu vereinfachen. Es ist nachrichtenbasiert und unterstützt asynchronen Nachrichtenaustausch.
Um SOAP-Nachrichten zu erzeugen, benutzt JAX-WS die SOAP with Attachments API for Java (SAAJ). Um die Umwandlung von XML-Datentypen in Java-Datentypen vorzunehmen, delegiert JAX-WS an JAXB weiter. Web Services werden bei JAX-WS mit hilfe von Annotationen entwickelt. Jeder Service muss dabei entweder mit *@WebService* oder mit *@WebServiceProvider* annotiert sein (siehe Listing 3.3) .

```
@WebService(name = "SentencesService")
public class Sentences {
   @WebMethod()
   public String[] getSentences(
               @WebParam(name = "startId") int startId,
               @WebParam(name = "endId") int endId) {
      String[] result = null;

      try {
         result = Finder.getSencesByStartAndEndId(
               startId, endId);
      }
      catch (SQLException e) {
         throw new WebServiceException("error while
            databaserequest", e);
      }
      return result;
   }
}
```

Listing 3.3: Implementierung SOAP Services in Metro 2.0

In Tabelle 3.7 sind die verschiedenen Module, aus denen JAX-WS besteht aufgelistet.
Das Modul Runtime betrachtet mit Server und Client zwei Sichtweisen des Nachrichtenflusses in einer Metro Umgebung. Ein Aufruf eines Web Services beginnt entweder mit der Klasse *WSServletDelegate* oder *ServerConnectionImpl* und den dazugehörigen Informationen des Services. Danach werden die Informationen über die empfangene Nachricht in Form einer *MessageInfo*-Instanz gesammelt. Wenn die MessageInfo-Instanz mit allen nötigen Informationen befüllt wurde, wird ein *MessageDispatcher* erstellt. Es gibt zwei Arten des Dispatchers: den *SOAPMessageDispatcher* und den *XMLMessageDispatcher*, die die empfangenen Nachrichten manipulieren. Der SOAPMessageDispatcher konvertiert das MessageInfo-Objekt in eine SOAP-Nachricht und ruft anschließend die Handler auf. JAX-WS unterstützt für die Nachrichtenverarbeitung, genauso wie Apache Axis 1 und 2, das Konzept der Handler. Handler sind Module, die genutzt werden können, um die JAX-WS Runtime zu erweitern. Das Handler Framework wurde durch das JAX-WS Protokoll Binding implementiert. Es existiert sowohl auf der Client- als auch auf der Serverseite. Der Lebenszyklus der verschiedenen Handler wird durch JAX-WS kontrolliert. Die Handler sind

Modul	Beschreibung
runtime	Enthält den Web Service Kern.
tools	Tools, um zum Beispiel aus WSDL Dateien Services zu erzeugen.
APT	Tool, um Annotationen zu verarbeiten.
Annotation Processor	Aus APT um Java Quellcode Dateien wie Web Services zu veröffentlichen.
Runtime SPI und Tools SPI	Teil von JAX-WS, der einen Vertrag zwischen JAX-WS-RI und Java EE definiert.
JAXB XJC-API	Schema Compiler.
JAXB runtime-API	Teil von JAXB, der den Vertrag zwischen JAXB und JAX-WS definiert.

Tabelle 3.7: Hauptmodule von JAX-WS

als eine geordnete Liste organisiert, welche als Handlerkette bezeichnet wird. Dabei werden die Handler immer dann aufgerufen, sobald eine Nachricht gesendet oder empfangen wird. Eingehende Nachrichten werden von Handlern bearbeitet, die vor dem Binding in der Kette stehen, ausgehende Nachrichten von Handlern, die nach dem Binding existieren. Handler werden mittels MessageContext aufgerufen, der Methoden besitzt, um ein- bzw. ausgehende Nachrichten zu verarbeiten. Die Eigenschaften des MessageContexts können verwendet werden, damit Handler untereinander kommunizieren können.
In JAX-WS werden zwei Arten von Handlern beschrieben, *logische Handler* und *Protokollhandler*. Logische Handler verarbeiten ausschließlich XML-Nachrichten und sind unabhängig vom Protokoll-Binding. Protokollhandler sind für die Abarbeitung von protokollspezifischen Eigenschaften einer Nachricht verantwortlich. Somit hat z.B. ein Protokollhandler für SOAP über die SAAJ API Zugriff auf die Nachrichtenstruktur von SOAP. Das Data Binding für die Handler hat dabei die Aufgabe der Instanziierung, den Aufruf und die Zerstörung der Handler sowie dem Management des MessageContexts. Zusätzlich fällt auch der Empfang und das Versenden von ein- bzw. ausgehenden Nachrichten in den Zuständigkeitsbereich des Bindings. Nach der Verarbeitung der Nachricht durch die Handler wird diese über eine Instanz von *WSConnection* zurückgeliefert. Diese Instanz wird durch MessageInfo erzeugt.
Auf der Clientseite ist die Verarbeitung ähnlich. Es wird wieder ein MessageInfo-Objekt mit allen nötigen Informationen gefüllt und anschließend über den MessageDispatcher an einen *SOAPEncoder* weitergeleitet. Es ist anzumerken, dass sich die Dispatcher auf der Clientseite von denen auf der Serverseite unterscheiden. Der Dispatcher trifft noch die Entscheidung über synchronen oder asynchronen Austausch der Nachricht und ruft die konfigurierten Handler auf, um den Request schließlich über WSConnection zu verschicken. Der Empfang des Response hat dabei den gleichen Ablauf wie der auf der Seite des Servers.
Das Data Binding wird an JAXB weitergeleitet. Der Vorgang des Data Bindings wird im folgenden Kapitel beschrieben.[32]

3.3.2 Data Binding

Das Data Binding erfolgt in Metro 2.0 über JAXB. Es ist die einzige Spezifikation für XML-Data Binding in Java. Mittels JAXB ist es möglich, ein XML-Schema einzulesen und daraus

[32] Vgl. (Hansen, 2007) S. 43 - 52 und S. 311 - 355, (Microsystems, 2010) Kapitel 2.9, (Kotamraju, 2009) und (Rajiv Mordani, 2006).

automatisch Java-Klassen zu generieren. JAXB besteht wiederum aus mehreren Komponenten, die in Tabelle 3.8 aufgezeigt werden.

Komponente	Beschreibung
Binding Compiler	Kern der JAXB Verarbeitung, der ein XML Schema transformiert oder bindet.
JAXB Binding Framework	Enthält Schnittstellen für Serialisierung, Deserialisierung und Validierung von XML.
Schema Klassen	Klassen, die durch den Compiler erzeugt wurden.

Tabelle 3.8: Hauptmodule von JAXB

Wenn Java-Klassen generiert werden, wird der Schema Compiler *xjc* verwendet. Nach dem Aufruf transformiert er eine Schemadefinition in eine Menge von Java-Klassen und kompiliert diese. Diese Klassen repräsentieren dann die Struktur der Schemadatei. Zusätzliche Metadaten wie Namespaces oder Kardinalitäten sind als Annotationen in den generierten Java-Klassen zu finden. Beim Prozess des Einlesens der XML-Daten muss zuerst der *JAXB-Unmarshaller* generiert werden, der von einer *JAXB-Context*-Instanz ausgeliefert wird. Diese Instanz stellt den Startpunkt der JAXB-API dar und enthält die Binding Informationen zwischen XML und Java. Die Deserialisierung wird durch das JAXB Binding Framework vollzogen. Wenn dies abgeschlossen wurde, wird der sogenannte *Contentbaum* erzeugt. Dies ist ein Baum von Instanzen der Klassen, die durch den Binding Compiler erzeugt wurden. Die Instanzen der Klassen repräsentieren die Struktur des XML-Dokumentes. Der Prozess der Validierung des XML-Dokumentes ist optional in JAXB. Wenn er ausgeführt wird, geschieht dies vor dem Aufbau des Contentbaumes. Nach der Erstellung des Baumes kann die Anwendung den Inhalt des Dokumentes modifizieren und auslesen, indem sie den zuvor erstellten Contentbaum bearbeitet. Um die Objekte nun in ein XML-Dokument zu schreiben, wird *JAXB-Marshaller* benötigt. Dieser ist ebenfalls von der JAXB-Context-Instanz erhältlich. Der eingelesene Contentbaum kann nun wieder zurück in XML transformiert werden.
Es bleibt anzumerken, dass JAXB zwei Möglichkeiten der Validierung vorsieht: Die *Unmarshal-Time Validierung* ermöglicht es, Informationen, wie Fehler und Warnhinweise, die während der Deserialisierung auftraten, zu erhalten. Die zweite Möglichkeit ist die *On-Demand Validierung*. Sie liefert Fehler und Warnungen, die direkt im Contentbaum vorkamen. Während die Unmarshal-Time Validierung nur zu Beginn der Deserialisierung ausgeführt werden kann, kann die On-Demand Validierung immer angestoßen werden. Dies ist vor allem dann sinnvoll, wenn der Contentbaum modifiziert wurde.[33]

3.3.3 Features von Metro 2.0

Entwicklung von RESTful Services

Metro 2.0 bietet ebenfalls Unterstützung von RESTful Services, bei der wie bei der Entwicklung von SOAP Services auch auf die Spezifikation JAX-WS zurückgegriffen wird.[34]

```
@WebServiceProvider
```

[33] Vgl. (Amrhein, 2010; Mehta, 2003; Microsystems, 2009) und (Hansen, 2007) S. 52 - 72 und S. 195 - 262.

[34] Recherchen haben gezeigt, dass Metro 2.0 auch JAX-RS und Jersey unterstützt. Jedoch konnten diese Behauptungen nicht nachvollzogen werden. Die Annotationen und Objekte von JAX-RS waren Standardmäßig nicht in Metro 2.0 verfügbar.

```
@BindingType(value=HTTPBinding.HTTP_BINDING)
public class Sentences implements Provider<Source> {
   @Resource
   protected WebServiceContext m_WsContext;

   public Source invoke(
      @WebParam(name = "request") Source request)  {
   try  {
      MessageContext mc =
         m_WsContext.getMessageContext();
      String path =
         (String)mc.get(MessageContext.PATH_INFO);
      String method =
         (String)mc.get(MessageContext.HTTP_REQUEST_METHOD);
      String queryString =
         (String)mc.get(MessageContext.QUERY_STRING);

      String[] keyValuePairs = queryString.split("&");
      int startId = keyValuePairs[0];
      int  endId = params.get("endId");

      if (method.equals("GET"))
         return doGet(startId, endId);

      if (method.equals("POST"))
         return doPost(startId, endId, mc);

      if (method.equals("PUT"))
         return doPut(request, mc);

      if (method.equals("DELETE"))
         return doDelete(request, mc);
           ...
```

Listing 3.4: Implementierung RESTful Services in Metro 2.0

Das Codebeispiel (Listing 3.4) zeigt, dass die Ressource, die es zu veröffentlichen gilt, das Interface *Provider* implementieren muss. Die Provider Schnittstelle bietet eine einzige Methode an, die es zu überschreiben gilt:

```
T invoke(T request)
```

Sie wurde generisch gehalten, wodurch es möglich ist, die Klassen *Provider(javax.xml.transform.Source)* und *Provider(javax.xml.soap.SOAPMessage)* für SOAP/HTTP Binding oder *Provider(javax.activation.DataSource)* und *Provider(javax.xml.transform.Source)* für XML/HTTP Binding zu verwenden.
Die *@Resource* Annotation wird verwendet, damit JAX-WS den *WebServiceContext* zu der Ressource hinzufügen kann. *@BindingType(value=HTTPBinding.HTTP_BINDING)* wird benutzt, um zu zeigen, dass die Ressource Sentences mittels HTTP-Binding veröffentlicht wird. Dies ist das Gegenstück zu *SOAPBinding.SOAP11HTTP_BINDING* oder *SOAPBin-*

ding.SOAP12HTTP_BINDING.
Mit hilfe des MessageContext besteht die Möglichkeit, den Querystring und den Pfad einer HTTP-Anfrage auszulesen. Der MessageContext ist wiederum in WebServiceContext enthalten und wird von JAX-WS in die Ressource eingefügt.
Im letzten Schritt muss nun in der Konfigurationsdatei *sun-jaxws.xml* der Endpunkt bekanntgegeben werden. [35]

Realisierung der Web Service Erweiterungen

WSIT ist eine Ansammlung von implementierten Technologien, wie zum Beispiel sichere Nachrichtenübertragung und Bootstrapping. Es wurde anfänglich unter dem Namen *Projekt Tango* geführt und ist bis heute unter diesem Namen weit verbreitet. WSIT gilt als die Erweiterung zu JAX-WS und implementiert die in Abbildung 3.7 aufgezeigten Techniken.

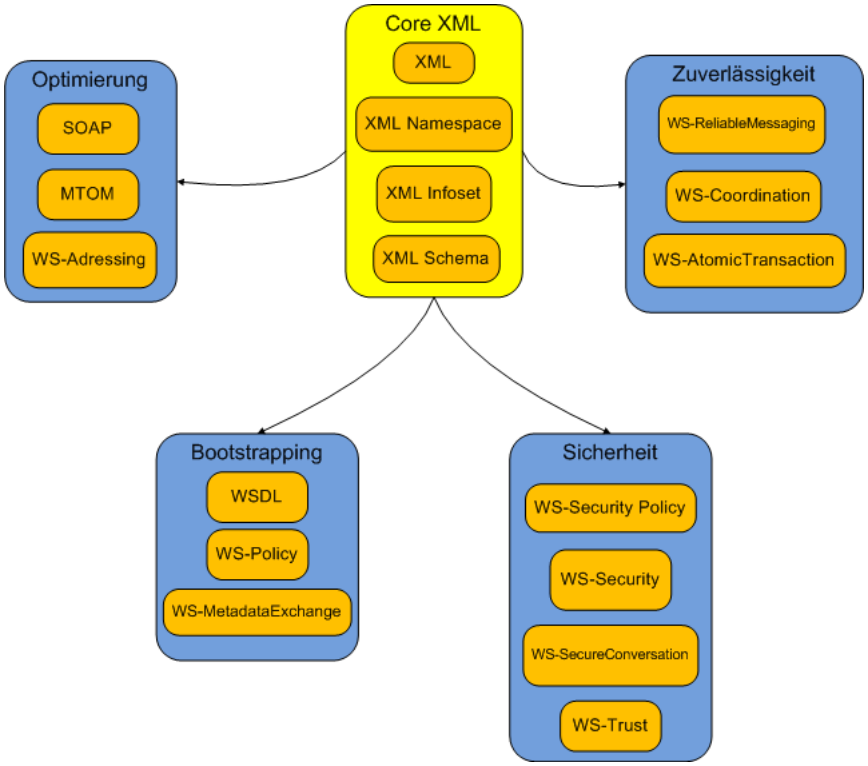

Abbildung 3.7: WSIT Web Service Features

Die Hauptaufgabe von WSIT ist die Sicherstellung der Interoperabilität der Services zu anderen Plattformen, wie zum Beispiel Microsofts WCF. Um die Technologien in Web Services bereitzustellen, benutzt es die bestehenden Modelle wie JAX-WS und Enterprise JavaBeans (EJB), um Sicherheit, Zuverlässigkeit und Transaktionen zu definieren. Dazu wird lediglich eine weitere Konfigurationsdatei verwendet. Die Namenskonvention der Datei ist im folgenden Listing gezeigt und muss sich entweder im META-INF- oder im WEB-INF-Ordner der Webanwendung befinden.

```
wsit-*.xml
```

Listing 3.5: Namenskonvention der Konfigurationsdatei

[35] Vgl. (Hansen, 2007) S. 85 - 136

Wie die Konfiguration vorzunehmen ist, wurde ausreichend dokumentiert. Des Weiteren bietet ein WSIT-Plugin für NetBeans die Möglichkeit die Konfiguration per Mausklick zu realisieren.[36] Für Eclipse existieren keine weiteren, nennenswerten Plugins. [37]

Nachrichtenoptimierung

WSIT realisiert die Nachrichtenoptimierung, indem MTOM und XOP implementiert werden. Dabei werden kleine Binärdaten im SOAP-Body verschickt; große Daten hingegen werden vom SOAP-Body extrahiert, kodiert und als Attachment in der SOAP-Nachricht gesendet. Auf der Empfängerseite werden die kodierten Daten dekodiert.

