ETH zürich

ETH Library

Web service execution for BioOpera

Master Thesis

Author(s):
Haller, Markus

Publication date:
2003

Permanent link:
https://doi.org/10.3929/ethz-a-004492044

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Informations- und Kommunikationssysteme

Web Service Execution for BioOpera

Markus Haller

Diplomarbeit

3.März 2003

Zuständiger Professor
Prof. Dr. G. Alonso

Betreuender Assistent
Cesare Pautasso

Zusammenfassung

Das Ziel dieser Arbeit ist die Integration von Web Services in das BioOpera-System, einer Programmier- und Laufzeitumgebung mit Ausführung- und Ablaufsteuerung für aufwendige Clusterberechnungen. In einem ersten Schritt entwickelten wir für eine Web Service Schnittstelle eine Programmdarstellung in OCR (Opera Canonical Representation), eine Softwarekomponente, die durch BioOpera mit Eingabeparametern aufgerufen werden kann und eine Menge von Ausgabeparametern zurückliefert. Eine Web Service Schnittstelle kann mit Hilfe eines UDDI (Universal Description, Discovery and Integration) Browser oder durch die Angabe eines WSDL (Web Service Description Language) Dokumentes importiert werden. In einem weiteren Schritt wurde das Opera Kernel Execution Subsystem um einen Mechanismus erweitert, der Web Service Aufrufe basierend auf SOAP (Simple Object Access Protocol) ausführen kann.

Abstract

Goal of this project is the integration of web services in the BioOpera system, a programming and runtime environment for cluster computing with the capability to define, execute, monitor and manage a broad range of large-scale, complex scientific computations. To do so we modeled a Web Service Interface as an OCR Program, that is a software component which can be called from BioOpera with a set of input parameters and returns back to BioOpera a set of output parameters. A Web Service can be imported using a UDDI browser or directly using a WSDL document address location. Furthermore the BioOpera kernel has been extended with a new subsystem, adding a basic mechanism to execute a Web Service call using the SOAP protocol.

Danksagung

Ich möchte mich an dieser Stelle herzlich bei allen, die mir bei dieser Arbeit geholfen haben, danken, namentlich Christian Rupp und Peter Amberg, Prof. Dr. Gustavo Alonso, Win Bausch und den Mitarbeitern am Institut für Informations- und Kommunikationssysteme an der ETH Zürich. Besonders bedanken möchte ich mich bei meinem Assistenten Cesare Pautasso für die hilfreiche Betreuung; er konnte mir immer sofort bei Problemen weiterhelfen.
Peter Matter danke ich für die LaTeX-Hilfe und für die lustigen Abwechslungen zwischendurch.
Zu allerletzt möchte ich mich bei meiner Familie bedanken, die mich während den letzten Jahren liebevoll unterstützt hat.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 BioOpera 1

1.2 Web Service Execution Subsystem 1

1.3 System Architektur 1

1.4 Web Services 2

1.4.1 Begriffsdefinition 2

1.4.2 Technologien 3

1.5 Aufgaben und Ziele 3

1.6 Aufbau 4

2 Web Services **5**

2.1 Architektur und Standards 5

2.1.1 WSDL 6

2.1.2 UDDI 8

2.2 Java API for XML Registries (JAXR) 11

2.2.1 Was ist JAXR 11

2.2.2 Architektur / API 11

3 WSDL - OCR Abbildung **13**

3.1 Anforderungen 13

3.2 Typsysteme 14

3.2.1 BioOpera 14

3.2.2 SOAP 14

3.3 Abbildungsverfahren 15

3.4 XML Format 15

3.4.1 BioOpera Programmdefinition 15

3.4.2 Codierung des Kommando-Attributes 16

4 UDDI Browser **21**

4.1 Architektur 21

4.1.1 WSDL Importfunktion 22

4.1.2 WSDL Parsing und Programmgenerierung 23

4.1.3 Einschränkungen 24

4.2 Entwicklungstool NetBeans 27

5 SOAP Subsystem **29**
5.1 Aufbau 29
5.2 Auswahl Web Service Toolkit 29
5.3 Architektur 30
5.4 Programmausführung 31
5.4.1 SOAP Call 31
5.4.2 Eingabeprogramm 32
5.4.3 Ausgabeprogramm 33
5.5 Interoperabilität / Fehlerbehebung 34

6 Leistungsmessungen **35**
6.1 Motivation 35
6.2 Versuchsaufbau 35
6.2.1 Serverkonfiguration 36
6.2.2 SOAP Proxy Konfiguration 37
6.2.3 Messprofil 37
6.2.4 Hardware Konfiguration 38
6.3 Ergebnisse 39
6.3.1 Allgemeine Bemerkung 39
6.3.2 Definition Auslastungsfaktor 39
6.3.3 Auslastungstabellen 40
6.3.4 Zeitmessungen 41
6.4 Schlussfolgerungen / Konfigurationsprofile 42

7 Erweiterungen und Ausblick **53**
7.1 Automatische OCR Generierung für Prozesse 53
7.2 UDDI Registry V3 53
7.3 Typsystem 54
7.4 Host based authentication 54
7.5 ExecutionMultiQuery 54
7.6 Ausblick 54
7.7 Persönlicher Schlusskommentar 55

A Aufgabenstellung: Web Service Execution for BioOpera **57**

B Program Execution Client Protocol **61**
B.1 Purpose 61
B.2 Protocol update 61
B.3 Communication topics 61
B.3.1 Resource Management 62
B.3.2 Program Execution 62
B.3.3 System Administration 62
B.4 Resource Manager Messages 62
B.4.1 Host state query 62
B.4.2 Full Host state query 63
B.4.3 Host state update 63
B.5 Program Execution Messages 64
B.5.1 Execute Command 64
B.5.2 Execution Done 64
B.5.3 Abort Execution 65

B.5.4 Suspend Execution 65
B.5.5 Resume Execution 66
B.5.6 Execution Query and Execution Query Recover 66
B.5.7 IsPersistent 67
B.6 System Administration Messages 67
B.6.1 PEC shutdown request 67
B.6.2 PEC shutdown when empty request 67
B.6.3 PEC information query 67
B.6.4 PEC information query XML 68

C Web Services Execution Subsystem for BioOpera 69
C.1 Introduction 69
C.2 Overview 69
C.3 Architecture 70
C.4 UDDIBrowser 71
C.4.1 Overview 71
C.4.2 Compilation Script 72
C.4.3 Installation 73
C.4.4 Configuration 73
C.4.5 Running the UDDIBrowser 73
C.4.6 Documentation 74
C.5 WSDL Import 75
C.5.1 OCR Program representation 75
C.5.2 Documentation 76
C.6 SOAP Proxy 77
C.6.1 Overview 77
C.6.2 Compilation Script 77
C.6.3 Installation 78
C.6.4 Configuration 78
C.6.5 Running the SOAPProxy 78
C.6.6 Documentation 79
C.7 Library Version Listing 80

D Messresultate 83
D.1 Messungen mit einem SOAPProxy 83
D.1.1 Testreihe 1 Kapazität 100 83
D.1.2 Testreihe 7 Kapazität 100 84
D.1.3 Testreihe 8 Kapazität 100 85
D.1.4 Testreihe 1 Kapazität 200 86
D.1.5 Testreihe 7 Kapazität 200 87
D.1.6 Testreihe 8 Kapazität 200 88
D.1.7 Testreihe MIX Kapazität 100 und 200 89
D.2 Messungen mit 4 SOAPProxy's 90
D.2.1 Testreihe 8 Kapazität 200 90
D.2.2 Testreihe MIX Kapazität 200 91

E XML Schema für SOAP Subsystem XML-Kommando 93

F Sourcecode Axis Test Web Service 95

Kapitel 1

Einleitung

1.1 BioOpera

BioOpera ist eine Programmier- und Laufzeitumgebung mit Ausführung- und Ablaufsteuerung für aufwendige Clusterberechnungen, mit der eine Vielzahl von komplexen wissenschaftlichen Berechnungen ausgeführt werden können. Berechnungen werden im BioOpera-System als Prozesse definiert - ein Prozess wird als gerichteter Graph modelliert. Knoten repräsentieren Aufgaben und Pfeile bestimmen den Kontroll- / Datenfluss zwischen den Aufgaben. Mit diesem Modell können Sequenzen von Programmausführungen (und deren Datenaustausch) in einer verteilten und heterogenen Umgebung ausgeführt werden.

1.2 Web Service Execution Subsystem

Mit Hilfe dieses Projektes wurde die Schnittstellentechnologie SOAP in das System integriert, welche (in Zukunft) die Kommunikation mit allen erdenklichen System erlaubt, da zum jetzigen Zeitpunkt beinahe jeder Systemlieferant eine Web Service Schnittstelle in seine Produkte integriert. Die Anforderungen an die Schnittstelle sind minimal: Für die Kommunikation wird im Normalfall lediglich eine HTTP-Verbindung über Port 80 benötigt. Damit ist es sehr einfach möglich, Berechnungen geographisch verteilt ausführen zu lassen und Rechenkapazitäten zusammenzuschliessen.
Mittels eines grafischen Tools kann eine ”UDDI Registry”, eine Art gelbe Seiten des Internet, nach Web Service Schnittstellen durchsucht werden; die ausgewählten Schnittstellen werden automatisch per Mausklick in die Prozess-Entwicklungsumgebung importiert und können in einen Prozess eingebunden werden.
Die Ausführung des einzelnen Web Services erfolgt (für das System transparent) auf einem Service-Proxy, welcher dem Cluster zugewiesen ist. Die gewählte Architektur unterstützt eine unlimitierte Anzahl dieser Service-Proxy’s und garantiert damit Skalierbarkeit des Systems.

1.3 System Architektur

BioOpera wurde als verteiltes System implementiert und besteht aus mehreren Komponenten, die spezifische Aufgaben innerhalb des Systems übernehmen. Zur Kommuni-

kation zwischen den Komponenten wurde eine eigenes, auf Sockets basierendes, Protokoll spezifiziert, das Opera Wire Protocol [1].

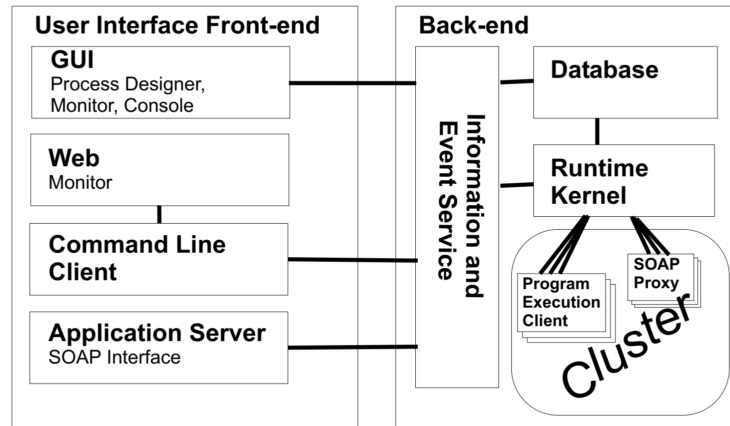


Abbildung 1.1: BioOpera Systemkomponenten

Back-end Komponenten:

Information Server: Datenverwaltung, Eventverwaltung, Kommunikation zu den Clients (API Schnitstelle)

Runtime Server: Prozessausführung, Scheduling, Überwachung des Clusters

Program Execution Client (PEC): Starten und Überwachen der Programme auf den Rechnern des Clusters, Messen der Belastung eines Rechners

SOAPProxy: Ausführen und Überwachen von Web Service Aufrufen

Front-end Komponenten (Clients):

Command Line Client: Command Line Tool zur Verwendung des APIs

GUI: Prozess-Design, Prozessmanagement (Starten, Abbrechen), Browsing in der Datenbank

WEB Interface: Prozessmanagement (Starten, Abbrechen), Browsing in der Datenbank

1.4 Web Services

1.4.1 Begriffsdefinition

Web Service ist ein genereller Begriff, der zur Zeit nicht durch eine geschlossene Menge von Technologien oder Architekturen definierbar ist. Web Services umfassen folgende Themen [2]:

[1] siehe [1] Kapitel ”Communication Protocols”

Distributed Objects and Application Integration: Web Service Anwendungen sind verteilte Applikation, die über ein Netzwerk miteinander kommunizieren (Austausch von Datenobjekten oder Aufruf von Schnittstellenfunktionen).

EDI / B2B: Austausch von elektronischen Geschäftsdokumenten über ein Netzwerk.

World Wide Web: Via HTTP-Protokoll können (für den Menschen lesbare) Dokumente abgerufen und Informations-, Produkt- und Serviceanfragen aufgegeben werden.

Die folgende, vom World Wide Web Consortium stammende Definition wurde aus dem Englischen übersetzt:

"Ein Web Service ist ein durch ein URI identifizierbares Software-System, dessen öffentliche Schnittstellen und Bindungen durch XML definiert und beschrieben sind. Diese Definition kann durch andere Software-Systeme gefunden werden. Software-Systeme können mit dem Web Service durch die in der Definition beschriebene Weise unter Benutzung von XML Nachrichten (basierend auf Internetprotokollen) interagieren." [2]

Vage gesagt, ist ein Web Service eine plattform- und sprachunabhängige Internet-basierte Applikation, die eine bestimmte Aufgabe gemäss einer Standard-Spezifikation ausführt. Die Applikation kann durch XML beschrieben werden und benutzt XML für die Kommunikation - mit den Vorteilen, dass sie einfach zu publizieren ist, auf eine standardisierte Weise auffindbar ist und dynamisch in einem verteilten System aufgerufen werden kann. Es ist nicht zwingend, dass HTTP oder SOAP für die Ausführung eines Web Service benutzt werden.

Wenn in dieser Arbeit von Web Services gesprochen wird, dann bezieht sich dies (falls nicht anders vermerkt) auf Web Services basierend auf den Technologien SOAP, WSDL und UDDI.

1.4.2 Technologien

Die momentan populärsten Web Service Technologien sind, nebst dem World Wide Web und XML, das SOAP Protokoll (Simple Object Access Protocol [3]) und WSDL (Web Services Description Language [4]) - über 70 Anwendungsplattformen bieten ein Toolkit für Web Services an. Das UDDI (Universal Description, Discovery and Integration) Protokoll [5] gehört zu den Kerntechnologien von Web Services, fand aber bis jetzt nicht den "vorgesehenen" Anklang im B2B-Bereich, die öffentlich zugänglichen UDDI Global Registries (IBM, Microsoft, HP, ...) enthalten nur ein paar tausend Einträge. Kapitel 2 enthält eine detaillierte Beschreibung von WSDL und UDDI.

1.5 Aufgaben und Ziele

Der folgende Abschnitt soll eine Übersicht über die Aufgaben und Ziele dieser Arbeit geben. Die offizielle Aufgabenstellung ist auf Seite 57 zu finden.

WSDL - OCR Mapping

Mittels der BioOpera Program Registration können Aufrufe von Standard-UNIX-Programmen definiert werden. Für das SOAP Subsystem musste ein ähnlicher Mechanismus unter Einbehaltung des bestehenden OCR-Formats erarbeitet werden. Dadurch wird ein Web

Service Aufruf im BioOpera Visual ProcessDesigner wie ein UNIX-Programmaufruf dargestellt.

Web Service Import Tool

Web Service Schnittstellen können mittels WSDL vollständig beschrieben werden. Eine Beschreibung in WSDL soll durch Angabe einer URL oder durch Laden einer Datei in den BioOpera Visual Process Designer importiert werden können.
In einem zweiten Schritt soll das Suchen von Web Service Schnittstellen durch einen Browser für UDDI Registries unterstützt werden. Die gefundenen Schnittstellen sollen per Mausklick in den BioOpera Visual Process Designer importiert werden können.

Web Service Execution System

Für die Ausführung und Überwachung von Web Service Aufrufen soll ein entsprechendes Subsystem in BioOpera realisiert werden. Das Subsystem soll ausführlich getestet werden.

Performance and Scalability

Messungen sollen die Leistungsfähigkeit und die Skalierbarkeit des SOAP Subsystem aufzeigen. Durch die Messresultate sollen optimale Konfigurationsdaten für das SOAP Subsystem gefunden werden.

1.6 Aufbau

- In Kapitel 2 wird auf die für die Realisation des ImportTools verwendeten Technologien eingegangen. Für allgemeine Information zum Thema Web Services sei auf die Diplomarbeit von Christian Rupp verwiesen ([6] Kapitel 2).
- Kapitel 3 definiert, wie eine WSDL Schnittstellendefinition in eine OCR-Darstellung (Opera Canonical Representation) abgebildet wird. Die Abbildung ist die Grundlage für die Implementation des WSDL ImportTools.
- Kapitel 4 behandelt das Design und die Implementation des WSDL ImportTools.
- Kapitel 5 enthält eine Übersicht über den Aufbau des SOAP Subsystems und über die Funktionsweise der Programmausführung.
- Kapitel 6 beschreibt den Versuchsaufbau der Leistungsmessungen und illustriert mit verschiedenen Grafiken die erhobenen Daten.
- Im letzten Kapitel werden mögliche Erweiterungen besprochen und Lösungsansätze skizziert.

Kapitel 2

Web Services

In diesem Kapitel wird auf die Technologien WSDL und UDDI detailliert eingegangen - dies soll dem Leser das nötige Wissen vermitteln, um die in den folgenden Kapiteln vorgestellten Implementationen nachvollziehen zu können. Zudem wird die Programmierschnittstellen JAXR (Java API for XML Registries) vorgestellt.

2.1 Architektur und Standards

Die Web Service Architektur ist darauf ausgerichtet, dass zwei unterschiedliche Informationssysteme miteinander kommunizieren können. Dies wird hauptsächlich durch die Verwendung von XML (Datenportabilität) und dem HTTP-Protokoll (universelle Transportmethode) erreicht. Weitere wichtige Faktoren in der Architektur sind Skalierbarkeit, Effizienz und Sicherheit. Eine Fülle von neuen Standards/Protokollen sind entstanden rsp. sind in der Entwicklung. Die momentan drei wichtigen Protokolle sind das **Simple Object Access Protocol** (SOAP [3]), die **Web Service Description Language** (WSDL [7]) und das **Universal Description, Discovery and Integration Protocol** (UDDI [5]) - sie machen den Kern der Web Service Architektur aus.

Der Vorteil von Web Services liegt klar in der (noch nicht vollständig vorhandenen) Interoperabilität zwischen verschiedenen Plattformen (siehe z.B. [8] Link Interop). Web Services haben das Potential, Einsparungen bei Integrationskosten zu fördern und die vereinfachte Entwicklung und Wiederverwendung von Komponentensoftware zu ermöglichen.

SOAP definiert wie XML-basierte Informationen in einem verteilten, dezentralisierten System zwischen einzelnen Softwarekomponenten ausgetauscht werden können. SOAP ist im Prinzip ein zustandsloses, unidirektionales Nachrichtenprotokoll - durch Kombination mit komplexeren Interaktionsmustern wie "request/response" (abhängig vom verwendeten Transportprotokoll) können verschieden Anwendungsgebiete abgedeckt werden. SOAP definiert keine Semantik für die auszutauschenden Nutzdaten, bietet aber ein erweiterbares Framework, mit dem applikationspezifische Daten befördert werden. SOAP beschreibt ausserdem die erforderlichen Massnahmen, welche eine Softwarekomponente, die eine SOAP-Nachricht empfängt, treffen muss [9]. Weitere Informationen finden sich in [6], [10] und [3].

Die Web Service Description Language definiert eine Schnittstellenbeschreibung, die XML Schema für die Datentypdefinition vorsieht. Weitere Information folgen in Sektion 2.1.1.
UDDI spezifiziert eine Plattform zur Lokalisierung von Web Services im Internet. Anwender und Applikation sollen Web Services schnell, einfach und dynamisch finden und verwenden können. Daneben existieren andere (proprietäre) "Service Discovery"-Protokolle und Methoden: IBM's ADS [11], Microsoft's DISCO, WS-Inspection [12], Service Broker wie XMethods (http://www.xmethods.net).

Das Zusammenspiel der Standards ist in Abbildung 2.1 dargestellt. Ein Serviceanbieter registriert in einem Verzeichnis die zur Verfügung gestellten Funktionen. Ein Konsument benutzt das Verzeichnis, um eine gewünschte Funktion ausfindig zu machen und benutzt diese unter Verwendung der beim Serviceanbieter vorhandenen Funktionsbeschreibung.

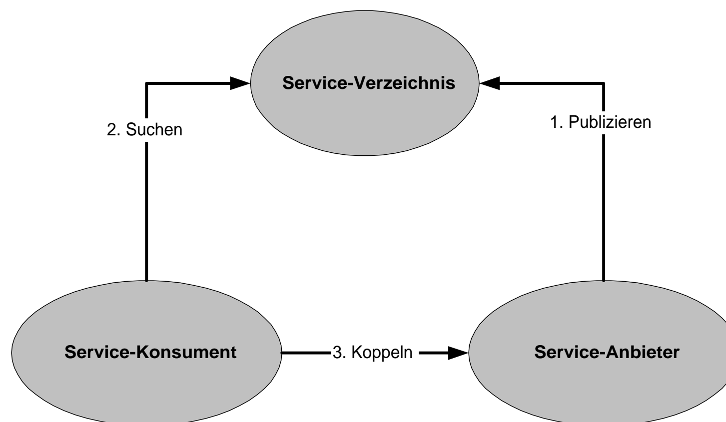


Abbildung 2.1: "publish, find, bind"

Die Web Service Standards werden hauptsächlich durch Arbeitsgruppen des W3C unter Einbezug der Öffentlichkeit ausgearbeitet und publiziert.

2.1.1 WSDL

Mittels der Web Service Description Language kann eine Web Service Schnittstelle durch ein XML Dokument beschrieben werden - im Wesentlichen besteht ein WSDL Dokument aus einer Syntaxbeschreibung der Schnittstelle und entsprechende Zugangs-Information. Dieses Kapitel geht auf die technische Seite von WSDL ein, eine allgemeine Zusammenfassung findet sich in [6] Kapitel 2.2.3.
Abbildung 2.2 zeigt den generellen Aufbau eines WSDL Dokumentes und die Beziehung zwischen den einzelnen WSDL-Elementen. Ein WSDL-Dokument besteht in seiner XML Struktur aus einem *definitions*-Element mit den Kindern *types*, *message*, *portType*, *binding* und *service*.

definitions bildet das Wurzelelement des WSDL Dokumentes und definiert einen oder mehrere Services. Eine Namensangabe ist optional. Das "`definitions`"-Element definiert zudem die im Dokument verwendeten Namensräume:

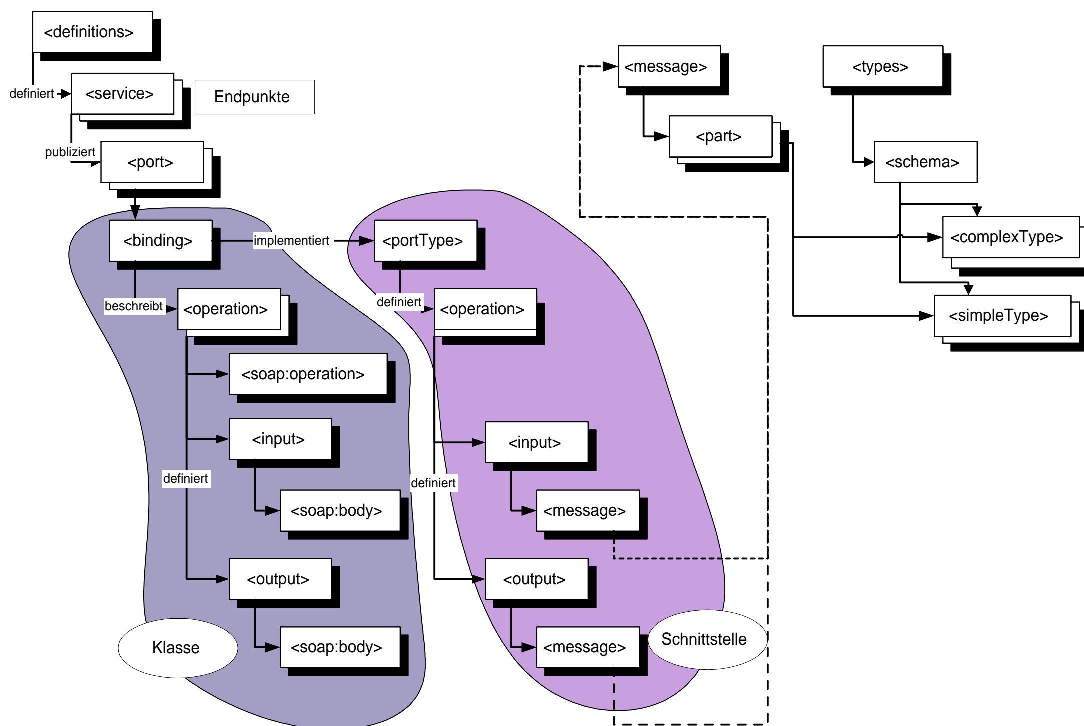


Abbildung 2.2: WSDL Struktur Diagramm [10]

`targetNamespace` Der logische Namensraum für Informationen für diesen Service (WSDL Dokumente können andere WSDL Dokumente importieren - die Verwendung eines eindeutigen Namensraums garantiert, dass keine Konflikte auftreten).

`xmlns` Definiert den Standardnamensraum für dieses WSDL Dokument mit dem Wert `http://schemas.xmlsoap.org/wsdl/`. Alle Elemente wie `types` oder `message` gehören diesem Namensraum an.

`xmlns:xsd` **und** `xmlns:soap` Standard Namensraum-Definitionen für SOAP - spezifische Informationen und Datentypen.

Beschreibung der einzelnen WSDL Elemente:

`types` Enthält die Definitionen der komplexen Datentypen, die im WSDL Dokument verwendet werden. Einfache Typen können auch direkt in den entsprechenden Elementen definiert werden. Das `types`-Element kann mehrere `schema`-Elemente enthalten, die einem **XML Schema** [7] entsprechen. Die Typdefinitionen können über die Namensangabe der Typdefinition referenziert werden. Das jeweilige Schema definiert seine eigenen Namensräume, die für die Typangaben verwendet werden.

`message` Abstrakte Definition der SOAP-Nachrichten. Ein Definition kennzeichnet sich durch einen Namen und einem (oder mehreren) `part`-Element aus. Das

part-Element entspricht im "rpc"-Modus einem Funktionsargument (die Namensangabe wird für die Identifikation des einzelnen Argumentes verwendet, d.h. in der XML Nachricht existiert ein Element, dass der Namensangabe entspricht), im "doc"-Modus wird stets nur ein part-Element verwendet, welches das zu transportierende XML Dokument repräsentiert (in diesem Fall wird die Namensangabe nicht verwendet). Informationen zum "rpc"/"doc"-Modus finden sich in der binding-Definition.

portType Abstrakte Definition einer Menge von Operationen, die einem oder mehreren Endpunkten zugeordnet sind. Endpunkte sind im service-Element definiert. Die einzelnen Operation sind abstrakt definiert und repräsentieren die durch diesen Service unterstützten Funktionen. Das operation-Element referenziert im input-Element und im output-Element je ein message-Element.

binding Das binding-Element beschreibt, welches konkrete Protokoll und welches Datenformat für eine Operation verwendet wird. Im Fall von SOAP wird die Transportmethode durch ein (wiederholtes) binding-Element spezifiziert. Die operation-Elemente spezifizieren den Informationstyp ("rpc" für Prozedur-orientiert oder "doc" für Dokument-orientiert [1]) der SOAP-Nachricht und das Datenkodierungsverfahren.

service Das service-Element enthält in den port-Elementen die Service-Endpunkte, die Portadresse. Ein Service ist eine Sammlung von Ports.