Deployment

Das Deployment für Web Services, die mit hilfe von Metro 2.0 erstellt wurden, erfordert einige Voraussetzungen. Die Services müssen in eine Web Application Archive (WAR)-Datei gepackt werden. Im WEB-INF Ordner existieren zwei Dateien. Die Datei web.xml stellt den Deploymentdescriptor für den Web Service Container wie Tomcat dar und die Datei *sun-jax-ws.xml* ist der Web Services Deploymentdescriptor für JAX-WS. Die kompilierten Javaklassen liegen im Unterordner *classes*. Das folgende Listing 3.6 zeigt ein Beispiel eines Deployment-Descriptors von JAX-WS.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints version="2.0" xmlns="http://java.sun.com/xml/ns/
    jax-ws/ri/runtime">
  <endpoint implementation="ServiceTypes.SoapServices.
      Sentences"
       name="SentencesService"
       url-pattern="/SentencesService"/>
  <endpoint name="SentencesResource"
       implementation="ServiceTypes.RestResources.Sentences"
       url-pattern="/SentencesResource/*"/>
</endpoints>
```

Listing 3.6: Beispiel der Konfigurationsdatei sun-jax-ws.xml

Das Element *endpoints* ist ein Container für einen oder mehrere Endpunkte, die es zu veröffentlichen gilt. Jeder Endpunkt repräsentiert einen Port der WSDL-Datei. In diesem Beispiel existieren zwei Endpunkte. Einer der SOAP- und einer der REST-basierte Anfragen entgegen nehmen kann. Das Element *endpoint* besitzt eine Attribut *implementation*, über das die Implementierung des Services angegeben wird. Die Implementierung, auf die referenziert wird, muss die *@WebService*-Annotation besitzen. Ein weiteres Attribut ist *name*, welches den Namen des Services definiert. Dieser muss mit dem angegebenen Namen in der eigentlichen Implementierung übereinstimmen. Ist dies nicht der Fall wird ein Fehler vom Sevrer zurückgeliefert und der Service kann nicht veröffentlicht werden.[38]

[36] Dokumentation zur Konfiguration der Sicherheitsfeatures: https://wsit-docs.dev.java.net/releases/1-0-FCS/WSITTutorial.pdf/.

[37] Vgl. (Sun Microsystems, 2007) und (Gupta, 2007b).

[38] Eine detailierter Beschreibung ist in https://metro.dev.java.net/guide/Deploying_Metro_endpoint.html zu finden.

3.4 Performanceevaluation der Web Service Frameworks

In diesem Abschnitt werden die Details der Implementierungen der Services, die Resultate sowie die Auswertungen der Tests dargestellt. Die aus den vorangegangenen Abschnitten erarbeiteten Unterschiede der Web Service Frameworks sind in der nachfolgenden Tabelle 3.9 zusammengefasst. [39]

	Apache Axis 1	Apache Axis 2	Metro 2.0
Data Binding	eigenes Typemapping Framework (Serialisierung, Deserialisierung auf Basis von SAX)	Unterstützung von ADB[40], JiBX[41], JAXB, JAXME, XML Beans[42]	JAXB
Unterstützung von WS-*	Integrierbar durch Module (Kandula, Kandesha, WSS4J)	Integrierbar durch Module (Kandula2, Muse, Sandesha2, Rampart, WS-Commons)	Bereits in WSIT enthalten
Unterstützung RESTful Services	Nein	Konfigurierbar in axis2.config	Implementierbar durch JAX-WS
Deployment	Automatisch (.jws), benutzerdefiniert (.war und server-config.wsdd)	Als .Axis Archive und .war, hot deployment und hot update	Als .war und sun-jaxws.xml, hot deployment
IDE Unterstützung	Mäßig in NetBenas 6.x und Eclipse	Mäßig in NetBeans 6.x und Eclipse	Sehr gut in NetBeans 6.x, mäßig in Eclipse
Java Unterstützung	Ab Version 1.4	Ab Version 1.4	Ab Version 1.5
Dokument-ation	Sehr gut	Sehr gut	Sehr gut
Open Source	Ja	Ja	Ja
Attachments	SwA	SwA und MTOM	SwA und MTOM
Erweiterbarkeit	Einfach, durch Entwicklung weiterer Handler bzw. Integration von Modulen	Einfach durch Entwicklung weiterer Handler bzw. Integration von Modulen	Schwierig durch Entwicklung weiterer Handler

Tabelle 3.9: Zusammenfassung der Features der Frameworks

Integration der Web Service Frameworks

Bei der Arbeit mit den Frameworks ist erwähnenswert, dass sie alle sehr einfach in den Tomcat Version 6.0.14 zu integrieren waren. Metro 2.0 bot eine build.xml an, die automatisch Metro 2.0 in Tomcat integrierte. Die Datei TOMCAT_HOME/conf/catalina.properties

[39] Diese Tabelle ist eine Zusammenfassung der in den vorangegangenen Abschnitten erarbeiteten Grundlagen. Aus diesem Grund wird von der Angabe von Quellen abgesehen.

musste vor dem Ausführen des Scriptes etwas abgeändert werden.[43] Bei Apache Axis 1 existierte ein Ordner namens *axis*, in dem Web Applikationsordner von Apache Axis 1. Dieser musste in den Web-Applikationsordner von Tomcat kopiert werden. Apache Axis 2 stellte eine *war*-Datei zur Verfügung, die ebenfalls in den Web-Applikationsordner von Tomcat kopiert werden musste. Diese wurde beim Start des Web Servers integriert.

SOAP-Dateien

Bei dem Request bzw. Response per SOAP gibt es keine gravierenden Unterschiede im Aufbau der SOAP-Dateien. Die einzelnen SOAP-Dateien der Web Service Frameworks sind im Anhang B zu sehen. Die einzigen Unterschiede, die erkennbar sind, sind die verwendeten Namespaces. Dies hat jedoch keine Auswirkung auf die Verarbeitung und soll aus diesem Grund auch nur erwähnt bleiben.

WSDL-Dateien

Beim Vergleich der drei WSDL-Dateien[44] fällt eines besonders auf: die Datei von Apache Axis 2 ist größer und somit auch umfangreicher als die Dateien von Metro 2.0 und Apache Axis 1. Der Grund dafür ist, dass in Apache Axis 2 standardmäßig auch die Schnittstelle für die REST-basierten Anfragen angegeben ist. Des Weiteren veröffentlich Axis 2 einen Endpunkt für SOAP Version 1.1 und SOAP Version 1.2. Auch dies ist in der Konfigurationsdatei axis.xml konfigurierbar und hat keinerlei Auswirkungen auf die Performance. Dennoch sind auch hier, ähnlich zu den SOAP-Dateien, keine weiteren Unterschiede für einfache WSDL-Dateien ersichtlich.

3.4.1 Service-Implementierung

Zur Durchführung der Load- bzw. Stresstests wurden drei Web-Anwendungen entwickelt. Jede dieser Anwendungen basierte auf einem anderen Framework und stellte eine SOAP-Schnittstelle zur Verfügung. Die beiden Anwendungen auf Basis von Apache Axis 2 sowie auf Metro 2.0 veröffentlichten zusätzlich einen RESTful-Service, da noch einmal Tests zum Vergleich von SOAP und REST durchgeführt werden sollten. Diese Tests hatten die Aufgabe, noch repräsentativer, als die aus dem zweiten Kapitel, zu sein, da hier sowohl SOAP- als auch RESTful-Service das gleiche Framework verwenden, um Anfragen zu empfangen bzw. Antworten zu verschicken.
Die SOAP-Serviceimplementierungen der drei Kernel als auch die beiden RESTful Services arbeiteten dabei alle nach der gleichen Logik. Sie konnten mit zwei Parameter aufgerufen werden. Diese Parameter bildeten dabei ein Intervall, mit dessen Hilfe die Sätze in der Tabelle *sentences* der Datenbank *TLG* identifiziert wurden.
Der Client ruft nun einen Service über die veröffentlichte Schnittstelle auf. Das Framework nimmt die Anfrage entgegen, verarbeitet sie und leitet sie an die Implementierung des eigentlichen Services weiter. Dieser ruft anschließend eine Methode auf, die eine Datenbankabfrage startet. Ein *ResultSet*-Objekt wird mit den Ergebnis dieser Abfrage befüllt und in einer Schleife durchlaufen. Jeder Schleifendurchlauf ließt nun den eigentlichen Satz aus dem Objekt aus und speichert es in einem *String-Array*. Die Methode, das komplette Ergebnis in

[43] Die genauen Schritte zur Integration von Metro 2.0 in Tomcat können in (Gupta, 2007a) nachgelesen werden.
[44] Siehe Anhang A.

einem String-Array zu speichern, ist nicht effizient, da der Speicher für große Datenmengen sehr belastet wird. Ein effektiverer Ansatz wäre hier mit Streams o.ä. zu arbeiten. Diese Methode der Verarbeitung wurde gewählt, um zum einen die Logik der Verearbeitung einheitlich zu entwickeln und zu anderen, damit der Speicher während der Verarbeitung richtig ausgelastet wird. Das befüllte String-Array wurde dann an das Framework zur internen Verarbeitung weitergereicht und anschließend an den Client geschickt. Dieser Sachverhalt ist noch einmal in der Abbildung 3.8 anhand eines Klassendiagramms verdeutlicht.

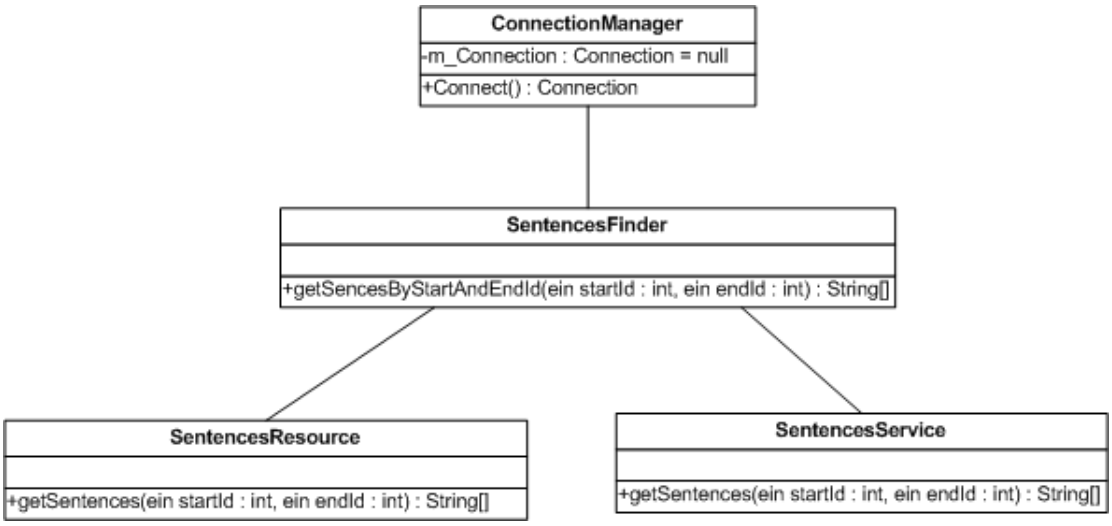

Abbildung 3.8: Klassendiagramm eines Services und einer Ressource

Als Data Binding Komponente wurde für den Kernel auf Basis von Apache Axis 1 das von Axis 1 implementierte Typemapping-Framework, für den Kernel auf Basis von Apache Axis 2 Axis Data Binding (ADB) und für den Kernel auf Basis von Metro 2.0 JAXB, verwendet.

3.4.2 Versuchsergebnisse

In den nachfolgenden Abschnitten werden die jeweiligen Ergebnisse der Tests sowie deren Auswertung präsentiert. Dabei ist anzumerken, dass hier drei verschiedene Tests, die in den jeweiligen Unterabschnitten erläutert werden, durchgeführt wurden.
Zur Verifizierung der Testergebnisse wurden die Tests mehrfach und, wo dies möglich war, auf verschiedenen Arten durchgeführt.

Versuchsergebnisse der Loadtests

Im folgenden sind die Ergebnisse der Tests tabellarisch aufgezeigt. Der Client wurde mit JMeter erzeugt. Dieser stellte sequentiell eine neue Abfrage an eine der drei Web-Anwendungen. Dabei variierte die Größe des angefragten Intervalls, zwischen 1-, 100-, 1.000-, 10.000- und 100.000 Elementen, welche jeweils 1.000 Mal ausgeführt wurden.
Die Gesamtverarbeitungszeit ist dabei die Zeit, die benötigt wird, einen kompletten Request-Response-Zyklus zu durchlaufen. Sie gibt somit an, wie viel Zeit vom Senden des Request bis zum Empfang des Responses vom Server vergeht. Die Serververarbeitungszeit ist die Zeit, die der Server benötigt, um eine der Anfrage entsprechende Antwort zu generieren. Hier fallen z.B. auch die Zeiten der Datenbankabfragen oder der Berechnung mit hinein. Der Durchsatz hingegen spiegelt die gestellten Anfragen pro Sekunde wider.

Anzahl der Elemente	Gesamtver-arbeitungszeit (in ms)	Serverver-arbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)
1	15	4	53,3
100	21	14	41,5
1.000	92	65	10,5
10.000	1.637	830	0,6
100.000	29.286	17.560	0,03

Tabelle 3.10: Versuchsergebnisse der Loadtests von Apache Axis 1

Anzahl der Elemente	Gesamtver-arbeitungszeit (in ms)	Serverver-arbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)
1	15	4	54,5
100	15	7	54,5
1.000	36	26	26,4
10.000	274	218	3,6
100.000	6.203	1.646	0,2

Tabelle 3.11: Versuchsergebnisse der Loadtests von Apache Axis 2

Anzahl der Elemente	Gesamtver-arbeitungszeit (in ms)	Serverver-arbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)
1	8	3	90,6
100	17	6	49,1
1.000	33	24	27,7
10.000	292	214	3,6
100.000	5.992	1.692	0,2

Tabelle 3.12: Versuchsergebnisse der Loadtests von Metro 2.0

Die Ergebnisse dieses Loadtests lieferten bereits eine wichtige Erkenntnis. Apache Axis 1 ist veraltet und es lohnt sich den eingangs dieser Arbeit erwähnten SOAP-Kernel, der auf Apache Axis 1 beruht, neu mit Apache Axis 2 bzw. Metro 2.0 zu entwickeln.
Bei einer Anfrage mit einem Element ist noch kein Unterschied zwischen Apache Axis 1 und 2, bzw. Metro ersichtlich. Aber schon bei 100 angefragten Elementen arbeitete Axis 2 und Metro 2.0 bis zu 50 % schneller als Axis 1. Bei 1.000 Anfragen vergrößerte sich die Gesamtverarbeitungszeit von Apache Axis 1 im Verhältnis zu Apache Axis 2 und Metro 2.0 auf 60%, bei 10.000 Anfragen auf 75% und bei 100.000 Anfragen sogar auf 80%. Die unterschiedlichen Zeiten zwischen Apache Axis 2 und Metro 2.0 können bei der Auswertung vernachlässigt werden, da die Tests der beiden Frameworks mit verschiedenen Data Binding Komponenten durchgeführt wurden.
Die Gründe für die langsamere Verarbeitung der Anfragen bei Apache Axis 1 liegen zum einen an der Implementierung des Typemapping-Framework. Zum anderen ist es das von Apache Axis 1 verwendeten Handlerkonzept. Dieses ist vermutlich schlechter umgesetzt als bei Apache Axis 2 und Metro 2.0.

Versuchsergebnisse der Stresstests

Die Stresstests verliefen stark abweichend zu den oben genannten Loadtests. Hier wurde alle 12 Minuten ein weiterer Benutzer, der ein Request stellte, sobald er ein Response vom Server erhalten hatte, erzeugt. Die Requests hatten dabei alle eine konstante Größe und forderten 1000 Elemente vom Service an. Nach 41 Stunden wurden die Testdurchläufe beendet. Die Anzahl der Nutzer war zu diesem Zeitpunkt auf 200 angestiegen. In den nachfolgenden Tabellen 3.13 und 3.14 sind die Statistiken der Stresstests auf der Serverseite dargestellt.

Getestet wurden bei den Stresstests:

- die Anzahl der gestellten Requests pro Minute,
- die durchschnittliche Verarbeitungszeit eines Requests,
- die aufgetretenen Systemfehler während der Verarbeitung,
- die aktiven Benutzer auf dem Server,
- der Speicherverbrauch der Web-Anwendung und
- die Auslastung der CPU während der Verarbeitung der Requests.

Bei der Auswertung fällt vor allem auf, dass auch hier die Ergebnisse der Services, die mit Apache Axis 1 erzeugt wurden, eine sehr viel schlechtere Performance aufweisen als die von Metro 2.0 und Apache Axis 2.
Die maximale Requestrate bei Apache Axis 1 liegt bei ca. 1.600 Anfragen, bei Apache Axis 2 bei ca. 6.500 und bei Metro 2.0 bei ca. 7.500. Die Requestrate spiegelt dabei die pro Minute verarbeiteten Anfragen des Servers wieder und beinhaltet auch die fehlerhaften Anfragen, die im Diagramm der Systemfehler dargestellt sind.
Die durchschnittliche Verarbeitungszeit ist die Zeit, die der Server vom Empfang des Requests bis zum Verschicken der Antwort benötigt. Aus den Diagrammen geht hervor, dass Apache Axis 1 nach 2.500 Minuten durchschnittlich ca. 4.000 Millisekunden für die Verarbeitung der Anfrage benötigt. Apache Axis 2 braucht dagegen maximal durchschnittlich ca. 1.600 Millisekunden und Metro 2.0 durchschnittlich ca. 1.250 Millisekunden.
Die Messung der Systemfehler ergibt, dass der erste Fehler bei Apache Axis 1 nach 1.700 Minuten auftritt. Bei Apache Axis 2 und Metro 2.0 treten die ersten Fehler schon nach 1.500 Minuten auf. Der Grund dafür liegt in der Anzahl der zu verarbeiteten Requests, denn Apache Axis 2 und Metro 2.0 verarbeiten zu diesem Zeitpunkt schon die vierfache Anzahl an Requests pro Minute.

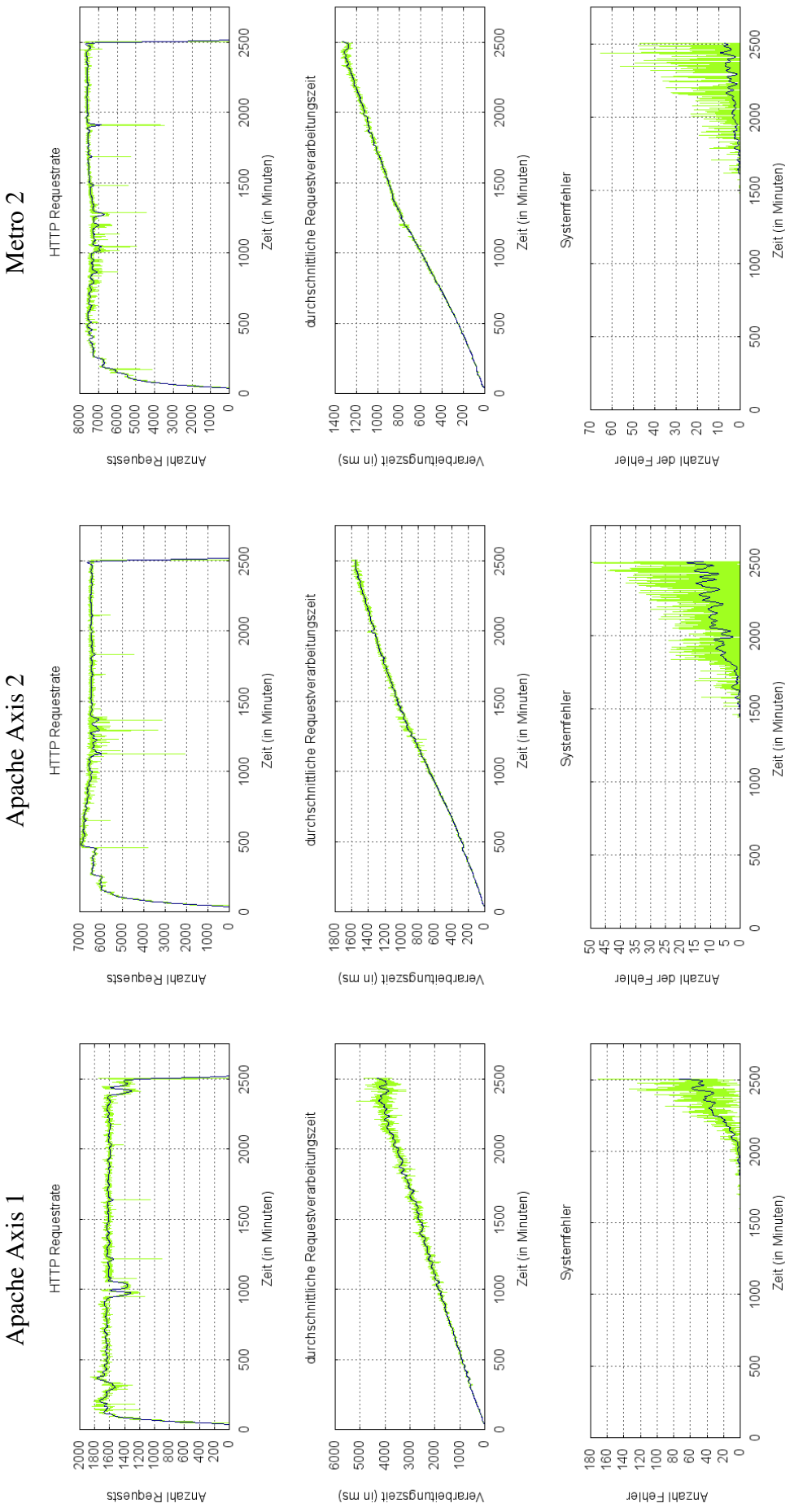

Tabelle 3.13: Statistiken der Web Service Kernel beim Stresstest (1)

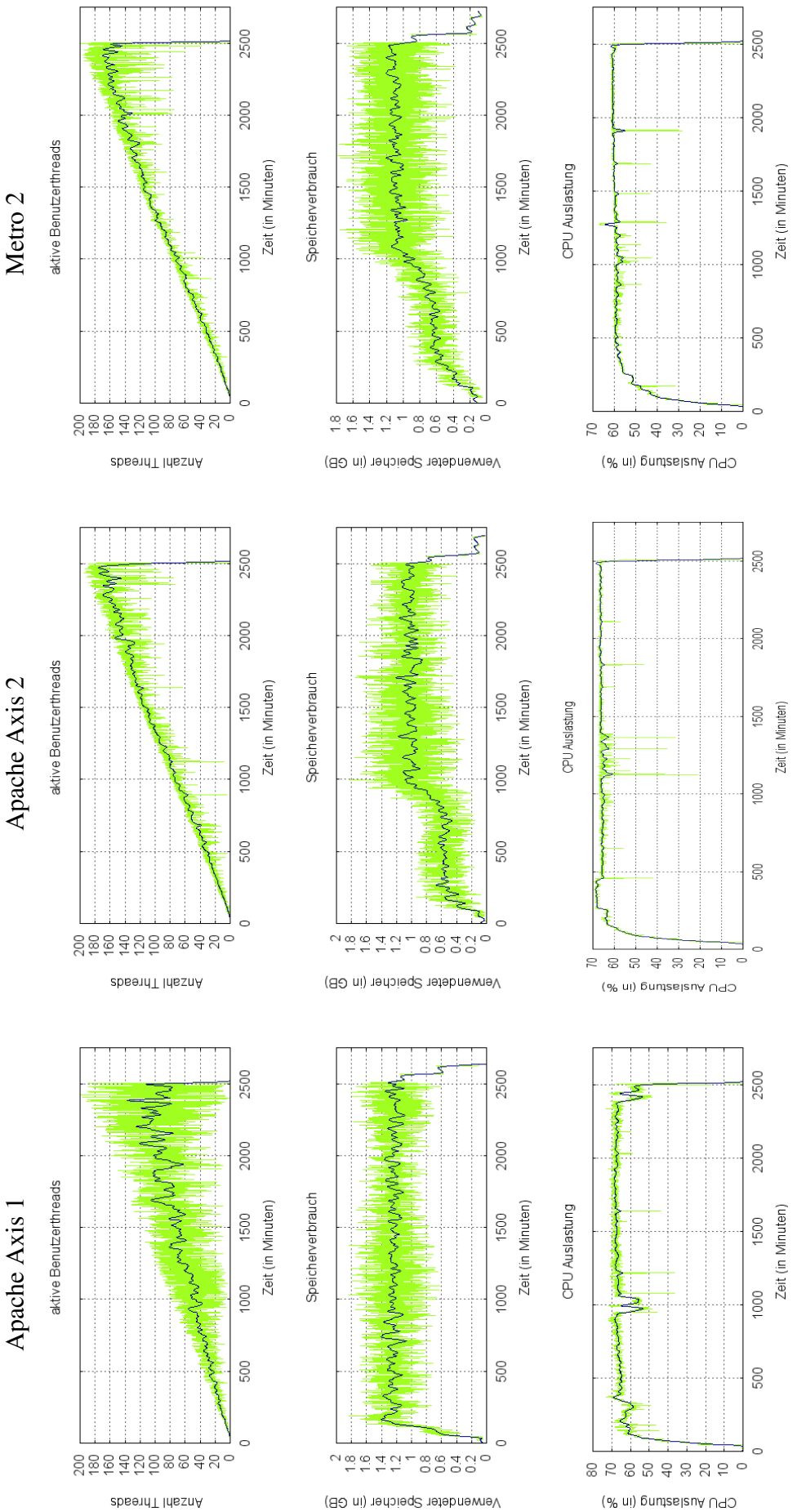

Tabelle 3.14: Statistiken der Web Service Kernel beim Stresstest (2)

Die Gesamtanzahl der Requests geht aus der Tabelle 3.15 hervor:

	Apache Axis 1	Apache Axis 2	Metro 2.0
Gesamtanzahl der Requests	3.927.466	15.691.734	17.905.524
durchschnittliche Verarbeitungszeit des Servers (in ms)	2214	850	743
Gesamtverarbeitungszeit (in ms)	3121	940	812
Anzahl der Fehler	13.542	6.452	11.064
durchschnittliche Fehlerquote (in %)	0,4	0,04	0,06

Tabelle 3.15: Zusammenfassung der Rampentestergebnisse

Obwohl Apache Axis 1 weniger Requests pro Minute verarbeitet, liegt die Fehlerquote deutlich höher als bei den andern beiden Web Service Frameworks.
Der maximale Speicher für Tomcat wurde auf 2.048 MB festgelegt. Bei Apache Axis 1 wird der vorher festgelegte Speicher sofort ausgenutzt. Apache Axis 2 und Metro 2.0 sind sich hierbei wieder ähnlich. Der Speicher wird bei diesen Web Services erst nach 1.000 Minuten vollständig ausgelastet.
Eines haben alle drei Web Service Frameworks gemeinsam: die Auslastung der CPU ist bereits nach 10 Nutzern voll erreicht. Der Grund dafür ist, dass die Tests in einer relativ homogenen Umgebung durchgeführt wurden. Server und Client befanden sich im gleichen Netzwerk, was zur Folge hatte, dass die Anfragen und Antworten schneller empfangen und verschickt werden konnten. Somit hatte der Server nicht genügend Zeit, um sich zu erholen.

Vergleich von SOAP und REST innerhalb von Apache Axis 2 und Metro 2.0

Wie am Ende des Kapitels 2.4 erwähnt, wurden die Kernel mit unterschiedlichen Frameworks entwickelt. Aus diesem Grund konnte die Vermutung aufgestellt werden, dass die gravierenden Performanceunterschiede zwischen SOAP und REST auch auf die unterschiedliche Arbeitsweise der Frameworks zurückzuführen ist. Diese Testergebnisse sollten lediglich einen Einblick über die Performanceunterschiede zwischen SOAP und REST geben. Um die Ergebnisse der Tests repräsentativ zu gestalten, wurden noch einmal zwei Kernel entwickelt, die jeweils einen SOAP- und einen RESTful-Service beinhalten. Der eine Kernel basierte auf Apache Axis 2 und der andere auf Metro 2.0. Somit konnte eine unterschiedliche Verarbeitung von SOAP- und REST-basierten Anfragen ausgeschlossen werden. In den Tabellen 3.16 und 3.17 ist das Resultat des erneuten SOAP-REST-Vergeleiches ersichtlich. Die eingeklammerten Werte repräsentieren dabei die Ergebnisse der SOAP-Tests. [45]

Anzahl der Elemente	Gesamtverarbeitungszeit (in ms)	Serververarbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)
1	6 (15)	3 (4)	151,2 (54,5)
100	8 (15)	6 (7)	111,6 (54,5)
1.000	25 (36)	24 (26)	38,6 (26,4)
10.000	228 (274)	(216) (218)	4,4 (3,6)
100.000	2.150 (6.203)	1.788 (1.646)	0,4 (0,2)

Tabelle 3.16: Versuchsergebnisse der Loadtests von SOAP und REST in Apache Axis 2

[45] Da der Test von SOAP und REST genauso wie die Loadtests erfolgte, wird hier von einer erneuten erläuterung abgesehen.

Anzahl der Elemente	Gesamtverarbeitungszeit (in ms)	Serververarbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)
1	8 (8)	6 (3)	106,5 (90,6)
100	11 (17)	10 (6)	83,4 (49,1)
1.000	33 (33)	30 (24)	29,9 (29,9)
10.000	269 (292)	263 (214)	3,7 (3,7)
100.000	2.867 (5.992)	2.628 (1.692)	0,4 (0,2)

Tabelle 3.17: Versuchsergebnisse der Loadtests von SOAP und REST in Metro 2.0

Die Testergebnisse ergaben ein eindeutiges Resultat. REST-basierte Anfragen hatten sowohl auf Basis von Apache Axis 2 als auch von Metro 2.0 eine bessere Performance. Bei Metro 2.0 waren die REST-basierten Anfragen bis zu 44% schneller als die SOAP-basierten; bei Apache Axis 2 sogar bis zu 65% schneller. Beim Vergleich der Serververarbeitungszeiten stellte sich heraus, dass sich diese bis auf wenige Millisekunden ähnelten. Aus dieser Tatsache heraus kann das Fazit gezogen werden, dass SOAP allein durch seinen Overhead an Daten eine schlechte Performance aufweist. Einen wichtigen Unterschied zur internen Verarbeitung von REST-basierten Nachrichten gibt es zwischen Axis 2 und Metro 2.0. Axis 2 packt diese in einen SOAP-Umschlag. Dieser Vorgang wird durchgeführt, da die Handler nur SOAP-Nachrichten verarbeiten können. Somit ist der Ansatz in Axis 2 nicht vollständig REST-basiert. Der direkte Vergleich der Verarbeitungszeiten der REST-basierten Anfragen bei Apache Axis 2 und Metro 2.0 sowie der Vergleich der Serververarbeitungszeiten von Metro 2.0 für REST und für SOAP zeigt, dass die Anfragen bei Metro 2.0 etwas langsamer sind. Das kann darauf zurückgeführt werden, dass die Verarbeitung der Anfragen in Metro 2.0 durch den Entwickler extra implementiert werden muss und diese Implementierung Performanceschwächen aufweist. Diese Ergebnisse bestätigen somit die Resultate aus Kapitel 2.4.

3.5 Zusammenfassung

In diesem Kapitel war die Hauptaufgabe, eine Entscheidungsgrundlage für die Neuentwicklung des bestehenden SOAP-Kernels zu finden. Apache Axis 1 hat wesentlich schlechter abgeschnitten als Apache Axis 2 und Metro 2.0. Jedoch nicht nur bei den Tests der Performance, auch bei den Erweiterungsmöglichkeiten durch bestehende Module ist Apache Axis 1 rückständig.
Aus den oben genannten Gründen ist eine Neuentwicklung des bestehenden SOAP-Kernels auf Basis eines anderen Frameworks anzustreben. Die Entscheidung darüber, ob für die Neuentwicklung Metro 2.0 oder Apache Axis 2 verwendet werden sollte, ist abzuwägen.
Zum einen ist Apache Axis 2 ein weit verbreitetes und sehr flexibles Web Service Framework, was jedoch eher schlecht durch Plugins für IDEs unterstützt wird. Zum anderen ist Metro 2.0 ein vergleichsweises starres Framework mit einer sehr guten IDE-Unterstützung in NetBeans 6.x und ebenfalls sehr verbreitet.
Hier muss der Entwickler abwägen, ob die Services mit dem .NET Framework kommunizieren sollen oder eine Vielzahl von WS-* Spezifikationen benötigt werden. In diesem Fall sollte eher Metro 2.0 verwendet werden, da es eine bewiesene Interoperabilität zu .NET besitzt, sowie die Module für die WS-* Spezifikation bereits integriert. Wenn dies nicht der Fall ist, so sollte der neue SOAP Kernel eher auf Basis von Apache Axis 2 entwickelt wer-

den, da dies der Nachfolger von Apache Axis 1 ist und aufgrund der Vorerfahrung mit der Entwicklung die Lernkurve wesentlich geringer ausfallen kann.

4 Programmierschnittstellen für XML

4.1 Document Object Model

Document Object Model (DOM) ist eine Spezifikation einer Schnittstelle, die den Zugriff auf Web Dokumente definiert. Sie gehört zu der Gruppe der sogenannten *baumbasierten Application Programming Interface (API)*. Diese laden das komplette Dokument in den Speicher und bauen dort aus diesem ein Objektmodell, das die Struktur des Dokuments aufweist, auf. Es wurde vom W3-Consortium standardisert und definiert einen Satz von Schnittstellen für die einzelnen Knotentypen des Baumes. Die Schnittstellen sind sprachenunabhängig und wurden mithilfe der Interface Definition Language (IDL) der Object Management Group (OMG) definiert.
DOM betrachtet ein XML-Dokument nicht nur als eine Menge von Zeichen, sondern auch als eine geordnete Menge von Objekten. Dafür stellt DOM für alle möglichen Komponenten eines Dokuments eine abstrakte Objektklasse mit einer genau definierten Schnittstelle zur Verfügung. Ein DOM-Objektmodell beschreibt unter anderem die Beziehung der einzelnen Objektklassen zueinander, welche und wie viel Unterklassen diese haben dürfen und ob die Eigenschaften wie Attribute oder Elementinhalte verändert werden dürfen. Die Objektklassen entsprechen dabei einem bestimmten Knotentyp und stellen Methoden zur Verfügung, mit denen die Attribute und die Namen eines Elementes festgestellt oder manipuliert werden können. [1]

4.1.1 Standadisierungen von DOM

DOM ist seit 1998 ein Standard und wurde seitdem immer wieder aktualisiert und erweitert. Die verschiedenen Versionen werden in Levels beschrieben. Nachfolgend ist eine Übersicht über die Standardisierung[2] von DOM dargestellt:

DOM Level 0: Ist keine W3C-Spezifikation. Es ist lediglich eine Definition der Funktionalität.

DOM Level 1: Baut auf DOM Level 0 auf und legt den Fokus auf HTML- und XML-Dokument-Modelle.

DOM Core: Definiert das Bewegen in einen DOM-Baum und die Manipulation von Knoten.

DOM HTML: Ist die Erweiterung für den Zugriff auf HTML-Dokumente.

[1] Vgl. (Vonhoegen, 2007) Seite 253 - Seite 267 und (W3C, 2005).
[2] Siehe (w3schools.com, 2010).

DOM Level 2: Enthält Erweiterungen für DOM-Core für u. a. XML-Namensraum-Unterstützung, für DOM-HTML für u. a. XHTML-Dokumente und definiert neue Elemente:

DOM Level 2 Style: Für das Manipulieren der Style Informationen im Dokument.

DOM Level 2 CSS: Für die Manipulation von Layoutinformationen.

DOM Level 2 Views: Definiert den Zugriff auf konkrete Wiedergabearten eines Dokumentes (z. B. Bilder).

DOM Level 2 Events: Definiert ein Eventmodell und standardisiert die Verarbeitung von Ereignissen im Dokument (z. B. Benutzerinteraktion).

DOM Level 2 Traversal und DOM Level 2 Range: Definieren das Durchlaufen des Knotenbaums anhand von bestimmten Auswahlkriterien und das Arbeiten mit Bereichen im Dokument, die bestimmte Elemente und Textknoten umfassen.

DOM Level 3: Erweiterung von DOM Core. Umfasst u. a. verbesserte Ausnahmebehandlung und Umgang mit Zeichenkodierungen. Es wird ein Contentmodel und Dokumentvalidierung spezifiziert. Zusätzlich wird das Laden und Sichern von Dokumenten, Dokument-Views, Dokument-Formatierung und Schlüsselevents definiert.

DOM Level 3 Core: Spezifiziert den Zugriff und die Manipulation von Inhalt, Struktur und Style in Dokumenten.

DOM Level 3 Events: Erweitert die Funktionalität aus DOM Level 2 Events, um neue Interfaces und neue Events.

DOM Level 3 Views und DOM Level 3 Formatting: Erlauben es, dynamisch auf den Inhalt, Struktur und Style zuzugreifen und diese zu ändern.

DOM Level 3 Load und DOM Level 3 Save: Ermöglicht die Serialisierung von Dokumenten oder Dokumentteilen sowie das Parsen von XML-Dokumenten in Zeichenketten in Dokument-Objekten.

DOM Level 3 XPath: Erlaubt das Auswählen von Knoten anhand von XPath-Ausdrücken.

DOM Level 3 Validation: Erlaubt das Prüfen, ob nach einer dynamischen Änderung (Hinzufügen oder Entfernen von Knoten) das DOM-Dokument valide bleibt.

4.1.2 DOM Schnittstellen und Knotentypen

Ein DOM-Parser baut im Speicher einen, dem XML-Dokument entsprechenden Baum aus Objekten auf. Dabei wird jedem erstellten Knoten gleichzeitig der entsprechende Knotentyp zugeordnet und festgelegt, welche Merkmale und Funktionen jeder Knoten besitzt. *Document* bildet die Wurzel des Baumes und repräsentiert das gesamte Dokument. Dieser Knoten ist der einzige, der keinen Elternknoten besitzt. Das Document-Interface erbt vom *Node*-Interface, dem zentralen Punkt von DOM. Es wird als Basisschnittstelle für alle Knotentypen verwendet und repräsentiert einen einzelnen Knoten im Dokumentenbaum. Document ist dabei nicht identisch mit dem Wurzelelement des eigentlichen XML-Dokumentes. Die Wurzel wird vielmehr mit einem *Element*-Knoten dargestellt. Auf der selben Ebene wie der erste Elementknoten können nur noch ein *Document-Type*-Knoten und *Processing-Instructions* auftauchen. Zeichendaten der einzelnen Elemente werden durch *Text*-Knoten dargestellt. Um die Plattformunabhängigkeit zu gewährleisten, existiert ein Standarddatentyp *DOMString*. Eine Besonderheit bilden die Attributknoten. Das sind Objekte der *Attr*-Schnittstelle, die

auch von der Nodeschnittstelle erbt. Attributknoten werden in DOM nicht wie Kindelemente von den Elementknoten, denen sie angehören, gehandhabt. Sie bilden Eigenschaften der Elementknoten. Es existiert auch keine Geschwisterbeziehung zwischen Attributen im selben Element, wie es bei Elementknoten auf der gleichen Ebene der Fall ist. Die Liste der Attribute eines Elementes ist ungeordnet. Im Listing 4.1 ist dieser Sachverhalt noch einmal verdeutlicht. Die DOM-Beispiele wurden mit Hilfe des Packages *org.w3c.dom*[3] aus der Java API for XML Processing (JAXP) erstellt.

```
private static Document createXml(ResultSet resultSet) {
    DocumentBuilderFactory factory = DocumentBuilderFactory.
        newInstance();
    DocumentBuilder builder =
        factory.newDocumentBuilder();
    Document document = builder.newDocument();
    Element root = document.createElement("result");

    document.appendChild(root);

    while (resultSet.next()) {
        Element sentence =
            document.createElement("sentence");
        sentence.setAttribute("id", (resultSet.getString(1)));
        sentence.appendChild(
            document.createTextNode(
                resultSet.getString(2)));

        root.appendChild(sentence);
    }
    return document;
}
```

Listing 4.1: Beispiel der Erstellung eines XML-Dokumentes mittels DOM

Der Methode in diesem Beispiel wird ein *ResultSet*-Objekt übergeben, dass die Ergebnisse aus einer Datenbankabfrage beinhaltet. Im ersten Schritt wird eine *DocumentBuilderFactory*-Instanz erstellt. Diese wird benötigt, um eine *DocumentBuilder*-Instanz zu erstellen, die wiederum das eigentliche Dokumenten-Objekt generiert. Mithilfe dieser Document-Instanzen werden dann alle weiteren Elemente erzeugt. Hier wird noch einmal deutlich, dass das eigentliche Wurzelelement des Dokuments lediglich ein einfaches Element ist. Dieses wird mit *Document.appendChild(root)* der Dokument-Instanz angehängt und somit als Wurzel deklariert. Alle weiteren Elemente werden nun an das Wurzelement angefügt. In einer Schleife wird das übergebene ResultSet durchlaufen und für jeden darin enthaltenen Datensatz ein neues DOM-Element erzeugt und mit Informationen befüllt. Dieser Vorgang unterscheidet sich nicht von der Erstellung des Wurzeldokumentes.
Mit *Element.appendChild(sentence)* wird das erzeugte Element dem Wurzelelement angefügt. Dabei ist anzumerken, dass die Beispiele in diesem Kapitel nur einen Einblick über die allgemeine Verwendung von DOM Implementierungen geben sollen, da sie alle ähnlich zu verwenden sind.

[3] Vgl. http://java.sun.com/javase/6/docs/api/org/w3c/dom/package-summary.html/.

Das Parsen eines XML-Dokumentes ist im nächsten Listing 4.2 verdeutlicht.