2.1.2 UDDI

Das Universal Description, Discovery and Integration of Web Services (UDDI) Protokoll [5] stellt die zwei Funktionen publish & find (publizieren und finden) und ergänzt damit WSDL, welches die Funktionen find & bind (finden und koppeln) bereitstellt. Eine UDDI Registry verwendet das UDDI Protokoll, um seine Funktionen anzubieten. Das Herzstück einer UDDI Registry ist das komplexe Datenmodell [13], das viele verschiedene Information (Abbildung 2.3: Organisationsdaten, Kontaktadressen, Servicebeschreibungen, WSDL-Links,...) aufnehmen soll. Implementationen einer UDDI Registry werden unter anderem von IBM, Microsoft, SAP oder Novell angeboten.

Das Informationsmodell (Abbildung 2.4), welches UDDI zu Grunde liegt, ist in einem XML Schema (Version 2: [14]) definiert. Das Schema definiert fünf grundlegende Typen [13]:

Business Entity Enthält Informationen zu einer Organisation, entspricht den "Weissen Seiten". Eine Business Entity enthält Assoziationen zu Business Service-Objekten.

Business Service Beschreibung einer bestimmten Gruppe von technischen Services. Entspricht den "Gelben Seiten".

Binding Template Technische Informationen und Spezifikation über den Aufbau eines Service. Binding Template's werden durch ein Business Service-Objekt aggregiert.

[1] siehe http://www.w3.org/TR/wsdl12-bindings/#_soap_operation_style

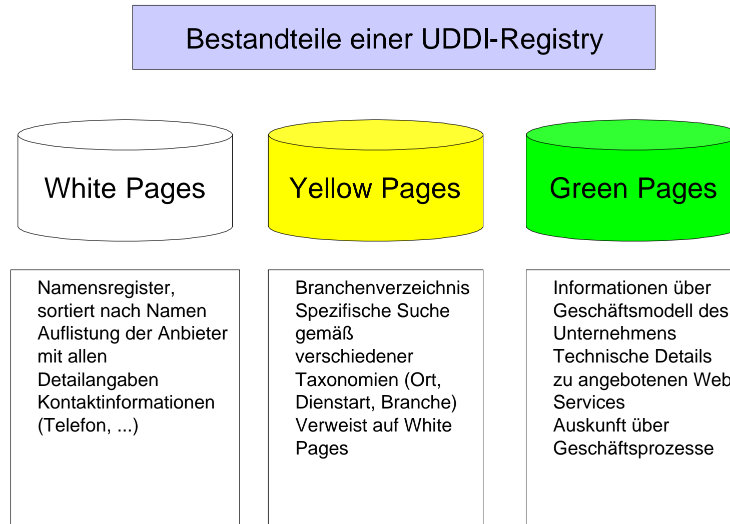


Abbildung 2.3: Bestandteile einer UDDI Registry

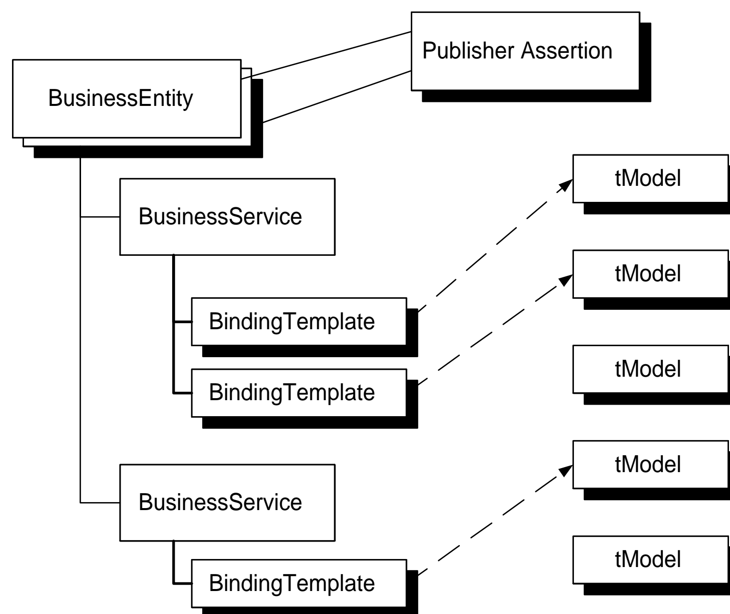


Abbildung 2.4: UDDI Datenmodell

`tModels` Ein `tModel` spezifiziert einen Service oder eine Taxonomie und bildet die Basis für technische Spezifikationen. Eine Schnittstellenspezifikation für einen Service wird aus den Referenzen der `Binding Templates` auf die `tModels` definiert. Entspricht den "Grünen Seiten".

`Publisher Assertion` Enthält Informationen zu einer Beziehung zwischen zwei Organisation. Mit diesem Typ können Unternehmen ihre Unternehmensstruktur (z.B. Konzern mit Divisionen Datenbanksoftware, Unternehmensberatung,...) abbilden.

Die UDDI-Spezifikation definiert eine Web Service Schnittstelle für den Zugriff auf die UDDI Registry. Die Schnittstelle ist in 2 Funktionsbereiche eingeteilt:

`Inquiry API` Definiert Zugriffs- und Suchfunktionen um ausgehend von Organisationen auf Informationen zu Web Services zuzugreifen. Eine Authentifizierung ist nicht notwendig.

`Publisher API` Definiert Funktionen für das Speichern, Ändern und Löschen von Daten in der UDDI Registry. Authentisierung ist für diese Funktionen zwingend vorgesehen, d.h. es muss eine vorgängige Registrierung beim Betreiber der UDDI Registry erfolgt sein.

Die Suche in einer UDDI Registry läuft meist nach folgendem Schema ab [15]:

1. Suche nach Unternehmen. Dies ist der Startpunkt für eine Abfrage, mit den zurückgelieferten Übersichtsinformationen (”BusinessInfo”, basierend auf der `BusinessEntitiy`-Datenstruktur) soll die Suche fortgesetzt werden (”TopDown”-Verfahren).

2. Im nächsten Schritt geht es mit den Servicekategorien weiter. Der Anwender oder die Softwarekomponente wählt eine Kategorie aus und fragt alle Services zu dieser Kategorie ab.

3. Das Ergebnis der Kategorienabfrage liefert eine Menge von Servicespezifikationen (`tModels`). Die Auswahl einzelner Spezifikationen beendet die Suche.

4. Der Anwender oder die Softwarekomponente ruft eine Liste der Zugriffsinformationen (`binding templates`) zu den ausgewählten Services ab, welche unter anderem die Adresse des WSDL Dokumentes enthalten können.

Die momentan verfügbaren Suchfunktionen sind relativ beschränkt. Es kann nur nach einem Kriterium gesucht werden (Organisationsname, Standort, Geschäftstyp, Servicetyp,...), was für den Progammierer sehr viel Aufwand bedeutet [16]. Die Suche kann auf Elemente einer bestimmten Taxonomie rsp. einen bestimmten Identifikationstyp beschränkt werden, darunter gehören auch die Konzepte WSDL (`wsdlSpec`), SOAP (`soapSpec`) und XML (`xmlSpec`). Das WSDL-Konzept kann dazu benutzt werden, Service Schnittstellen zu suchen, die durch ein WSDL Dokument definiert sind. Zu den bekannten Taxonomien gehören ”North American Industry Classification System” (NAICS), ”Universal Standard Products and Services Classification” (UNSPSC), ”Standard Industrial Classification ” (SIC) und GEO (Geographischer Code ISO 3166), zu den bekannten Identifikationstypen gehören ”Dun & Bradstreet D-U-N-S® numbers” (D-U-N-S), ”Thomas Register numbers” und US Tax ID. Für weitere Information über Taxonomien und Identifikationstypen sei auf die Spezifikation [17] verwiesen.

2.2 Java API for XML Registries (JAXR)

Die in diesem Kapitel vorgestellte Programmierschnittstelle wird in der UDDIBrowser-Implementation in Kapitel 4 verwendet, um auf eine UDDI Registry zuzugreifen. Das Kapitel soll einen Einblick in die Funktionalität und die Verwendung von JAXR geben.

2.2.1 Was ist JAXR

Das `Java API for XML Registries` (JAXR) ist ein standardisiertes Java API, um universell auf verschiedene Implementationen einer `XML Registry` zugreifen zu können. Eine `XML Registry` ist eine Infrastruktur für das Entwickeln, Publizieren und Finden von Web Services. Es existieren mehrere Spezifikation für `XML Registries`, beispielsweise der `ebXML Registry and Repository standard` [18] oder die aus dem vorherigen Kapitel bekannte UDDI Spezifikation.

JAXR bietet dem Programmierer ein einfaches und abstraktes API für den Zugriff auf verschiedene `XML Registries` durch das JAXR Informationsmodell, welches den Inhalt und die Metadaten einer `XML Registry` beschreibt. Eine Softwarekomponente, die JAXR benutzt, ist portabel bezüglich verschiedenen Implementationen von `XML Registries` und erweitert zusätzlich den Funktionsumfang, der durch eine `XML Registry` zur Verfügung steht. Die aktuelle JAXR Spezifikation beinhaltet detaillierte Abbildungen zwischen dem JAXR Informationsmodell und den Spezifikationen ebXML und UDDI Version 2.

JAXR kennzeichet sich durch ein hohes Mass an Metadaten für Klassifikationen und Assoziationen und durch hohe Suchfunktionalität aus. JAXR gehört zur Familie der Java XML API's wie das `Java API for XML Processing` (JAXP), die `Java Architecture for XML Binding` (JAXB), das `Java API for XML-based RPC` (JAX-RPC) oder das `Java API for XML Messaging` (JAXM). Information zur Programmierung eines JAXR Clients findet sich im Tutorial [19].

2.2.2 Architektur / API

Die JAXR-Architektur besteht aus 2 Teilen, einem abstrakten Teil für die Clientprogrammierung und einem Provider-Teil, einer Implementation der JAXR-Schnittstelle, die den Zugriff auf eine bestimmte Klasse von Verzeichnisdiensten, welche die gleiche Spezifikation erfüllen, ermöglichen. Ein Provider muss zwei `packages` implementieren: **javax.xml.registry** (API Schnittstellen und Klassen die den Zugriff auf ein Verzeichnis definieren) und **javax.xml.registry.infomodel** (Schnittstellen für die Definition des Informationsmodells; Typendefinition für Verzeichnisobjekte und Assoziationsmodellierung). Das grundlegende Objekt in dieser `package` ist das `RegistryObject`, welches Typen wie Organisation, Service oder Bindung zu Grunde liegt.

Die wichtigsten Elemente der Package **javax.xml.registry** sind:

`connection` Diese Schnittstelle repräsentiert eine Verbindung mit einem `Registry` Provider. Alle Kommunikationsvorgänge werden über diese Schnittstelle ausgeführt.

`RegistryService` Ein `RegistryService`-Objekt wird dazu benutzt, Schnittstellenobjekte zu generieren, die den Zugriff auf eine `Registry` ermöglichen. Die zwei wichtigsten Schnittstellen sind `BusinessQueryManager` (Suchfunktionen) und `BusinessLifeCycleManager` (Datenaktualisierungen durch

die zwei Funktionen Speichern und Löschen). Fehlersignalisation wird generell durch die Klasse `JAXRException` (oder deren Subklassen) umgesetzt. Viele Klassen im JAXR API verwenden Kollektionen (`Collection`) als Argumente oder Rückgabewerte, um mit mehreren `Registry` Objekten gleichzeitig arbeiten zu können [20].

Kapitel 3

WSDL - OCR Abbildung

Die Opera Canonical Representation beschreibt Prozesse, Programme und Ressourcen; diese kann in XML Syntax oder in Textform vorliegen. Der erste Schritt in der Entwicklung des SOAP Subsystems bestand in der Abbildung einer Web Service Schnittstelle (definiert durch ein WSDL Dokument) auf eine OCR Darstellung.

3.1 Anforderungen

Die Anforderungen an die Abbildungen waren einerseits durch das BioOpera-System gegeben, andererseits durch den Informationsbedarf des Programmcodes für die Web Service Ausführung und für die Generierung der SOAP Inhaltsnachricht. Massgebende Kriterien für die Abbildung waren Einfachheit, Erweiterbarkeit, Robustheit und Kompatibilität zum bestehenden OCR-XML-Format.

BioOpera benötigt für eine Programmdefinition Information über die Eingabeparameter, die Ausgabeparameter, die Resourcenzuteilung und eine Kommandozeile beliebigen Inhalts. Die Baumstruktur einer OCR Programmdefinition ist in der folgenden Abbildung dargestellt:

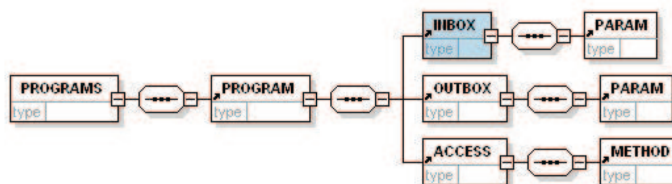


Abbildung 3.1: XML Struktur der OCR Programmdefinition

Ein Programm wird mit einem INBOX-Element für die Eingabeparameter, einem OUTBOX-Element für die Ausgabeparameter und einem ACCESS-Element für die Kommandozeile/Ressourcenangabe assoziiert. Das INBOX-Element und das OUTBOX-Element enthalten eine beliebige (0 inkl.) Anzahl von Parameterdefinitionen, die aus einem Namen, einer Typ-Angabe und optional einem Standardwert bestehen. Die Ty-

pangabe erlaubt die Typen `String` und `String[]` (String-Array). Das ACCESS-Element enthält mindestens ein METHOD-Element, welches das auszuführende Kommando, die zu verwendende Ressource und eine Subsystemangabe enthält.

Um eine Web Service Operation ausführen zu können, sind folgende Information erforderlich:

- **Adresse (URL) der WSDL Schnittstellendefinition**
- **Service-Name**
- **Port-Name**
- **Operation-Name**
- **Eingabeparameternamen- und werte**

Durch die Angabe des Service-, Port- und Operationsnamen ist eindeutig spezifiziert, welche Operation im WSDL Dokument referenziert wird. Mittels des WSDL Dokuments kann ein Web Service Toolkit (sofern unterstützt) eine Autokonfiguration vornehmen. Dies hat mehrere Vorteile: durch die Autokonfiguration wird der Umfang des Programmcodes in der Client-Software klein und Änderungen an der WSDL Schnittstellenbeschreibung werden zu einem Teil automatisch berücksichtigt. Im Falle eines statischen Clients müsste bei einer Änderung der WSDL Schnittstellenbeschreibung neuer Programmcode (`Stubs`) generiert werden, das ganze Softwareprojekt müsste überarbeitet und neu kompiliert werden.
Die Ein- und Ausgabeparameter können einfachen oder komplexen Typs sein.

3.2 Typsysteme

Beim Design der Abbildung galt es dem Problem der unterschiedlichen Typsysteme Rechnung zu tragen - unter Berücksichtigung des verwendeten Toolkits (siehe Kapitel 5.3) zur Ausführung von Web Services.

3.2.1 BioOpera

BioOpera kennt die beiden Typen `String` und `String[]` und unterstützt indirekt alle möglichen Typen von Datenstrukturen - es liegt keine Beschränkung für den Inhalt von Parameterwerten vor. Um unterschiedliche Programme gegenseitig aufzurufen, müssen die Parameterwerte gegenfalls durch Hilfsprogramme konvertiert werden. Komplexe Datenstrukturen können durch das Verwenden eines Codierungsverfahrens (CSV [21], XML,...) in eine Zeichenfolge abgebildet werden.

3.2.2 SOAP

SOAP Nachrichten sind in XML kodiert - können also jede beliebige Datenstruktur enthalten. Die Definition der Datenstrukturen befindet sich im WSDL Dokument (siehe Kapitel 2.1.1 Seite 6).
Die XML Kodierung einer Datenstruktur ist Sache der SOAP-Implementierung, für Basistypen wie String, Integer,... sind diese Kodierungsmechanismen meistens bereits in der SOAP-Implementierung automatisiert und bedürfen keiner weiteren Programmierung. Komplexe Datentypen werden durch eine SOAP-Implementierung mittels

verschiedener Mechanismen in eine XML-Darstellung umgewandelt; im Falle von Java werden komplexe Datenstrukturen durch Java-Klassen implementiert, die durch "Serializer"/"Deserializer" in XML geschrieben und gelesen werden. Diese Java-Klassen lassen sich durch einen WSDL-Compiler (z.B. Axis WSDL-Compiler [8]) statisch erzeugen, andernfalls kann man eigene Java-Klassen verwenden, welche gewissen Anforderungen gerecht werden (im Falle von AXIS müssen sie dem Standard JavaBean Pattern [22] folgen) rsp. man schreibt eigene "Serializer"/"Deserializer"-Klassen. Diese Methode eignet sich gut für statisch eingebundene Web Service Schnittstellen.

3.3 Abbildungsverfahren

Axis bietet die Möglichkeit, die zu versendende SOAP-Nachricht direkt zu modifizieren. Dies kann mitunter benutzt werden, um eine bereits vorhandene XML-Zeichenfolge als Grundlage eines Web Service Aufrufs zu verwenden.
Bei "normaler" Verwendung des Axis Toolkits werden die Eingabeparameter mittels Java-Typen instanziert, für jeden Typen wird ein `typemapping` registriert und die instanzierten Eingabeparameter werden einem `Call`-Objekt zur Ausführung des SOAP-Aufrufs übergeben. Sinnvollerweise sollten die Eingabe-Typinstanzen bereits aus einer Berechnung vorliegen, im Falle des SOAP Subsystems ist dies aber nicht der Fall. Die Java-Klassen für die Eingabeparameter müssten in einem ersten Schritt mittels des WSDL-Compilers generiert werden, in einem zweiten Schritt müssten die Instanzen mit den BioOpera-Parameterwerten initialisiert werden.
Die SOAP Subsystem Implementierung verwendet den Mechanismus der direkten SOAP-Nachrichtmodifizierung - mit Unterstützung von automatisch generierten Eingabe- / Ausgabeprogrammen. Diese Eingabeprogramme generieren aus Eingabewerten eine XML-Zeichenfolge, durch das hintereinanderschalten der Eingabeprogramme können komplexe XML Strukturen erzeugt werden. Die Ausgabeprogramme bilden analog eine XML Struktur auf einzelne Parameterwerte ab. Das eigentliche Web Service Programm akzeptiert als Eingabeparameter eine XML Nachricht, der Ausgabeparameter ist wiederum eine XML Nachricht.
Durch diesen Mechanismus kann der (oben beschriebene) umständliche Vorgang der Klassen-Generierung für die Ein- / und Ausgabeparameter umgangen werden. Der Zeitbedarf für die Ausführung der Eingabe- und Ausgabeprogramme im Vergleich zum Zeitbedarf für die Generierung der Java Klassen wäre sehr wahrscheinlich kleiner, wenn die Java Klassengenerierung bei jedem Web Service Aufruf neu ausgeführt werden muss, andernfalls wäre die Lösung "WSDL-Compiler" wahrscheinlich effizienter.

3.4 XML Format

In den folgenden Kapiteln ist die XML Struktur für die SOAP Subsystem-Kommandos definiert. Die Kommandos sind mit einer Kommandozeile für das UNIX Subsystem vergleichbar.

3.4.1 BioOpera Programmdefinition

Eine BioOpera-Programmdefinition ist durch die folgende XML Syntax auszudrücken:

```
<PROGRAM OID="" NAME="">
```

```
    <INBOX>
       <PARAM OID="" TYPE="" NAME=""/>
    </INBOX>
    <OUTBOX>
       <PARAM OID="" TYPE="" NAME=""/>
    </OUTBOX>
    <ACCESS>
       <METHOD OID="" SYS="" RES="" CMD=""/>
    </ACCESS>
 </PROGRAM>
```

Die für das SOAP Subsystem zu verwendenden Attribute sind: Programmname, Parametername, "command" (**CMD**) und "resource" (**RES**) der Zugriffmethode. Für die Implementierung des SOAP Subsystems konnte der Mechanismus der Kommando-Zeile benutzt werden, um alle notwendigen Informationen für den Web Service Aufruf rsp. die Eingabe/Ausgabeoperationen an die SOAP Subsystem-Implementierung zu übergeben. Die Information werden einfachheitshalber in XML codiert. Die drei verschiedenen Formate sind im folgenden Kapitel definiert. Zwischen der Kommandozeile und der Programmdefinition bestehen folgende Relationen:

Für jeden in der `INBOX` enthaltenen Parameter ist in der Kommandozeile ein Platzhalter der Form %<<Name des Parameters>>% vorhanden, der während der Laufzeit durch den entsprechenden Parameterwert substituiert wird.

Für jeden in der `OUTBOX` enthaltenen Parameter wird eine entsprechende XML Zeichenfolge vom SOAP Subsystem zurückgeliefert, d.h. wenn eine Programmdefinition einen Ausgabeparameter namens "test" enthält, wird die XML Zeichenfolge eine XML Teilzeichenfolge der Form `<test> .... </test>` enthalten. BioOpera erkennt auf diese Weise die Ausgabeparameter und weist diese automatisch den entsprechenden Parametern zu. Mehrfach auftretende XML Elemente müssen in einen Array umgewandelt werden (Informationen dazu im Anhang C.6.6).

3.4.2 Codierung des Kommando-Attributes

Mit der Abbildung eines Web Service auf eine OCR Darstellung werden mehrere XML-Kommandos erzeugt (Kapitel 4.1.2), die je dem `CMD`-Attribut im `METHOD`-Element einer Programmdefinition zugewiesen werden. Der Inhalt des CMD-Attributes ist gemäss XML-Spezifikation [23] zu codieren - Zeichen wie `<, >, oder "` dürfen nicht in einem Attributwert vorkommen. Ein XML Schema, welches die 3 XML Strukturen für die Kommandozeile definiert, ist im Anhang E enthalten.

Definition SOAP Aufruf

Abbildung 3.2 zeigt die XML Struktur des XML-Kommandos für einen SOAP Aufruf. Das Kommando enthält die Informationsfelder WSDL URL, Service-Name, Port-Name, Operationsname und die als SOAP Request zu verwendende XML-Zeichenfolge.

Beispiel:

```
<ACCESS>
   <METHOD SYS="SOAP" RES="soap"
    CMD="&lt;OPERAWEBSERVICEREQUEST&gt;
    &lt;WSDLLOCATION&gt;http://test.homeftp.net:
    8080/axis/BioOperaPerfTest.jws?WSDL
```

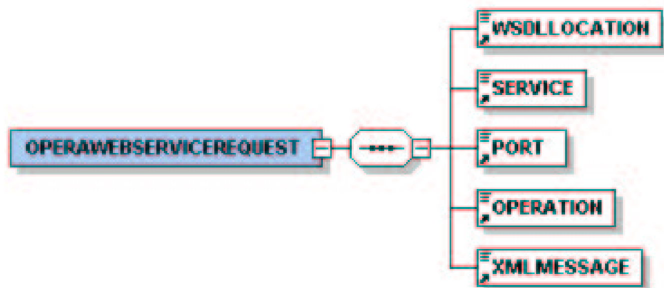


Abbildung 3.2: XML-Kommando für Web Service Aufruf

```
    &lt;/WSDLLOCATION&gt;&lt;SERVICE&gt;
    BioOperaPerfTestService&lt;/SERVICE&gt;
    &lt;PORT&gt;BioOperaPerfTest&lt;/PORT&gt;
    &lt;OPERATION&gt;echo&lt;/OPERATION&gt;
    &lt;XMLMESSAGE&gt;%xml%&lt;/XMLMESSAGE&gt;
    &lt;/OPERAWEBSERVICEREQUEST&gt;"/>
</ACCESS>
```

Das Kommando ist mit einem Eingabe- und einem Ausgabeparameter in der BioOpera-Programmdefinition assoziiert. Der Eingabeparametername ist (immer) `xml` - im Kommando wird zur Laufzeit der Ausdruck `%xml%` durch eine XML Zeichenfolge ersetzt. Der Ausgabeparametername ist ebenfalls immer `xml`.
Decodierter Inhalt des CMD-Attributes:

```
<OPERAWEBSERVICEREQUEST>
   <WSDLLOCATION>http://test.homeftp.net:8080/
      axis/BioOperaPerfTest.jws?WSDL</WSDLLOCATION>
   <SERVICE>BioOperaPerfTestService</SERVICE>
   <PORT>BioOperaPerfTest</PORT>
   <OPERATION>echo</OPERATION>
   <XMLMESSAGE>%xml%</XMLMESSAGE>
</OPERAWEBSERVICEREQUEST>
```

Definition Eingabeprogramm

Abbildung 3.3 zeigt die XML Struktur des XML-Kommandos für ein Eingabeprogramm. Das Kommando kann mehrere Informationsfelder zu den Eingabeparametern im Element <PARAMETERS> enthalten. Ein Parameter wird durch die Felder Name (Typname), Namespace (XML Namespace dieses Elements), Targetnamespace (Namespaceangabe für Dokument-orientierten Nachrichtenmodus), Type (BioOpera Typ) und Value (Parameterwert) definiert.
Dabei entspricht jede Parameterdefinition im <PARAMETERS>-Element einer Eingabeparameterdefinition in der OCR-Programmdarstellung (vgl. Beispiel):

```
<PROGRAM OID="" NAME="">
   ...
   <INBOX>
      <PARAM OID="1" TYPE="String" NAME="echo"/>
   </INBOX>
   ...
</PROGRAM>
```

Zudem enthält das XML Kommando ein Informationsfeld zum Wurzelelement der XML-Nachricht, welches dem Ausgabeparameter entspricht.

```
<PROGRAM OID="" NAME="">
   ...
   <OUTBOX>
      <PARAM OID="3" TYPE="String" NAME="xml"/>
   </OUTBOX>
   ...
</PROGRAM>
```

Im Gegensatz zum SOAP Aufruf Programm ist der Ausgabeparameter nicht fest als `xml` definiert, sondern hängt vom jeweiligen Eingabeprogramm ab (siehe folgendes Kapitel). Die Informationsfelder WSDL URL, Service-Name, Port-Name, Operationsname haben reinen informativen Charakter, sie werden für den **Debug**-Modus verwendet.