```
private static String parseDocument(String xmlDocument) {
    ByteArrayInputStream inputStream = new
        ByteArrayInputStream(xmlDocument.getBytes("UTF-8"));
    DocumentBuilderFactory factory = DocumentBuilderFactory.
        newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.parse(inputStream);
    NodeList sentences = document.getElementsByTagName("
        sentence");

    for (int i = 0; i < sentences.getLength(); i++) {
        ...
    }
    ...
}
```

Listing 4.2: Beispiel des Parsens eines XML-Dokumentes mittels DOM

Hier wird das XML-Dokument, dass es zu parsen gilt, anhand einer Zeichenkette übergeben und in ein *ByteArrayInputStream* geladen. Andere Repräsentationen des Dokumentes, zum Beispiel als *File*-Objekt, sind auch denkbar. Mit *DocumentBuilder.parse(inputStream)* wird das *Document*-Objekt erstellt und befüllt. Hier ist zu sehen, dass die Anwendung erst wieder weiter arbeiten kann bzw. darf, wenn das Dokument vollständig eingelesen ist. Der Zugriff auf die einzelenen Elemente des Dokumentes wird mit *Document.getElementsByTagName("sentence")* erhalten. Nach Aufruf dieser Methode wird eine Liste mit den Elementen, die dem angefragten Namen entsprechen, zurückgeliefert.
Wie auch schon beim Generieren von XML wird zuerst eine *DocumentBuilderFactory*- und anschließend eine *DocumentBuilder*-Instanz erzeugt.
Im nachfolgenden Abschnitt dieses Kapitels werden verschiedene DOM-Implementierungen vorgestellt. Hierbei ist noch zu erwähnen, dass sich die Benutzung der Implementierungen sehr ähnlich sind und keine nennenswerten Unterschiede aufweisen.

4.1.3 Implementierungen von DOM

Da DOM nur eine abstrakte Schnittstelle ist, die Regeln zur Implementierung von DOM-basierten Anwendungen beschreibt, kann bis zu diesem Zeitpunkt keine praktische Nutzung stattfinden. In den nachfolgenden Unterabschnitten werden einige ausgewählte DOM-Implementierungen vorgestellt. Diese werden ebenfalls in den Benchmarktests betrachtet und ausgewertet. Alle vorgestellten Implementierungen haben das Ziel den Zugriff, die Generierung und die Manipulation von XML- oder HTML-Dokumenten zu vereinfachen.

JDOMl

Java Document Object Model (JDOM) ist eine frei verfügbare Java-Bibliothek, um XML-Daten zu manipulieren. Dabei wurde JDOM nicht nach dem DOM-Konzept von W3C entwickelt, sondern steht für ein alternatives Objektmodell. JDOM ist nicht, wie in der DOM-Spezifikation vorgeschrieben, sprachenunabhängig, sondern javaspezifisch. Daraus ergibt sich der Vorteil, dass die Datenmanipulation direkt für Java angepasst und optimiert wurde.

Somit kann JDOM auf die Features und Vorteile, wie zum Beispiel Collections, Reflection oder Methodenüberladung, der Sprache zurückgreifen. Die JDOM Bibliothek besteht dabei aus sechs Paketen, die in der nachfolgenden Tabelle kurzen erläutert werden.

Paket	Beschreibung
org.jdom	Beinhaltet Klassen, die die verschiedenen Knotentypen repräsentieren.
org.jdom.input	Beinhaltet Klassen, um Dokumente zu erstellen.
org.jdom.output	Beinhaltet Klassen, um XML-Dokumente auszulesen.
org.jdom.transform	Beinhaltet Klassen, für XSLT-Transformationen.
org.jdom.xpath	Beinhaltet Klassen, um mittels XPath zu operieren.
org.jdom.adapters	Beinhaltet Helper-Klassen für DOM.

Tabelle 4.1: JDOM-Komponenten

JDOM stellt dabei einige Parser, wie z.B. *SAXOutputter* und *DOMOutputter* sowie verschiedene Builder, wie z.B. *DOMBuilder* und *SAXBuilder*, für das effiziente Einlesen und Generieren von XML-Dokumenten bereit.[4]

DOM4J

Ebenfalls wie JDOM ist Document Object Model for Java (DOM4J) eine frei verfügbare Javabibliothek, um XML zu verarbeiten und wurde ausschließlich für die Sprache Java entwickelt.[5]

XOM

XML Object Model (XOM) ist eine weitere baumbasierte XML-Programierschnittstelle und bietet zwei Besonderheiten: zum einen ist sowohl eine streaming basierte als auch eine baumbasierte API. Dies bedeutet, dass während der Erstellung des Dokumentes bereits einzelne Knoten verarbeitet werden können. Somit ist es möglich, dass Programme, die mit XOM arbeiten, meist so schnell arbeiten können, wie der verwendete Parser Daten verarbeiten kann. Zum anderen gilt es als sehr speichereffizient. Dies kann erreicht werden, indem vor dem Prozess des Parsens des Dokumentes ein Filter definiert wird. Dieser ermöglicht es, dass nur die benötigten Teile des Dokumentes geladen werden.
Wie auch die vorherigen XML-Programmierschnittstellen ist XOM von einem SAX Parser abhängig, der die Daten in eine Baumstruktur bringt. Hier gibt die Dokumentation vor, dass theoretisch jeder SAX2 Parser mit XOM zusammenarbeiten kann, jedoch Xerces der einzige ist, mit dem ein korrektes Arbeiten garantiert wird.[6]

4.2 Streaming API for XML

Für das Arbeiten mit XML-Infosets existieren im Allgemeinen zwei Programmiermodelle: *Document Streaming* und das im vorherigen Abschnitt behandelte *Document Object Model*.

[4] Vgl. (Hunter, 2002).
[5] Vgl. (Hari, 2009).
[6] Vgl. (Harold, 2009).

Wie bereits erwähnt, ist es mit DOM möglich im Speicher ein Objektmodell zu erstellen, dass das eigentliche Dokument repräsentiert. Der Vorteil, der sich daraus ergibt, ist, dass somit frei durch das Dokument navigiert werden kann. Somit ist dieser Ansatz sehr flexibel und benutzerfreundlich. Die Flexibilität und Anwenderfreundlichkeit haben jedoch einen Nachteil. Der Speicherverbrauch und die Auslastung der CPU sind im Vergleich zum Streamen von Dokumenten sehr hoch, da das erstellte Objektmodell im Speicher gehalten werden muss. Dieser Nachteil ist allerdings zu vernachlässigen, wenn mit kleinen Dokumenten gearbeitet wird. Die Bearbeitung von sehr großen Dokumenten ist jedoch fast nicht möglich.
Das Streamen von Dokumenten ist ein Programmiermodell, in dem XML-Infosets seriell übertragen und geparst werden. Streambasierte Parser haben die Fähigkeit Elemente von Dokumenten zu übergehen, falls sie nicht benötigt werden. Der Vorteil dieses Programmiermodells ist, ein geringerer Speicherverbrauch, geringere CPU Auslastung und eine höhere Performance. Aus diesen Vorteilen ergibt sich auch ein gravierender Nachteil. Der Progrmmierer muss wissen, welche Teile des Dokumentes verarbeitet werden sollen, bevor begonnen wird, das Dokument einzulesen. Ein willkürliches Navigieren und Manipulieren ist nicht möglich. [7]

4.2.1 Pull Parsing und Push Parsing im Vergleich

Streaming Pull Parsing ist ein Programmiermodell, in dem die Anwendung eine Methode einer Parsing Bibliothek aufruft, sobald sie Informationen aus einem XML-Infoset benötigt. Streaming Push Parsing ist das entgegengesetzte Programmiermodell zu Pull Parsing. Hier sendet der XML-Parser XML-Daten zu der Anwendung, egal ob sie diese benötigt oder nicht. Daraus ergeben sich Vorteile für Pull Parsing gegenüber Push Parsing. Zum einen hat hier die Anwendung die Kontrolle Methoden dann aufzurufen, wenn es notwendig ist. Zum anderen können XML-Dokumente gefiltert werden, was zur Folge hat das nur bestimmte Teile eines Dokumentes gelesen und gespeichert werden kann.

4.2.2 StAX-API

Streaming API for XML (StAX) wurde von BEA mit Unterstützung von Sun Microsystems entwickelt, um die Nachteile, die DOM und SAX haben, aufzuheben. Es läuft unter der Spezifikationsnummer JSR-173. Dabei unterstützt StAX mit der iterativen- und der ereignisbasierten Verarbeitung zwei Möglichkeiten der Manipulation von XML-Dokumenten. Die StAX-API besteht mit der *StAX-Cursor API* und der *StAX-Event-Iterator-API* aus zwei separaten APIs. Beide dieser APIs bestehen jeweils aus zwei Schnittstellen. Die Cursor-API besitzt *XMLStreamReader* und *XMLStreamWriter*, die Event-Iterator-API *XMLEventReader* und *XMLEventWriter* als Schnittstelle. Die separaten APIs können als Iteratoren über eine Menge von Ereignissen betrachtet werden. Als Events werden Elemente eines XML-Dokumentes gesehen. Dies können zum Beispiel Attribute, Textknoten, Kommentare oder Namespaces sein.
Die Cursor-API bewegt einen Cursor durch ein XML-Dokument. Dieser ist dabei vorwärtsgerichtet und kann demnach nicht zurückspringen. Die Events sind direkt im Cursor eingebettet. Trifft der Cursor nun auf das von der Anwendung angeforderte Event, liefert er alle nötigen Informationen zu diesem. Bei der Erstellung von XML-Dokumenten durch die Cursor-API wird das XMLStreamWriter Interface verwendet. Es spezifiziert wie XML erstellt wird, jedoch ohne Prüfung auf Wohlgeformtheit.

[7] Vgl. (Harold, 2003) und (Oracle, 2010).

Die Iterator-API baut auf der Cursor-API auf, verfolgt aber ein anderes konzeptionelles Modell. Sie enthält Objekte, die Events repräsentieren, die auch in der Cursor-API verwendet werden können. Die Events sind dabei eine Abstraktion des XML-Infosets und können der Reihe nach von der Anwendung angefordert werden. Beim Finden gewisser Knoten, werden die dazugehörigen Informationen abgespeichert. Eine Anwendung kann nun über das XML-Dokument iterieren, indem es das nächste Event anfragt.
Im nachfolgenden Listing 4.3 ist ein Beispiel[8] für die Verwendung der StAX-Cursor-API gegeben. Dabei wurde das Package *javax.xml.stream* für die Erstellung der Beispiele verwendet.

```
private static void createXml(ResultSet resultSet,
   OutputStream outputStream) {
   XMLOutputFactory outputFactory = XMLOutputFactory.
      newInstance();
   XMLStreamWriter streamWriter =
      outputFactory.createXMLStreamWriter(outputStream);
   streamWriter.writeStartDocument();
   streamWriter.writeStartElement("result");

   while (resultSet.next()) {
      streamWriter.writeStartElement("sentence");
      streamWriter.writeAttribute("id", (new Integer(
         resultSet.getRow() + startId - 1).toString()));
      streamWriter.writeCharacters(resultSet.getString(2));
      streamWriter.writeEndElement();
   }

   streamWriter.writeEndElement();
   streamWriter.writeEndDocument();

   streamWriter.close();
}
```

Listing 4.3: Beispiel der Erstellung eines XML-Dokumentes mittels StAX und der Cursor-API

Hier wird, ähnlich wie im Beispiel 4.1, einer Methode ein *ResultSet*-Objekt mit dem Ergebnis einer Datenbankabfrage und ein *OutputStream* übergeben. Somit ist es gleich möglich das zu erstellende Dokument in einen Stream zu schreiben. Dies erleichtert die Umwandlung des resultierenden Dokumentes in eine Zeichenkette. Im ersten Schritt wird eine Instanz von *XMLOutputFactory* erstellt, mit deren Hilfe im nächsten Schritt eine Instanz von *XMLStreamWriter* erzeugt wird. In StAX muss beim Generieren von Dokumenten lediglich definiert werden, wo der Beginn und das Ende des Dokumentes bzw. des Elementes ist. Dies geschieht mit *XMLStreamWriter.writeStartDocument()* und *XMLStreamWriter.writeEndDocument();* bzw. mit *XMLStreamWriter.writeStartElement("sentence")* und *XMLStreamWriter.writeEndElement()*.
Das Parsen eines XML-Dokumentes ist im Listing 4.4 gezeigt.

[8] Die Beispiele wurden hier lediglich mit der StAX-Cursor-API entwickelt, da nur diese in den Tests verwendet wurde. Die StAX-Iterator-API soll an dieser Stelle nur erwähnt bleiben.

```
private static String parseDocument(String xmlDocument) {
   XMLInputFactory inputFactory = XMLInputFactory.newInstance
      ();
   StringReader reader = new StringReader(response);
   XMLStreamReader streamReader = inputFactory.
      createXMLStreamReader(reader);

   for (int event = streamReader.next();
         event != XMLStreamConstants.END_DOCUMENT;
         event = streamReader.next()) {
      if (event == XMLStreamConstants.START_ELEMENT
            && streamReader.getLocalName().equals("sentence")
               ) {
               int sentenceId = Integer.parseInt(streamReader
                  .getAttributeValue(0));
      }
   }
   ...
}
```

Listing 4.4: Beispiel des Parsens eines XML-Dokumentes mittels StAX Cursor API

Wie aus diesem Beispiel hervorgeht, wird für das Einlesen, im Gegensatz zum Erstellen von XML-Dokumenten, zuerst eine Instanz vom Objekt *XMLInputFactory* erstellt. Mithilfe dieser wird dann anschließend eine Instanz von *XMLStreamReader* erzeugt. Diese Instanz kann dann für jedes Ereignis durchlaufen werden. Mit *XMLStreamReader.next()* wird das nächste Element des Dokumentes eingelesen. Diese Methode gibt einen *Integer* zurück. Anhand dieses Wertes kann die Art des Elementes ermittelt werden. Das Enum *XMLStreamConstants* erleichtert die Identifizierung des derzeitigen Elementes. *XMLStreamReader* bietet darüber hinaus zahlreiche Methoden, um Informationen, wie z. B. den Namen, die Attribute oder den Wert des derzeit sich im Stream befindlichen Elementes abzufragen. [9]

Vergleich der Iterator- und Cursor-API

Mithilfe der Iterator-API ist es möglich, unveränderbare Objekte zu erzeugen und in Arrays, Listen und Maps zu speichern. Diese können dann durch die Anwendung hindurch gereicht werden, sobald der Parser nachfolgende Events verarbeitet. Des Weiteren ist es wesentlich einfacher Events einem XML-Event-Stream hinzuzufügen bzw. von einem Event Stream zu entfernen als mit der Cursor API.[10]

4.2.3 Anwendungsgebiete von StAX

Die StAX-Spezifikation findet in zahlreichen Gebieten ihren Einsatz, welche nachfolgen exemplarisch aufgezählt werden.[11]

Data Binding

[9] (Oracle, 2010)
[10] (Oracle, 2010)
[11] Aus (Oracle, 2010)

- Unmarshalling eines XML-Dokumentes.
- Marshalling eines XML-Dokumentes.
- Parallele Dokumentenverarbeitung.

SOAP-Nachrichtenverarbeitung

- Parsen simpler, planbarer Strukturen.
- Parsen von Graphrepräsentationen mit Vorwärtsrichtung.
- Parsen von WSDL.

virtuelle Datenquellen

- Betrachtet als XML-Daten, gespeichert in Datenbanken.
- Betrachte Daten in Java-Objekten, generiert durch Data Binding.
- Navigiert durch einen DOM-Baum, als Stream von Events.

Parsen spezifischer XML-Vokabulare.

4.2.4 StAX Implementierungen

SJSXP

Sun Java Streaming XML Parser (SJSXP) ist Suns Implementierung von JSR-173. Er zählt zu den nichtvalidierenden Parsern, der auf Xerces2 aufgebaut wurde. Dabei wurden die unteren Schichten von Xerces2, besonders die Scanner und die mit ihm in Verbindung stehenden Klassen, neu entwickelt, um das Pull Parser Programmiermodell einzubetten. [12]

Javolution

Javolution ist ein Open-Source-Projekt, das hauptsächlich für Echtzeit- und Embedded- Systeme gedacht ist. Es liefert einige Klassen aus, die gegenüber den bestehenden Klassen der Java Standard- bzw. -Mircro Edition einige Vorteile hinsichtlich der Benutzbarkeit und Performance versprechen. Javolution bietet auch einen Stax-ähnlichen Parser, der jedoch nicht vollständig JSR-173 erfüllt.[13]

Woodstox und StAX RI

Woodstox ist ein von der Firma Codehaus frei verfügbarer, validierender StAX Parser. Er wurde mit Java entwickelt und basiert auf der Spezifikation JSR-173.
Streaming API for XML Reference Implementation (StAX RI) wurde ebenfalls von der Firma Codehaus entwickelt und ist die Referenzimplementierung von StAX.[14]

[12] Vgl. (Team, 2005) Kapitel 3.
[13] Vgl. http://javolution.org/.
[14] Vgl. http://woodstox.codehaus.org.

4.3 Performanceevaluation der XML-Schnittstellen

Im diesem Abschnitt werden die Testergebnisse zum Vergleich der Programmierschnittstellen für XML ausgewertet, um festzustellen, welche Schnittstelle die bessere Performance besitzt. Für die Ermittliung, wurden 1.000 Requests pro Nachrichtengröße, die zwischen 1, 100, 1.000, 10.000 und 100.000 Elementen variierte, gestellt. Die XML- Programmierschnittstellen wurden in einer REST-basierten Umgebung implementiert. Dafür wurden acht verschiedene Webanwendungen erstellt, von denen jede eine der bereits erwähnten Implementierungen der DOM- bzw. StAX-Spezifikationen verwendet.

Service-Implementierung

Alle der acht entwickelten Web-Anwendungen arbeiteten dabei nach der gleichen Logik. Es wurde wieder, der bereits bekannte, SentencesService verwendet. Dieser konnte mit Hilfe von zwei Parameter aufgerufen werden. Eine anschließende Datenbankabfrage lieferte dann ein *ResultSet*-Objekt zurück. Dieses wurde in einer Schleife durchlaufen, wobei eine Zeile die Id und den dazugehörigen Satz beinhaltete. Diese Informationen wurden in dem Dokument abgespeichert.
Alle untersuchten Implementierungen der Spezifikationen von DOM und StAX bauen auf Parser auf. Theoretisch könnten diese mit allen zur Verfügung stehenden Parsern, wie z.B. Piccolo, Xerces, Crimson, usw., laufen. Für die repräsentative Aussage der Tests wurde für jede getestete Implementierung Xerces als Parser verwendet[15].

Versuchsergebnisse

In diesem Abschnitt werden die erzielten Resultate präsentiert und ausgewertet. Die Gesamtverarbeitungszeit wurden dabei auf der Clientseite gemessen und spiegelt die Zeit vom Versenden des Requests bis hin zum Empfang des Response vom Servers wieder. Die Serververarbeitungszeit ist die Zeit, die der Server zur Verarebitung einer Anfrage und zur Generierung einer Antwort benötigt.
In den Tabellen 4.2 und 4.3 sind die Ergebnisse der Testdurchläufe tabellarisch aufgelistet.

		Serververarbeitungszeit pro Anzahl Elemente				
		1	100	1.000	10.000	100.000
DOM	DOM	4	10	27	233	2384
	JDOM	2	6	25	198	2480
	DOM4J	3	7	23	204	2461
	XOM	3	8	29	297	2811
StAX	Javolution	2	6	20	179	1773
	SJSXP	3	7	24	187	2230
	StAX RI	2	6	21	179	2040
	Woodstox	2	5	19	172	2048

Tabelle 4.2: XML-Schnittstellen Verarbeitungszeiten auf dem Server

Wie schon bereits in den vorherigen Abschnitten vermutet, besteht zwischen kleineren Dokumenten kein signifikater Unterschied in der Gesamtverarbeitungszeit der einzelnen XML-Schnittstellen. Erst bei größeren Dokumenten ab 1.000 Elementen kristallisiert sich heraus,

[15] Eine Untersuchung der Performance der verschiedenen Parser wurde in (Büchler, 2005) durchgeführt.

		Gesamtverarbeitungszeit pro Anzahl Elemente				
		1	100	1.000	10.000	100.000
DOM	DOM	8	13	30	235	2537
	JDOM	6	10	27	224	2632
	DOM4J	6	10	27	212	2617
	XOM	6	11	31	299	2932
StAX	Javolution	6	9	23	182	1937
	SJSXP	6	10	29	193	2301
	StAX RI	6	9	24	182	2148
	Woodstox	5	8	22	174	2170

Tabelle 4.3: XML-Schnittstellen Gesamtverarbeitungszeiten

dass DOM-Implementierungen längere Zeit für die Erstellung der Ergebnisdokumente benötigen als die StAX-Implementierungen. Negativ hervorzuheben ist in diesem Test XOM, da diese relativ viel Zeit für die Bearbeitung der Anfragen benötigt. Am schnellsten verarbeitetet die Webanwendung die Anfragen als Woodstox zur Generierung von XML verwendete wurde.

4.4 Zusammenfassung

In diesem Kapitel wurde StAX mit DOM verglichen. Hier ist zu erwähnen, dass die Integration und Verwendung aller Implementierungen sehr einfach war. Die Entscheidung über die Verwendung von DOM oder StAX, hängt von der Größe der zu generierenden Dokumente und der zur Verfügung stehenden Hardware ab. So lange die Dokumente klein genug sind[16], gibt es keinen signifikanten Unterschied zwischen den beiden Programmierschnittstellen. Erst wenn diese größer werden[17], ist StAX zu bevorzugen, da hier aufgrund des geringeren Speicherverbrauches und der schnelleren Verarbeitung die bessere Performance besteht.

[16] Bis 100 Elemente.
[17] Ab 1.000 Elemente.

5 XML Repräsentationen

Ein XML-Dokument kann mithilfe von Standards und durch Variierung der Strukturierung der Elemente verschiedenartig aufgebaut werden, jedoch immer den gleichen Aussagegehalt haben. Dieses Kapitel soll anfänglich einen Einblick über die beiden Markup-Arten Inline und Stand-Off geben und anschließend den Standard TEI untersuchen. Dieses wird schließlich ebenfalls in Inline und Stand-Off Syntax unterschieden.
Ziel dieses Abschnitts ist ein Vergleich beider Markup-Arten unter den Aspekten der Größe der erzeugten Dokumente, des Schwierigkeitgrades der Implementierung und der Peformance bei der Generierung. Erwartet wird dabei, dass Stand-Off-Markup einfacher zu generieren ist, als Inline-Markup, da hier die neuen Informationen einfach an das Ende des Dokuments angehängt wird. Des Weiteren sollte Stand-Off schneller zu generieren sein, jedoch die größeren Dokumente erzeugen.

5.1 Arten von Markup

5.1.1 Inline Markup

Inline Markup erlaubt es, dass sich die Struktur- und Zusatzinformationen innerhalb des Textes befinden. Der Nachteil, der sich daraus ergibt, ist die Manipulation bzw. Beschädigung des Originaltextes. Außerdem wird die Lesbarkeit durch das Hinzufügen von Markup beeinträchtigt. Allerdings verringert sich der Aufwand der Darstellung des Inhaltes des Dokumentes, da sich hier alle Informationen an den Elementen befinden, zu den sie gehören.
Bei der Erstellung eines Dokumentes im Inline- Format existieren zwei Lösungsansätze. Der Erste verwendet ausschießlich Attribute, um weitere Informationen einem Objekt hinzuzufügen, was im Codebeispiel 5.1 ersichtlich ist. Hier ist das Ergebnis eines Aufrufs einer Servicekette zu sehen[1]. Es stellt einen Satz dar, der zu jedem enthaltenen Wort die Frequenz angibt. Das Dokument ist am Beispiel des Wortes Follow folgendermaßen zu lesen: *„Das Wort Follow mit der Id 17.185 des Satzes mit der Id 3 hat die Frequenz 5“*.

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <sentence s_id="3">
      <token frequency="5" w_id="17185">Follow</token>
      <token frequency="160837" w_id="1">the</token>
      <token frequency="113" w_id="2213">colors</token>
      <token frequency="94352" w_id="2">of</token>
      <token frequency="160837" w_id="1">the</token>
      <token frequency="39" w_id="4982">Nineteenth</token>
```

[1] Detailierte Beschreibung der Servicekette Siehe Kapitel 5.1.3.

```
    </sentence>
</result>
```

Listing 5.1: Inline Markup mit Attributen

Möglichkeit zwei ist im Listing 5.2 zu sehen. Dieses Beispiel ist eine gekürzte Variante für die ersten beiden Wörter des Satzes. Es hat die gleiche Aussagekraft wie das erste Beispiel, jedoch werden hier die Informationen über die Frequenz eines Wortes als Element und nicht als Attribut gespeichert.

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
   <sentence s_id="3">
      <frequency count="5">
         <token w_id="17185">Follow</token>
      </frequency>
      <frequency count="160837">
         <token w_id="1">the</token>
      </frequency>
      <frequency count="113">
      ...
      </frequency>
   ...
   </sentence>
</result>
```

Listing 5.2: Inline Markup mit Elementen

In diesen beiden vorangegangenen Beispielen ist somit sichtbar, dass die Angabe von Strukturinformationen in Elementen bzw. in Attributen geschehen kann. Dies hat erhebliche Auswirkungen auf die Größe und somit auf die Geschwindigkeit des Übertragens und des Parsens des resultierenden Dokumentes.

5.1.2 Stand-Off Markup

Stand-Off- oder externes Markup wird verwendet, wenn das Markup außerhalb des Textes eingesetzt werden soll. Die Gründe hierfür könnten beispielsweise sein, dass keine Schreibrechte auf die Daten bestehen oder die Daten beinhalten bereits Markup, das jedoch inkompatibel zu dem Markup ist, mit dem das Dokument erweitert werden soll. Es besteht allerdings die Möglichkeit jedes externe Markup in internes umzuwandeln, indem einfach das externe Markup direkt in den Text hinzugefügt und die Referenzen auf die Objekte mit den eigentlichen Objekten ersetzt werden.
Auch für Stand-Off-Markup existieren zwei Möglichkeiten der Realisierung. Die Erste ist, dass der reine Text als gesondertes Dokument betrachtet wird und ein zweites Dokument existiert, welches das Markup enthält. Die einzelnen Elemente enthalten dann Referenzen auf Textpassagen, Wörter o. ä., die mithilfe von *XInclude*, *XPointer* und *XLink* erzeugt bzw. ausgelesen werden können. Die zweite Variante zur Generierung von externen Markup ist, dass sich alles in einem Dokument befindet und die Informationen, die dem Dokument hinzugefügt werden sollen, einfach an das Dokumentende angehängt werden. Für die Durchführung der Tests kommt lediglich die zweite Variante zum Einsatz, da alle Dokumente neu

generiert werden. Somit können inkompatibeles Markup und fehlende Schreibrechte ausgeschlossen werden. Das nachfolgende Codebeispiel 5.3 zeigt einen Ausschnitt aus einem Dokument im Stand-Off Format.

```
<?xml version="1.0" encoding="UTF-8"?>
<result>
<sentence s_id="3"> Follow the colors of the Nineteenth.</
   sentence>
<sentence s_id
   <tokens>
      <token w_id="4982" s_id="3">Nineteenth</token>
      <token w_id="1" s_id="3">the</token>
      <token w_id="2213" s_id="3">colors</token>
      ...
   </tokens>
   <frequencies>
      <frequency w_id="4982">39</frequency>
      <frequency w_id="1">160837</frequency>
      <frequency w_id="2213">113</frequency>
      ...
   </frequencies>
</result>
```

Listing 5.3: Stand-Off Markup

Während beim Inline Markup am Ende der Servicekette nicht mehr erkennbar ist, welcher Service welche Informationen hinzugefügt hat, ist es hier noch sichtbar, da Informationen einfach angehängt werden. Der erste Service hat die angefragten Sätze in das Dokument geschrieben. Dies sind alle Elemente, die mit sentence beginnen und enden. Der zweite Service fügt dann seine Informationen hinzu. Diese befinden sich innerhalb des *tokens*-Elementes. Schließlich fügte der dritte Service das *frequencies*-Element hinzu.

5.1.3 Performanceevaluation der Markup-Arten Inline und Stand-Off

Das in diesem Test gewählte Szenario zur Performanceanalyse von Inline- und Stand-Off-Markup, ist angelehnt an einen Prozess derDeutsche Sprachressourcen-Infrastruktur (D-SPIN)[2]. D-SPIN ist der deutsche Beitrag zum europäischen Common Language Resources and Technology Infrastructure (CLARIN)[3]. Beide Projekte besitzen das Hauptziel, eine gesamteuropäische Infrastruktur für Sprachressourcen aufzubauen.
In dem gewählten Szenario wird einem speziellen Import-Service ein Text geschickt. Der Service überführt diesen mit bestimmten Operationen in das D-SPIN-Textkorpus-Format. Das so entstandene neue Dokument wird anschließend auf einen *Tokenizer*-Service angewandt, der den Text segmentiert. Im Anschluss daran wird der segmentierte Text an einen *POS-Tagger*-Service geschickt. Dieser hat die Aufgabe, die Wörter des Textes zu Wortarten zu zuordnen. Eine derartige Verkettung von Services ist keine Seltenheit in einer linguistischen Umgebung. [4] Aus diesem Grund wurde das Szenario in abgewandelter Form für die

[2] Siehe http://weblicht.sfs.uni-tuebingen.de/.

[3] http://www.clarin.eu/external/

[4] Auch in anderen Bereichen ist das sogenannte *Service-Chaining* weit verbreitet.

Performancetests gewählt.
Die Abwandlung fand zum einen im ersten Service statt. Hier wurde kein Service verwendet der Text übergeben bekommt, sondern, wie auch in den vorherigen Tests, der bereits entwickelte SentencesService, da für den repräsentativen Aussagegehalt der Tests viel Inhalt generiert werden musste. Zum anderen wurde von der Verwendung von einem POS-Tagger-Service abgesehen und stattdessen ein Service entwickelt, der die Frequenzen der in einem Satz enthaltenen Wörter angibt. Die letzte Änderung betraf den Ablauf des Aufrufs der Services. In der Praxis ruft ein Client der Reihe nach die benötigten Services auf und übergibt dabei das vom vorherigen Service modifizierte Dokument bzw. Teile davon. Dieses Szenario ist in Abbildung 5.1 verdeutlicht.

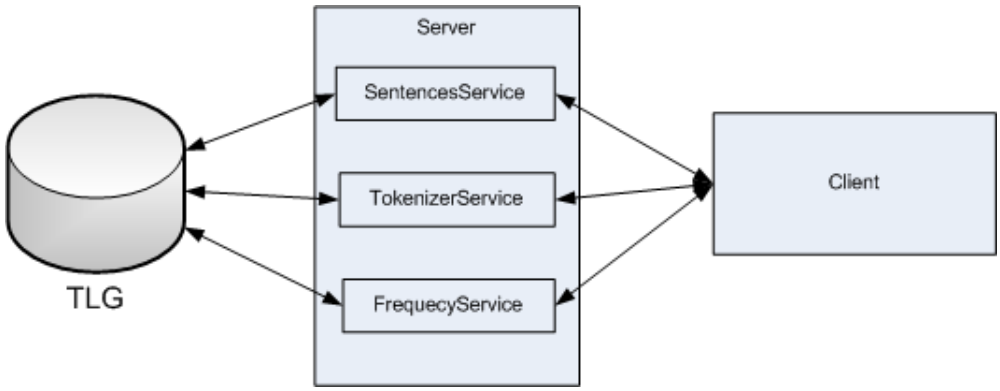

Abbildung 5.1: Aufruf einer Kette von Services durch den Client

Aus Vereinfachungsgründen, wurde von einer Client-Server-Interaktion abgesehen. In dem entwickelten Szenario ruft der Client lediglich den *SentencesService* anhand einer Start-Id und einer End-Id auf. Dieser generiert ein XML-Dokument, das eine Liste der Sätze mit den dazugehörigen Ids enthält. Anschließend wird durch den SentencesService der *TokenizerService* aufgerufen, der dann jeden Satz segmentiert. Am Ende der Servicekette steht der *FrequencyService*. Er erhält das erweiterte Dokument des TokenizerServices, fügt schließlich zu jedem Wort die Information über die Frequenzen hinzu und liefert danach das erweiterte Dokument an den Client zurück.
Weiterhin waren beim Vergleich der beiden Markup-Arten vor allem die Kriterien Größe der Dokumente, Zeit vom Senden des Requests bis hin zum Empfangen des Response vom Server sowie der Schwierigkeitsgrad der Entwicklung relevant.

Service-Implementierung

Bei der Implementierung des Testszenarios wurden drei RESTful Services entwickelt, die die Dokumente mithilfe von StAX generierten.[5] Für die Simulation der wechselseitigen Client-Server-Interaktion musste auf der Serverseite ein Simulator entwickelt werden der nacheinander die Services auf rief. Die ersten beiden Services konvertierten dabei das XML-Dokument in eine Zeichenkette. Der FrequencyService war dabei der letzte Service in der Kette. Er übergab eine *OutputStream*-Instanz, die das endgültige Dokument beinhaltete. Bei der Erstellung des Ergebnisdokumentes wurde in jedem Service und auch bei jeder Markup-Variante eine *BufferedWriter*-Instanz an eine *XMLStreamWriter*-Instanz übergeben, welche schließlich das Dokument erzeugte. Die größe der Puffers der BufferedWriter-Instanz wurde

[5]Dabei wurde SJSXP verwendet.

dabei auf 32.768 Bytes festgelegt. Sowohl der TokenizerService als auch der FrequencyService bekamen das Ergebnisdokument in Form eines String-Objektes übergeben und laßen dieses mithilfe einer *BufferedReader*-Instanz ein. Das übergebene String-Objekte wurde dabei sfort auf *null* gesetzt, so dass es bei der nächsten Ausführung des *Garbage Collectors* aus dem Speicher entfernt werden kann. Für die Abfrage der Datensätze aus der Datenbank wurden *PreparedStatements* verwendet. Die *ResultSet*-Instanz, die das Ergebnis der Abfrage enthielt wurde nach jedem Durchlauf auf null gesetzt. Somit war ein resourcenschonendes Arbeiten der Services gewährleistet.
Bei der Entwicklung der Services ist aufgefallen, dass es schwieriger war, Inline-Markup zu generieren. Denn hier musste im TokenizerService und im FrequencyService immer wieder das Dokument modifiziert werden. Dabei wurde es mittels DOM oder StAX geeparst, für jeden enthaltenen Satz die dazugehörigen Wörter oder die Wörter mit den Frequenzen aus der Datenbank geholt und in das Dokument eingefügt. Das Modifizieren der Dokumente hatte immer das selbe Schema. Der Inhalt des Sentences-Element wurde entfernt und der neue Inhalt aus dem TokenizerService oder FrequencyService hinzugefügt.
Bei der Erweiterung des Dokuments durch den TokenizerService bzw. FrequencyService mussten sich beim Stand-Off-Markup hingegen lediglich die IDs der Sätze, die vom SentencesService identifiziert wurden, gemerkt werden. Aufgrund dessen, dass immer ein zusammenhängendes Intervall von Sätzen abgefragt werden konnte, wurde in diesem Szenario nur die kleinste und die größte Id gespeichert. Die Folge daraus war, dass nur eine Datenbankabfrage benötigt wurde. Die Generierung des Inline Markups hätte ebenfalls so entwickelt werden können, dass eine Datanbankabfrage benötigt wird. Dies hätte jedochein zweimaliges Durchlaufen des Dokumentes zur Folge gehabt. Einmal, um die Start- und End-Id herauszufiltern und einmal, um das Ergebnis dieser Datenbankabfrage in das jeweilige Sentence-Element zu schreiben. [6]
Während der Entwicklung der Services war es schwierig Vorhersagen über den Ausgang der Tests zu treffen. Denn auf der einen Seite war das augenscheinlich kleinere Dokument im Inline-Format, das jedoch durch die mehrfachen Modifizierungen und die häufigeren Datenbankabfragen eventuell mehr Zeit für die Erzeugung benötigte. Auf der anderen Seite steht das Dokumnet mit Stand-Off-Markup, das aufgrund der Größe mehr Zeit für die Übertragung benötigt.
Eines ist schließlich noch anzumerken. Mit StAX war es nicht möglich die Dokumente zu modifizieren. Sie mussten immer geparst und der Inhalt in ein neues Dokument geschrieben werden. Mit DOM hingegen gab es die Möglichkeit das Dokument zu modifizieren. Jedoch war der Speicherverbruch so hoch, dass die Tests nur bis 100.000 Elemente durchgeführt werden konnten. Auch die Zeiten der Generierung der Dokumente war wesentlich höher. Aus diesem Grund wurde von der Präsentation dieser Ergebnisse abgesehen und nur die StAX-Ergebnisse dargestellt und ausgewertet.

Versuchsergebnisse

Es wurden sechs verschiedene Nachrichtengrößen mit 1-, 100, 1.000, 10.000, 100.000 und 200.000 Elementen mit jeweils 1.000 Requests getestet. Die Zeiten wurden auf der Clientseite gemessen und spiegeln die Zeit vom Senden des Requests bis zum Empfang des Response wieder.

[6] Zum Zeitpunkt der Entwicklung der Services schien eine Datenbankabfrage pro existierendem Satz im Dokument die effektivere Variante, weshalb diese auch entwickelt wurde.

Anzumerken ist hierbei, dass Dokumente mit Stand-Off-Markup mit den zur Verfügung stehenden Mitteln, maximal mit 300.000 Elementen erzeugt werden konnten. Für größere Dokumente lieferte der Server eine Exception vom Typ *java.lang.OutOfMemoryError*. Dokumente mit Inline-Markup hingegen konnten bis zu 500.000 Elemente beinhalten. Danach wurde ebenfalls die eben genannte Exception zurück geliefert.
Neben den Zeiten (Siehe Tabelle 5.2) wurden auch die Größen der Nachrichten untersucht, die in Tabelle 5.1 dargestellt sind. In den Abbildungen 5.2 und 5.3 wurden die Ergebnisse normalisiert und aufgrund der besseren Übersichtlichkeit in einem Diagramm veranschaulicht.

	Anzahl der Elemente im Dokument							
	1	100	1.000	10.000	100.000	200.000	400.000	500.000
Inline mit Elementen	2	122	1.093	14.138	111.893	211.129	436.257	543.039
Inline mit Attributen	2	92	823	10.623	84.292	159.041	328.435	408.834
Stand-Off	3	146	1.191	13.703	104.692	195.994	-	-

Tabelle 5.1: Nachrichtengröße der Markup-Arten

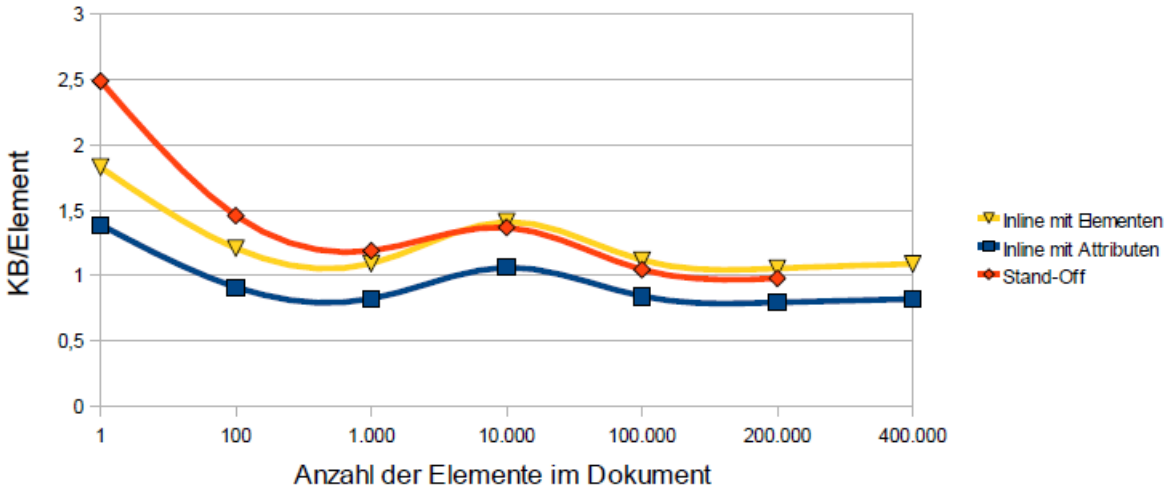

Abbildung 5.2: Durchschnittliche Nachrichtengröße der Markup-Arten pro Element

Bei der Auswertung des Vergleichs der Nachrichtengröße ist aufgefallen das keinere Dokumente im Stand-Off Format bis zu 60% größer sind als Dokumente im Inline-Format. [7] Erst bei einer höheren Anzahl an Elementen im Dokument wird dieses verhältnismäßig kleiner. Der Grund dafür ist die Arbeitsweise des FrequencyService. Denn dieser beinhaltet für Dokumente im Stand-Off-Format, jedes Wort das in den abgefragten Sätzen vorkommt

[7] Aus den eingangs erwähnten Gründen konnten die Dokumente mit Stand-Off Markup nur bis 300.000 und die Dokumente im Inline-Format bis 500.000 Elementen generiert werden.

genau einmal. Während bei Dokumenten mit Inline-Markup jedes Wort mit den Frequenz Informationen versehen wird, egal wie oft es in den vorherigen Sätzen enthalten war.
Tabelle 5.2 und die dazugehörige Abbildung 5.3 zeigen die durchschnittliche Gesamtverarbeitungszeit und die Normalisierung der gemessenen Zeiten.

	Anzahl der Elemente im Dokument					
	1	100	1.000	10.000	100.000	200.000
Inline mit Elementen	9	148	1.277	13.514	118.197	238.446
Inline mit Attributen	10	147	1.274	13.469	114.887	223.313
Stand-Off	12	76	577	6.722	99.667	274.143

Tabelle 5.2: Gesamtverarbeitungszeit der Markup-Arten

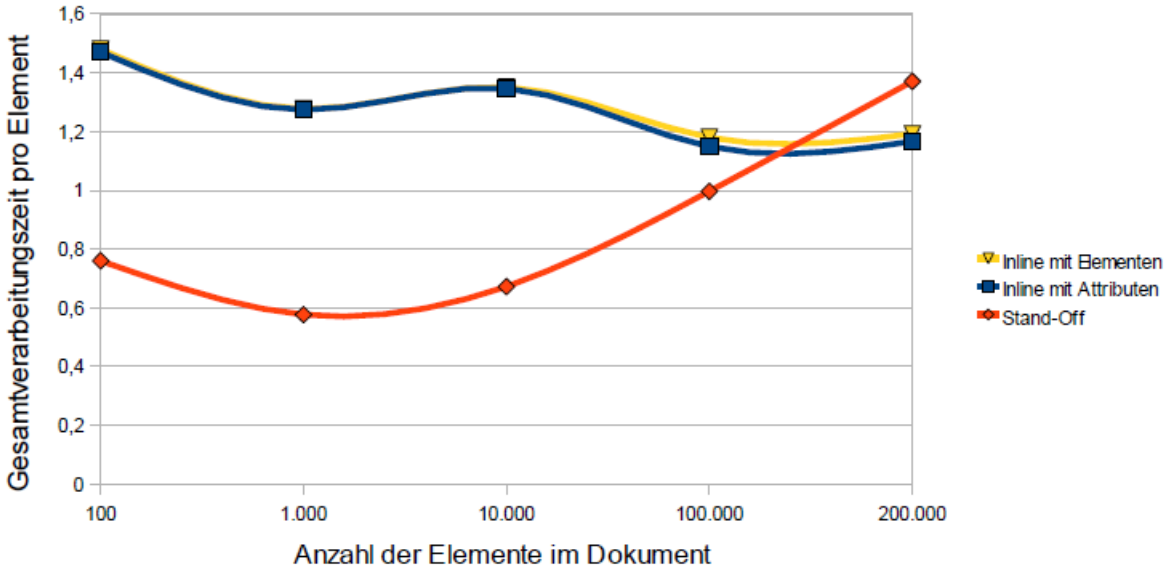

Abbildung 5.3: DurchschnittlicheGesamtverarbeitungszeit der Markup-Arten pro Element

Die Zeiten der Request-Response-Zyklen zeigen deutlich, dass bis 100.000 Elementen die Dokumente im Stand-Off-Format bis zu 50% schneller generiert, übertragen und vom Client empfangen werden können. Jedoch zeichnet sich bereits ab 1.000 Elementen ab, dass die durchschnitlliche Generierungszeit für Dokumente im Stand-Off-Format ansteigt. Der anfängliche zeitiche Vorteil hängt mit der geringeren Anzahl an Datenbankanfragen zusammen. Des Weiteren werden weniger Zeichenkettenoperationen ausgeführt. Dies bedeutet, dass wenn ein Sentences-Element gefunden wird, wird dieses beim Stand-Off Markup einfach nur kopiert und in das neue Dokument eingefügt. Während beim Inline Markup der Inhalt des Sentences-Elementes entfernt wird und der neue durch den jeweiligen Service hinzugefügt. Ab 200.000 Elementen weist der Request-Response-Zyklus eine höhere Zeit als bei den beiden Inline-Varianten auf. Dies liegt vorallem daran, dass das Dokumente mit Stand-Off Markup einen höheren Verbrauch des Speichers haben, was an der zu Beginn dieses Abschnittes erwähnten Exception belegt werden kann. Aus diesem Grund dauert die Generierung der Dokumente immer länger, je mehr Elemente sich im Dokument befinden.
Beim direkten Vergleich der beiden Inline Varianten fällt vorallem auf, dass die Dokumente im Inline-Markup mit Attributen bis zu 25% kleinere Dokumente generieren. Der Grund

dafür ist, dass beim Inline-Markup mit Attributen nicht so viel Zeichen für die Zusatzinformationen verwendet werden, wie beim Inline-Markup mit Elementen. Aus diesem Grund sind auch die Zeiten der Request Response-Zyklen bis zu 15% geringer, da hier das Netzwerk nicht so sehr belastet wird und der Client weniger Bytes empfangen muss.
Eines ist noch anzumerken: die durchgeführten Testreihen für die Dokumente mit 400.000 und 500.000 Elementen sind nicht repräsentativ, da JMeter hier einen sehr hohen Speicherverbrauch beim Empfang der Nachrichten hatte und Java den kompletten zur Verfügung stehenden Speicher belegte. Hier wurde teilweise mehr Zeit für den Empfang der Nachricht aufgewendet, als für die Generierung und die Übertragung.
Aufgrund der unterschiedlichen Resultate der beiden Varianten der Dokumente mit Inline-Markup wurden zusätzlich die Zeiten des Parsens der Dokumente auf der Seite des Clients betrachtet. Diese sind in Tabelle 5.3 gezeigt.

	Anzahl der Elemente im Dokument				
	1	100	1.000	10.000	100.000
Inline mit Elementen	0,4	4,6	37,8	462,3	3663,8
Inline mit Attributen	0,5	3,8	30,8	393,7	3115,3

Tabelle 5.3: Zusammenfassung der Inline Markup Parsingergebnisse

Zur Durchführung der Parsertests wurde auf Basis von SJSXP ein Parser entwickelt, der in Form einer *FileInputStream*-Instanz das Dokument erhielt. Ab diesem Zeitpunkt wurde die Stopuhr gestartet. Der Parser speicherte dabei immer zu einem Wort im Dokument die dazugehörige Frequenz. Wenn er am Ende des Dokumentes angelangt war, wurde die Stopuhr wieder angehalten und die Zeiten dieses Durchlaufes gespeichert. Dabei ist auch hier ersichtlich, dass die Dokumente mit Inline-Markup mit Elementen aufgrund ihrer Größe etwas länger benötigen. Der Parser muss hier mehr Bytes einlesen und verarbeiten. Ein weiterer Grund für den zeitlichen Vorteil der Dokumente mit Inline-Markup mit Attributen ist, dass lediglich zwei Anweisungen notwendig sind, um die erforderlichen Informationen aus dem Dokument zu filtern. Dieser Sachverhat ist in den beiden nachfolgenden Codebeispielen 5.4 und 5.5 ersichtlich.