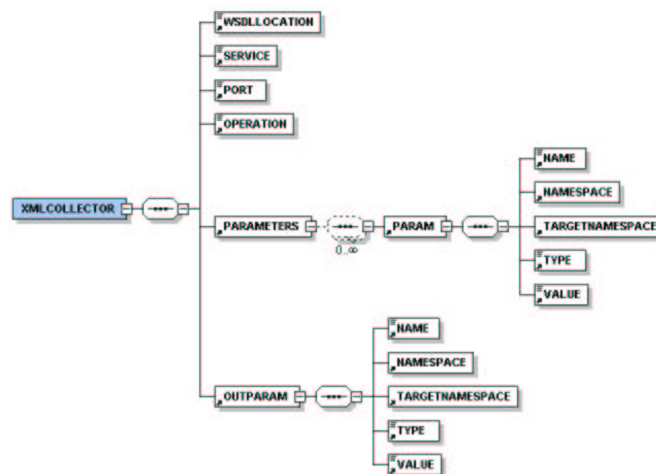


Abbildung 3.3: XML-Kommando für Generierung der SOAP Nachricht

Beispiel:

```
<ACCESS>
   <METHOD SYS="SOAP" RES="soap"
   CMD="&lt;XMLCOLLECTOR&gt;&lt;WSDLLOCATION&gt;
```

```
   http://test.homeftp.net:8080/axis/
   BioOperaPerfTest.jws?WSDL&lt;/WSDLLOCATION&gt;
   &lt;SERVICE&gt;BioOperaPerfTestService
   &lt;/SERVICE&gt;&lt;PORT&gt;BioOperaPerfTest
   &lt;/PORT&gt;&lt;OPERATION&gt;echo_InputCollector
   &lt;/OPERATION&gt;&lt;PARAMETERS&gt;&lt;PARAM&gt;
   &lt;NAME&gt;echo&lt;/NAME&gt;&lt;NAMESPACE&gt;
   &lt;/NAMESPACE&gt;&lt;TARGETNAMESPACE&gt;
   &lt;/TARGETNAMESPACE&gt;&lt;TYPE&gt;STRING []
   &lt;/TYPE&gt;&lt;VALUE&gt;%echo%
   &lt;/VALUE&gt;&lt;/PARAM&gt;&lt;/PARAMETERS&gt;
   &lt;OUTPARAM&gt;&lt;NAME&gt;xml&lt;/NAME&gt;
   &lt;NAMESPACE&gt;&lt;/NAMESPACE&gt;
   &lt;TARGETNAMESPACE&gt;&lt;/TARGETNAMESPACE&gt;
   &lt;TYPE&gt;STRING []&lt;/TYPE&gt;
   &lt;VALUE&gt;&lt;/VALUE&gt;
   &lt;/OUTPARAM&gt;&lt;/XMLCOLLECTOR&gt;"/>
</ACCESS>
```

Der Eingabeparametername ist "echo" - im Kommando wird zur Laufzeit der Ausdruck "%echo%" durch einen beliebigen Eingabewert ersetzt.

Decodierter Inhalt des CMD-Attributes:

```
<XMLCOLLECTOR>
   <WSDLLOCATION>http://test.homeftp.net:8080/
      axis/BioOperaPerfTest.jws?WSDL</WSDLLOCATION>
   <SERVICE>BioOperaPerfTestService</SERVICE>
   <PORT>BioOperaPerfTest</PORT>
   <OPERATION>echo_InputCollector</OPERATION>
   <PARAMETERS>
      <PARAM>
         <NAME>echo</NAME>
         <NAMESPACE/>
         <TARGETNAMESPACE/>
         <TYPE>STRING []</TYPE>
         <VALUE>%echo%</VALUE>
      </PARAM>
   </PARAMETERS>
   <OUTPARAM>
      <NAME>xml</NAME>
      <NAMESPACE/>
      <TARGETNAMESPACE/>
      <TYPE>STRING []</TYPE>
      <VALUE/>
   </OUTPARAM>
</XMLCOLLECTOR>

Zu generierende XML-Struktur:
   <xml><echo>...</echo></xml>
```

Definition Ausgabeprogramm

Abbildung 3.4 zeigt die XML Struktur des XML-Kommandos für ein Ausgabeprogramm. Es enthält lediglich ein Informationsfeld, welches die XML Zeichenfolge enthält.

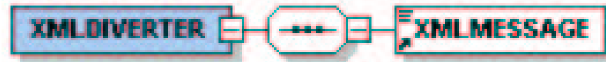


Abbildung 3.4: XML-Kommando für Auslesen der SOAP Nachricht

Beispiel:

```
<ACCESS>
<METHOD SYS="SOAP" RES="soap" CMD="&lt;XMLDIVERTER&gt;
&lt;XMLMESSAGE&gt;%echoRequest%&lt;/XMLMESSAGE&gt;&lt;/XMLDIVERTER&gt;"/>
</ACCESS>
```

Decodierter Inhalt des CMD-Attributes:

```
<XMLDIVERTER>
   <XMLMESSAGE>%echoRequest%</XMLMESSAGE$>
</XMLDIVERTER>
```

Analog zum Eingabeprogramm ist der Ausdruck `%echoRequest%` als Eingabeparameter in der BioOpera-Programmdefinition vorhanden - zur Laufzeit wird als Eingabe des Programms eine XML Zeichenfolge der Form `<echoRequest> ... </echoRequest>` erwartet. Die Ausgabeparameter der BioOpera-Programmdefinition entsprechen den XML Kinder-Elementen von `<echoRequest>`.

```
   <PROGRAM OID="" NAME="">
      <INBOX>
         <PARAM OID="1" TYPE="String" NAME="echoRequest"/>
      </INBOX>
      <OUTBOX>
         <PARAM OID="3" TYPE="" NAME="a"/>
         <PARAM OID="3" TYPE="" NAME="b"/>
      </OUTBOX>
      ...
   </PROGRAM>

Erwartete XML-Zeichenfolge:
   <echoRequest>
      <a>...</a>
      <b>...</b>
   </echoRequest>
```

Kapitel 4

UDDI Browser

Das zweite Ziel dieser Diplomarbeit ist die Entwicklung eines Tools, welches automatisch eine Web Service Schnittstelle (basierend auf WSDL) importiert und als Programmbibliothek im BioOpera Visual Designer zur Verfügung stellt.

Auf Basis eines WSDL Prozessors, der alle relevanten Information für eine einzelne Operation eines Web Services sammelt und in einer dem WSDL-OCR Mapping (siehe Kapitel 3) entsprechenden Datenstruktur ablegt, wurde eine Applikation entwickelt, die als Baustein für BioOpera-Softwarekomponenten verwendet werden kann wie auch als alleinstehende Applikation mit einer graphischen Benutzeroberfläche funktioniert.

Die graphischen Anwenderschnittstelle greift mittels SOAP auf eine UDDI Registry (Version 2) zu und erlaubt das Suchen nach Web Services, die durch ein WSDL Dokument beschrieben sind. Die Applikation filtert dabei automatisch mittels einer Konzeptsuche (Konzept `wsdlSpec`) alle Web Services heraus, die nicht durch ein WSDL Dokument beschrieben sind.

Die Dokumentation dieser Anwendungen findet sich in [1] (und im Anhang C.4), es wird darauf verzichtet, die Dokumentation an dieser Stelle zu wiederholen. In den folgenden Kapiteln wird auf den WSDL Prozessor, die OCR Programmgenerierung und die graphische Benutzeroberfläche eingegangen.

4.1 Architektur

Die Architektur der Applikation kann in drei Teile gegliedert werden:

Package für grafische Oberfläche: `ch.ethz.inf.bioopera.ws.uddibrowser.gui`

Package für Programmlogik und Suchfunktion: `ch.ethz.inf.bioopera.ws.uddibrowser`

Package für WSDL Verarbeitung: `ch.ethz.inf.bioopera.ws.tools`

Abbildung 4.1 zeigt die wichtigsten Methoden, die in der Applikation vorkommen. Zentrale Klasse ist `UDDIBrowser`, die durch die `main` -Methode aufgerufen werden kann; die Klasse beinhaltet neben einer Methode, die die Suche auslöst, Funktionen, die das Verarbeiten eines WSDL Dokumentes starten, um eine einzelne Operation rsp. alle Operationen eines Web Services zu importieren. Methode `exportOCRML()` ist die zentrale Funktion, die die OCR Programmrepräsentation aus den Metainformationen

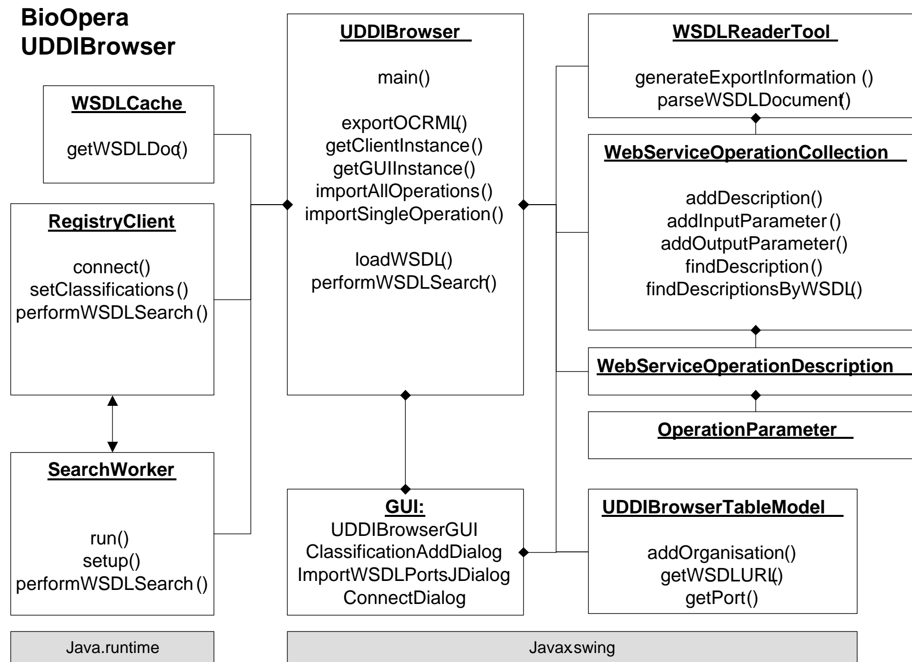


Abbildung 4.1: Klassendiagramm ImportTool

des WSDL Prozessors generiert. Abhängig von den Programm-Startargumenten (siehe Anhang C.4.5) wird entweder die grafische Benutzeroberfläche aufgerufen oder das WSDL-Konvertierungsprogramm wird gestartet.

Für WSDL Dokumente wird ein SessionCache geführt, der durch die beteiligten Klassen benutzt wird. Zu diesen Klassen gehören `RegistryClient` und `Search Worker`, welche den Zugriff auf eine Registry regeln und eine Suche konkret ausführen und die Suchresultate für die Anzeige in den GUI-Klassen aufbereiten. Suchresultate werden in der Klasse `UDDIBrowserTableModel` abgelegt.

4.1.1 WSDL Importfunktion

Die Klassen `WSDLReaderTool`, `WebServiceOperationCollection`, `WebServiceOperationDescription` und `OperationParameter` bilden zusammen den WSDL Prozessor. Die Klasse `WSDLReaderTool` ist für das Parsing (mittels JDOM) des WSDL Dokumentes zuständig und generiert während des WSDL-Baumtraversierens Beschreibungen von Web Service Operationen (Klasse `Web ServiceOperationDescription`). Instanzen der Klasse `WebService OperationDescription` werden indirekt über die Klasse `WebService OperationCollection` generiert und verwaltet, die entsprechende Methoden bereitstellt. Ein- und Ausgabeparameter werden durch die Klasse `OperationParameter` modelliert, die bereits den Ein/Ausgabeparametern der OCR Programmdarstellung entsprechen.

Die Klasse `WSDLReaderTool` generiert pro gültiger Web Service Operationsdefinition im WSDL Dokument im Minimum 5 Instanzen der Klasse

WebServiceOperationDescription (1 x SOAP Aufruf, 1 x Eingabeprogramm für den xml-Parameter des SOAP Aufruf-Programms, 1 x Ausgabeprogramm für den xml-Parameter des SOAP Aufruf-Programms, mindestens 1 x Eingabeprogramm für einen Eingabewert und mindestens 1 x Ausgabeprogramm für einen Ausgabewert). Eine abstrakte Algorithmusbeschreibung folgt im nächsten Kapitel. Die Instanzen der Klasse WebServiceOperationDescription lassen sich in die drei Klassen Eingabeprogramm, Ausgabeprogramm und SOAP Aufruf (vgl. Kapitel 3.4.2) einteilen.

Das "SOAP Aufruf"-Programm repräsentiert die eigentliche Web Service Operation. Eingabe- und Ausgabeparameter ist immer **xml**, d.h. im Visual Designer erscheinen die Parameter unter dem Namen "xml", zur Laufzeit wird als Parameterwert eine XML Struktur der Form:
<xml> <...> </xml> erwartet.

Um die "xml"-Parameter des "SOAP Aufruf"-Programms im Visual Designer mit anderen Programmen verbinden zu können, wird ein Eingabeprogramm generiert, welches als Ausgabe-Parameter den xml-Parameter hat. Als Eingabe-Parameter wird dieses Programm einen Parameter haben, der dem SOAP Body-Element des Requests entspricht.
Gegengleich wird ein Ausgabeprogramm mit dem Eingabeparameter xml generiert; der Ausgabe-Parameter wird dem SOAP Body-Element der SOAP Response entsprechen.
Das zweite (und folgende) Eingabeprogramm baut die XML Struktur mit den Eingabewerten von BioOpera auf. Im Gegenzug wird der XML-Rückgabewert durch die Ausgabeprogramme auf einzelne BioOpera-Parameter abgebildet.

4.1.2 WSDL Parsing und Programmgenerierung

Abbildung 4.2 illustriert den ersten Schritt des Programmgenerierungsvorgangs. Für jede gültige Port-Deklaration wird die entsprechende Binding-Definition gesucht - für jede Operation des Bindings wird eine Programmgenerierung der Form "SOAP Aufruf", Eingabeprogramme, Ausgabeprogramme vorgenommen. In jedem Schritt des Parsings werden nebst dem Überprüfen der logischen Bedingungen (z.B. dass für eine Operationsdefinition im <binding> auch eine entsprechende Deklaration im <portType> vorhanden ist) die benötigten Informationsfelder für die Programmgenerierung ausgelesen.

Gemäss [24] wird im "rpc style"-Modus der Operationsname für das Wurzelelement des SOAP Nachrichteninhaltes verwendet (Abbildung 4.3 Schritt 1), im "doc style"-Modus treten die einzelnen Typdefinition direkt unter dem <soap:body> -Element auf.

Abbildung 4.3 illustriert die Eingabeprogramm-Generierung. Eine Typdefinition im <message> -Element kann in direkter Form vorliegen (dem aktuellen Eingabeprogramm wird ein entsprechender Parameter hinzugefügt) oder kann einen komplexen Typ enthalten; in diesem Fall werden weitere Eingabeprogramme generiert.

Abbildung 4.4 illustriert die Ausgabeprogramm-Generierung, die analog der Eingabeprogramm-Generierung abläuft.

Die Namen der Ein-/Ausgabeparameter werden nach folgendem Schema erzeugt:

- Für das SOAP Aufruf Programm ist der Ein- und Ausgabeparameter "xml".
- Für das in Abbildung 4.3 Schritt 1 generierte Eingabeprogramm ist der Ausgabeparametername "xml", der Eingabeparametername entspricht dem Operationsnamen (nur "rpc"-Modus). Das in Schritt 2 generierte Programm hat folglich als Ausgabeparametername den Operationsnamen, die Eingabeparameter entsprechen den `<part>`-Elementen des `<message>`-Elementes.
  Im "doc"-Modus wird ein Program mit Ausgabeparameter "xml" und den Eingabeparametern, die direkt den Typnamen der `<part>`-Elemente entsprechen, generiert.
- Ist das `<part>`-Element eine direkte Typdefinition, so ergibt sich der Eingabeparametername aus dem Namensattribut des `<part>`-Elementes.
- Ist das `<part>`-Element eine komplexe Typdefinition, so ergibt sich der Eingabeparametername aus dem Namensattribut des `<part>`-Elementes und es wird ein weiteres Programm generiert, dass als Ausgabeparametername den Namen aus dem Namensattribut des `<part>`-Elementes hat, die Eingabeparameter entsprechend den `<element>`-Namen der komplexen Typdefinition. Bei allen weiter generierten Programmen entsprechen die Ein- und Ausgabenparameternamen den `<element>`-Namen.
- Für das in Abbildung 4.4 Schritt 1 generierte Eingabeprogramm ist der Eingabeparametername "xml", der Ausgabeparametername entspricht dem Namen des `<message>`-Elementes (nur "rpc"-Modus). Das in Schritt 2 generierte Programm hat folglich als Eingabeparametername den `<message>`-Namen, die Ausgabeparameter entsprechen den `<part>`-Elementen des `<message>`-Elementes. Die Namensgebung für die `<part>`-Elemente erfolgt analog wie bei den Eingabeprogrammen.

4.1.3 Einschränkungen

WSDL erlaubt das Importieren von zusätzlichen Typdefinitionen (als XML Schema, im WSDL Dokument durch eigenen Namensraum erkennbar), da dies aber bei keinem der getesteten Web Services Verwendung fand und bei vielen Entwicklern Unklarheit besteht [1], wie die Importfunktion zu verwenden ist, wurde auf eine entsprechende Ein/Ausgabe-Programmgenerierung verzichtet. Sollte während dem Parsen der Typdefinition eine Typdefinition nicht weiter aufgelöst werden können rsp. bei nicht korrekt definierten Typangaben (z.B. nicht vorhandenes Namens-Attribut), wird ein Ein/Ausgabeparameter mit dem Namen "anyType" generiert.

[1] z.B. Yahoo WSDL Message Board http://groups.yahoo.com/group/wsdl/message/112

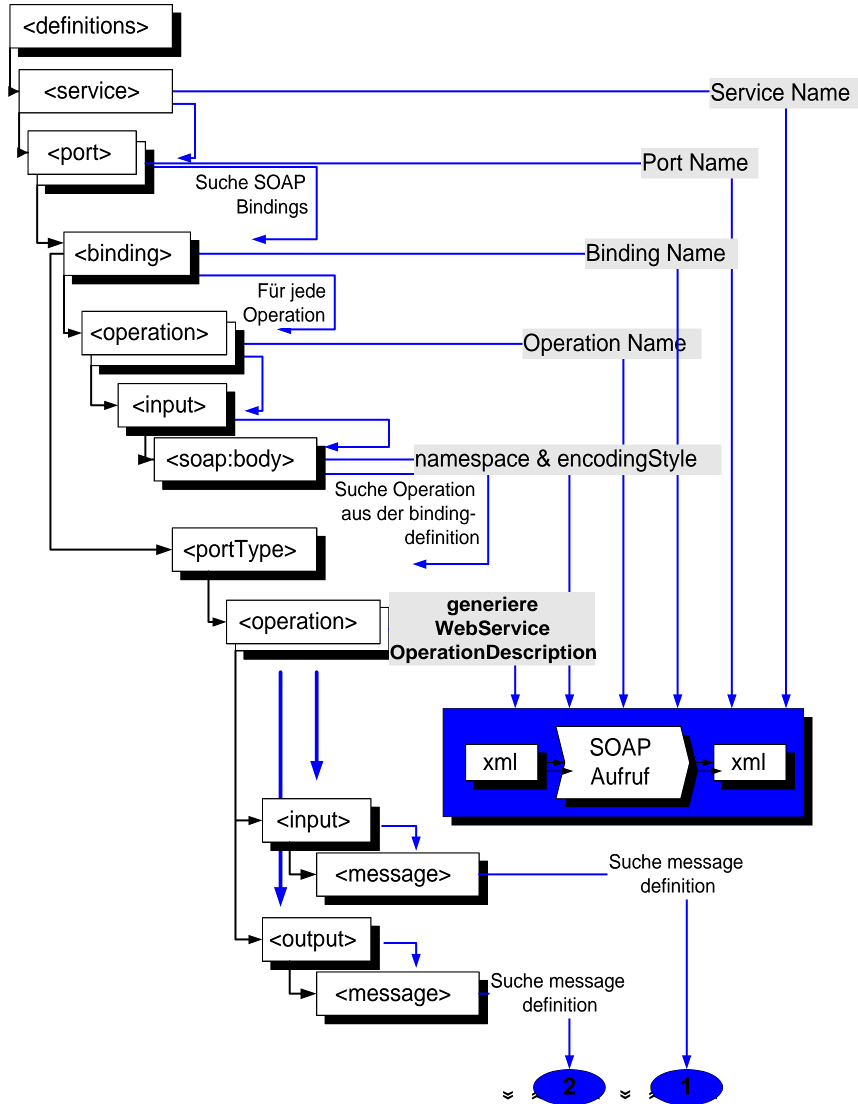


Abbildung 4.2: WSDL Parsing Algorithmus Schritt 1: SOAP Aufruf

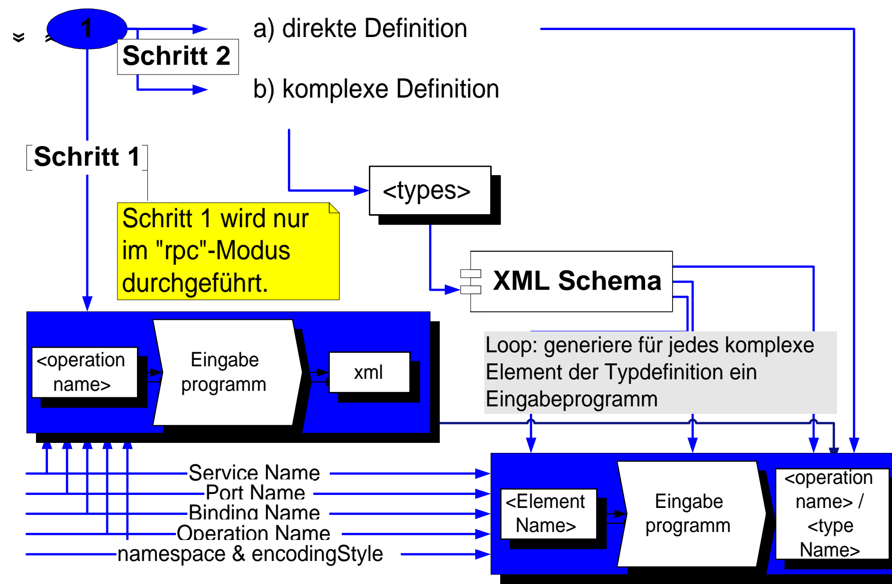


Abbildung 4.3: WSDL Parsing Algorithmus Schritt 2: Eingabeprogramm

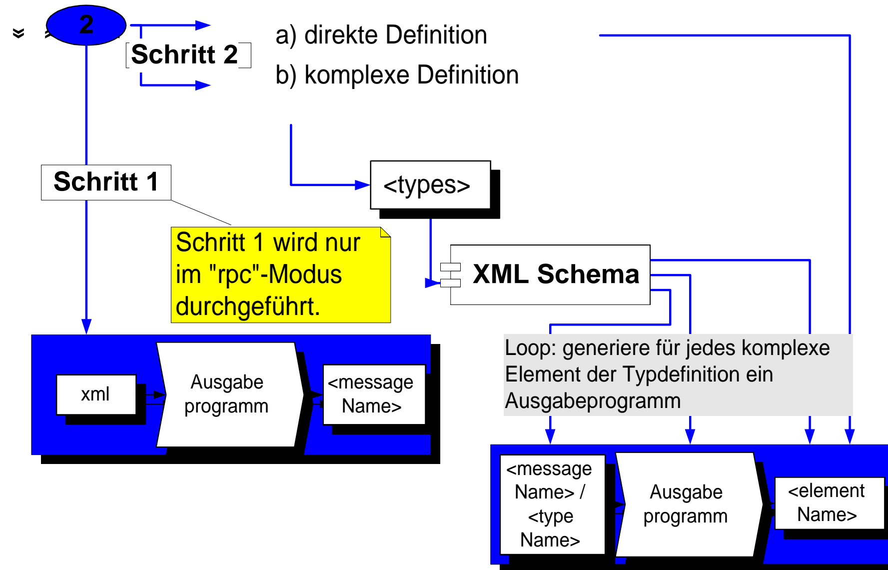


Abbildung 4.4: WSDL Parsing Algorithmus Schritt 3: Ausgabeprogramm

4.2 Entwicklungstool NetBeans

Die grafische Benutzeroberfläche wurde unter Java Swing mit Unterstützung des Entwicklerstools NetBeans [25] [2] entwickelt. NetBeans bietet einen Form-Editor, mit dem man per Drag&Drop grafische Komponenten zusammensetzen kann. Jede Komponente kann zudem durch Eigenschaften angepasst werden, die automatisch in JavaCode umgesetzt werden. Ein grosser Teil des notwendigen JavaCodes wurde dadurch automatisch generiert und sollte auch nicht von Hand editiert werden. Zu den GUI-Klassen existiert je eine ".form"-Datei, die von NetBeans zur Darstellung im Form Editor verwendet wird (Abbildung 4.5).

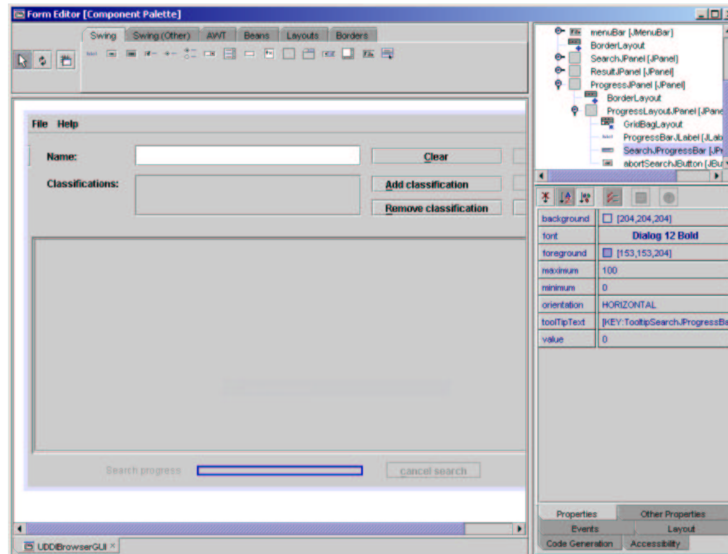


Abbildung 4.5: NetBeans Form-Editor

[2]frei erhältlich unter www.netbeans.org.

Kapitel 5

SOAP Subsystem

Die dritte Teilaufgabe dieser Arbeit bestand in der Entwicklung und Integration des Web Service Ausführungssystems. Die Kernkomponente, der SOAPProxy, wurde auf Basis des bestehenden Java Programm Execution Client weiterentwickelt. Die Integrationsarbeit bestand in der Aktualisierung der `PEC Protocol` Dokumentation, die im Anhang B zu finden ist, und der Implementation dieses Protokolls auf der Clientseite. Das `PEC Protocol` definiert die Kommunikationcodes zwischen dem Runtime-Server, dem BioOpera Resource Monitoring Tool, dem Linux Programm Execution Client und dem SOAPProxy.
Die SOAPProxy-Dokumentation findet sich in [1] (und im Anhang C.6).

5.1 Aufbau

Das SOAP Subsystem besteht aus 2 Teilen:

UnixSystem Serverseitige Komponente für die Kommunikation (`PEC Protocol`) mit den `PECs` und `SOAPProxies`. Die Unterscheidung zwischen `PEC`- und `SOAPProxy`-Kommunikation erfolgt durch die Verwendung von zwei verschiedenen Portnummern.