```
<?xml version="1.0" encoding="UTF-8"?>
...
if (event != XMLStreamConstants.START_ELEMENT)
{
    continue;
}

if (streamReader.getLocalName().equals("frequency"))
{
    streamReader.next();
    frequency = streamReader.getAttributeValue(0);
    continue;
}

if (streamReader.getLocalName().equals("token"))
{
    streamReader.next();
```

```
    word = streamReader.getAttributeValue(0);
    continue;
}
...
```

Listing 5.4: Parsing Logik für Inline-Markup mit Elementen

```
...
if (event != XMLStreamConstants.START_ELEMENT)
{
    continue;
}

if (streamReader.getLocalName().equals("token"))
{
    frequency = streamReader.getAttributeValue(0);
    word = streamReader.getElementText();
    continue;
}
...
```

Listing 5.5: Parsing Logik für Inline-Markup mit Elementen

5.2 Repräsentation von XML mittels TEI

Text Encoding Initiative (TEI) ist ein Konsortium, dass sich mit der Entwicklung eines Standards beschäftigt, um Texte in digitaler Form darzustellen. Dazu liefert es eine Menge von Richtlinien, die Methoden für das Kodieren von maschinen-lesbaren Texten der Linguistik und Geistenswissenschaft definiert. Seit 1994 werden diese Richtlinien hauptsächlich von Museen, Bibliotheken und Verlegern eingesetzt. TEI ist eine Non-Profit Organisation, die sich aus akademischen Institutionen und Forschungsprojekten aus der ganzen Welt zusammensetzt. In den TEI-Richtlinien wird ein *Encoding Scheme* definiert, das in eine formale Markupsprache eingebettet ist.
Die TEI-Sprache wurde in mehreren Schritten standardisiert und weiterentwickelt. Die einzelnen Weiterentwicklungen werden von P1 bis P5 dargstellt. In den ersten drei Etappen (P1 bis P3) wurde noch Standard Generalized Markup Language (SGML)[8] für die Beschreibung verwendet. In P4 hatte der Anwender die Möglichkeit zwischen XML und SGML zu wählen. In P5 schließlich wurde nur noch XML verwendet. Somit beruht P5 stark auf den zur Verfügung stehenden XML-Standards, wie der Schema Sprache und den Tools wie XQuery und Extensible Stylesheet Language Transformation (XSLT).
Die TEI-Sprache definiert eine Menge von XML-Elemente, die für die Kodierung von Texten verwendet werden. Zusätzlich existieren in der Spezifikation zahlreiche XML-Attribute, um diese zu erweitern.
Ziel der TEI-Richtlinien ist es, für Texte in beliebiger Sprache, eines beliebigen Genres und einer beliebigen Epoche ein Framework zur Kodierung bereitzustellen. Aus diesem Grund sind zur Zeit fast 500 Elemente vordefiniert. Die Elemente in dieser vordefinierten Menge

[8] Standard Generalized Markup Language ist eine Metasprachen, mit deren Hilfe verschiedene Markup-Sprachen (wie XML) für Dokumente definiert werden.

unterteilen sich in zwei Kategorien. In der einen Kategorie finden sich Elemente wieder, die die Metadaten eines Textes beinhalten. Unter Metadaten eines Textes sind zum Beispiel Daten über den Autor, bibliografische Informationen oder die Beschreibung des Manuskriptes gemeint. Die andere Kategorie beinhaltet Elemente, die die Struktur eines Textes selbst wiederspiegeln. Dazu gehören u.a. Informationen über Kapitel, Überschriften, Paragraphen, Hervorhebungen usw.
Die TEI-Sprache der 5. Version ist die, zum Erstellungszeitpunkt dieser Arbeit, aktuellste und wurde modular aufgebaut. Sie besitzt ein *Core Modul*, dass die Grundelemente von TEI beinhaltet. Das Core Modul kann mit weiteren Modulen kombiniert und auch um nichtbenötigte Elemente gekürzt werden. Der Benutzer hat ebenfalls die Möglichkeit die Sprache zu erweitern. Dazu werden sowohl Dokumente von TEI als auch Tools zur Verfügung gestellt.

5.2.1 Einführung TEI

In dieser Einführung wurde die Sprache TEI Version 5 verwendet. Das Bereitstellen eines Vokabulars zur Beschreibung von Texten, ist die Philosophie von TEI. In TEI (Version 5) stehen 22 Module zur Verfügung, die in der Tabelle 5.4[9] aufgelistet sind.

Modul	Beschreibung
analysis	Beinhaltet analytische Mechanismen.
certainty	Gewiss und Ungewiss.
core	Beinhaltet Grundelemente von TEI.
corpus	Erweiterungen für den Header für Korpora.
declarefs	System-Deklarationen.
dictionaries	Gedruckte Wörterbücher.
drama	Drehbücher.
figures	Tabellen, Formulare, Abbildungen.
gaiji	Dokumentation von Buchstaben und Glyphen.
header	Header von TEI.
iso-fs	Strukturen.
linking	Ausrichtung, Segmentierung und Verlinkung.
msdescription	Beschreibung von Manuskripten.
namesdates	Namen und Daten.
nets	Graphen, Netzwerke und Bäume.
spoken	Transkripierte Sprache.
tagdocs	Dokumentation von TEI-Modulen.
tei	Deklaration von Datentypen, Klassen und Makros.
textcrit	Textkritiken.
textstructure	standardmäßige Textstruktur.
transcr	Transkription von primären Quellen.
verse	Struktur von Versen.

Tabelle 5.4: Module in TEI P5

Jedes TEI-Dokument besteht aus einem Kopf, dem *teiHeader*, der das Titelblatt darstellt, sowie einem Textkörper, *text*, der den elektronischen Textinhalt enthält. Diese Struktur ist in

[9] Vgl. (Georg Braungart, 2009).

Abbildung 5.6 verdeutlicht[10].

```
<?xml version="1.0" encoding="UTF-8"?>
<TEI>
    <teiHeader>
    ...
    </teiHeader>
    <text>
    ...
    </text>
</TEI>
```

Listing 5.6: Grundstruktur eines TEI- Dokumentes

Texte lassen sich in TEI auf zwei Möglichkeiten gruppieren. Wenn die Texte alle den gleichen Header verwenden können, wäre folgende Struktur für das Dokument möglich:

```
<?xml version="1.0" encoding="UTF-8"?>
<TEI>
    <teiHeader>
    ...
    </teiHeader>
    <text>
        <group>
            <text>
            ...
            </text>
          <text>
          ...
          </text>
        </group>
    </text>
</TEI>
```

Listing 5.7: Gruppierung von Texten mit gleichen Header in TEI

Bei Texten, die einen eigenen Header verwenden, sieht die Dokumentation vor, dass die Elemente nochmals unterteilt werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<TEICorpus>
    <teiHeader>...</teiHeader>
    <TEI>...</TEI>
    <TEI>...</TEI>
    ...
</TEICorpus>
```

Listing 5.8: Gruppierung von Texten mit unterschiedlichen Header

[10]Es wird hier nur auf die nötigsten Elemente von TEI eingegangen, da die Erläuterung aller Elemente den Rahmen dieser Arbeit sprengen würde.

In Listing 5.7 ist verdeutlicht, dass das text-Element mehrfach auftaucht. In der obersten Hierarchie fasst das text-Element alles außerhalb des Headers zusammen. In der unteren Hierarchie werden die einzelnen Texte mittels des *group*-Elementes gebündelt. Diese Form der Formatierung findet häufig bei Werken eines Autors statt. In Listing 5.8 hat jedes TEI-Element seinen eigenen Header.
In TEI ist es möglich, den Text zu untergliedern. So kann beispielsweise bei der Wiedergabe eines Buches mit den Elementen <*front*>, <*body*> und <*back*> dieses unterteilt werden. Front würde das Titelblatt, body den eigentlichen Text und back das Stichwortverzeichnis oder Glossar des Buches enthalten. Mit dem Element <*div*> würden dann die einzelnen Kapitel dargestellt. Attribute im div-Element beschreiben die Art der Untergliederung, was im folgenden Beispiel 5.9 zu sehen ist.

```
...
<text>
   <body>
      <div type="Fragment" n="t">
         <ab>
            <lb n="1t"/>
            <w n="TLG\_wID\_9219">
               <note type="Frequency">16</note>
               constituting
            </w>
         </ab>
      </div>
      ...
   </body>
</text>
...
```

Listing 5.9: Untergliederung von Textenr

Zusätzlich bietet TEI die Möglichkeit, mit nummerierten und unnummerierten div-Elementen zu arbeiten. Die einfachere Verwaltung ist bei nummerierten Elementen vorteilhaft. Die Schachtelungstiefe ist jedoch begrenzt. TEI bietet nur die Möglichkeit mit acht Hierarchien[11] zu arbeiten. Unnummerierte div-Elemente können dahingegen beliebig tief geschachtelt werden, sind aber dafür schwerer zu verwalten.
Für die Kodierung von Zeilenumbrüchen und Seitenwechsel gibt TEI mit <*lb*> für linebreak und <*pb*> für pagebreak zwei weitere Elemente vor.
Der TEI-Header stellt das Titelblatt des Textkörpers dar. Er wird verwendet, um alle relevanten Informationen zum Text bzw. zum Dokument zu speichern. Der minimale TEI-Header muss Angaben über den Autor und den Editor des Textes sowie den Titel beinhalten. In dem nachfolgenden Codebeispiel ist die Struktur des TEI-Headers verdeutlicht.

```
<teiHeader>
   <fileDesc></fileDesc>
   <encodingDesc></encodingDesc>
   <profilDesc></profilDesc>
   <revisionDesc></revisionDesc>
```

[11] Von div0 bis div7.

```
</teiHeader>
```

Listing 5.10: Grundstruktur des TEI-Headers

Das Element *fileDesc* beinhaltet bibliografische Angaben zum im Textkörper enthaltenen Text. In *encodingDesc* werden Beziehungen zwischen dem Text und der Vorlage dokumentiert. *profilDesc* enthält unter anderem Angaben zur verwendeten Sprache oder zur Entstehung des Textes. In *revisionDesc* kann schließlich die Überarbeitungsgeschichte des Textes dokumentiert werden.

5.2.2 Markup-Arten in TEI

Im vorherigen Abschnitt wurden verschiedene Markup-Arten für XML-Dokumente vorgestellt und genauer untersucht. Auch in TEI ist es möglich, TEI-Richtlinien konforme Dokumente sowohl im Stand-Off- als auch im Inline-Format zu erzeugen. Jedoch existiert aufgrund dieser Richtlinien keine Möglichkeit mehr zwei verschiedene Inline-Formate zu generieren, da sie vorgeben, Informationen, die das Objekt näher beschreiben, in XML-Attribute zu speichern. Des Weiteren müssen weiterführende Informationen zu einem Objekt in Unterelementen gespeichert werden. Aus diesen Tatsachen heraus ist anzumerken, dass TEI im Inline-Format eine Mischung zwischen den beiden Inline-Varianten aus Kapitel 5.1.1 ist. Im Anhang C.1 und C.2 sind Auszüge aus einem TEI konformen Dokument im Inline- und Stand-Off-Format präsentiert. Hier ist wieder deutlich der Unterschied zwischen diesen beiden Markup-Arten zu erkennen. Im Stand-Off Format werden alle Informationen beibehalten und das Dokument wird durch jeden Service lediglich erweitert, indem neue Informationen an das Ende des bestehenden Dokumentes angehängt werden, während im Inline-Format die einzelnen Elemente erweitert werden.