SOAPProxy Proxy für die Ausführung von Web Services mittels SOAP. Zusätzlich werden Ein-/Ausgabeprogramme für Web Services auf dem Proxy ausgeführt.

5.2 Auswahl Web Service Toolkit

Als Toolkit für die Ausführung von Web Services kamen verschiedene Implementationen in Frage: JAX-RPC [26], Apache SOAP [1], Apache Axis [2] oder IBM WSTK [3]. Aus zwei Gründen wurde Apache Axis als Toolkit ausgewählt:

- Kompatibilität zum BioOpera Application Server [6], wo Axis als Servicekomponente eingesetzt wird. Dies soll Probleme ausschliessen, die sich durch die Verwendung von 2 verschiedenen Toolkits (Interoperabilitätsprobleme) ergeben.

[1] http://ws.apache.org/soap/
[2] http://ws.apache.org/axis/
[3] http://www.alphaworks.ibm.com/tech/webservicestoolkit

- Direkter Zugriff (lesend und schreibend) auf die SOAP-Nachricht wird durch das API unterstützt.

5.3 Architektur

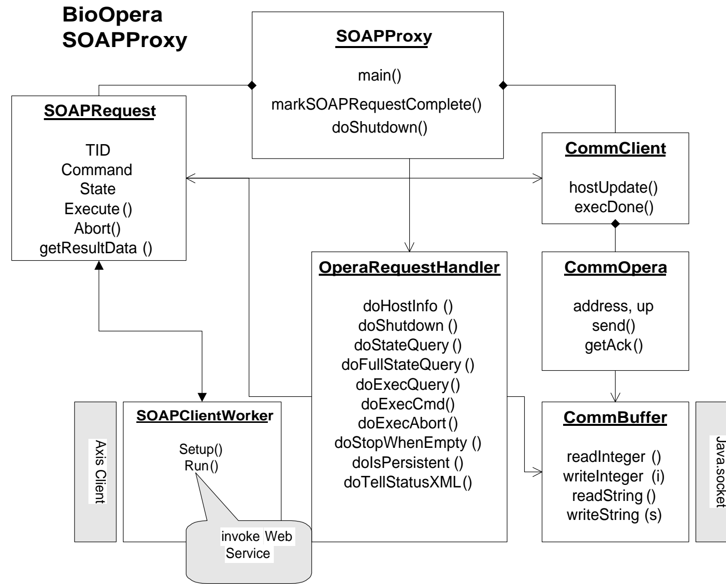


Abbildung 5.1: SOAPProxy Klassendiagramm

Abbildung 5.1 zeigt die Klassenarchitektur des SOAPProxy, die sich in drei Kategorien unterteilen lässt [4]:

Kommunikation Die Kommunikationsklassen bieten Funktionen für das Senden und Empfangen von Daten auf der Basis einer `Socket`-Verbindung. Die Kommunikationsklassen sprechen automatisch mehrere BioOpera-Server an (Aktualisierung von Statusinformationen).

CommClient Service-Klasse als Schnittstelle zwischen BioOpera und dem SOAP-Proxy. Die Klasse verwaltet eine Liste von bekannten Servern. Mit der `execDone()`-Methode wird ein abgeschlossener Web Service Aufruf mit dem Resultat an BioOpera übergeben.

CommOpera Diese Klasse verwaltet Verbindungen zu einem einzelnen BioOpera-Server und bietet die Funktionen verbinden, Befehl senden, Empfangsbestätigung empfangen und Verbindung schliessen.

CommBuffer Implementiert die Kommunikationsfunktionen Senden und Empfangen. Unterstützt werden Integer-Codes (4 Byte) und Strings beliebiger Länge.

[4] Die Systemdokumentation ist im Anhang C.6 zu finden.

Logik Alle nicht Web Service relevanten Funktionen und Informationen finden sich in den folgenden zwei Klassen.

SOAPProxy Die Hauptklasse des Projekts. Die Hauptaufgabe dieser Klasse besteht im Akzeptieren von Serverkommandos und dem Starten eines Threads, der dieses Serverkommando verarbeitet. Sämtliche Statusinformationen und die aktiven Web Service Programminstanzen sind als Datentrukturen in dieser Klasse abgelegt. Zusätzlich ist diese Klasse für die Ausgabe von Programminformationen und das Verarbeiten von Befehlseingaben auf der Kommandozeile zuständig.

OperaRequestHandler Jedes Serverkommando wird in einer Instanz dieser Klasse verarbeitet. Die erlaubten Kommandos sind durch das *PEC*-Protokoll definiert.

Web Service Ausführung Eine Web Service Programminstanz ist durch eine Klasse als Datenstruktur für alle relevanten Informationen und durch einen Thread für die Ausführung des `SOAP Calls` mittels des Axis Toolkits umgesetzt.

SOAPRequest Enthält alle relevanten Daten zu einem auszuführenden Kommando (Resultat, Fehlernachrichten, Status der Ausführung, Start- und Endzeit, etc.), die über entsprechende Funktionen abgefragt werden können.

SOAPClientWorker Führt das vom BioOpera-Server gesendete Kommando aus: Im Fall eines SOAP Aufrufs wird der Axis Client gemäss den Angaben im Kommando konfiguriert und der Web Service wird aufgerufen. Im Falle eines Ein- oder Ausgabeprogramms wird die entsprechende Berechnung (Generieren einer XML Zeichenfolge aus den Angaben im Kommando, Abbilden einer XML Zeichenfolge auf Ausgabeparameter) bereits in der `SOAPRequest`-Klasse durchgeführt.

5.4 Programmausführung

Der SOAPProxy kann drei verschiedene Programme verarbeiten - die Definition der Programme befindet sich in Kapitel 3. Das System kann auf einfache Weise durch neue Programme erweitert werden, indem eine neue XML Struktur für das neue Programm definiert wird und eine entsprechende Routine zur Verarbeitung in der Klasse `SOAPRequest` implementiert wird. Die Unterscheidung der verschiedenen Programmtypen basiert auf dem Wurzelelement der XML Struktur (`<OPERAWEBSERVICEREQUEST>`, `<XMLCOLLECTOR>` und `<XMLDIVERTER>`).

5.4.1 SOAP Call

Falls das Kommando eine XML Zeichenfolge der Form

```
  <OPERAWEBSERVICEREQUEST>
<WSDLLOCATION></WSDLLOCATION>
<SERVICE></SERVICE>
<PORT></PORT>
```

```
<OPERATION></OPERATION>
<XMLMESSAGE></XMLMESSAGE>
</OPERAWEBSERVICEREQUEST>
```

enthält, wird eine Threadinstanz der Klasse `SOAPClientWorker` mittels der Methode `setup()` mit den Angaben im Kommando konfiguriert. Der Axis Client nimmt durch die Angaben im WSDL Dokument (Adressangabe in `<WSDLLOCATION>`) eine Autokonfiguration vor - anschliessend wird die SOAP XML Nachricht dem `SOAP Request` zugewiesen und der `SOAP Call` wird ausgeführt. Nach Erhalt der `SOAP Response` wird der Inhalt des `SOAP Body`-Elements

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   <soap:Body>
      ...
   </soap:Body>
</soap:Envelope>
```

ausgelesen und als Resultat der Programmausführung zugewiesen.

5.4.2 Eingabeprogramm

Das Eingabeprogramm generiert aus Eingabeparametern (beliebige Zeichenkette) und einem Ausgabeparameter eine XML Zeichenfolge. Das Kommando enthält dazu alle notwendigen Information in den Felder des `<PARAM>`-Elements.

```
   <XMLCOLLECTOR>
<WSDLLOCATION></WSDLLOCATION>
<SERVICE></SERVICE>
<PORT></PORT>
<OPERATION></OPERATION>
<PARAMETERS>
<PARAM>
<NAME></NAME>
<NAMESPACE/>
<TARGETNAMESPACE/>
<TYPE></TYPE>
<VALUE></VALUE>
</PARAM>

   ...

   <PARAM>
<NAME></NAME>
<NAMESPACE/>
<TARGETNAMESPACE/>
<TYPE></TYPE>
<VALUE></VALUE>
</PARAM>
```

```
</PARAMETERS>
<OUTPARAM>
<NAME></NAME>
<NAMESPACE/>
<TARGETNAMESPACE/>
<TYPE></TYPE>
<VALUE/>
</OUTPARAM>
</XMLCOLLECTOR>
```

Die XML Zeichenfolge setzt sich wie folgt zusammen:

1. Für jedes `<PARAM>`-Element wird ein neues XML-Element mit dem angegeben Namen erstellt. Falls das `<NAMESPACE/>`-Element nicht leer ist, wird dem neuen XML-Element ein Prefix vorangestellt und eine Namesraumdefinition wird dem Element hinzugefügt. Falls das `<TARGETNAMESPACE/>`-Element nicht leer ist, wird dem neuen Element ein Attribut der Form `xmlns=""` hinzugefügt. Schliesslich wird der Inhalt des `<VALUE>`-Elementes dem neuen Element als Text zugeordnet.

2. Mit den Angaben im `<OUTPARAM>`-Element wird analog Schitt 1 das Wurzelelement der neuen XML Zeichenfolge generiert und die aus Schritt 1 generierten XML-Elemente werden diesem Wurzelelement zugewiesen.

3. Die generierte XML Zeichenfolge wird als Resultat der Programmausführung zugewiesen.

Aus Gründen der Performance wurden diese Schritte mittels `String`-Operation implementiert, alternativ hätte JDOM für die Generierung der XML Zeichenfolge verwendet werden können (Overhead durch Aufbauen einer Baumstruktur und den daraus zusätzlich anfallenden Operationen).

5.4.3 Ausgabeprogramm

Das Ausgabeprogramm ist im Prinzip eine UNIX echo Operation.

```
  <XMLDIVERTER>
<XMLMESSAGE></XMLMESSAGE>
</XMLDIVERTER>
```

Das BioOpera-System liest Resultatwerte automatisch aus einer XML Zeichenfolge heraus: Kommt das Element `<test>` im Resultat vor und wurde für das Programm ein Ausgabeparameter namens "test" definiert, so weist BioOpera dem Parameter "test" als Wert den Inhalt des XML Elementes `<test>` zu. Um mehrfach auftretenden Elemente zu unterstützen, konvertiert das Ausgabeprogramm diese zu einem einzigen Element, der Inhalt des Elements ist ein `String []` [5] bestehend aus den Werten des mehrfach aufgetretenen Elementes.

[5] Die Kodierung eines `String []` ist im Anhang C.6.6 beschrieben.

5.5 Interoperabilität / Fehlerbehebung

Obwohl Interoperabilität zwischen verschiedenen Systemen zu den Merkmalen von Web Services gehört, sind in der Realität noch nicht alle SOAP Implementationen zueinander vollständig kompatibel (siehe dazu auch [6] und [6]). Manche Kompatibilitätsprobleme können eventuell durch das manuelle Anpassen der OCR-Programmbeschreibung für einen Web Service gelöst werden. Änderungen (löschen, hinzufügen) können an den Namensraum-Angaben (`<NAMESPACE>` und `<TARGETNAMESPACE>`) für Ein / Ausgabetypen gemacht werden.

Eine häufige Fehlerquelle sind auch nicht korrekt definierte WSDL Dokumente, bei Fehlern dieser Kategorie stimmen oft die Typangaben nicht mit der SOAP Request/Response XML Zeichenfolge überein. BioOpera erwartet in der XML Zeichenfolge ein Element, dass dem Namen der Typdefinition entspricht - durch einen Vergleich der XML Zeichenfolge und des Parameternamens kann das entsprechende Programm angepasst werden [7]. Zu beachten ist auch der Fall der Typangabe ”anyType” (siehe dazu auch Sektion 4.1.3): XML Schema erlaubt die Definition eines Elements unbestimmten Typs, in diesem Fall kann erst zur Laufzeit entschieden werden, welche XML Struktur durch den betreffenden Service zurückgeliefert wird.

[6] Apache Soap 2.3+ and Apache Axis Client Interop Results http://www.apache.org/ rubys/ApacheClientInterop.html

[7] siehe http://otn.oracle.com/ws/otnnews?WSDL Der `message`-Name `getRss0Response` tritt in der XML Zeichenfolge als `getRssResponse` auf.

Kapitel 6

Leistungsmessungen

6.1 Motivation

Die Performance-Messungen sollen die Leistungsfähigkeit und die Skalierbarkeit des SOAP Subsystems zeigen. Durch die Messungen sollen zudem optimale Konfigurationswerte rsp -profile gefunden werden.

Auf Performance Messungen für den UDDIBrowser wurde verzichtet, da die zeitkritischen Elemente

- Suchen in der Registry
- Laden der WSDL Dokumente

von der Leistungsfähigkeit der Registry und von der Verbindungsgeschwindigkeit und Auslastung des Webapplikationsservers stark abhängig sind.

6.2 Versuchsaufbau

Insgesamt wurden 8 Web Services getestet:

T1: http://www.tinetics.com/webservices/get_states_xml.cfc?wsdl Operation ”getstates”

T2: http://wavendon.dsdata.co.uk:8080/axis/services/SBGGetAirFareQuote?wsdl Operation ”getAirlines”

T3: http://www.cosme.nu/services/dns.php?wsdl Operation ”dns”

T4: http://otn.oracle.com/ws/otnnews?WSDL Operation ”getRss”

T5: http://www.tinetics.com/webservices/password.cfc?wsdl Operation ”hashme”

T6: http://www.xmlme.com/WSVideoGames.asmx?WSDL Operation ”FindVideoGames”

T7: http://www.webservicex.net/UDDIBusinessFinder.asmx?WSDL Operation ”FindBusiness”

T8: Axis Test Service [1] Operation "echo"

Die Auswahl der Web Services erfolgte nach den Kriterien `style` (T1 bis T4 und T8 sind Web Services des Typs "rpc-style", T5, T6 und T7 sind vom Typ "doc-style") und Grösse der SOAP Response-Message (T1, T2, T4, T7 zwischen 1000 Bytes und 100 KBytes; T3,T5, T6 und T8 $<$ 1000 Bytes).
T8 simuliert eine 10 Minuten andauernde Berechnung, im Gegensatz zu den Web Services T1 bis T7, die eine sehr kurze Antwortzeit ($<$ 2 Sekunden) haben.
Es wurden 6 Testreihen durchgeführt:

1. T1, T7 und T8 wurden auf einem SOAProxy mit der Kapazität 100 einzeln ausgeführt.
2. T1, T7 und T8 wurden auf einem SOAProxy mit der Kapazität 200 einzeln ausgeführt.
3. T1 bis T8 wurden auf einem SOAProxy mit der Kapazität 100 parallel ausgeführt.
4. T1 bis T8 wurden auf einem SOAProxy mit der Kapazität 200 parallel ausgeführt.
5. T8 wurde auf 4 SOAPProxies mit der Kapazität 200 ausgeführt.
6. T1 bis T8 wurden auf 4 SOAPProxies mit der Kapazität 200 parallel ausgeführt.

In den folgenden Ausführung wird die parallele Ausführung von T1 bis T8 mit Testreihe MIX bezeichnet.

Die Testreihen wurden je einmal mit Eingabe/Ausgabeoperation und einmal ohne Eingabe/Ausgabeoperation ausgeführt. Als Laufzeitumgebung wurde unter Linux Java 1.3.1 verwendet, unter Windows 2000 kam Java 1.4.1 zum Einsatz.

6.2.1 Serverkonfiguration

Für die Serverkonfiguration wurden die folgenden Einstellungen abgeändert, alle anderen Einstellungen wurden standardmässig belassen:

- runtime.isserver.numThreads: 4
- runtime.db.numThreads : 10
- runtime.sched.placepolicy : PROB
- runtime.sched.selectpolicy : SNO
- runtime.sched.transferpolicy : TTWO

[1] Der Sourcecode für den Axis Service der Testreihe 8 befindet sich im Anhang F.

6.2.2 SOAP Proxy Konfiguration

Für die SOAPProxy-Konfiguration wurden folgende Werte verwendet:

- RAM = 256
- SWAP = 512
- cpu = 2
- capacity = 100 und 200
- debug = no

Die SOAPProxies wurden im interaktiven Modus gestartet.

6.2.3 Messprofil

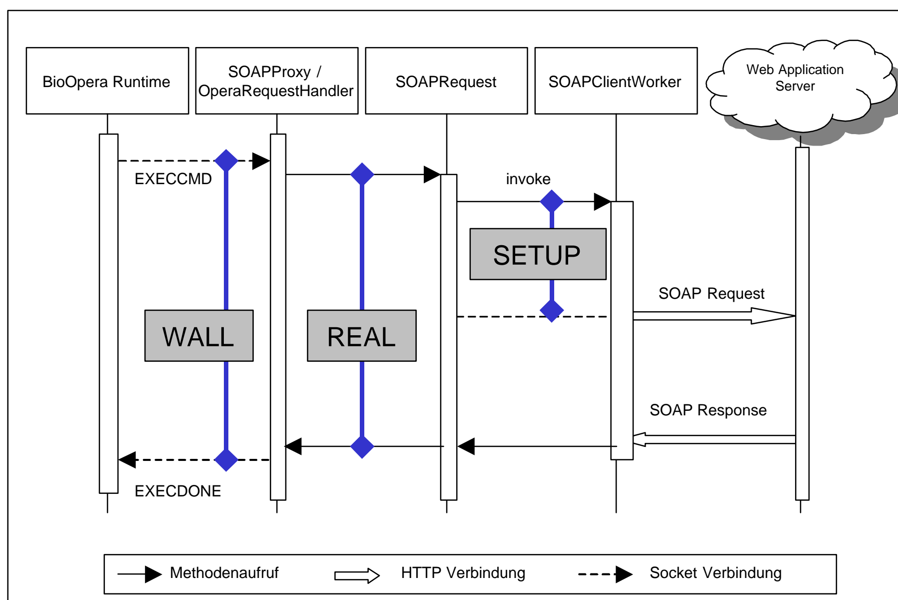


Abbildung 6.1: Messprofil

Abbildung 6.1 illustriert die erhobenen Messgrössen.

SETUP Ausführungszeit für die Konfiguration des Axis Client (beinhaltet das Laden des WSDL Dokumentes vom Server oder aus dem Cache). Bei Eingabe/Ausgabeprogrammen ist dies die Zeit für die Berechnung der XML Zeichenfolge.

REAL Gesamte Ausführungszeit eines Kommandos.

WALL Ausführungszeit der BioOpera Prozessinstanz.

Die Zeiten wurden mit der Java-Methode `System.currentTimeMillis()` ermittelt und in der Resultatstruktur (SETUP-Zeit in <cputime> und Kommandoausführungszeit in <realtime>) an die BioOpera Runtime zurückgeliefert, die Zeiten können dadurch durch die BioOpera API-Funktionen ausgelesen werden.

Die SOAPProxies wurden für jede Testreihe neu gestartet. Dies soll die Effekte durch den WSDL Cache des Axis Clients messbar machen. Der Axis Client muss das WSDL Dokument laden und verarbeiten, bevor ein Web Service ausgeführt werden kann. Werden mehrere Prozessintanzen gleichzeitig gestartet, wird das WSDL Dokumente eventuell mehrmals geladen rsp. die einzelnen Threads müssen warten, bis das WSDL Dokument im Cache verfügbar ist.
Die dadurch enstehenden Wartezeiten sind in den Abbildung in Sektion 6.3.4 zu finden.

6.2.4 Hardware Konfiguration

Verwendete Hosts			
Zweck	**OS**	**Prozessoren**	**RAM**
BioOpera DB	RedHat 7.3 Linux	2 PIII 1GHz	1 GB
	SunOS 5.6	2 UltraSparc-II 338MHz	512 MB
BioOpera IS & RT	RedHat 7.1 Linux	1 PIII 450MHz	512 MB
SOAPProxy	2 x RedHat 7.1 Linux	2 PIII 600MHz	512 MB
	2 x Windows 2000	1 PIII 1 GHz	512 MB

Mit den verwendeten Hosts wurde eine mögliche Netzwerkkonfiguration (Abbildung 6.2) mit Firewall simuliert: Der Host für BioOpera DB, der Host für BioOpera IS & RT und die 2 Hosts für den SOAPProxy (LINUX) waren durch ein 100 MBit/s Intranet untereinander vernetzt, die 2 Hosts unter Windows 2000 waren durch eine 512/128 KB/s Internetanbindung an das Intranet angebunden. Die SOAPProxies im Intranet würden nur auf vertrauenswürdige Applikationsserver (BioOpera Applikationsserver) im Internet zugreifen, die SOAPProxies vor der Firewall würden beliebige Web Services im Internet ansprechen. In der Messungen wurde auf die Unterscheidung von vertrauenswürdigen/nicht vertrauenswürdigen Hosts verzichtet.

Einen merklichen Unterschied zwischen der Ausführung auf den SOAPProxies im Intranet und im Internet konnte nicht festgestellt werden. Die Bandbreite der 512/128 KB/s Leitung war genügend gross für die Kommunikation zwischen BioOpera, den SOAPProxies und einem Internet-Testapplikationsserver. Auf eine vergleichende Messung wurde verzichtet.

Unglücklicherweise stand der Host für den BioOpera Datenbankserver nicht die ganze Zeit für die Tests zur Verfügung - es musste auf einen weniger leistungsfähigen Rechner ausgewichen werden. Betroffen sind die Testreihen:

1. T1 bis T8 auf einem SOAProxy, Kapazität 200, parallele Ausführung

2. T8 auf 4 SOAProxy's, Kapazität 200

3. T1 bis T8 auf 4 SOAProxy's, Kapazität 200, parallele Ausführung

Aus diesem Grund sind die Vergleiche zwischen Tests dieser 3 Testreihen und der ersten 3 Testreihen zu relativieren.

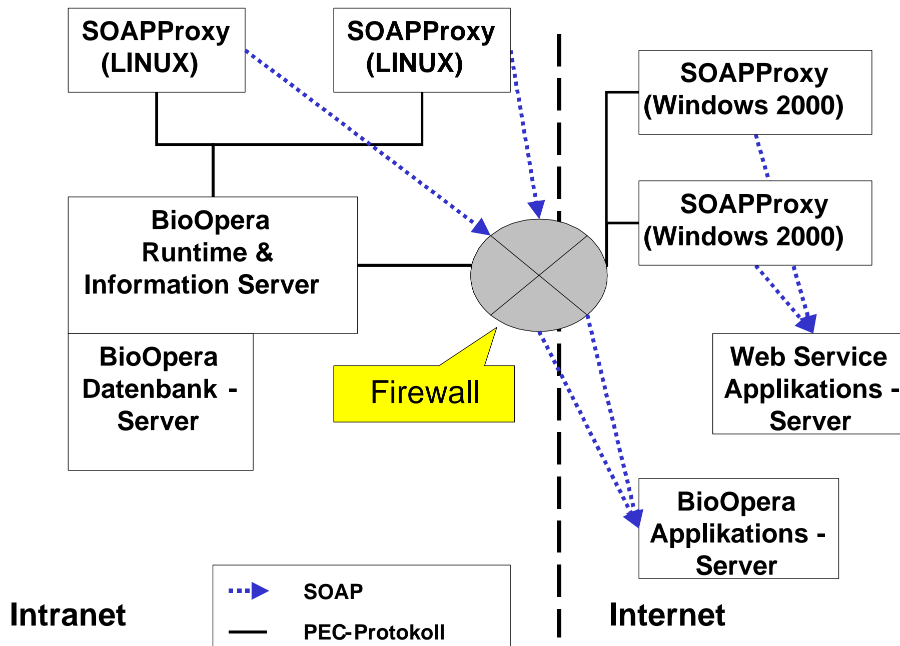


Abbildung 6.2: Netzwerkkonfiguration

6.3 Ergebnisse

6.3.1 Allgemeine Bemerkung

Die Anzahl fehlgeschlagener SOAP Calls lag im Bereich von bis zu 7 Prozent. Dies hat natürlich auch Auswirkungen auf die berechneten Durchschnittswerte. Mögliche (häufigste) Fehler-Ursache ist sehr wahrscheinlich eine Überlastung des Applikationsservers (beobachtet bei Testreihe 8 auf lokalem TOMCAT-Server: Zurücksetzen von Verbindungen, zu kleiner Thread-Pool von Request-Handlern, Probleme im Axis Web Service Stack).
Die Daten der fehlgeschlagenen SOAP Calls wurden bewusst in die Berechnungen der Durchschnittswerte einbezogen: da der Fehler-Anteil sehr hoch lag, wären viele Messreihen davon betroffen gewesen. Diese hätten ausgelassen werden müssen (unvollständige Graphiken). Die Fehler sind (teils als massive Ausschläge) in den Graphiken sehr gut erkennbar.

6.3.2 Definition Auslastungsfaktor

Für jede Testreihe wurde der maximal aufgetretene Auslastungsfaktor registriert. Der Auslastungsfaktor ist definiert durch die Anzahl aktiver Kommandoinstanzen geteilt durch die Kapazität:

$$f = \frac{A}{K}$$

mit

- A = Anzahl aktiver Instanzen

- K = Kapazität (maximale Anzahl parallel auszuführender Programme auf einem SOAPProxy)

Der maximal erreichbare Auslastungswert in der vorliegenden Testkonfiguration beträgt 2.24. Dies ergibt sich aus der Konfiguration des SOAPProxy (Anzahl Prozessoren = 2) und des BioOpera Servers (`runtime.sched.transferpolicy:TTWO`). Die BioOpera Runtime wird solange neue Kommandos an einen SOAPProxy senden, bis dessen Auslastungsfaktor (**dividiert durch die Anzahl Prozessoren**) den Wert 1.12 überschreitet (12% Überbelastung).

6.3.3 Auslastungstabellen

Testreihe	OS	Auslastung ohne Ein/Ausgabe	Auslastung mit Ein/Ausgabe
	1 SOAPProxy Kapazität: 100 CPU: 2		
T1	RedHat Linux	0.020	0.020
T7	RedHat Linux	0.070	0.020
T8	RedHat Linux	2.210	2.180
MIX	RedHat Linux	0.510	0.250
	1 SOAPProxy Kapazität: 200 CPU: 2		
T1	RedHat Linux	0.075	0.045
T7	RedHat Linux	0.045	0.010
T8	RedHat Linux	1.265	1.105
MIX	RedHat Linux	0.240	0.240
	4 SOAPProxies Kapazität: 200 CPU: 2		
T8	RedHat Linux	1.030	0.540
	RedHat Linux	0.970	0.445
	Windows 2000	0.835	0.485
	Windows 2000	0.775	0.425
MIX	RedHat Linux	0.165	0.095
	RedHat Linux	0.085	0.065
	Windows 2000	0.090	0.145
	Windows 2000	0.060	0.080

Bei Web Services mit einer Antwortzeit von weniger als 1 Sekunde werden die Kommandos praktisch sequentiell auf dem SOAPProxy ausgeführt. Nur bei langandauernden Berechnungen durch den Web Service (Testreihe 8) werden mehrere Web Services gleichzeitig auf dem SOAPProxy ausgeführt.

Die Kapazitätsgrenze an parallel ausführbaren Instanzen erreicht nur Testreihe T8 bei Kapazität 100 auf einem SOAPProxy (221 bzw. 218). Bei Testreihe T8 Kapazität 200 auf einem SOAPProxy wird zwar absolut ein höherer Wert erreicht (253 bzw. 221), bei der maximal möglichen absoluten Anzahl von 448 parallelen Instanzen ergibt dies aber ”nur” eine Auslastung von 1.265. Daraus kann geschlossen werden, dass die Leistungsgrenze des SOAPProxy bei einer absoluten Anzahl an parallel ausgeführten Web Services bei ca. 250 liegt.

6.3.4 Zeitmessungen

In den folgenden Ausführungen handelt es sich bei Zeitangaben stets um Durchschnittswerte.

WSDL-Cache Effekte

Abbildung 6.3 und 6.4 zeigen deutlich den WSDL Cache-Effekt des Axis Clients, der durchschnittliche Konfigurationszeitbedarf sinkt ab, während der maximale Zeitbedarf ansteigt (dadurch dass beim ersten Zugriff auf den Web Service bei steigender Anzahl Kommandos immer mehr Instanzen gleichzeitig das WSDL Dokument anfordern). Bei der oberen Grafik in Abbildung 6.3 sind starke Schwankungen durch Fehler entstanden. Die Eingabe- / Ausgabeoperation können sich positiv auf die SETUP-Zeit auswirken. Während ohne Ein- / Ausgabeoperation zu Beginn gleich mehrere Axis Clients auf das WSDL Dokument zugreifen wollen, ist dies bei den Testreihen ohne Ein/Ausgabe nicht der Fall: die Zeit zwischen 2 SOAP Call-Programmen ist dementsprechend gross, dass der Cache für folgende Axis Clients früher bereit ist (siehe auch Abbildung 6.5 blaue Fläche). Ohne Ein- / Ausgabeprogramme sinkt die SETUP-Kurve, während sie bei gleicher Anzahl an gestarteten Prozessinstanzen ohne Ein- / Ausgabeprogramme zuerst ansteigt und dann langsam fällt.

Ab einer Anzahl von 64 Prozessinstanzen, die in BioOpera gestartet wurden, zeigt der WSDL Cache den maximalen Nutzen.

Realzeit

Aus Abbildung 6.6 ist die Zeit für die SOAP Call Ausführung (inkl. Kommunikationszeit mit BioOpera) ersichtlich. Bei Web Services mit kurzer Antwortzeit (obere Grafik) ist ein klarer Unterschied durch die Ein- / Ausgabeprogramme erkennbar (Annahme: bei beiden Testreihen war die Antwortzeit des SOAP Calls gleich). Das heisst das die 4 Ein- /Ausgabeprogramme durchschnittlich einen Zeitbedarf von ca. 2 Sekunden haben (Overhead durch den Mechanismus der Ein- / Ausgabeprogramme). Bei langandauernden SOAP Calls ist der zusätzliche Zeitbedarf durch die Ein- / Ausgabeprogramme vernachlässigbar klein (in 2. Grafik in Abbildung 6.6 abgesehen durch die starken Schwankungen durch Fehler nicht erkennbar). Interessant ist, dass bei geringer Auslastung der grösste Teil der Zeit für die Konfiguration des Axis Clients verbraucht wird. Das Ausführen des SOAP Calls liegt in der Testreihe T7 im Bereich von unter einer Sekunde. Bei der 2. Grafik in Abbildung 6.5 wird das Zeitbedarfs-Verhältnis durch die Ein/Ausgabeprogramme merklich verändert; der Unterschied von ca. zwei zusätzlichen Sekunden ist v.a. durch die Kommunikation mit BioOpera und Thread-Wartezeiten zu erklären, da die Ausführungszeit der Ein- / Ausgabeprogramme-Berechnungen vernachlässigbar klein ist.

Abbildung 6.7 vergleicht nochmals die REAL-Zeit für die Testreihe 7 (Kapazität 200) und 8 (Kapazität 100), um die Differenz durch die Ein/Ausgabeprogramme zu illustrieren. Auf die Ausführungszeit von Testreihe 8 haben die Ein/Ausgabeprogramme durch die verhältnismässig lange Dauer des SOAP Calls praktisch keinen Einfluss mehr. Bei der Verwendung von mehreren SOAPProxies (Abbildung 6.8) fällt der Overhead durch die Ein/Ausgabeprogramme deutlich kleiner aus, da die Programme auf

mehreren Hosts verteilt ausgeführt werden.

Abbildung 6.9 vergleicht die Real-Zeit für Testreihe 7 und 8 bei Kapazität 100 und 200. Die Kapazitätsänderung hat keinen wesentlichen Einfluss auf die Ausführungszeit (die Prozessorbelastung durch den AxisClient ist minimal).

Gesamtausführungszeit

Abbildung 6.10 vergleicht die Gesamtausführungszeit WALL für Testreihe 7 und 8 bei Kapazität 100 und 200. Die Kapazitätsänderung hat einen wesentlichen Einfluss auf die Gesamtzeit bei langandauernden Web Services (Durchsatz). Dies kann mit der Überbelastung der Kommunikation zwischen SOAPProxy und Subsystem begründet werden. Noch stärker verdeutlicht wird dies durch Abbildung 6.11: bei der Verteilung der Kommandos auf 4 SOAPProxies liegen die Auslastungsfaktoren um 0.5 [2] herum (entspricht ca. 100 parallelen Ausführungen). Das theoretische Maximum an parallelen Ausführungen bei der Ausführung auf 1 SOAPProxy liegt bei Kapazität 200 und 2 Prozessoren bei 448 (Kapazität 200 bei 2 Prozessoren und 12% Überlast), das in den Test aufgetretene Maximum liegt bei 253 parallelen Ausführungen (Test 8 ohne Ein/Ausgabeprogramme). Das Verhältnis der Ausführungszeit auf einem gegenüber der Ausführungszeit auf vier SOAPProxies steigt stark mit der Anzahl Prozessinstanzen in BioOpera, bei 1024 Instanzen beträgt das Verhältnis mehr als 4.
Bei Web Services mit kurzer Antwortzeit hat die Kapazitätsänderung keinen wesentlichen Einfluss auf die Gesamtausführungszeit.

6.4 Schlussfolgerungen / Konfigurationsprofile

Aus den vorliegenden Messungen lässt sich interpretieren, dass die optimale Konfiguration von den auszuführenden Web Services abhängig ist.

Die Optimierung kann für 2 Grössen vorgenommen werden: Minimierung der Gesamtausführungszeit (Durchsatz) oder Minimierung der REAL-Zeit (SOAP Call Ausführungszeit). Ersteres wird durch möglichst hohen Parallelitätsgrad (Anzahl SOAPProxies) erreicht, was sich aber schlecht auf den WSDL-Cache Effekt und damit auf die REAL-Zeit auswirken kann. Minimierung der SOAP Call Antwortzeit erreicht man durch möglichst hohe Auslastungsfaktoren (möglichst kleiner Parallelitätsgrad).

Die reale Kapazitätsgrenze (= Kapazität multipliziert mit der Anzahl Prozessoren) sollte auf jeden Fall 200 nicht überschreiten; es sollte ein Wert zwischen 100 und 200 in Betracht gezogen werden, wie sich in Sektion 6.3.3 und 6.3.4 gezeigt hat, wirkt sich eine absolute Kapazität von über 200 (Leistungsgrenze des SOAPProxy) schlecht auf die Gesamtausführungszeit aus.

Bei Web Services mit einer langen Antwortzeit sollten möglichst viele SOAPProxies dem Cluster zur Verfügung stehen, da sich dies positiv auf die Ausführungszeit (Durchsatz) einer BioOpera Prozessinstanz auswirkt, bei kurzen Antwortzeiten und einer hohen Anzahl an Prozessinstanzen sollten 2 Faktoren beachtet werden: Um Synergie-Effekte durch den Axis WSDL-Cache nutzen zu können, sollte die Anzahl an SOAPProxies nicht zu hoch gewählt werden. Die Anzahl an SOAPProxies kann auch dazu dienen, den angesprochenen Applikationsserver vor Überlast zu schützen.

[2] siehe Tabelle 6.3.3.

Die Messungen haben gezeigt, dass der Overhead für die Ein- / Ausgabeprogramme im Bereich von 2 Sekunden (für je 2 Ein- und Ausgabeprogramme) liegt. Bei einer hohen Anzahl an Prozess-Instanzen mit kurzer Antwortzeit wird das Verhältnis des Overheads zur SOAP Call Ausführungszeit 2:1, was als negative Eigenheit des Mechanismus der Ein- / Ausgabeprogramme beurteilt werden kann. Bei langer Antwortzeit hingegen ist der Overhead vernachlässigbar klein.

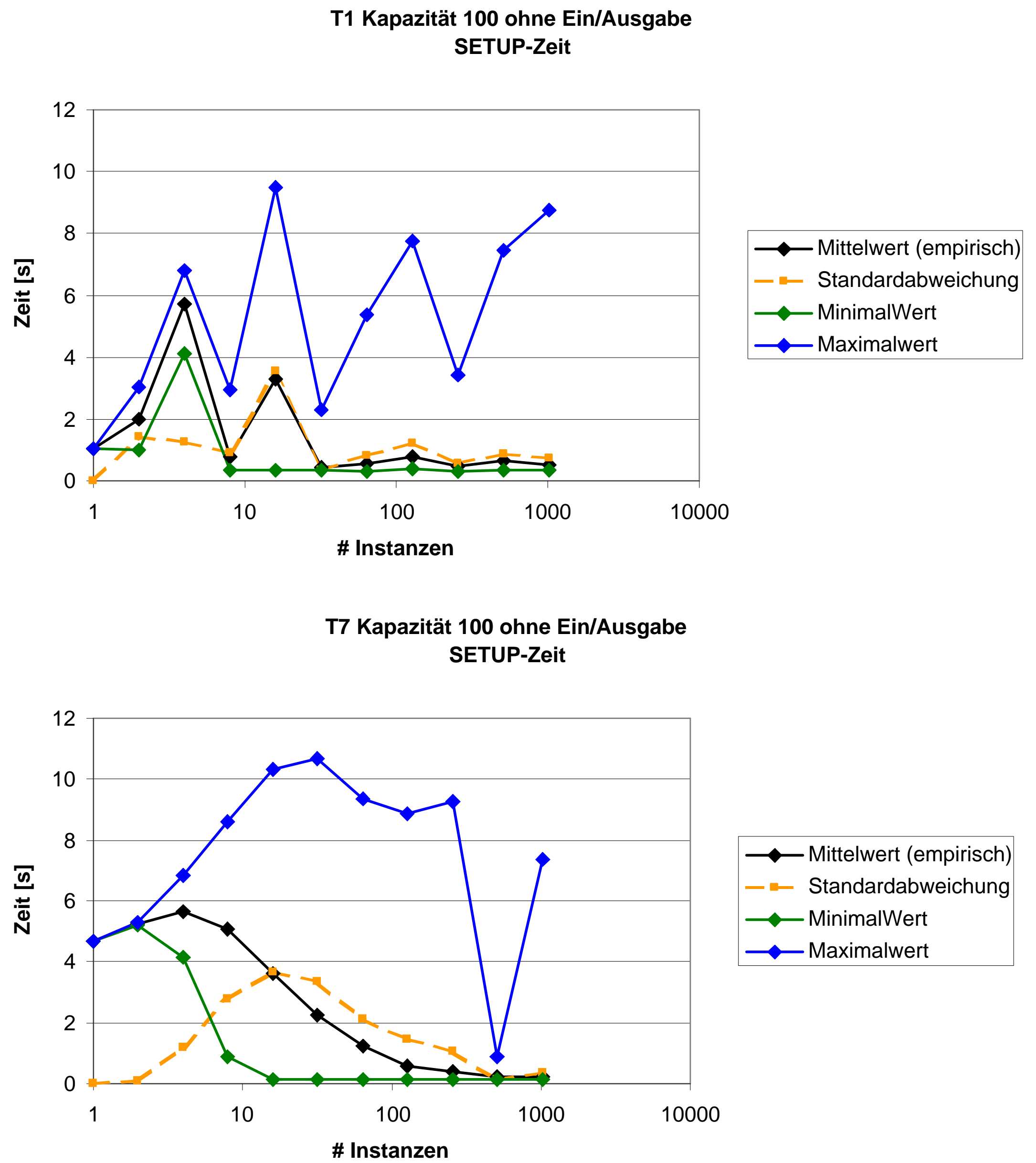


Abbildung 6.3: Testreihe 1 und 7 auf einem SOAPProxy: WSDL-Cache Effekt

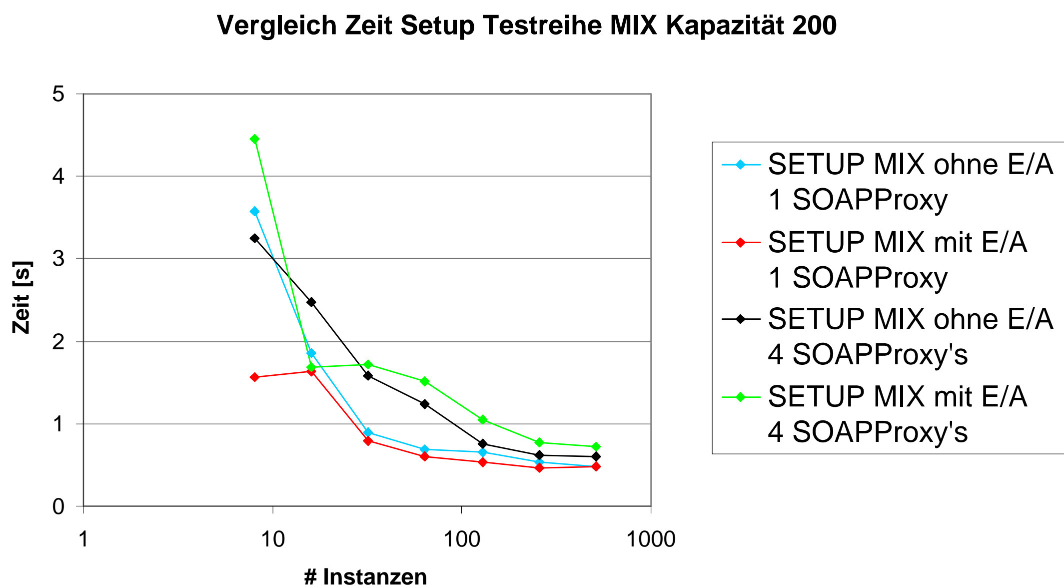


Abbildung 6.4: Testreihe MIX auf einem und 4 SOAPProxies: WSDL-Cache Effekt

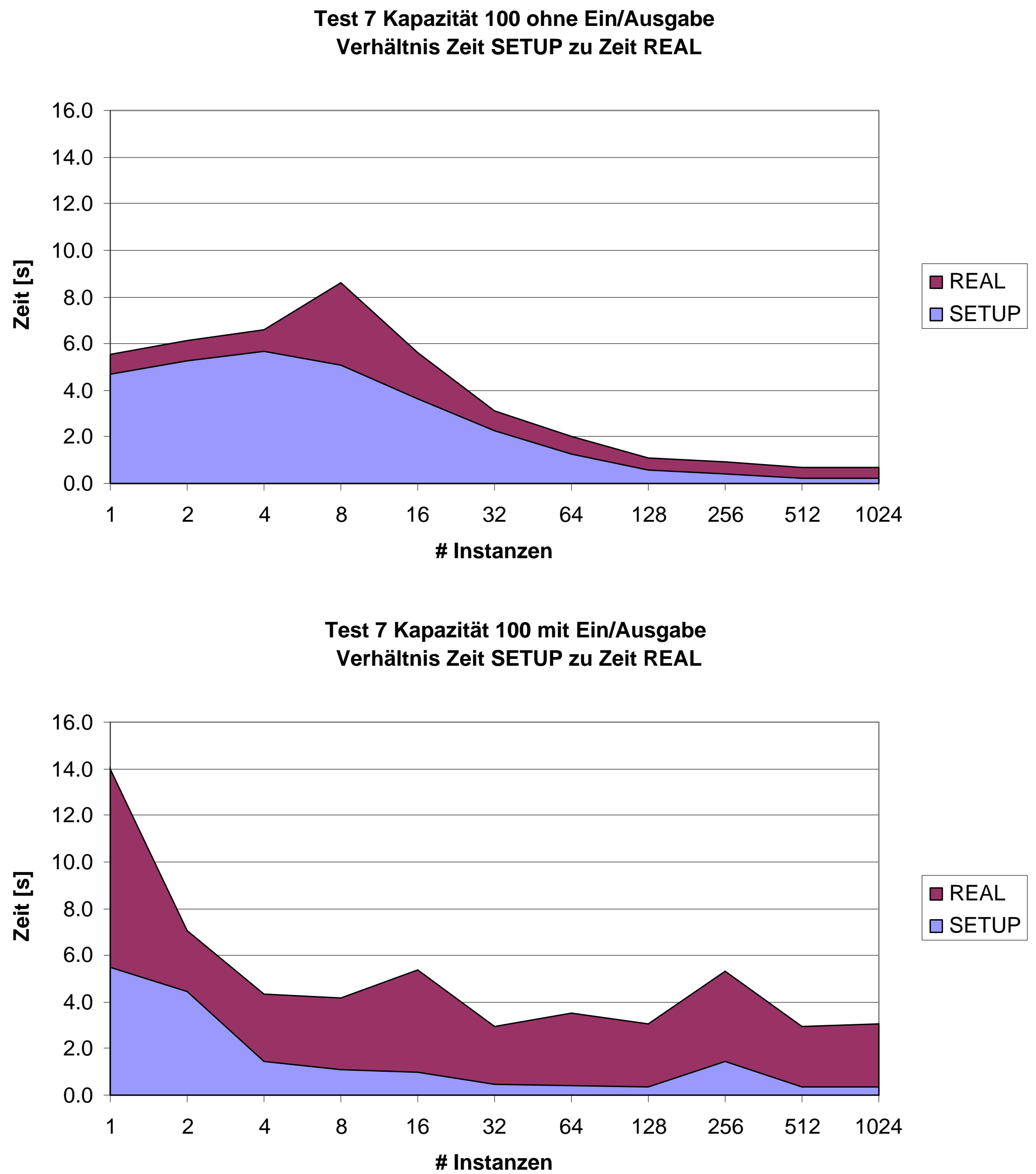
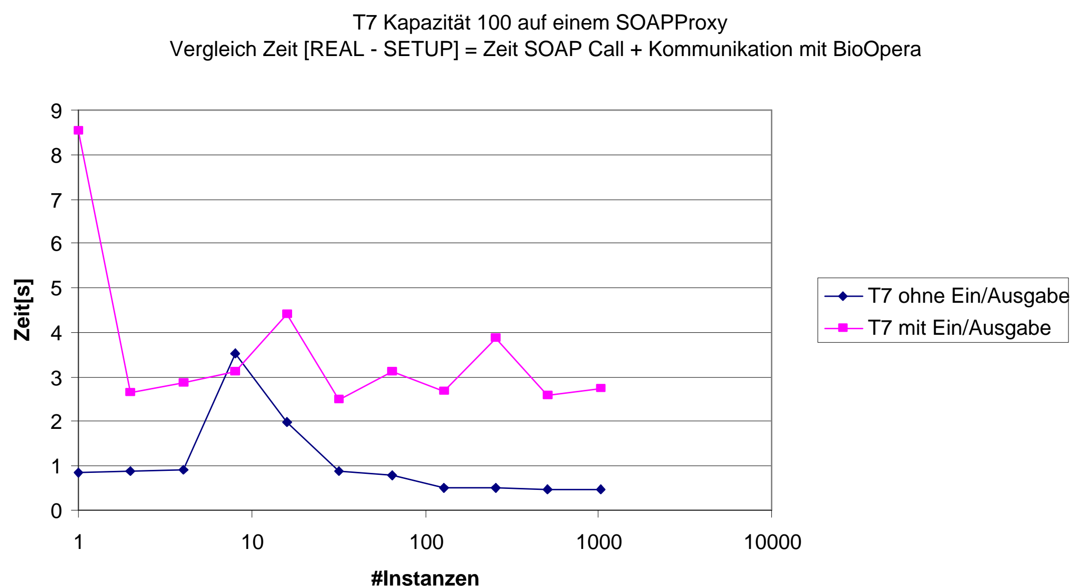


Abbildung 6.5: Testreihe 7 auf einem SOAPProxy: Differenz Setup- zu Realzeit



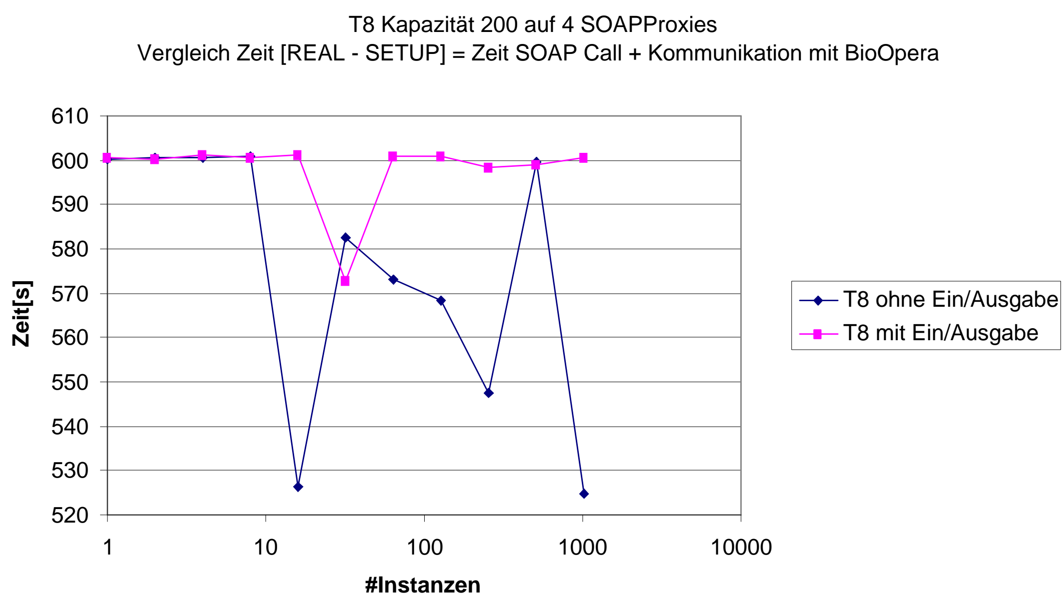


Abbildung 6.6: Testreihe 7 und 8: Berechnete Ausführungszeit für SOAPCall, Kommunikation mit BioOpera, Threadwartezeiten

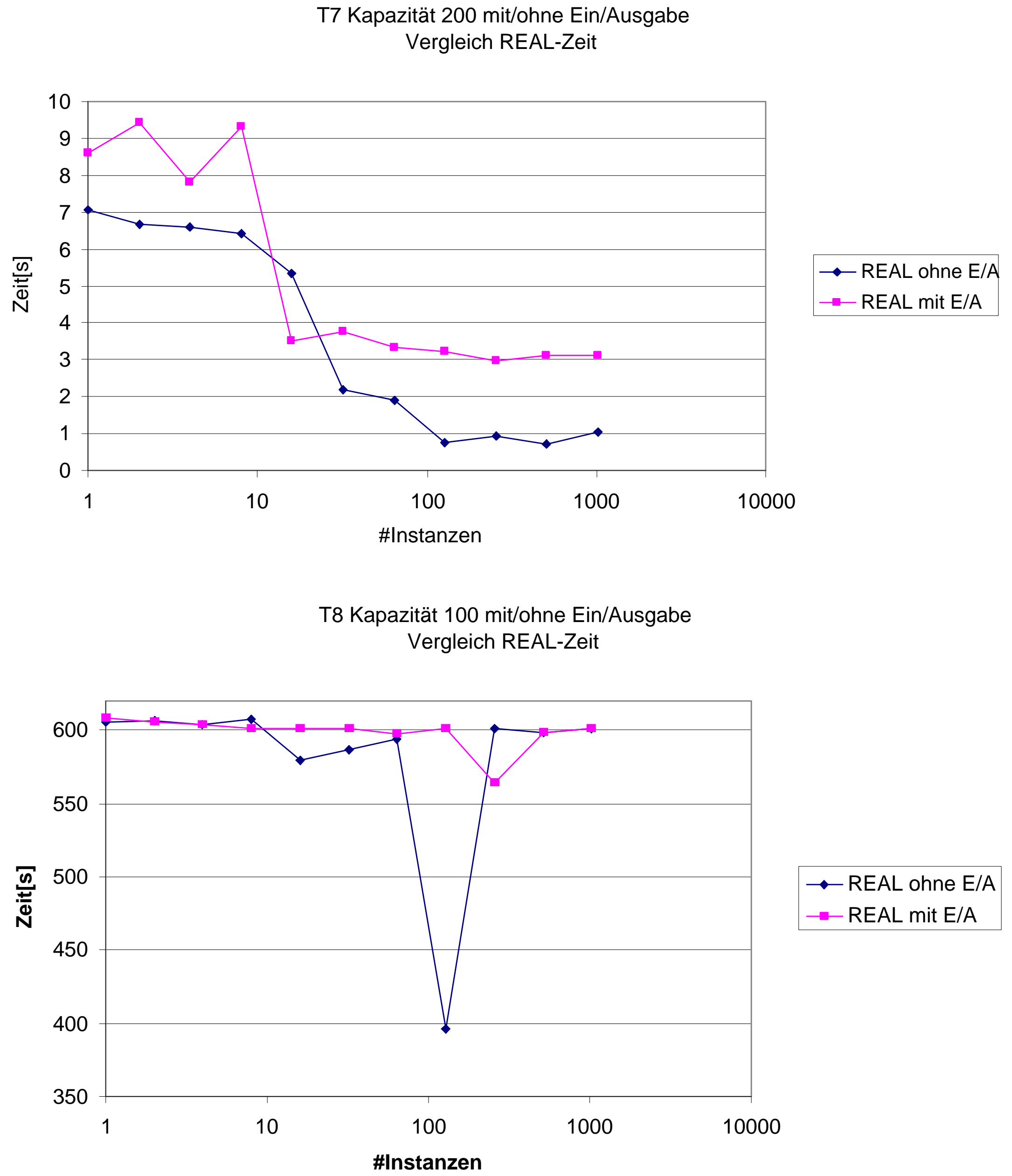


Abbildung 6.7: Testreihe 7 und 8 auf einem SOAPProxy: Differenz Realzeit mit/ohne Ein/Ausgabe

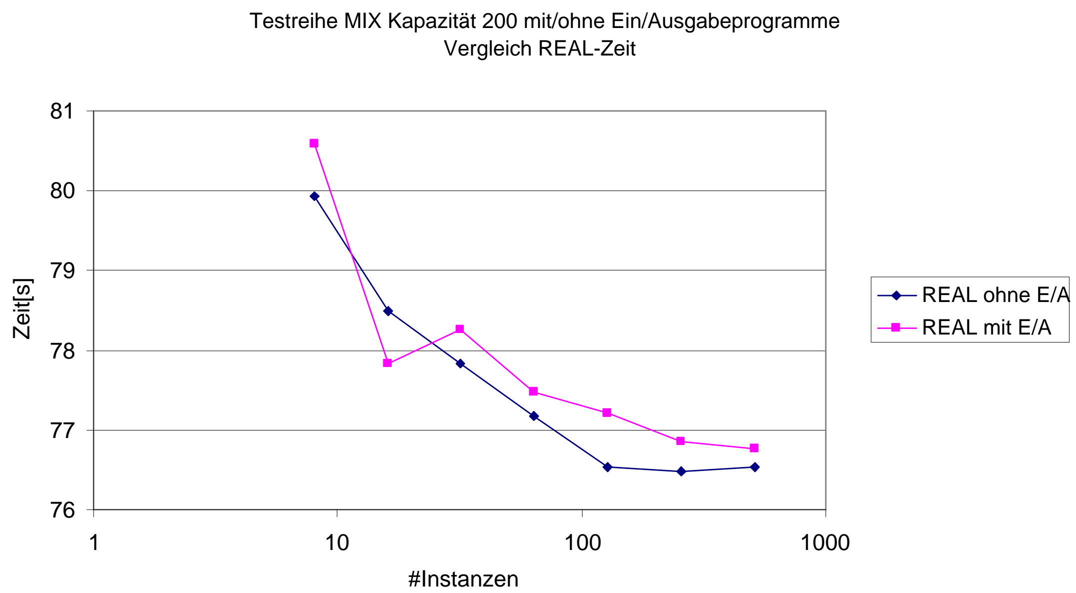


Abbildung 6.8: Testreihe MIX auf 4 SOAPProxies: Differenz Realzeit mit/ohne Ein/Ausgabe

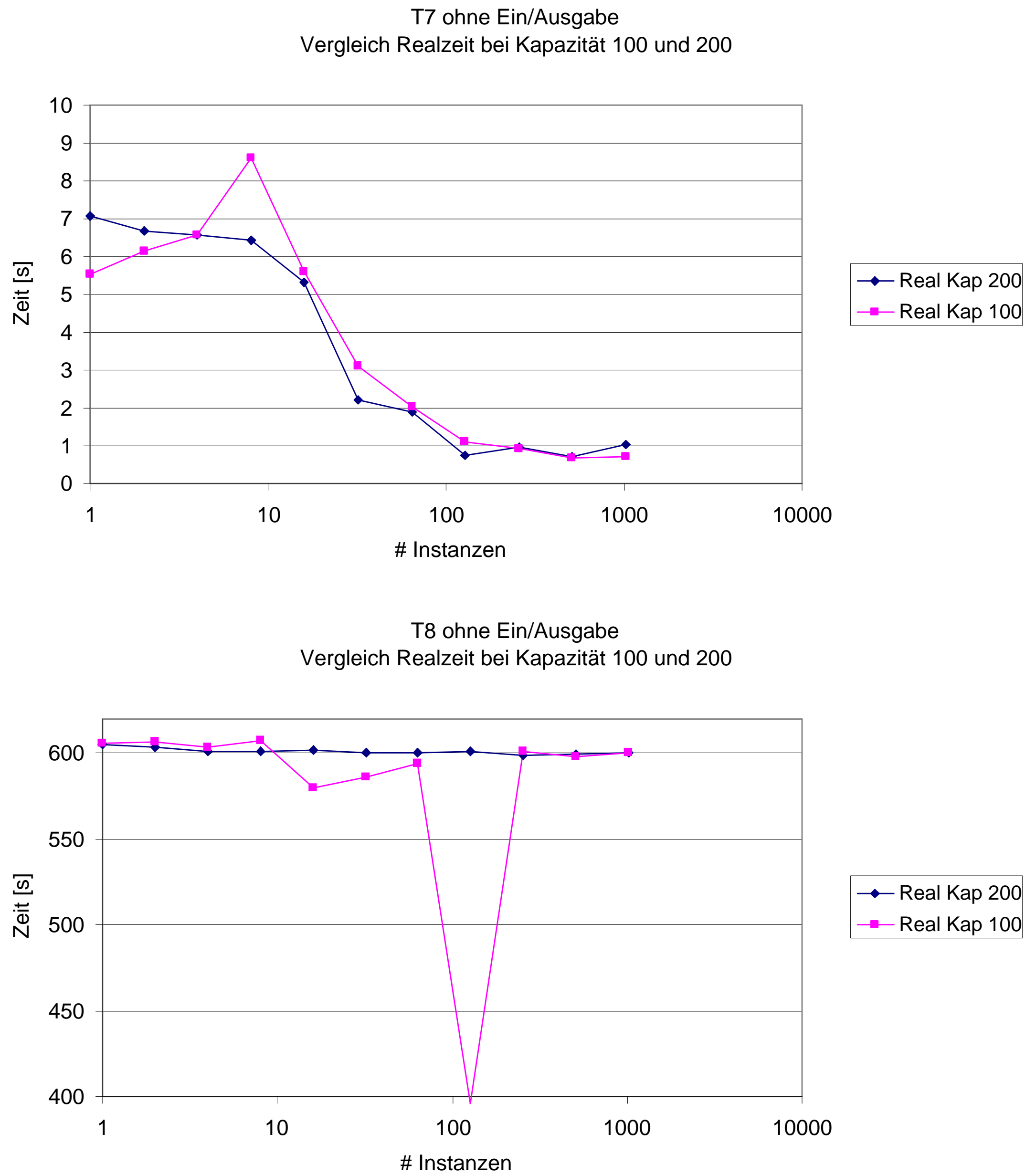
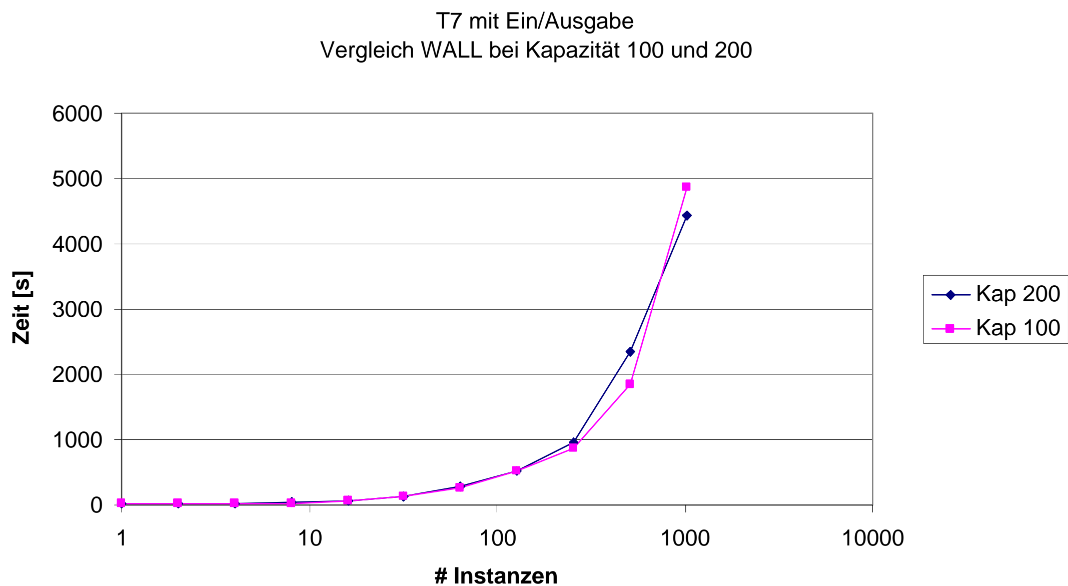


Abbildung 6.9: Testreihe 7 und 8 auf einem SOAPProxy: Unterschied Realzeit bei Kapazität 100 und 200



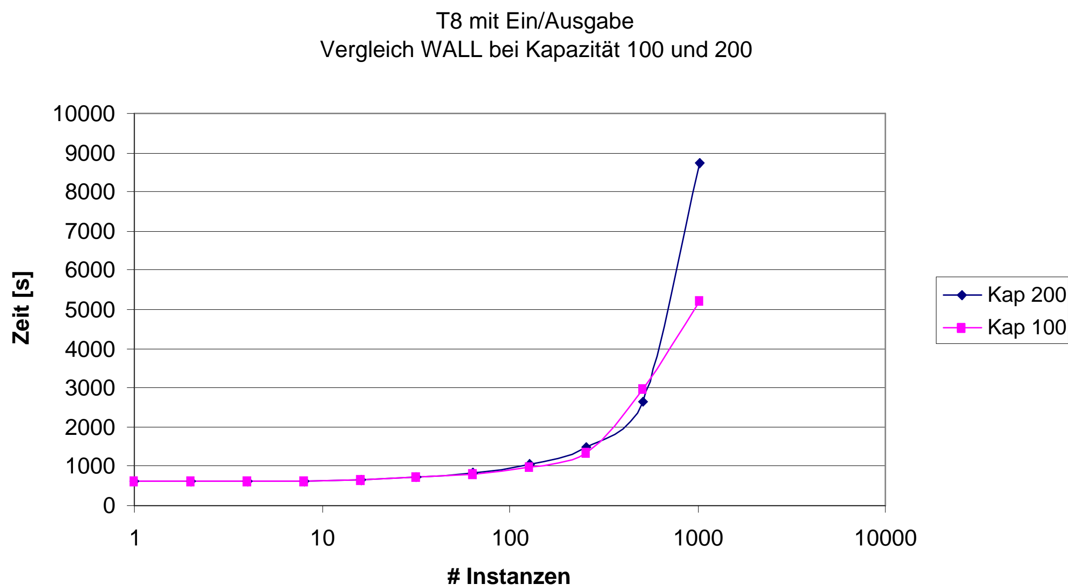


Abbildung 6.10: Testreihe 7 und 8 auf einem SOAPProxy: Gesamtausführungszeit-Unterschied bei Kapazität 100 und 200

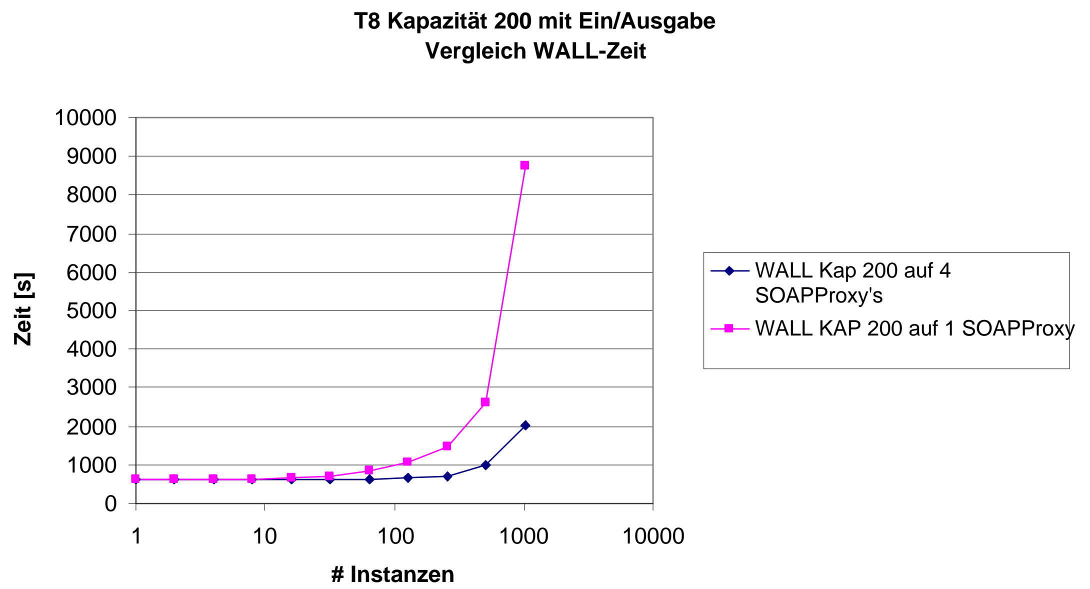


Abbildung 6.11: Testreihe 8 auf 1 und 4 SOAPProxies: Gesamtausführungszeit-Unterschied bei Kapazität 200

Kapitel 7

Erweiterungen und Ausblick

Dieses Kapitel gibt einen Überblick über die möglichen Erweiterung des UDDIBrowser und des SOAP Subsystems.

7.1 Automatische OCR Generierung für Prozesse

Die automatisch generierten Programme enthalten viele Ein- und Ausgabeprogramme, die manuel zusammengesetzt werden müssen. Da dies nicht sehr anwenderfreundlich ist (ca. 10 Mausklicks pro Programm), sollte eine Automatisierung entwickelt werden. Zwei mögliche Ansatzpunkte sind:

- **UDDIBrowser** Zusätzlich zu den Programmen wird eine entsprechende Prozess-Beschreibung generiert. Die Hauptaufgabe liegt in der korrekten Verlinkung der verschiedenen Elemente einer Prozessrepräsentation durch die `OID`-Attribute im XML Dokument.

- **Visual OCR Designer** Während des Programm-Imports durch den BioOpera Visual Process Designer werden automatisch Prozess-Wrapper generiert. Dies kann aufgrund der Namensgebung der Programme und der Programm-Parameter geschehen (siehe C.5.1).

7.2 UDDI Registry V3

Die Entwicklung von UDDI wird Änderungen und neue Funktionalität bringen, die eventuell eine Anpassung des UDDI Browser erfordern. Zum jetzigen Zeitpunkt ist die UDDI Spezifikation Version 2 im Einsatz, Version 3 wurde publiziert. Durch das Austauschen der JAXR Java-Archive kann ein einfaches Aktualisieren möglich sein, falls die neue JAXR Version abwärtskompatibel ist.

UDDI Spezifikation und WSDL Beschreibung	
UDDI Version 3	http://www.oasis-open.org/committees/uddi-spec/tcspecs.shtml#uddiv3

7.3 Typsystem

BioOpera kennt keine komplexen Datenstrukturen, wie sie in Web Services vorkommen. Der Ansatz der Ein- und Ausgabeprogramme könnte optimiert werden, da diese einen hohen Kommunikationsaufwand generieren. Wie eine solche Typerweiterung aussehen könnte, liegt ausserhalb meines Horizontes, da eine Änderung am Typsystem sicher mehrere Kernelkomponenten betreffen wird. Ein Ansatz wäre möglicherweise die Verwendung von XML Schema als Beschreibung/Definition von Parametertypen.
Alternativ dazu könnte die Implementation des SOAPProxies mittels statischer Stub-Generierung (WSDL-Compiler) in Betracht gezogen werden. Dies würde die Ein- / Ausgabeprogramme hinfällig machen und den Kommunikationsaufwand stark verkleinern (im Minimum um Faktor 5).

7.4 Host based authentication

Der SOAPProxy implementiert noch nicht die im Program Execution Client implementierten Sicherheitsmassnahmen (Überprüfen der Addressmaskierungen) zur Absicherung gegen unerlaubte Zugriffe. Eine Erweiterung würde zusätzliche Funktionen im Code und einen zusätzlichen Konfigurationsteil benötigen. Nähere Information finden sich in [1] im Kapitel ”Security”.

7.5 ExecutionMultiQuery

Die Belastung des SOAPProxies durch `EXECQUERY` könnte durch eine Erweiterung des `PEC`-Protokolls optimiert werden. Im Falle von langandauernden Web Services wird der SOAPProxy periodisch durch BioOpera nach dem Status eines einzelnen Jobs angefragt. Diese Anfragen treten bei einer grossen Anzahl an gleichzeitig gestarteten Prozessinstanzen sequentiell auf; während dieser Zeit (meist mehrere Sekunden) wird die Kommunikation des SOAPProxies vollständig ausgelastet.
Eine Erweiterung des `PEC`-Protokolls definiert eine neue Query-Message, die als Argument eine Liste oder einen Range von Job-ID’s enthält. Der SOAPProxy antwortet auf diese Query-Message mit einer Liste von ID - Status Informationen.

7.6 Ausblick

Mit der generischen Web Service Schnittstelle wurde die Grundlage für das Einbinden von Fremdsystemen geschaffen. Die Weiterentwicklung sollte hier ansetzen und in den Bereichen Interoperabilität, Sicherheit und der Unterstützung von weiteren Standards fortgeführt werden.
Interessant wäre auch die Anwendung einer privaten UDDI Registry gewesen, in der bestehende BioOpera Prozesse dynamisch registriert werden und mittels UDDIBrowser in neue Prozesse eingebaut werden können.
Die Performance-Tests des SOAP Subsystems haben gezeigt, dass das System durch die Verwendung eines Clusters skaliert und der Durchsatz für Prozessausführungen maximiert werden kann.
Die hohe Fehlerquote bei den getesteten Web Services weist darauf hin, dass die meisten Web Service Applikationensserver noch nicht sehr stabil sind.

7.7 Persönlicher Schlusskommentar

Rückblickend hat diese Arbeit ihre Ziele für mich erfüllt, die formal gestellten Aufgaben sind erfüllt und die Grundlage für weitere Arbeiten in diesem Bereich wurde geschaffen. Im grossen und ganzen konnte ich dem vorgeschlagenen Zeitplan folgen: anfangs konnte ich einige Zeit gutmachen, da ich bereits mit Web Services Erfahrungen gesammelt hatte. In der zweiten Phase (Entwicklung UDDIBrowser) benötigte ich diese Zeit leider wieder, um mich mit Java Swing auseinanderzusetzen. Die Entwicklung des UDDIBrowsers war zudem auch vom Aspekt der WSDL Spezifikation anspruchsvoll, da diese viele Eigenschaften nicht zwingend fordert und dementsprechend der Unterschied zwischen verschiedenen Implementation von Web Service Plattformen einen grossen Aufwand an Funktionstest mit sich brachte (Ziel war, mind. 80% der momentan verfügbaren Plattformen zu unterstützen - mit dem Ergebnis, dass mehr als 90% der getesteten Web Services mit dem SOAPProxy kompatibel waren). Durch diese fortlaufenden Anpassungen zur Unterstützung von möglichst vielen Plattformen erforderte die Entwicklung des SOAP Subsystem mehr Zeit als geplant war.
Trotzdem investierte ich gegen Ende der Arbeit einiges an Zeit für Leistungstest, um die Entwicklungsarbeit abzurunden. Persönlich konnte ich viel aus der Arbeit lernen, insbesondere durch die vertiefte Auseinandersetzung mit unzähligen Standards hat sich mein technischer Horizont stark erweitert.

Anhang A

Aufgabenstellung: Web Service Execution for BioOpera

Diplomarbeit
Student: Markus Haller <busi@smile.ch>
Start date: 4.11.2002
End date: 3.3.2003
Assistant: Cesare Pautasso <pautasso@inf.ethz.ch>

Motivation

BioOpera integrates calls to standard UNIX applications into a process. This Project will extend the execution subsystem to support calling a Web Service from within a process using the SOAP protocol (or equivalent). Some more work is also needed to make BioOpera understand a Web Service Interface Description and use it to call such a Web Service.

Goals

1. Design a mapping between a Web Service (specified in WSDL) and a BioOpera Program Registration (described in OCR)

2. Implement a tool to automatically import a Web Service Interface into the BioOpera Program Library.

3. Develop, integrate and test the Web Services execution subsystem in the BioOpera Kernel.

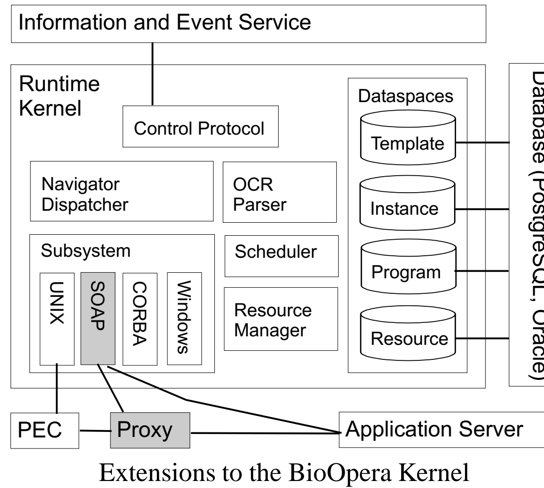
4. Measure the performance and scalability of the solution.

Background

This project will enable the BioOpera System to integrate any set of Web Services within a Process. To do so we need to model a Web Service Interface as an OCR

Program, that is a software component which can be called from BioOpera with a set of input parameters and returns back to BioOpera some output parameters. Furthermore, the appropriate extensions to the Kernel Execution Subsystem need to be developed adding to BioOpera a basic mechanism to execute a Web Service call using the SOAP protocol.

In order to connect BioOpera with an Application Server, and make it call a Web Service, various solutions can be considered:



Extensions to the BioOpera Kernel

1. A direct connection between the Kernel Subsystem and the Application Server.

2. A three layered Subsystem-Proxy-Application Server setup, with the advantage of offloading some of the work to a separate component.

3. A call through an external proxy program invoked through the existing BioOpera PEC component.

The feasibility, as well as the advantages and disadvantages of the previous solutions will be analyzed and the appropriate solution will be developed and benchmarked.

Timeline

The Diplomarbeit lasts 4 months. This time can be organized as follows:

Week	Phase
1-3	Learning how to use the system and studying the Web Services Specifications.
4-6	Design and implementation of the import tool.
7-11	Implementation and integration of the Web Services Execution Subsystem.
12-13	Performance Measurements.
14-16	Writing the Final Report.

Guidelines

You are free to organize your working time and environment independently. You are encouraged to keep a log of your activities, we shall meet at least once a week to discuss your progress.

This work is part of a software research and development project, it is important to write clear, well documented code which seamlessly integrates with the rest of the system.

You can write the final report in german or english. The BioOpera Kernel code is programmed in C++, existing Web Services code is written in Java based on the Sun Toolkit.

The final grade will depend from the quality of both the software solution and the final report.

Appendix B

Program Execution Client Protocol

B.1 Purpose

The PEC protocol defines the messages (encoded using the Opera Wire Protocol) between the Opera information server, the Opera runtime server, the program execution client and the BioOpera Graphical User Interface.

B.2 Protocol update

The new protocol supports asynchronous execution requests of programs. The advantage is that a socket is not kept open between OPERA and the PEC during the execution of a program, which typically lasts a significant amount of time, thus freeing valuable communication resources. Further optimization may be needed for short lasting programs, for which the asynchronous requests may involve extra overhead. The OpCodes values are defined in the file ProgClientOpCodes.h [1] and in the file OpCodes.java [2], whereas ProgClientOpCodes.h is authoritative.

B.3 Communication topics

The main topics of the messages exchanged between OPERA and a PEC are:

1. Resource Management.
2. Program Execution.
3. System Administration.

This protocol is implemented in the UnixSubSystem, the progclient-cpp/ module (see `ProgClientOpCodes.cc`) and the Java SOAPProxy module (see `OperaRequestHandler.java`).

[1] located in bioopera/server/include/dispatcher
[2] located in bioopera/java/src/ch/ethz/inf/bioopera/ws/soapproxy

B.3.1 Resource Management

This exchanged information is about the state of a host (**up**, **down**), and its utilization indexes (uptime, job count, free memory,...). The PEC informs OPERA of relevant load changes and OPERA periodically polls the PEC for updates. No permanent connections are involved. Messages involved: HOSTSTATE and FULLHOSTSTATE.

B.3.2 Program Execution

Asynchonous program execution requests involve these messages: `EXECCMD, EXECDONE, EXECABORT, EXECSUSPEND, EXECRESUME, EXECQUERY, EXECQUERYRECOVER and ISPERSISTENT`.

The first two are needed to communicate an execution request from OPERA to the PEC, and to communicate the results of the program execution from the PEC to OPERA. The third is used to abort the execution of a running program. EXECSUSPEND and EXECRESUME are used to interupt and resume a running execution, the next two messages are used by OPERA to poll the PEC for updates on the state of the execution of a program, in case of OPERA-side timeouts or during recovery. The `ISPERSISTENT` message signals the client that the program result has been stored on the server side and the client can delete all information about it.

All these messages must contain the activity's ID (or equivalent).

B.3.3 System Administration

These messages are used for system administration purposes. In order to simplify the management of the running PEC over the nodes of a cluster, these messages allow to remotely control its PEC execution, and query its state.

B.4 Resource Manager Messages

B.4.1 Host state query

This message is used to query the state of a host. This message is used when OPERA boots, and periodically when the hosts are being polled by the Resource Manager.

OPERA TO PEC:

```
<OpCodes.HOSTSTATE>
```

There are two possible outcomes of this message:

up OPERA is able to connect to the PEC which will answer with this message:

PEC TO OPERA:

```
<OpCodes.HOSTUPDATE> <HOSTNAME> <LOADSTRING>

where
<HOSTNAME> = the workstation's hostname.
<LOADSTRING> = B<number>,J<number>,M<number>,F<number>
```

The message contains the measured load index (uptime), the number of running programs, the amount of free memory and the amount of free swap space.

down OPERA is unable to connect. This means that the machine is down or that the PEC is not started on it. Therefore the host state is considered to be down, at this point OPERA should check for programs that were supposed to be running on this host and restart them somewhere else.

B.4.2 Full Host state query

This message is used to query the full state of a host.

OPERA TO PEC:

```
<OpCodes.FULLHOSTSTATE>
```

PEC TO OPERA:

```
<OpCodes.HOSTUPDATE> <HOSTNAME> <LOADSTRING>

where
<HOSTNAME>   = the workstation's hostname.
<LOADSTRING> = B<number>,J<number>,M<number>,F<number>,C<number>,P<number>,
               R<number>,S<number>,U<string>,O<string>,V<string>,T<string>
```

The message contains the measured load index, the number of running programs, the amount of free memory, the amount of free swap space, the number of processors, the performance index, the total amount of ram, the total amount of swap space, the user name, the operating system, the client version and the client start time.

B.4.3 Host state update

This message is sent by the PEC to OPERA whenever there is a significant change in the state of the host and when the PEC is started.

PEC TO OPERA:

```
<OpCodes.HOSTUPDATE> <HOSTNAME> B<number>,J<number>,M<number>,F<number>
```

The message contains one or more of the host state items, depending on which one changed.

The same message is also used when the PEC is being stopped by the user:

PEC TO OPERA:

```
<OpCodes.HOSTUPDATE> <HOSTNAME> down
```

In this case the content of the message clearly indicates that the host is not going to be available anymore.

The answer from OPERA to this message is an acknowledgement:

OPERA TO PEC:

```
<OpCodes.ACK_HOSTUPDATE_OK> or <OpCodes.ACK_HOSTUPDATE_UNKNOWNHOST>
```

where the result is generally **ok** , and only if the host is unknown to OPERA it will be **unknownhost**. In this case both the PEC and OPERA should warn the user that the host has not been registered with the Resource Manager.

B.5 Program Execution Messages

B.5.1 Execute Command

To remotely execute a program OPERA will send the following message to the PEC: OPERA TO PEC:

```
<OpCodes.EXECCMD> <TID> <Command>
```

where TID is the activity's ID and Command is the UNIX command line or the SOAP-Proxy command with the input parameter already in place. The program execution client will immediately respond with an acknowledgement on the same socket. PEC TO OPERA:

```
<OpCodes.ACK_EXECCMD_OK> or <OpCodes.ACK_EXECCMD_ERROR>
```

depending on the client state:

ok the PEC was able to start the program succesfully. In this case OPERA may perform the state transition to Running for the activity.

error the PEC was not able to start the program. In this case OPERA needs to deal with the failure, possibly rescheduling the program somewhere else.

B.5.2 Execution Done

When the program has completed its execution the PEC notifies OPERA with this message:

PEC TO OPERA:

```
<OpCodes.EXECDONE> <TID> <pecdata>
```

This message contains the activity's ID, and the output data of the program, as it was previously encoded. This includes the program's exit code, and a copy of its standard output and standard error streams.