5.2.3 Performanceevaluation der Markup-Arten in TEI

In diesem Abschnitt werden die Tests der Performance der Markup-Arten, Inline und Stand-Off, in der TEI-Syntax dargestellt. [12]

5.2.4 Versuchsergebisse

In den folgenden Tabellen 5.5, und 5.6 werden die Ergebnisse der Versuchsreihen präsentiert.

Anzahl der Elemente	Gesamtverarbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)	Nachrichtengröße (in KB)
1	9	67,5	2,24
100	118	7,9	117
1.000	1.053	0,9	1.061
10.000	12.146	0,08	13.615
100.000	115.022	0,01	108.720

Tabelle 5.5: Versuchsergebnisse der Loadtests von TEI mit Inline-Markup

[12] Für die Analyse wurde das gleiche Testszenario wie in Abschnitt 5.1.3 der Markup-Arten-Tests gewählt.Aus diesem Grund wird von einer erneuten Erläuterung des Testszenarios abgesehen.

Anzahl der Elemente	Gesamtverarbeitungszeit (in ms)	Durchsatz (in Requests/Sekunde)	Nachrichtengröße (in KB)
1	9	98,5	3,23
100	92	10,7	152
1.000	625	1,6	1.177
10.000	6.729	0,15	12.262
100.000	96.342	0,01	91.005

Tabelle 5.6: Versuchsergebnisse der Loadtests von TEI mit Stand-Off-Markup

Die Tabellen bestätigen nocheinmal die Resultate der Markuptest aus dem vorangegangenen Abschnitt. Für kleiner Dokumente, welche bis zu 1.000 Elemente beinhalten, sind diese mit Stand-Off-Markup bis zu 30% gößer. Jedoch werden sie kleiner, je mehr Elemente enthalten sind. Auch hier liegt der Grund in der Arbeitsweise des FrequencyServices. Die Auswertung der Gesamtverarbeitungszeit zeigt, dass je höher die Anzahl der angefragten Sätze ist, desto länger dauert die Verarbeitungszeit von Dokumenten im Stand-Off Format gegenüber Dokumenten im Inline-Format.
In den Abbildungen 5.4 und 5.5 wurde zur Verdeutlichung der Ergebnisse, die Größe der Nachrichten sowie die Gesamtverarbeitungszeit normalisiert und dargestellt.

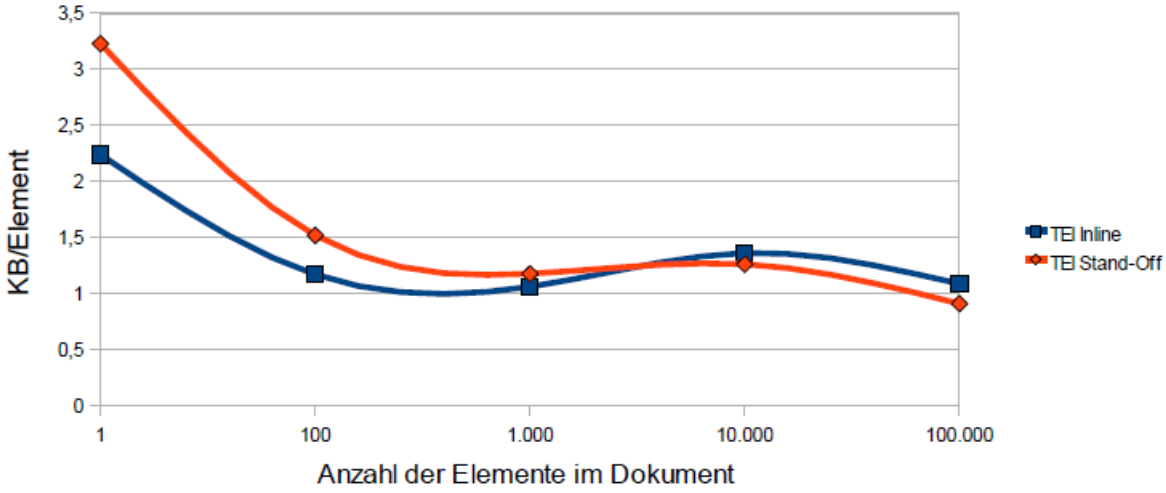

Abbildung 5.4: Durchschnittliche Nachrichtengröße pro Element

Hier werden die vorangegangenen Behauptungen nocheinmal verdeutlicht. Je mehr Sätze, d.h. Elemente, ein Dokument im Stand-Off Format enthält, desto geringer ist seine Größe, im Verhältnis zu dem Dokument mit Inline- Markup. Des Weiteren ist zusehen, dass die durchschnittliche Gesamtverarbeitungszeit bei steigender Anzahl von Elementen ebenfalls ansteigt.

5.3 Zusammenfassung

In diesem Kapitel wurden die beiden Markup-Arten Inline und Stand-Off erläutert und miteinander verglichen. Das Ergebnis dieses Vergleiches ist, dass die Entwicklung von Services,

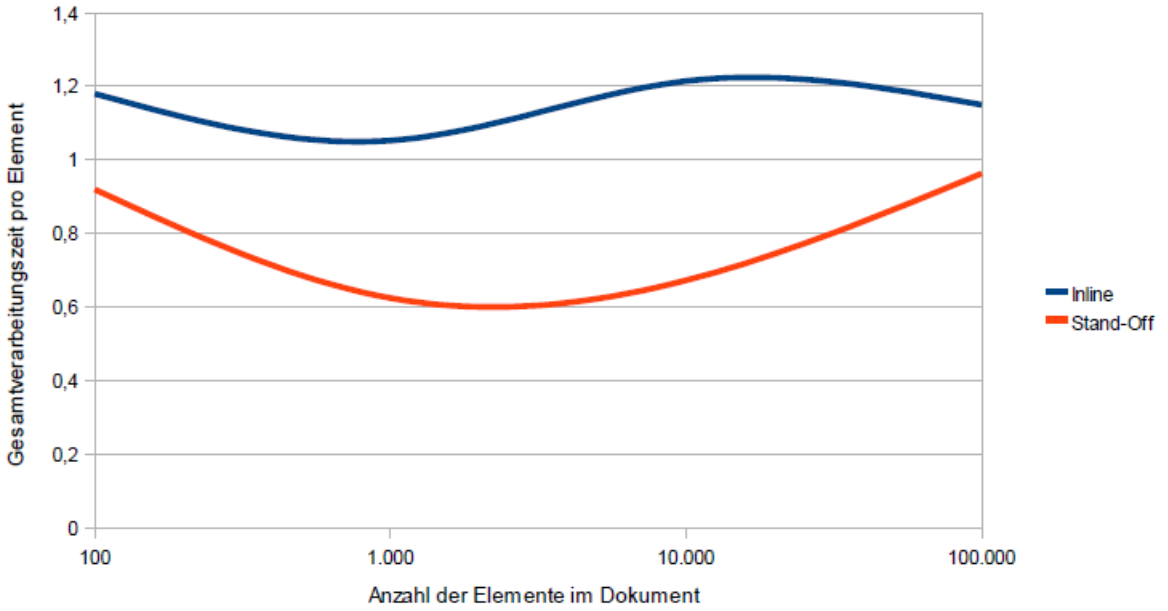

Abbildung 5.5: Durchschnittliche Gesamtverarbeitungszeit pro Element

welche Dokumente mit Stand-Off -Markup generieren, einfacher ist als bei Services die Dokumente mit Inline-Markup erzeugen. Jedoch wurde durch die Exception, die bereits bei mehr als 300.000 Elementen auftrat gezeigt, dass diese Services einen wesentlich höheren Ressourcenbedarf haben. Denn Dokumente mit Inline-Markup konnten mehr als 500.000 Elemente enthalten. Dieser Sachverhalt ist nicht nur auf der Serverseite zu beobachten. Auf der Clientseite wäre ebenfalls ein erhöhter Bedarf an Speicherressourcen notwendig, um die Informationen der Frequenzen zu den verschiedenen Wörtern zuzuordnen. Auch die Tests der Gesamtverarbeitungszeiten zeigten, dass nach einer gewissen Anzahl von Sätzen, die Dokumente im Stand-Off-Format höher ist als, bei Dokumenten im Inline-Format.
Diese Punkte zeigen, dass es besser ist mehr Aufwand in die Entwicklung der Services zu investieren, so dass diese Dokumente mit Inline-Markup generieren. Das schont Ressourcen sowohl auf der Seite des Clients als auch auf der Seite des Servers.

6 Fazit

Ziel dieser Arbeit war es zu zeigen, mithilfe welcher Technologien, Web Service Frameworks, XML-Schnittstellen und XML-Repräsentationen es möglich ist XML-Dokumente serverseitig zu generieren. Dazu wurde zu erst SOAP und REST auf ihren Grundlagen, Features und Performance genauer betrachtet. Die Motivation dafür bildete der existierende SOAP-Kernel, der mit RESTful Services angereichert werden sollte. Die Schwierigkeit in diesem Abschnitt lag vorallem in dem Finden eines gemeinsamen Nenners zur Durchführung des Vergleiches für beide Technologien. Dies gestaltete sich insofern schwierig, da es sich zum einen bei SOAP um eine Art Protokoll handelt, was definiert wie XML-Dokumente aufgebaut sein müssen damit sie von zwei Systemen interpretiert werden können. Zum anderen ist REST ein Architekturstil, der lediglich abstrakt beschreibt, wie Ressourcen-orientierte Architekturen aufgebaut sein müssen. Des Weiteren mussten geeignete Vergleichskriterien ausgearbeitet werden. Dies wurde u.a. mit der Betrachtung der wichtigsten WS-* Spezifikationen von SOAP erreicht. Anschließend wurden Möglichkeiten für REST ausgearbeitet, diese zu realisieren. Das Ergebnis der Untersuchung der Grundlagen und Features war, dass REST eine adäquate Alternative zu SOAP für die Realisierung von Web Services ist. Vorallem durch die weite Verbreitung von HTTP-Bibliotheken für eine Vielzahl von Programmiersprachen hat REST eine Vorteil gegenüber von SOAP. SOAP hingegen besitzt, durch die Vielzahl von Spezifikationen eine Vorteil für die Realisierung von komplexen Web Services. Die Performance wurde mithilfe eines linguistischen Service getestet. Das Resultat dieser Untersuchung war, dass für einfache Services eher REST zu verwenden ist, da hier kein weiterer Overhead besteht. Da die Aufgabe dieser Arbeit war, die Geschwindigkeitsvorteile einfacher Datenbank-Lookups von REST gegenüber von SOAP zu erarbeiten, fanden andere wichtige Aspekte wie das Hinzufügen, das Aktualisieren oder das Entfernen von Datensätzen keine weitere Beachtung. Dies könnte eine weiterführende Tätigkeit über diese Arbeit hinaus sein, um weitere Vor- bzw. Nachteile der Web Service Techniken gegenüberzustellen. Das darauf folgende Kapitel hatte zur Aufgabe mit Apache Axis 2 und Metro 2.0 die zwei am weit verbreitetsten Web Service Frameworks für Java mit Apache Axis 1 zu vergleichen. Ziel war es dabei, eine Entscheidungsgrundlage für eine eventuelle Neuentwicklung des existierenden SOAP-Kernels auf Basis eines der beiden anderen betrachteten Web Service Frameworks zu finden. Für den Vergleich der Frameworks wurden u.a. Kriterien wie die Möglichkeiten zur Erweiterung von SOAP-Services, Deployment-Mechanismen, Architektur oder die Entwicklung von SOAP bzw. RESTful Services in Betracht gezogen. Des Weiteren wurde die Performance mittels eines linguistischen Referenz-Services getestet. Die Erkenntnis das eine Neuentwicklung des SOAP-Kernels auf Basis von Apache Axis 2 oder Metro 2.0 die Performance dieses Kernels erheblich steigern kann war das Resultat dieser Tests. Die Entscheidung, ob die Neuentwicklung auf Basis von Apache Axis 2 oder Metro 2.0 zu erfolgen hat, muss der zuständige Entwickler treffen. Beide haben ihre Vor- und Nachteile wobei die Performance beider Frameworks ähnlich gut ist. Ein Kritikpunkt für diesen Abschnitt der Arbeit ist, dass es beispielsweise mit dem *Spring Framework*, oder *Apache*

CXF weitere nennenswerte Frameworks gibt die nicht Teil dieser Untersuchung waren. Weiterhin wurden, aufgrund der Aufgabenstellung, lediglich einfache Services die Datenbank-Lookups ausführen getestet und somit das Manipulieren oder Löschen von Datensätzen nicht mit betrachtet.
Anschließend wurden die XML-Schnittstellen StAX und DOM verglichen. Auch hier bildete die Grundlagen beider Schnittstellen den Beginn des Kapitels. Zur Durchführung der Tests wurden verschiedene Web-Anwendungen mit verschiedenen Implementierung der Schnittstellen entwickelt. Sie stellten alle einen RESTful Service zur Verfügung, den ein Client aufrief. Ergebnis dieser Untersuchung war, dass auch für die Generierung von XML-Dokumenten StAX die bessere Performance aufweist. DOM ist jedoch einfacherer anzuwenden, falls eine Manipulierung bestehender Dokumente notwendig ist. Der Kritikpunkt des Kapitels liegt darin, dass die Zeiten für das Parsen der Dokumente nicht mit untersucht wurden. Die Begründung für diese Entscheidung liegt auch hier in der Hauptaufgabe dieser Arbeit. Dies war die Untersuchung darauf, mit welchen Techniken und Verfahren XML serverseitig effektiv generiert werden kann. Ein weiterer Kritikpunkt ist, dass der Speicherverbrauch beim Erzeugen von XML nicht betrachtet wurde, da aufgrund der geringen Entfernung zwischen Server und Client die Request-Response-Zyklen zu niedrig für das minütliche Logging-Intervall von JavaMelody waren. [1]
Den Schluss dieser Arbeit bildete dann die Untersung der Repräsentationen von XML. Dabei wurden die Möglichkeien XML mit Inline-Markup und Stand-Off Markup zu erzeugen anhand von Beispielen voneinander abgegrenzt. Daran anschließend wurde mit TEI eine XML-Syntax eingeführt, mit der ebenfalls Inline- und Stand-Off-Markup untersucht wurde.
Der größte Kritikpunkt dieser Arbeit liegt in der Durchführung der Tests. Die Konzeption beinhaltete anfänglich den Aspekt der verschiedenen Größen der Nachrichten und verschiedener Entfernungen zwischen Server und Client. Dies war jedoch aufgrund mangelnder Hardware Ressourcen nicht möglich. Deshalb wurden alle Tests im Haus der Universität Leipzig durchgeführt. Sowohl Server als auch Client befanden sich innerhalb eines Netzwerkes, was es teilweise nicht möglich machte, den Server ausreichend mithilfe von JavaMelody zu überwachen. Weiterenhin wurde für die Messung der Zeiten auf der Clientseite nur *JMeter* verwendet. Ein weiteres wichtiges und weit verbreitetes Test-Tool ist *soapUI*[2] von eviware. Aufgrund der verschiedenen Art und Weise zur Messung der Zeiten wurde in zahlreichen Quellen vorgeschlagen, beide Test-Tools zu verwenden und den Durchschnitt aus den Ergebnissen zu berechnen[3]. Wegen der Komplexität der Tests wurde jedoch auf die Hinzunahme von soapUI verzichtet. Für die Messung der Geschwindigkeiten zur Übertragung von linguistischen Texten wurde lediglich XML betrachtet. Dies gilt als zusätzlicher Kritikpunkt der Arbeit denn die Möglichkiet der Erzeugung und Übertragung von *Plain Text* wurde außer Acht gelassen. Die Begründung findet sich auch hier in dem Ziel der Arbeit wieder. Denn es sollte nur die Generierung von XML untersucht werden. Dennoch ist dieser Aspekt ebenfalls wichtig und sollte im Rahmen weiterer Tätigkeiten untersucht werden.

[1] Es wurde mehrfach experimentiert, um dieses Problem zu umgehen, jedoch konnte das Intervall minimal auf sekündlich gestellt werden, was jedoch den Server zu sehr belastete und somit die Ergebnisse sehr beeinflusste.

[2] Vgl. http://www.soapui.org/.

[3] Vgl. http://www.soapui.org/userguide/loadtest/comparison.html/.

A Anhang

A WSDL Dateien

A.1 Apache Axis 1

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions targetNamespace="http://ServiceTypes"
       xmlns:apachesoap="http://xml.apache.org/xml-soap"
       xmlns:impl="http://ServiceTypes"
       xmlns:intf="http://ServiceTypes"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
       xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>
    <schema elementFormDefault="qualified"
         targetNamespace="http://ServiceTypes"
         xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="getSentences">
        <complexType>
          <sequence>
            <element name="startId" type="xsd:int" />
            <element name="endId" type="xsd:int" />
          </sequence>
        </complexType>
      </element>
      <element name="getSentencesResponse">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded"
                     name="getSentencesReturn"
                     type="xsd:string" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
  <wsdl:message name="getSentencesRequest">
```

```
    <wsdl:part element="impl:getSentences"
          name="parameters" />
  </wsdl:message>
  <wsdl:message name="getSentencesResponse">
    <wsdl:part element="impl:getSentencesResponse"
          name="parameters" />
  </wsdl:message>
  <wsdl:portType name="Sentences">
    <wsdl:operation name="getSentences">
      <wsdl:input message="impl:getSentencesRequest"
            name="getSentencesRequest" />
      <wsdl:output message="impl:getSentencesResponse"
            name="getSentencesResponse" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SentencesSoapBinding"
          type="impl:Sentences">
    <wsdlsoap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="getSentences">
      <wsdlsoap:operation soapAction="" />
      <wsdl:input name="getSentencesRequest">
        <wsdlsoap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="getSentencesResponse">
        <wsdlsoap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SentencesService">
    <wsdl:port binding="impl:SentencesSoapBinding"
          name="Sentences">
      <wsdlsoap:address location="http://localhost:8080/
          WebServicesKernelAxis1/services/Sentences" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing A.1: Automatisch generierte WSDL Datei von Apache Axis 1

A.2 Apache Axis 2

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl
    /"
      xmlns:ns1="http://org.apache.axis2/xsd"
      xmlns:ns="http://ServiceTypes"
```

```
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:ax21="http://sql.java/xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    targetNamespace="http://ServiceTypes">

<wsdl:documentation>Please Type your service description
   here</wsdl:documentation>
<wsdl:types>
  <xs:schema xmlns:ax22="http://ServiceTypes"
       attributeFormDefault="qualified"
      elementFormDefault="qualified"
      targetNamespace="http://sql.java/xsd">

    <xs:import namespace="http://ServiceTypes" />
    <xs:complexType name="SQLException">
      <xs:complexContent>
        <xs:extension base="ax22:Exception">
          <xs:sequence>
            <xs:element minOccurs="0" name="SQLState"
                nillable="true" type="xs:string" />
            <xs:element minOccurs="0" name="errorCode" type
               ="xs:int" />
            <xs:element minOccurs="0" name="nextException"
               nillable="true"
                type="ax21:SQLException" />
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:schema>
  <xs:schema xmlns:ax23="http://sql.java/xsd"
       attributeFormDefault="qualified"
       elementFormDefault="qualified"
       targetNamespace="http://ServiceTypes">
    <xs:import namespace="http://sql.java/xsd" />
    <xs:complexType name="Exception">
      <xs:sequence>
        <xs:element minOccurs="0" name="Exception"
            nillable="true" type="xs:anyType" />
      </xs:sequence>
    </xs:complexType>
    <xs:element name="SQLException">
      <xs:complexType>
        <xs:sequence>
```

```
            <xs:element minOccurs="0" name="SQLException"
                nillable="true" type="ax21:SQLException" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="getSentences">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" name="startId" type="
               xs:int" />
            <xs:element minOccurs="0" name="endId" type="
               xs:int" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="getSentencesResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0"
                name="return" nillable="true" type="xs:string
                   " />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>
  <wsdl:message name="getSentencesRequest">
    <wsdl:part name="parameters" element="ns:getSentences" />
  </wsdl:message>
  <wsdl:message name="getSentencesResponse">
    <wsdl:part name="parameters" element="
       ns:getSentencesResponse" />
  </wsdl:message>
  <wsdl:message name="SQLException">
    <wsdl:part name="parameters" element="ns:SQLException" />
  </wsdl:message>
  <wsdl:portType name="SentencesPortType">
    <wsdl:operation name="getSentences">
      <wsdl:input message="ns:getSentencesRequest"
          wsaw:Action="urn:getSentences" />
      <wsdl:output message="ns:getSentencesResponse"
          wsaw:Action="urn:getSentencesResponse" />
      <wsdl:fault message="ns:SQLException" name="
         SQLException"
          wsaw:Action="urn:getSentencesSQLException" />
    </wsdl:operation>
  </wsdl:portType>
```

```
<wsdl:binding name="SentencesSoap11Binding" type="
    ns:SentencesPortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/
     http" style="document" />
  <wsdl:operation name="getSentences">
    <soap:operation soapAction="urn:getSentences" style="
       document" />
    <wsdl:input>
      <soap:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal" />
    </wsdl:output>
    <wsdl:fault name="SQLException">
      <soap:fault use="literal" name="SQLException" />
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="SentencesSoap12Binding" type="
    ns:SentencesPortType">
  <soap12:binding transport="http://schemas.xmlsoap.org/
     soap/http" style="document" />
  <wsdl:operation name="getSentences">
    <soap12:operation soapAction="urn:getSentences" style="
       document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
    <wsdl:fault name="SQLException">
      <soap12:fault use="literal" name="SQLException" />
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
<wsdl:binding name="SentencesHttpBinding" type="
    ns:SentencesPortType">
  <http:binding verb="POST" />
  <wsdl:operation name="getSentences">
    <http:operation location="Sentences/getSentences" />
    <wsdl:input>
      <mime:content type="text/xml" part="getSentences" />
    </wsdl:input>
    <wsdl:output>
      <mime:content type="text/xml" part="getSentences" />
    </wsdl:output>
  </wsdl:operation>
```

```
  </wsdl:binding>
  <wsdl:service name="Sentences">
    <wsdl:port name="SentencesHttpSoap11Endpoint" binding="
        ns:SentencesSoap11Binding">
      <soap:address location="http://localhost:8080/
          WebServicesKernelAxis2/services/Sentences.
          SentencesHttpSoap11Endpoint/" />
    </wsdl:port>
    <wsdl:port name="SentencesHttpSoap12Endpoint" binding="
        ns:SentencesSoap12Binding">
      <soap12:address location="http://localhost:8080/
          WebServicesKernelAxis2/services/Sentences.
          SentencesHttpSoap12Endpoint/" />
    </wsdl:port>
    <wsdl:port name="SentencesHttpEndpoint" binding="
        ns:SentencesHttpBinding">
      <http:address location="http://localhost:8080/
          WebServicesKernelAxis2/services/Sentences.
          SentencesHttpEndpoint/" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing A.2: Automatisch generierte WSDL Datei von Apache Axis 2

A.3 Metro 2.0

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

<definitions xmlns:wsu="http://docs.oasis-open.org/wss
    /2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy
        "
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://SoapServices.ServiceTypes/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://SoapServices.ServiceTypes/"
    name="SentencesService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://SoapServices.ServiceTypes
          /"
          schemaLocation="http://localhost:8080/
              WebServicesKernelMetro2/SentencesService?xsd=1"
```

```
            />
    </xsd:schema>
  </types>
  <message name="getSentences">
    <part name="parameters" element="tns:getSentences" />
  </message>
  <message name="getSentencesResponse">
    <part name="parameters" element="tns:getSentencesResponse
       " />
  </message>
  <portType name="SentencesService">
    <operation name="getSentences">
      <input wsam:Action="http://SoapServices.ServiceTypes/
         SentencesService/getSentencesRequest"
          message="tns:getSentences" />
      <output wsam:Action="http://SoapServices.ServiceTypes/
         SentencesService/getSentencesResponse"
          message="tns:getSentencesResponse" />
    </operation>
  </portType>
  <binding name="SentencesServicePortBinding" type="
     tns:SentencesService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/
       http" style="document" />
    <operation name="getSentences">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <service name="SentencesService">
    <port name="SentencesServicePort" binding="
       tns:SentencesServicePortBinding">
      <soap:address location="http://localhost:8080/
         WebServicesKernelMetro2/SentencesService" />
    </port>
  </service>
</definitions>
```

Listing A.3: Automatisch generierte WSDL Datei von Metro 2.0

B SOAP-Nachrichten

B.1 Apache Axis 1

```
POST /WebServicesKernelAxis1/services/Sentences HTTP/1.0
Host: localhost:6666
Content-Type: text/xml; charset=utf-8
Content-Length: 372
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema
   -instance"
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope
         /"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soapenv:Body>
      <getSentences xmlns="http://ServiceTypes">
         <startId>1</startId>
         <endId>1</endId>
      </getSentences>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing A.4: Request des Clients

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Date: Sun, 22 Aug 2010 16:55:34 GMT
Connection: close

<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/
   soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <getSentencesResponse xmlns="http://ServiceTypes">
         <getSentencesReturn>Hallo Welt!</getSentencesReturn>
      </getSentencesResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing A.5: Response des Servers

B.2 Apache Axis 2

```
POST /WebServicesKernelAxis2/services/Sentences HTTP/1.1
Content-Type: application/soap+xml;
              charset=UTF-8; action="urn:getSentences"
User-Agent: Axis2
Host: localhost:6666
Transfer-Encoding: chunked

<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/
   soap-envelope">
  <soapenv:Body>
    <ns1:getSentences xmlns:ns1="http://ServiceTypes">
      <ns1:startId>1</ns1:startId>
      <ns1:endId>1</ns1:endId>
    </ns1:getSentences>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.6: Request des Clients

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/soap+xml; action="
   urn:getSentencesResponse";charset=UTF-8
Transfer-Encoding: chunked
Date: Tue, 06 Apr 2010 16:03:52 GMT

<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/
   soap-envelope">
  <soapenv:Body>
    <ns:getSentencesResponse
        xmlns:ns="http://ServiceTypes"
        xmlns:ax21="http://rmi.java/xsd" xmlns:ax22="http://io
            .java/xsd">
      <ns:return>Hallo Welt!</ns:return>
    </ns:getSentencesResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing A.7: Response des Servers

B.3 Metro 2.0

```
POST /WebServicesKernelMetro2/SentencesService HTTP/1.0
```

```
Host: localhost:6666
Content-Type: text/xml; charset=utf-8
Content-Length: 269
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope
    /">
    <S:Body>
        <ns2:getSentences xmlns:ns2="http://SoapServices.
            ServiceTypes/">
            <startId>1</startId>
            <endId>1</endId>
        </ns2:getSentences>
    </S:Body>
</S:Envelope>
```

Listing A.8: Request des Clients

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml;charset=utf-8
Content-Length: 577
Date: Sun, 22 Aug 2010 17:00:48 GMT
Connection: close

<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope
    /">
    <S:Body>
        <ns2:getSentencesResponse xmlns:ns2="http://
            SoapServices.ServiceTypes/">
            <return>Hallo Welt!</return>
        </ns2:getSentencesResponse>
    </S:Body>
</S:Envelope>
```

Listing A.9: Response des Servers

C Markup-Arten für TEI Dokumente

C.1 TEI konformes Beispieldokument mit Inline-Markup

```
<?xml version="1.0" ?>
<TEI>
  <teiHeader>...</teiHeader>
```

```
  <text>
    <body>
      <div type="fragment">
        <ab>
          <lb n="1">
            <w n="TLG_wId_113174">
              <note type="frequency">32</note>Hallo
            </w>
            <w n="TLG_wId_179336">
              <note type="frequency">17</note>Welt
            </w>
          </lb>
          <milestone n="TLG_sId_1" unit="sentence"></
             milestone>
        </ab>
      </div>
    </body>
  </text>
</TEI>
```

Listing A.10: Inline-Markup in TEI

C.2 TEI konformes Beispieldokument mit Stand-Off-Markup

```
<?xml version="1.0" ?>
<TEI>
  <teiHeader>...</teiHeader>
  <text>
    <body>
      <div type="fragment">
        <ab>
          <lb n="1">Hallo Welt!<milestone n="TLG_sId_1" unit=
             "sentence"></milestone></lb>
        </ab>
      </div>
      <div type="fragment">
        <ab>
          <lb n="1t">
            <w n="TLG_wId_113174">Hallo</w>
            <w n="TLG_wId_28662">Welt</w>
          </lb>
        </ab>
      </div>
      <div type="fragment">
        <ab>
          <lb>
            <w n="TLG_wId_113174">
```

```
            <note type="frequency">32</note>
          </w>
          <w n="TLG_wId_28662">
            <note type="frequency">177</note>
          </w>
        </lb>
      </ab>
    </div>
  </body>
 </text>
</TEI>
```

Listing A.11: Stand-Off-Markup in TEI

Literaturverzeichnis

[Amrhein 2010] AMRHEIN, Beatrice: *JAXB Java Architecture for XML Binding*, 2010. – URL `http://www.sws.bfh.ch/~amrhein/Skripten/XML/JAXBSkript.pdf`

[Büchler 2005] BÜCHLER, Marco: *Analyse der XML-Performanz für Suchanfragen*. 2005. – Manuskript, Universität Leipzig

[Burke 2010] BURKE, Bill: *RESTful Java with JAX-RS*. 1. Auflage. United States of America : O'Eeilly, 2010. – ISBN 978-0-596-15804-0

[Dapeng Wang 2004] DAPENG WANG, Thilo Frotscher Marc T.: *Java Web Services mit Apache Axis*. 1. Auflage. Software & Support Verlag GmbH, 2004

[Davis 2005] DAVIS, Doug: *WS-RM and WS-R: Can SOAP be reliably delivered from confusion?* W3C (Veranst.), 2005. – URL `http://www.ibm.com/developerworks/library/ws-rmpaper/`

[Dörnemann 2008] DÖRNEMANN, Kay: *Grid Computing*. Universität Marburg (Veranst.), 2008. – URL `http://www.uni-marburg.de/fb12/verteilte_systeme/lehre/ws0809/vl/gc/ch04_1.pdf`

[Erik Christensen 2001] ERIK CHRISTENSEN, Greg Meredith Sanjiva W.: *Web Services Description Language (WSDL) 1.1*. W3C (Veranst.), 2001. – URL `www.w3.org/TR/wsdl/`

[Feingold 2005] FEINGOLD, Ma: *Web Services Atomic Transaction (WS-AtomicTransaction)*. W3C (Veranst.), 2005. – URL `http://specs.xmlsoap.org/ws/2004/10/wsat/wsat.pdf`

[Fielding 2000] FIELDING, Roy T.: Architectural Styles and the Design of Network-based Software Architectures / University of Irvine, California. 2000. – Dissertation

[Foundation 2005] FOUNDATION, The Apache S.: *WebServices - Axis*. The Apache Software Foundation (Veranst.), 2005. – URL `http://ws.apache.org/axis/java/user-guide.pdf`

[Foundation 2006] FOUNDATION, The Apache S.: *Axis2 Databinding Framework*. The Apache Software Foundation (Veranst.), 2006. – URL `http://ws.apache.org/axis2/1_0/adb/adb-howto.html`

[Foundation 2009] FOUNDATION, The Apache S.: *OM Tutorial*, 2009. – URL `http://ws.apache.org/commons/axiom/OMTutorial.html`

[Francisco Curbera 2004] FRANCISCO CURBERA, Don B.: *Web Services Adressing (WS-Addressing)*, 2004. – URL http://www.w3.org/Submission/ws-addressing/

[Georg Braungart 2009] GEORG BRAUNGART, Fotis J.: *TEI in der Praxis*, 2009. – URL http://computerphilologie.uni-muenchen.de/praxis/teiprax.html/

[Grimm 2007] GRIMM, Timo: *Axis1 vs Axis2 - Middleware der nächsten Generation*. PRACTICALweb (Veranst.), 2007. – URL http://www.practicalweb.de/2009/06/axis1-axis2-middleware-der-nachsten.html

[Gupta 2007a] GUPTA, Arun: *Metro on Tomcat 6.x*, 2007. – URL http://blogs.sun.com/arungupta/entry/metro_on_tomcat_6_x

[Gupta 2007b] GUPTA, Arun: *Project Tango: Adding Quality of Service and .NET Interoperability to the Metro Web Services Stack*. Sun Microsystems (Veranst.), 2007. – URL https://wsit.dev.java.net/docs/tango-overview.pdf

[Gustavo Alonso 2004] GUSTAVO ALONSO, Harumi Kuno Vijay M.: *Web Services*. 1. Auflage. Deutschland : Springer-Verlag, 2004. – ISBN 3-540-44008-9

[Hansen 2007] HANSEN, Mark D.: *SOA Using Java Web Services*. 1. Auflage. USA : Pearson Education, 2007. – ISBN 0-13-044968-7

[Hari 2009] HARI, Sri: *Introduction to DOM4J*. javabeat (Veranst.), 2009. – URL http://www.javabeat.net/articles/44-introduction-to-dom4j-1.html

[Harold 2003] HAROLD, Elliotte R.: *Streaming API for XML*. O'Reilly (Veranst.), 2003. – URL http://www.xml.com/pub/a/2003/09/17/stax.html

[Harold 2009] HAROLD, Elliotte R.: *XOM*, 2009. – URL http://www.xom.nu/

[Hunter 2002] HUNTER, Jason: *JDOM and XML Parsing, Part 1*, 2002. – URL http://www.jdom.org/docs/oracle/jdom-part1.pdf

[IBM 20004] IBM: *Web Services Transactions specifications*, 20004. – URL http://www.ibm.com/developerworks/library/specification/ws-tx/

[IBM 2004] IBM: *Web Services Security*, 2004. – URL http://www.ibm.com/developerworks/library/specification/ws-secure/

[Iwasa 2004] IWASA, Kazunori: *Web Services Reliable Messaging TC WS-Reliability 1.1*. OASIS (Veranst.), 2004. – URL http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

[James Snell 2002] JAMES SNELL, Pavel K.: *Programming Web Services with SOAP*. 1. Auflage. United States of America : O'Eeilly, 2002. – ISBN 0-596-00095-2

[JavaMelody 2010] JAVAMELODY: *JavaMelody User Guide*. Evernat (Veranst.), 2010. – URL http://code.google.com/p/javamelody/wiki/UserGuide/

[Kotamraju 2009] KOTAMRAJU, Jitendra: The Java API for XML-Based Web Services (JAX-WS) 2.2. (2009). – URL http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index3.html

[Martin Gudgin 2005a] MARTIN GUDGIN, Mark Nottingham Hervé R.: *SOAP Message Transmission Optimization Mechanism*. W3C (Veranst.), 2005. – URL `http://www.w3.org/TR/soap12-mtom/`

[Martin Gudgin 2005b] MARTIN GUDGIN, Mark Nottingham Hervé R.: *XML-binary Optimized Packaging*. W3C (Veranst.), 2005. – URL `http://www.w3.org/TR/xop10/`

[Mehta 2003] MEHTA, Bhakti: Java Architecture for XML Binding (JAXB). (2003). – URL `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`

[Microsystems 2009] MICROSYSTEMS, Sun: *JAXB Architecture*, 2009. – URL `http://java.sun.com/webservices/docs/1.5/tutorial/doc/JAXBWorks2.html`

[Microsystems 2010] MICROSYSTEMS, Sun: *Metro Users Guide*, 2010. – URL `https://metro.dev.java.net/guide/`

[Oracle 2010] ORACLE: *The Java Web Services Tutorial*. Oracle (Veranst.), 2010. – URL `http://java.sun.com/webservices/docs/2.0/tutorial/doc/StAX2.html`

[Pareigis 2000] PAREIGIS, Prof. Dr. B.: *Lineare Algebra für Informatiker*. 1. Auflage. Deutschland : Springer-Verlag, 2000. – ISBN 3-540-67533-7

[Project 2005] PROJECT, Apache Web S.: *Axis Architecture Guide*. Apache Web Services Project (Veranst.), 2005. – URL `http://ws.apache.org/axis/java/architecture-guide.html`

[Project 2006] PROJECT, Apache Web S.: *Axis Reference Guide*. Apache Web Services Project (Veranst.), 2006. – URL `http://ws.apache.org/axis/java/reference.html#DeploymentWSDDReference`

[Project 2010] PROJECT, The Apache J.: *Apache JMeter*. The Apache Jakarta Project (Veranst.), 2010. – URL `http://jakarta.apache.org/jmeter/usermanual/intro.html`

[Rajiv Mordani 2006] RAJIV MORDANI, Robert E.: *Introducing JAX-WS 2.0 With the Java SE 6 Platform (Part 1)*. Sun Microsystems (Veranst.), 2006. – URL `http://java.sun.com/developer/technicalArticles/J2SE/jax_ws_2/`

[Roy T. Fielding 2000] ROY T. FIELDING, Richard N. T.: *Principled Design of the Modern Web Architecture*. University of California (Veranst.), 2000. – URL `http://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf`

[Ruby 2007] RUBY, Leonard Richardson & S.: *RESTful Web Services*. 1. Auflage. United States of America : O'Reilly, 2007. – ISBN 0-596-52926-0

[Schlimmer 2006] SCHLIMMER, Jeffrey: *Web Services Policy 1.2 - Framework (WS-Policy)*, 2006. – URL `http://www.w3.org/Submission/WS-Policy/`

[Sirinivas 2006] SIRINIVAS, Devanum: *Axis2/Java – Performance Testing Round #1*. WSO2 Oxygen Tank (Veranst.), 2006. – URL `http://wso2.org/library/91`

[Sun Microsystems 2007] SUN MICROSYSTEMS, Inc.: *The WSIT Tutorial*. Sun Microsystems, Inc. (Veranst.), 2007. – URL `http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/reference/tutorials/wsit/doc/WSITTutorial.pdf`

[Team 2005] TEAM, Java Web Services P.: *Streaming APIs for XML Parsers*. Sun Microsystems (Veranst.), 2005. – URL `http://java.sun.com/performance/reference/whitepapers/StAX-1_0.pdf`

[Thilo Frotscher 2007] THILO FROTSCHER, Dapeng W.: *Java Web Services mit Apache Axis2*. 1. Auflage. Deutschland : entwickler.press, 2007. – ISBN 978-3-93-5042-81-9

[Tobias Hauser 2004] TOBIAS HAUSER, Ulrich M. L.: *Web Services - Die Standards*. 1. Auflage. Deutschland : Galileo Press, 2004. – ISBN 3-89842-393-X

[Unbekannt 2010] UNBEKANNT: *Introduction to MTOM A hands-on Approach*. crossXcheck networks (Veranst.), 2010. – URL `http://www.crosschecknet.com/intro_to_mtom.php`

[Vonhoegen 2007] VONHOEGEN, Helmut: *Einstieg in XML*. 4. aktualisierte und erweiterte Auflage. Deutschland : Galileo Press, 2007

[W3C 2005] W3C: *Document Object Model (DOM)*, 2005. – URL `http://www.w3.org/DOM/`

[W3C 2007] W3C: *W3C Recommendation (Second Edition) 27 April 2007*. W3C (Veranst.), 2007. – URL `http://www.w3.org/TR/soap/`

[w3schools.com 2010] W3SCHOOLS.COM: *W3C DOM Activities*. w3schools.com (Veranst.), 2010. – URL `http://www.w3schools.com/w3c/w3c_dom.asp/`

Eidesstattliche Erklärung

“Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.”

Ort Datum Unterschrift