```
<pecdata> is a xml structure with 5 elements:

<pecdata>
<stdout></stdout>
<realtime></realtime>
<cputime></cputime>
<stderr></stderr>
<retval></retval>
</pecdata>
```

where retval contains a integer number:

return value = 0 program execution was sucessful

return value != 0 an error occurred while running the program

OPERA must acknowledge the message on the same socket with the answer:
OPERA TO PEC:

```
<OpCodes.ACK_EXECDONE_OK>
```

If this does not happen, there is a communication problem with OPERA (the server may be down) and the PEC should store this information, so that it can send it to the server when it recovers. If the TID of the finished program is unknown to opera, then the message it is simply ignored.

B.5.3 Abort Execution

This message is used in two cases, when the user is requesting to abort a particular program, or a process is being rolled back and its running activities need to be aborted.

OPERA TO PEC:

```
<OpCodes.EXECABORT> <TID>
```

The message contains the Activity's ID of the program to be aborted. The PEC immediately answers with a return code:

PEC TO OPERA:

```
result
```

where result is one of the following:

`<OpCodes.ACK_EXECABORT_OK>` the PEC killed the program.

`<OpCodes.ACK_EXECABORT_ERROR>` the PEC attempted to kill the program, but was not succesful.

`<OpCodes.ACK_EXECABORT_UNKNOWNID>` the PEC could not find a program with the given id, this is a serious failure: a program got lost!!.

There may be times, for administrative purposes, when it may be useful to clean the system of all running programs. For instance, when the OPERA server is started and could not perform recovery. In this case the EXECABORT message is sent with a special <TID>0.0.0.0</TID> parameter, meaning that the PEC should kill all of its running programs. The PEC answers with the same result codes.

B.5.4 Suspend Execution

This message is used to interupt a program.

OPERA TO PEC:

```
<OpCodes.EXECSUSPEND> <TID>
```

The message contains the Activity's ID of the program to be aborted. The PEC immediately answers with a return code:

PEC TO OPERA:

```
result
```

where result is one of the following:

`<OpCodes.ACK_EXECSUSPEND_OK>` the PEC was able to suspend the program.

`<OpCodes.ACK_EXECSUSPEND_ERROR>` the PEC attempted to suspend the program, but was not succesful.

<OpCodes.ACK_EXECSUSPEND_UNKNOWNID> the PEC could not find a program with the given id.

For administrative purposes, it may be useful to halt all of the running programs. In this case the EXECSUSPEND message is sent with a special <TID>0.0.0.0</TID> parameter, meaning that the PEC should suspend all of its running programs. The PEC answers with the same result codes.

B.5.5 Resume Execution

This message is used to resume a program.

OPERA TO PEC:

```
<OpCodes.EXECRESUME> <TID>
```

The message contains the Activity's ID of the program to be resumed. The PEC immediately answers with a return code:

PEC TO OPERA:

```
result
```

where result is one of the following:

<OpCodes.ACK_EXECRESUME_OK> the PEC was able to resume the program.

<OpCodes.ACK_EXECRESUME_ERROR> the PEC attempted to resume the program, but was not succesful.

<OpCodes.ACK_EXECRESUME_UNKNOWNID> the PEC could not find a program with the given id.

Similar to EXECSUSPEND, it may be useful to resume all of the running programs. In this case the EXECRESUME message is sent with a special <TID>0.0.0.0</TID> parameter, meaning that the PEC should resume all of its running programs. The PEC answers with the same result codes.

B.5.6 Execution Query and Execution Query Recover

This message is used so that OPERA can query the PEC for the execution state of an already running program:

OPERA TO PEC:

```
<OpCodes.EXECQUERY> <TID>
```

or in case of opera recovering from a crash:

```
<OpCodes.EXECQUERYRECOVER> <TID>
```

where <TID> is the activity's ID.

The immediate answer of the PEC on the same socket looks like:

PEC TO OPERA:

```
result
```

where result is one of the following:

<OpCodes.ACK_EXECQUERY_STILLRUNNING> the PEC is still waiting for the program to complete its execution.

<OpCodes.ACK_EXECQUERY_DONE> the program has terminated its execution but the PEC has not yet told OPERA, a new <EXECDONE> message is sent to OPERA.

<OpCodes.ACK_EXECQUERY_UNKNOWNID> the PEC could not find a program with the given id, this is a serious failure: a program got lost!!.

B.5.7 IsPersistent

This message signals the PEC that a program has been processed by OPERA (results are stored persistent). The client can remove the program from his active job list.

OPERA TO PEC:

```
<OpCodes.ISPERSISTENT> <TID>
```

where <TID> is the activity's ID.

B.6 System Administration Messages

B.6.1 PEC shutdown request

This message is needed for system administration purposes. It allows to remotely stop the PEC.

OPERA TO PEC:

```
<OpCodes.HOSTSTOP>
```

When the PEC receives this message it should abort all running jobs and terminate its own execution, notifying opera that the host is going down in the usual way.

B.6.2 PEC shutdown when empty request

This message is used to remotely stop the PEC when it runs out of active jobs.

OPERA TO PEC:

```
<OpCodes.HOSTSTOPWHENEMPTY>
```

When the PEC receives this message it should wait for all running jobs to finish and only then terminate its own execution, notifying opera that the host is going down in the usual way.

B.6.3 PEC information query

This message returns detailed information on the state of the PEC.

OPERA TO PEC:

```
<OpCodes.HOSTINFO>
```

The PEC answers with the following data:

```
<HOSTINFO_ACK>
   <COMMSETUP>comminfo</COMMSETUP>
   <LOADINFO>loadinfo</LOADINFO>
   <JOBINFO>
     <JOBTOTAL>number</JOBTOTAL>
     <JOBTOTALTIME>milliseconds</JOBTOTALTIME>
     <JOBLIST>
        <TID>prefix.parent.key</TID>
        <COMMAND>commandLine</COMMAND>
     </JOBLIST>
   </JOBINFO>
</HOSTINFO_ACK>
```

B.6.4 PEC information query XML

OPERA TO PEC:

```
<OpCodes.OUI_TELLSTATUSXML>
```

Remark: for the c++-programclient, this OpCode is defined in the file ClientProtocol.h (located in bioopera/server/include/operaisd/)

The program execution client will respond on the same socket with an acknowledgement and a message similar to the above using XML Syntax.

PEC TO OPERA:

```
<OpCodes.OUI_OK>
<XMLMessage>
```

The <XMLMessage> contains various state and version information, a joblist and the PEC configuration parameters.

Appendix C

Web Services Execution Subsystem for BioOpera

C.1 Introduction

The Opera system has been extended by a new subsystem which supports web service calls (using the SOAP protocol). The SOAP subsystem is equivalent to the existing UNIX (PEC) subsystem regarding system architecture and communication (it uses the same communication protocol as the PEC - the Program Execution Client Protocol. Based on this new subsystem, a wide range of new functionality (considering the fact that almost every software functionality can be published as a web service) can be integrated into BioOpera processes.

C.2 Overview

The Web Services Execution Subsystem includes these applications:

- UDDIBrowser (Graphical Client for a Universal Description, Discovery and Integration registry)
- WSDL to OCR program conversion tool (import functionality in BioOpera Visual Process Designer)
- BioOpera SOAP subsystem

The UDDIBrowser is a graphical Java tool to query a UDDI V2 registry for web services. The UDDIBrowser does automatic filtering: only those services matching the concept of WSDL (Web Services which are defined through a WSDL document) will be shown.
The Web Service Description Language (WSDL V1.1) provides a model and an XML format for describing Web services. WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as ”how” and ”where” that functionality is offered. The WSDL to OCR program conversion tool uses this information to generate a OCR program which can be exported directly into the BioOpera Graphical Process Designer.

The BioOpera SOAP subsystem handles the actual SOAP call and the input/output parameter mapping, which deals with the fact that BioOpera has no support for complex types.

Information about the technology and the defined standards can be found on the internet:

UDDI Version 2	
Official Web Site	http://www.uddi.org
Web Service Architecture	
Definition W3C Web Services Site	http://www.w3.org/TR/2002/WD-ws-arch-20021114/#whatisws http://www.w3.org/2002/ws
SOAP V1.2	
Specification Introduction	http://www.w3.org/2000/xp/Group http://developer.java.sun.com/ developer/ technicalArticles/ xml/ webservices
WSDL	
V1.1 Specification V1.2 Working Draft Introduction	http://www.w3.org/TR/wsdl http://www.w3.org/2002/ws/desc http://otn.oracle.com/tech/ webservices/ htdocs/ wsvsm/ wsdlover.html

C.3 Architecture

Figure C.1 on page 70 shows the architectural overview:

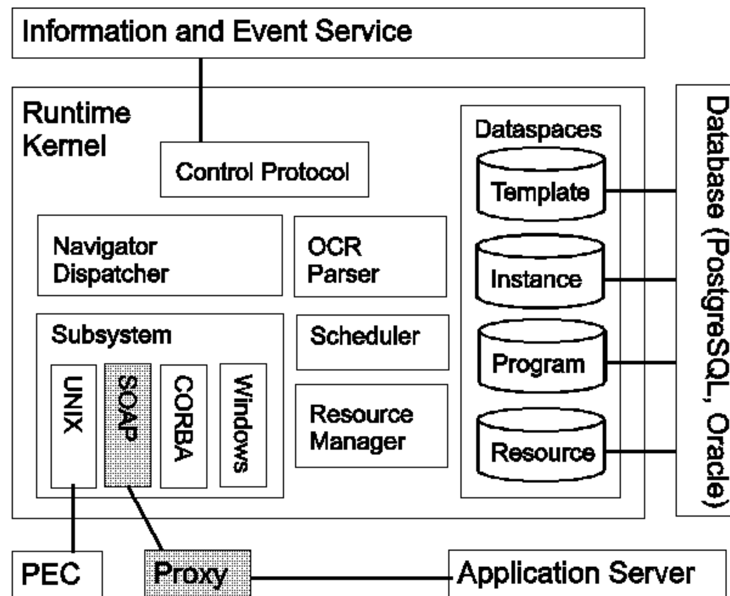


Figure C.1: Web Services Execution Subsystem Architecture

The new system components are:

SOAP subsystem Same as the UNIX subsystem except that another port for communication is used. The subsystem handles a collection of hosts running the Proxy and is responsible for job assignment and controlling (start, abort, suspend, resume, query,..).

SOAP proxy The Proxy (SOAPProxy) executes the SOAP call to a web service application server based on the information contained in the command argument(WSDL url, operation name, parameters). The SOAPProxyprovides also the funtionality to generate the xml message contained in the SOAP envelope and to collect the result parameters.

C.4 UDDIBrowser

”The Universal Description, Discovery and Integration (UDDI) protocol is one of the major building blocks required for successful Web services. UDDI creates a standard interoperable platform that enables companies and applications to quickly, easily, and dynamically find and use Web services over the Internet. UDDI also allows operational registries to be maintained for different purposes in different contexts. UDDI is a cross-industry effort driven by major platform and software providers, as well as marketplace operators and e-business leaders within the OASIS standards consortium.” [27]

The BioOpera UDDIBrowser can be used for searching web services using one of the two main public available UDDI Version 2 registries:

IBM	http://uddi.ibm.com/testregistry/inquiryapi
Microsoft	http://uddi.microsoft.com/inquire

or by using a private UDDI Registry. This is especially interesting in the context of a BioOpera UDDI registry where BioOpera processes are published as web services (see [6]).
The UDDIBrowser is programmed in Java 1.4 using Swing for the graphical user interface and Sun’s JAXR (JAXR 1.0_02 Reference Implementation) for accessing the registry. Additionally, it uses Apache’s JDOM Beta 8 for WSDL documents processing.

C.4.1 Overview

The source files for UDDIBrowser are located in $OPERAHOME/java. The following table explains where the single files can be found:

/classes	folder where you will find the java class files after running the compile script
/config	java property files (UDDIBrowser.properties and UDDIBrowserUserSettings.properties)
/lib	JAR-files
/log	DTD for displaying xml log files
/manifest	manifest UDDIBrowser.mf for building the UDDIBrowser JAR file
/script	ANT script (UDDIBrowserAntScript.xml), Windows batch file (UDDIBrowser.bat) and Linux shell script (UDDIBrowser.sh)
/src	java source and NetBeans form files (package ch.ethz.inf.bioopera.ws.uddibrowser and ch.ethz .inf.bioopera.ws.tools)

The GUI was developed using the NetBeans IDE, a lot of the GUI code was generated automatically by the IDE, so if you intend to extend the GUI classes I recommend to use NetBeans (otherwise the form files will become useless). NetBeans is an open-source project and is freely available on the website `http://www.netbeans.org` : ”The NetBeans IDE is a world-class development environment written in Java. It can be used to develop code in Java, HTML, XML, JSP, C/C++ and other languages. The IDE is modular, and there is a huge variety of commercial and free extensions to it to support various technologies.” [28]

C.4.2 Compilation Script

With the ANT script ”UDDIBrowserAntScript” located in folder java/scripts the following tasks can be performed:

Task:

- **init:** Used to set the classpath in a script variable.
- **compile:** Compiles all the necessary classes and places them in the folder java/classes. Additionally all necessary files (JAR’s, config, log,...) will be copied to the bin/java/uddibrowser folder.
- **jar:** Creates the JAR (bin/java/uddibrowser/lib/UDDIBrowser.jar) using the Manifest manifest/UDDIBrowser.mf.
- **javadoc:** Builds the JavaDoc package for the UDDIBrowser classes and places them in bin/java/doc
- **build’n’run:** Executes the tasks init, compile and jar and starts the UDDIBrowser with the default start parameters.
- **run:** Starts the UDDIBrowser with the default start parameters.
- **all:** Executes the tasks init, compile, jar, javadoc and run.
- **clean:** Deletes all compiled class-files in the classes folder and removes the built application in the bin/java folder.

C.4.3 Installation

To deploy the application copy the folder $OPERAHOME/bin/java/uddibrowser to the target directory. A Windows batch file and a Linux shell script are included in the uddibrowser folder.

To integrate the UDDIBrowser in the BioOpera Visual Process Designer follow these steps:

1. Open the BioOpera Visual Process Designer options window (tools -> options).
2. Expand "advanced" and click on UDDIBrowser.
3. In the field "Initial directory" specify the directory where you have put your UDDIBrowser installation.
4. In the field "Command to start UDDIBrowser" enter "java -jar lib\UDDIBrowser.jar".
5. Optionally specify a UDDI registry address to connect.

In order to use the operactl-script you should leave the uddibrowser folder in $OPERAHOME/bin/java/

C.4.4 Configuration

All configuration data is stored in the properties-file "UDDIBrowserUserSettings.properties" in folder "uddibrowser/config".
Changeable parameters are:

- **RegistryCount:** Number of registry entries available in the dialog choice list.
- **Registry0, Registry1, ...:** The registry entries selectable in the connect choice dialog.
- **DefaultRegistry:** The default registry to connect to.
- **ResourceGroup:** This property is useful to change the resource group name for the access method in programs generated by the OCR export method (see 74). By default. the resource group name will be "soap".

C.4.5 Running the UDDIBrowser

The UDDIBrowser can be started in 2 modes - in graphical or in batch mode. See C.5 for batch mode. In graphical mode you can search for web service interfaces using a UDDI V2 registry. The UDDIBrowser is intented for running under Java 1.4.1.
Command line start parameters:

```
java -jar UDDIBrowser.jar -import <oml file> <parameter id>
<UDDI registry url>
```

- `<oml file>:` The file (and path) where the result will be stored. If omitted, results will be written to current directory to the file temp.oml
- `<parameter id>:` Parameter numeration will start using this number. If omitted numeration will start with 0.

- `<UDDI registry url>:` UDDI Registry to connect to. If omitted, the UDDIBrowser will connect to the registry specified in properties file `UDDIBrowser UserSettings.properties`, key `DefaultRegistry`.

Working in the BioOpera Visual Process Designer, you can start the UDDIBrowser by using the menu action Edit - Import programs from UDDI Registry (only visible if the program tab is active).

By starting the UDDIBrowser with the command line arguments ”-import” the GUI will be loaded. After the window has displayed on screen, the classification are loaded which will take some seconds, until then the UDDIBrowser can't be used.

Start a search by entering a name in the corresponding field - it depends on the registry what kind of information (except for the organization name) will be found. You can use the ”%” operator as a wildcard (if the UDDI registry supports this feature). To narrow your search, add classifications to your search by clicking on the ”Add classification” button. A new window will show up with 3 root classification names. Select one and expand the node that will appear to see more classifications. It is possible to add more than one classification.

To start a search click on the ”Search” button. It will take some time until the first result are showed in the table below the button section. While the search is in progress, you can abort it by clicking on the button ”cancel search” next to the progress bar.

The tool will only show operations using the SOAP protocol, so it is possible that an interface may contain no operations!

Once there are some search results displayed, use the button ”Import” to select one (or more) of the operations or use the button ”Import all” to select all operations for import.

Finally you can return to the BioOpera Visual Process Designer by using the menu action File -> Exit or by using the button ”Return to designer”. At this point the OCR program representation will be generated and written to the file specified by the command line (usually c:\uddi.oml).

To connect to a different UDDI registry open the connect dialog (File -> Connect to a registry). Enter the address of the new UDDI registry or choose from one in the dialog list.

C.4.6 Documentation

The project source files are organized in the packages ch.ethz.inf.bioopera.ws.uddibrowser, ch.ethz.inf.bioopera.ws.uddibrowser.gui and ch.ethz.inf.bioopera.ws.tools.

- `ch.ethz.inf.bioopera.ws.uddibrowser` The classes contained in this package represent the program logic for the GUI and the import functionality. Most important class is UDDIBrowser, which contains the OCR export method. Class RegistryClient handles the UDDI Registry access, SearchWorker is a thread-class which executes a WSDL concept search. Search results are stored using class UDDIBrowserTableModel.

- `ch.ethz.inf.bioopera.ws.uddibrowser.gui` GUI classes for the main UDDIBrowser Window and dialogs windows for connecting to a UDDI-Registry, importing single operations from a Web Service interface and adding classifications to a WSDL concept search.

- `ch.ethz.inf.bioopera.ws.uddibrowser.tools` Contains the WSDL processing classes. The necessary information about a Web Service operation is stored in an instance of class WebServiceOperationDescription, class WebServiceOperationCollection manages a collection of these WebServiceOperationDescription's and provides the functions create, update, find and delete.

For more information about the API build the JavaDoc using the Ant-script which places the JavaDoc files in folder bin/java/doc.

C.5 WSDL Import

If the WSDL source address is already known, the OCR program representation can be generated using the following command:

```
java -jar UDDIBrowser.jar -convert <WSDL address> <oml file>
<parameter id>
```

- **<WSDL address>:** URL of the WSDL document. Can be a web location (`http://host/test.wsdl`) or a local file reference (`file:///c:/test.wsdl`)
- **<oml file>:** The file (and path) where the result will be stored. If omitted, results will be written to current directory to the file temp.oml
- **<parameter id>:** Parameter numeration will start using this number. If omitted numeration will start with 0.

C.5.1 OCR Program representation

The import tools will generate at least 5 programs for each valid web service operation - one for the actual web service call, one for each input/output message and one for each input/output message type. The programs are associated by their parameter names and the program name. The program names are generated by the following rules:

1. The SOAP call executing program will be named after the port and operation name in the WSDL document (concatenated by underscore). This is the basic program name, all input and output program names will start with this name. This program will always have an input parameter named xml and an output parameter named xml - expecting as input the request SOAP envelope body content; return parameter is the response SOAP envelope body content.
2. An input program named by adding "_InputCollector" and an output program named by adding "_OutputGenerator" to the basic program name - matching the xml parameter to the type parameter of the following programs.
3. For each input type (and subtype) a program will be generated und named by adding "_INPUT_" followed by the type name to the basic program name. These programs compose a xml message which will finally form the SOAP request body part.
4. For each output type (and subtype) a program will be generated und named by adding "_OUTPUT_" followed by the type name to the basic program name. These programs will decompose a xml message and assign the values to the BioOpera output parameters.

Outgoing parameter names match inbound parameter names of a subsequent program (see figure C.2), which makes it easy to figure out how parameter connections (edges) in a visual process template have to be set.
The import will not generate OCR program representations for bindings other than SOAP (e.g. HTTP Get, HTTP Post).
Example using a sample echo operation:

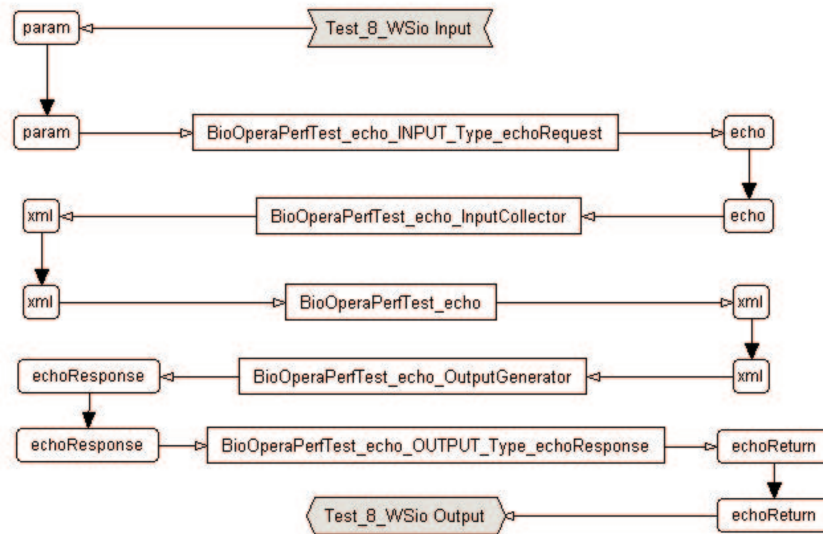


Figure C.2: Process template example

- **BioOperaPerfTest_echo:** Program for executing the SOAP call.
- **BioOperaPerfTest_echo_InputCollector:** Program for mapping the input message to the xml parameter in the SOAP executing call program.
- **BioOperaPerfTest_echo_INPUT_Type_echoRequest:** Composes the input xml message.
- **BioOperaPerfTest_echo_OutputGenerator:** Program for mapping the xml parameter of the SOAP executing call program to the output message.
- **BioOperaPerfTest_echo_OUTPUT_Type_echoResponse:** Filters the output parameter values.

C.5.2 Documentation

Please refer to C.4.6 - the WSDL import uses the same classes as the UDDIBrowser.

C.6 SOAP Proxy

The SOAPProxy, similar to the Program Execution Client, is responsible for the execution of process activities. The SOAPProxy is part of the SOAP subsytem, it uses the Program Execution Client Protocol for communication with the BioOpera server.

By default, the programs generated by the WSDL ImportTool have already the group ”soap” specified as resource. You can change that at will (see C.4.4).

C.6.1 Overview

The source files for SOAPProxy are located in $OPERAHOME/java. The following table explains where the single files can be found:

/classes	folder where you will find the java class files after running the compile script
/config	java property files (SoapProxySettings.properties and log4j.properties)
/lib	JAR-files
/log	DTD for displaying xml log files
/manifest	manifest SOAPProxy.mf for building the UDDIBrowser JAR file
/script	ANT script (SOAPProxyAntScript.xml), Windows batch file (SOAPProxy.bat) and Linux shell script (SOAPProxy.sh)
/src	java source files (package ch.ethz.inf.bioopera.ws.soapproxy and ch.ethz.inf.bioopera.ws.tools)

C.6.2 Compilation Script

With the ANT script ”SOAPProxyAntScript” located in folder java/scripts the following tasks can be performed:

Task:

- **init:** Used to set the classpath in a script variable.
- **compile:** Compiles all the necessary classes and places them in the folder java/classes. Additionally all necessary files (JAR’s, config, log,...) will be copied to the bin/java/soapproxy folder.
- **jar:** Creates the JAR (bin/java/soapproxy/lib/SOAPProxy.jar) using the Manifest manifest/SOAPProxy.mf.
- **javadoc:** Builds the JavaDoc package for the SOAPProxy classes and places them in bin/java/doc.
- **build’n’run:** Executes the tasks init, compile and jar and starts the SOAPProxy.
- **run:** Starts the SOAPProxy.
- **all:** Executes the tasks init, compile, jar, javadoc and run.
- **clean:** Deletes all compiled class-files in the classes folder and removes the built application in the bin/java folder.

C.6.3 Installation

To deploy the application copy the folder $OPERAHOME/bin/java/soapproxy to the target directory. A Windows batch file and a Linux shell script are included in the soapproxy folder. In order to use the operactl-script you should leave the soapproxy folder in $OPERAHOME/bin/java/.

C.6.4 Configuration

All configuration data is stored in the properties-file "SoapProxySettings.properties" in folder "soapproxy/config". Important properties are: Opera (BioOpera server address), capacity (number of concurrent requests) and debug (control logging of debug information).

- **OperaPort:** BioOpera server port to connect to. Default is 5682.
- **SoapProxyPort:** Listener port of the SoapProxy. Default is 4443.
- **RAM:** The amount of memory which will be shown to BioOpera. Default is 256.
- **SWAP:** The amount of swap space to show to BioOpera. Default is 512.
- **capacity:** The maximum number of concurrent requests running on the SOAPProxy. This parameter is used to compute the load. Please be aware that if the number of cpu's is for example 2, BioOpera may place up to 400 requests on the SOAPProxy. Default is 100.
- **cpu:** The number of processors to show to BioOpera. Default is 2.
- **Opera:** The initial BioOpera server address to connect to.
- **IPBasedSecurity:** Specify "yes" or "no". In case of activation, only connections using a IP address starting with 129.132 will be accepted. In future, this should be extended by the **host based authentication** mechanism.
- **debug:** Enable or disable debug mode. Allowed options are "yes" or "no". Default is "no".

C.6.5 Running the SOAPProxy

The SOAPProxy can be started through the operactl-script using the parameters "start soapxy" or directly by issuing the command "java -jar lib/SOAPProxy.jar" on the command line (in the directory $OPERAHOME/bin/java/soapproxy). You should use at least Java version 1.3.1.

In interactive mode, the SOAPProxy allows one of the following commands:

whoami	Prints information about the SOAPProxy
about	Show help information
quiet	Disables interactive mode
stop	Shuts the SOAPProxy down

The SOAPProxy starts by default in daemon mode (no output). In interactive mode,

runtime information will written to the command line.
Here is a description of all the command line start parameters:

```
java -jar lib/SOAPProxy.jar <address to bioopera server>
<bioopera server port> -i -d
```

All start parameters are optional. To change the default bioopera server address specify as first argument the hostname of the machine running the BioOpera server. Second argument must be the bioopera server port. To enable interactive mode add the parameter ”-i”. You can also set the debug mode to on by adding the parameter ”-d”.

C.6.6 Documentation

The project source files are organized in the packages ch.ethz.inf.bioopera.ws.soapproxy and ch.ethz.inf.bioopera.ws.tools.

- `ch.ethz.inf.bioopera.ws.soapproxy` The main project class is SOAPProxy: the main method contains the program loop to accept BioOpera server requests (see figure C.3 on page 80) For each server request, a thread instance of class OperaRequestHandler will be created - in case of a EXECCMD-message a new instance of class SOAPRequest holds all the necessary information to execute the command. The SOAP call execution will be handled by a thread instance of class SOAPClientWorker. Communication is done using the classes CommOpera, CommClient and CommBuffer.

- `ch.ethz.inf.bioopera.ws.uddibrowser.tools` Contains the WSDL processing classes. Additionally , some helpful functions are defined in class WSDLReaderTool for String escaping.

For more information about the API build the JavaDoc using the Ant-script which places the JavaDoc files in folder bin/java/doc.

StringArray Encoding

The BioOpera Runtime uses a encoding scheme for representing a `string[]` structure. A `string[]` is a character string where single values are seperated by a space. This scheme has been extended to allow values to contain spaces - such values will be delimited by double quotes and seperated by space from other values.

examples:

- `a b c d` = [a,b,c,d]

- `a "b c" d` = [a,b c,d]

- `"a" "b c d"` = [a,b c d]

In extension, the SOAPProxy supports the following encodings:

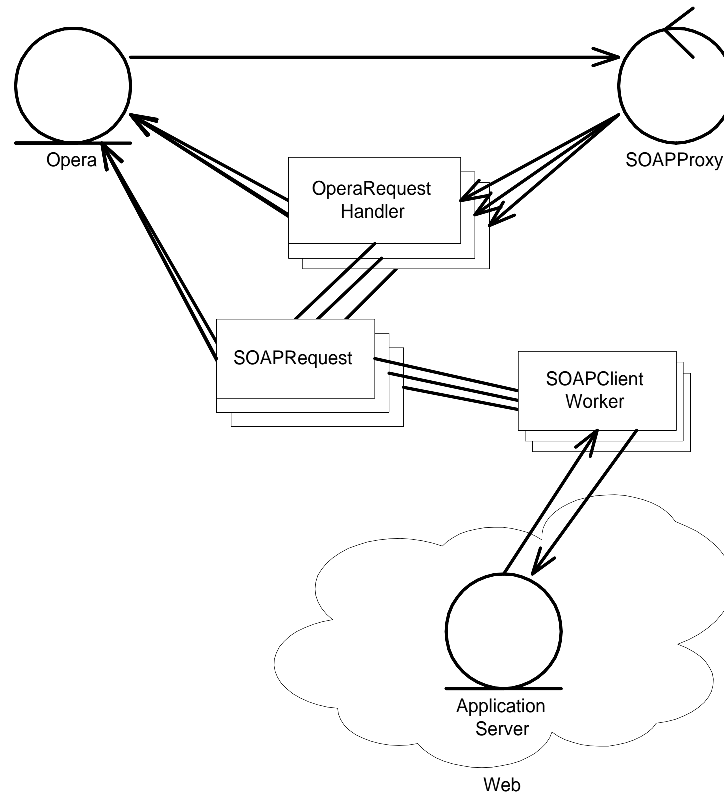


Figure C.3: SOAPProxy architecture

space	` `
ampersand &	`&`
singlequote ‘	`'`
doublequotes ”	`"`
lessthan <	`<`
greaterthan >	`>`

C.7 Library Version Listing

Most Java Archive Files come shipped with the Java Web Service Developer Pack resp. the Apache Axis Distribution.

UDDIBrowser	
commons-logging.jar	
jaxm-client.jar	
jaxr-api.jar	All shipped with Java Web Services
jaxr-ri.jar	Developer Pack 1.0_01 [29]
saaj-api.jar	
saaj-ri.jar	
castor-0.9.3.9-xml.jar	
activation.jar	
mail.jar	
dom4j.jar	
xerces.jar	Xerces Version 2.3.0
jdom.jar	JDOM Version Beta 8
SOAPProxy	
axis11.jar	
wsdl4j.jar	
jaxrpc.jar	All shipped with Apache
log4j-1.2.4.jar	Axis Version 1.1 Beta [8]
saaj.jar	
commons-discovery11.jar	
commons-logging.jar	
jdom.jar	JDOM Version Beta 8
xerces.jar	Xerces Version 2.3.0

Appendix D

Messresultate

D.1 Messungen mit einem SOAPProxy

D.1.1 Testreihe 1 Kapazität 100

Testreihe 1: Webservice ohne Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	1.023	1.632	1.820	0.000	0.000	0.000	1.023	1.632	1.820	1.023	1.632	1.820
2	2.009	2.601	3.166	1.426	1.394	1.385	1.000	1.615	2.186	3.017	3.586	4.145
4	5.719	7.316	8.367	1.235	1.591	0.832	4.104	5.350	7.423	6.808	8.952	9.319
8	0.780	1.349	5.393	0.927	0.943	2.231	0.340	0.886	2.634	2.942	3.546	8.499
16	3.314	4.908	11.362	3.532	4.027	1.470	0.338	0.888	8.975	9.506	11.890	13.852
32	0.430	1.024	17.824	0.362	0.368	10.131	0.327	0.872	1.824	2.300	2.870	34.214
64	0.571	1.414	17.998	0.811	1.009	9.955	0.323	0.869	2.465	5.386	5.953	38.497
128	0.775	1.619	36.479	1.211	1.359	21.352	0.385	1.024	2.096	7.738	9.139	66.890
256	0.493	1.259	24.976	0.566	0.742	12.444	0.322	0.878	1.459	3.436	4.765	46.493
512	0.650	1.539	28.936	0.862	1.153	14.862	0.329	0.876	1.582	7.472	9.272	55.951
1024	0.541	1.316	54.858	0.726	0.941	27.335	0.329	0.870	1.572	8.755	11.070	120.048

Testreihe 1: Webservice mit Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	1.015	1.607	6.254	0.000	0.000	0.000	1.015	1.607	6.254	1.015	1.607	6.254
2	1.669	2.291	13.215	1.872	1.937	0.700	0.345	0.921	12.720	2.992	3.661	13.710
4	0.539	1.175	23.011	0.315	0.428	1.814	0.342	0.913	20.523	1.005	1.813	24.762
8	1.044	1.800	30.168	1.144	1.126	5.654	0.340	0.922	16.508	3.329	3.898	33.463
16	0.505	1.094	57.756	0.514	0.515	9.623	0.326	0.899	24.596	2.322	2.910	64.289
32	0.362	0.957	110.411	0.118	0.125	20.216	0.326	0.889	30.804	1.005	1.578	122.930
64	0.549	1.250	213.750	0.612	0.703	46.811	0.327	0.892	26.897	2.661	3.375	244.961
128	0.625	1.591	459.245	0.679	0.932	101.068	0.421	1.100	147.043	5.722	6.965	545.433
256	0.562	1.297	792.614	0.695	0.905	228.603	0.323	0.888	18.678	6.068	6.663	1056.700
512	0.492	1.236	1651.681	0.519	0.707	462.221	0.329	0.889	12.326	2.568	4.705	2178.000
1024	0.488	1.257	3620.188	0.520	0.745	775.447	0.326	0.891	59.833	5.131	6.258	4372.640

D.1.2 Testreihe 7 Kapazität 100

Testreihe 7: Webservice ohne Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	4.680	5.538	5.852	0.000	0.000	0.000	4.680	5.538	5.852	4.680	5.538	5.852
2	5.257	6.150	6.497	0.079	0.081	0.277	5.201	6.092	6.301	5.313	6.207	6.693
4	5.663	6.589	7.458	1.211	1.241	0.851	4.130	4.997	6.465	6.817	7.810	8.486
8	5.064	8.595	10.822	2.761	2.817	1.570	0.880	4.150	8.337	8.585	12.133	13.050
16	3.621	5.614	10.433	3.652	4.419	1.814	0.143	0.610	7.826	10.338	13.074	13.973
32	2.231	3.119	16.828	3.358	3.655	5.869	0.138	0.579	5.396	10.655	12.311	27.322
64	1.247	2.031	24.884	2.117	2.463	11.291	0.138	0.585	3.682	9.350	11.403	41.206
128	0.576	1.090	42.383	1.448	1.620	21.663	0.136	0.576	2.054	8.877	9.836	67.239
256	0.411	0.915	13.412	1.045	1.225	4.515	0.138	0.572	2.121	9.249	10.787	21.358
512	0.215	0.682	1.280	0.130	0.236	0.414	0.137	0.570	0.796	0.882	4.368	4.680
1024	0.235	0.714	1.203	0.343	0.521	0.617	0.136	0.573	0.788	7.377	10.280	10.738

Testreihe 7: Webservice mit Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	5.478	14.002	17.869	0.000	0.000	0.000	5.478	14.002	17.869	5.478	14.002	17.869
2	4.440	7.073	12.971	1.007	1.042	1.428	3.728	6.336	11.961	5.152	7.810	13.980
4	1.468	4.344	19.790	2.175	2.101	1.541	0.194	2.770	17.565	4.724	7.443	21.084
8	1.079	4.182	30.839	1.500	3.009	8.509	0.155	1.276	13.419	4.667	8.952	38.347
16	0.960	5.362	72.761	1.417	2.963	11.963	0.149	2.712	42.930	5.648	13.303	81.726
32	0.450	2.949	131.289	0.784	1.609	21.965	0.139	0.827	51.071	4.650	7.994	146.764
64	0.379	3.495	262.260	0.607	2.197	48.086	0.137	0.798	88.089	4.978	9.718	293.010
128	0.349	3.039	513.994	0.167	1.683	119.594	0.140	1.131	35.772	0.935	8.874	623.297
256	1.418	5.297	871.327	3.571	7.792	227.653	0.000	0.138	78.604	21.881	43.271	1200.960
512	0.324	2.918	1856.932	0.500	2.022	466.366	0.000	0.138	64.283	9.824	15.253	2423.820
1024	0.329	3.067	4864.365	0.251	1.727	1383.741	0.137	0.739	186.859	6.212	9.077	7452.380

D.1.3 Testreihe 8 Kapazität 100

Testreihe 8: Webservice ohne Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			geschätze Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	5.220	605.539	605.892	0.000	0.000	0.000	5.220	605.539	605.892	5.220	605.539	605.892
2	6.494	606.817	607.227	0.100	0.090	0.135	6.423	606.753	607.131	6.564	606.880	607.322
4	3.578	603.795	604.706	0.652	0.648	0.246	2.818	603.093	604.509	4.199	604.474	605.024
8	7.280	607.655	611.039	3.081	2.961	2.580	1.984	602.384	608.419	11.796	611.941	615.579
16	15.573	579.751	586.009	7.845	152.417	151.871	0.000	8.631	16.603	29.694	629.842	630.194
32	4.367	586.155	598.792	5.441	106.991	107.627	0.000	0.680	9.504	18.627	621.235	626.435
64	3.054	594.190	617.472	5.286	75.004	77.628	0.403	5.233	13.164	19.861	620.431	648.251
128	15.350	396.206	423.065	15.411	297.883	293.653	0.000	0.054	5.782	65.238	675.345	714.807
256	0.580	600.750	690.920	0.884	1.049	137.819	0.393	600.511	601.233	8.041	611.676	1050.180
512	0.495	598.313	816.474	0.601	37.240	262.985	0.393	1.707	1.990	10.302	610.446	1371.480
1024	0.478	600.646	713.537	0.336	0.411	89.065	0.391	600.497	600.762	7.316	608.105	897.687

Testreihe 8: Webservice mit Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	8.085	608.442	610.785	0.000	0.000	0.000	8.085	608.442	610.785	8.085	608.442	610.785
2	5.395	605.777	611.295	1.768	1.771	1.225	4.144	604.524	610.428	6.645	607.029	612.161
4	3.135	603.410	614.958	3.903	4.006	2.005	0.749	600.881	612.000	8.955	609.362	616.301
8	1.097	601.270	624.342	1.668	1.736	6.896	0.430	600.577	611.641	5.223	605.564	630.330
16	0.926	601.469	653.281	1.254	1.232	16.331	0.417	600.643	609.674	5.357	605.713	665.896
32	0.647	600.813	707.654	0.902	0.950	30.777	0.404	600.539	609.883	5.538	605.971	733.568
64	4.670	597.232	803.208	12.166	75.171	114.625	0.000	13.428	57.398	66.190	668.183	868.361
128	0.457	600.934	980.813	0.408	0.794	130.077	0.386	600.536	609.311	5.015	605.369	1103.890
256	1.338	564.291	1348.575	5.552	146.111	309.514	0.000	0.047	422.464	40.663	649.426	1717.850
512	0.453	598.268	2967.637	0.218	0.263	507.887	0.394	600.517	1607.510	5.123	605.488	4074.250
1024	0.538	600.774	5214.862	0.590	0.893	1415.007	0.344	600.506	952.370	7.278	611.445	7722.420

D.1.4 Testreihe 1 Kapazität 200

Testreihe 1: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			geschätze Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	5.009	5.824	6.175	0.000	0.000	0.000	5.009	5.824	6.175	5.009	5.824	6.175
2	5.589	6.489	6.867	0.125	0.088	0.180	5.500	6.427	6.740	5.677	6.551	6.994
4	5.290	6.210	7.168	1.253	1.224	0.602	3.428	4.458	6.526	6.121	7.310	7.858
8	3.492	5.285	7.728	2.680	3.038	1.644	0.355	1.199	5.903	7.749	9.866	10.548
16	2.417	3.683	10.097	2.623	2.938	2.050	0.335	0.898	7.828	7.519	9.213	14.643
32	1.638	2.522	16.462	2.363	2.727	6.859	0.329	0.883	6.051	8.103	9.740	28.251
64	1.059	2.091	29.199	1.840	2.225	16.046	0.328	0.869	2.096	8.244	10.243	54.978
128	0.825	1.659	83.878	1.153	1.523	47.181	0.373	0.981	8.120	7.465	10.476	164.432
256	0.680	1.531	126.408	0.987	1.265	71.220	0.374	0.974	7.126	7.732	10.211	247.916
512	0.612	1.384	247.857	0.882	1.048	154.292	0.326	0.870	3.424	8.339	9.818	489.485
1024	0.654	1.496	727.659	0.686	0.866	442.098	0.422	1.110	8.494	7.735	9.594	1469.140

Testreihe 1: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			geschätze Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	6.526	7.582	11.018	0.000	0.000	0.000	6.526	7.582	11.018	6.526	7.582	11.018
2	5.379	7.672	12.709	0.701	1.251	1.345	4.883	6.787	11.758	5.875	8.556	13.660
4	1.931	2.736	16.933	3.160	3.212	0.717	0.343	0.936	16.051	6.670	7.549	17.779
8	1.319	1.939	33.003	2.735	2.830	2.601	0.347	0.923	28.488	8.088	8.943	35.189
16	1.019	1.898	66.201	1.734	1.781	7.692	0.334	0.914	39.672	7.022	7.880	71.295
32	0.561	1.273	117.132	0.896	0.980	19.572	0.330	0.913	44.897	5.078	5.925	129.612
64	0.682	1.402	236.983	1.131	1.225	34.247	0.330	0.898	95.107	7.966	8.823	262.163
128	0.763	1.601	452.866	0.938	1.137	51.237	0.375	0.997	143.388	5.340	8.914	496.808
256	0.519	1.425	841.335	0.531	0.766	201.653	0.377	1.016	92.514	5.423	6.617	1007.910
512	0.509	1.261	1722.365	0.576	0.798	330.584	0.327	0.887	202.714	5.190	7.384	2079.040
1024	0.701	1.597	5522.338	2.271	3.004	806.745	0.303	0.788	816.260	45.650	46.381	6759.130

D.1.5 Testreihe 7 Kapazität 200

Testreihe 7: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	6.174	7.066	7.425	0.000	0.000	0.000	6.174	7.066	7.425	6.174	7.066	7.425
2	5.815	6.667	7.080	0.124	0.144	0.161	5.727	6.565	6.966	5.903	6.768	7.194
4	5.677	6.578	7.588	1.178	1.240	0.614	4.174	5.018	7.006	6.749	7.711	8.354
8	5.268	6.431	9.161	2.760	2.676	0.780	1.229	2.736	8.248	8.649	9.887	10.493
16	3.879	5.338	11.944	3.885	4.097	1.386	0.142	0.587	9.715	10.373	11.820	14.670
32	1.491	2.198	14.751	2.553	2.887	5.497	0.138	0.580	2.209	8.662	10.233	22.787
64	1.015	1.910	26.090	1.985	2.643	13.995	0.138	0.582	2.132	8.694	10.275	46.726
128	0.253	0.755	105.471	0.187	0.336	62.892	0.136	0.588	1.513	1.451	3.361	213.893
256	0.446	0.948	112.362	1.172	1.362	62.004	0.135	0.566	8.635	9.577	10.966	219.361
512	0.242	0.701	184.148	0.157	0.175	116.922	0.136	0.567	1.500	1.118	1.658	364.053
1024	0.436	1.041	497.962	0.538	0.683	289.488	0.281	0.796	4.273	9.000	10.421	995.654

Testreihe 7: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	5.062	8.619	12.287	0.000	0.000	0.000	5.062	8.619	12.287	5.062	8.619	12.287
2	6.052	9.426	15.774	0.112	0.104	1.247	5.972	9.352	14.892	6.131	9.499	16.656
4	4.261	7.832	20.225	2.632	2.603	1.565	1.298	5.440	17.975	6.631	10.752	21.522
8	3.990	9.317	37.803	6.686	7.539	2.822	0.158	4.238	34.591	16.096	23.191	42.406
16	0.825	3.529	66.579	1.466	2.083	9.466	0.146	2.145	43.369	5.238	8.299	74.537
32	0.450	3.757	140.632	0.798	2.591	21.217	0.138	2.112	81.808	4.718	12.906	157.294
64	0.412	3.343	274.363	0.610	2.024	48.141	0.140	1.127	95.056	4.826	12.591	304.902
128	0.332	3.209	516.898	0.419	1.933	125.637	0.137	0.788	97.070	4.729	8.542	626.400
256	0.344	2.961	963.857	0.345	1.623	252.478	0.137	1.032	99.301	5.236	12.060	1291.880
512	0.305	3.112	2351.252	0.543	1.964	767.314	0.136	1.034	286.309	12.081	20.125	3439.520
1024	0.391	3.128	4429.112	0.266	1.526	614.662	0.160	0.881	836.110	5.458	9.554	5074.460

D.1.6 Testreihe 8 Kapazität 200

Testreihe 8: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	4.086	604.990	605.436	0.000	0.000	0.000	4.086	604.990	605.436	4.086	604.990	605.436
2	2.944	603.613	603.581	0.269	0.092	0.167	2.754	603.548	603.463	3.134	603.678	603.699
4	0.772	601.337	602.825	0.419	0.527	0.950	0.401	600.734	601.875	1.192	602.006	604.136
8	0.453	601.032	605.147	0.398	0.509	2.154	0.160	600.503	602.812	1.182	601.875	608.638
16	1.209	602.240	609.170	0.824	1.184	3.634	0.420	601.105	604.821	3.034	604.679	615.395
32	0.293	600.601	616.088	0.229	0.554	9.164	0.141	600.244	601.017	1.016	603.375	630.901
64	0.194	600.290	631.165	0.152	0.195	19.546	0.080	600.133	600.835	0.691	601.134	664.142
128	0.875	601.126	664.959	1.224	1.458	37.556	0.396	600.518	603.793	9.937	610.063	731.137
256	0.729	598.600	733.934	1.768	37.173	92.044	0.000	6.879	7.113	17.374	617.544	878.911
512	0.562	599.553	1117.086	1.109	26.245	418.917	0.000	7.559	7.815	13.878	614.710	1820.240
1024	0.301	600.333	2023.922	1.153	1.229	955.457	0.031	600.046	606.459	15.360	616.234	3643.820

Testreihe 8: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	2.183	602.987	607.251	0.000	0.000	0.000	2.183	602.987	607.251	2.183	602.987	607.251
2	3.260	603.758	609.906	0.276	0.212	0.295	3.065	603.608	609.697	3.455	603.908	610.114
4	0.581	601.272	611.697	0.347	0.493	2.775	0.401	600.744	608.126	1.102	601.936	613.949
8	0.749	600.945	622.663	0.623	0.686	7.995	0.416	600.548	606.451	2.249	602.557	629.306
16	0.751	601.173	652.434	1.960	2.549	14.926	0.177	600.373	622.131	8.084	610.697	665.349
32	0.179	600.291	714.005	0.129	0.138	37.563	0.070	600.143	612.677	0.521	600.644	745.532
64	0.595	602.243	830.605	0.669	2.345	63.739	0.410	600.587	609.125	5.315	613.742	875.907
128	0.552	600.718	1051.776	0.471	0.521	123.979	0.406	600.542	616.868	5.116	605.913	1144.930
256	0.430	600.600	1483.333	0.067	0.211	255.385	0.394	600.528	978.456	1.362	603.164	1718.370
512	0.512	598.906	2627.968	0.663	1.187	720.130	0.394	600.455	1203.860	11.480	611.910	3683.020
1024	0.369	600.398	8749.480	0.359	0.405	1110.405	0.031	600.062	3935.760	11.187	612.593	10826.100

D.1.7 Testreihe MIX Kapazität 100 und 200

Testreihe MIX: Webservice ohne Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	2.907	79.681	82.204	2.024	212.922	212.124	0.959	1.821	4.733	6.270	606.598	607.127
16	1.924	77.875	84.309	2.066	205.298	204.691	0.348	0.608	2.082	6.675	606.984	609.638
32	1.046	77.176	89.694	1.697	201.636	200.896	0.000	0.594	2.427	6.826	607.220	619.696
64	0.767	76.760	94.050	1.270	199.894	199.840	0.000	0.590	3.271	7.274	607.598	629.648
128	0.716	76.584	123.633	1.128	199.083	197.805	0.142	0.588	1.875	7.932	608.832	687.722
256	0.482	76.237	122.184	0.487	198.644	202.086	0.139	0.561	1.218	5.998	606.350	689.827
512	0.533	76.289	171.544	0.646	198.540	209.890	0.134	0.554	1.954	7.402	607.760	850.442

Testreihe MIX: Webservice mit Input/Output Operationen CPU 2 Capacity 100

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	1.589	77.510	114.555	1.447	213.335	202.137	0.765	1.154	36.357	5.102	605.483	614.702
16	1.298	77.278	154.175	2.091	206.189	188.257	0.342	0.601	58.202	8.938	610.077	650.367
32	0.744	76.958	226.047	1.256	201.571	180.185	0.000	0.624	64.110	7.269	602.641	722.102
64	0.624	76.581	365.742	0.815	199.882	172.274	0.000	0.595	29.847	5.471	605.799	845.249
128	0.542	76.373	711.436	0.572	199.026	194.446	0.217	0.575	55.491	5.058	605.399	1165.210
256	0.484	76.330	1229.345	0.400	198.688	413.824	0.178	0.600	103.263	5.009	606.359	2119.860
512	0.449	76.275	2573.867	0.299	198.414	729.225	0.139	0.574	66.580	5.165	605.511	3998.170

Testreihe MIX: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	3.573	79.685	81.659	2.305	213.143	212.524	1.087	1.629	4.461	6.861	607.162	607.620
16	1.859	77.982	83.778	2.094	205.278	204.363	0.347	0.692	5.195	6.687	607.032	607.583
32	0.890	76.908	85.745	1.208	201.620	199.861	0.341	0.605	1.953	5.682	606.038	610.916
64	0.684	76.525	96.240	0.933	200.178	198.904	0.178	0.648	3.285	5.985	607.004	635.333
128	0.660	76.456	106.258	0.864	199.009	199.447	0.178	0.554	1.214	5.927	606.275	654.048
256	0.536	76.342	135.649	0.658	198.613	203.206	0.139	0.555	2.354	6.553	606.883	713.551
512	0.488	76.308	124.815	0.456	198.413	200.839	0.135	0.555	1.615	6.241	606.568	690.123

Testreihe MIX: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	1.572	77.443	113.354	1.413	213.293	202.283	0.744	1.113	32.689	4.980	605.312	613.874
16	1.630	77.528	156.102	2.165	205.710	188.638	0.345	0.627	63.974	7.274	608.344	652.566
32	0.797	76.647	238.846	0.930	201.668	173.962	0.340	0.651	100.360	5.074	605.416	726.940
64	0.597	76.436	393.584	0.653	199.920	158.283	0.338	0.624	91.882	5.049	605.355	854.303
128	0.525	76.283	703.218	0.511	199.064	195.342	0.146	0.589	112.787	5.147	605.500	1156.410
256	0.469	76.299	1218.884	0.360	198.615	412.626	0.141	0.576	66.527	4.979	605.302	2101.020
512	0.481	76.325	2618.135	0.478	198.392	725.602	0.141	0.569	65.902	8.019	608.912	4066.960

D.2 Messungen mit 4 SOAPProxy's

D.2.1 Testreihe 8 Kapazität 200

Testreihe 8: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	8.109	608.450	608.892	0.000	0.000	0.000	8.109	608.450	608.892	8.109	608.450	608.892
2	5.092	605.752	607.778	0.713	0.383	1.774	4.587	605.481	606.523	5.596	606.023	609.032
4	8.530	609.055	609.574	1.306	1.367	1.491	6.985	607.306	607.640	10.092	610.395	610.805
8	12.270	613.039	613.277	3.474	3.309	3.415	6.420	607.043	607.582	17.215	618.029	618.581
16	24.831	551.080	552.304	12.293	211.140	211.155	0.000	3.716	4.915	59.696	659.806	660.612
32	6.683	589.150	613.604	3.883	106.446	133.523	0.441	6.195	20.717	15.574	615.884	1060.490
64	5.332	578.269	593.275	6.435	127.913	129.527	0.000	0.425	12.898	22.983	624.177	629.342
128	5.729	574.106	606.203	6.974	137.813	153.804	0.000	0.449	6.422	52.187	652.351	1113.670
256	4.949	552.564	585.006	4.664	173.518	181.657	0.000	0.176	12.118	27.269	648.342	1075.570
512	31.522	631.038	635.100	103.027	121.665	123.006	0.000	2.753	3.455	432.251	1035.790	1036.290
1024	19.630	544.430	559.703	52.755	246.629	248.752	0.000	0.093	0.719	504.055	1273.340	1274.040

Testreihe 8: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
1	4.519	604.992	606.946	0.000	0.000	0.000	4.519	604.992	606.946	4.519	604.992	606.946
2	10.105	610.457	612.574	2.167	2.187	1.827	8.572	608.910	611.282	11.637	612.003	613.866
4	7.462	608.666	613.322	2.390	3.252	1.679	4.844	605.060	611.076	10.331	611.790	614.919
8	7.497	608.011	612.861	3.414	3.382	3.087	2.399	603.396	608.750	11.687	612.592	617.472
16	5.975	607.019	617.277	3.400	3.588	3.362	1.369	601.507	612.096	14.678	615.161	625.060
32	2.538	575.344	619.913	2.586	116.895	148.976	0.000	0.460	21.496	10.265	610.395	1192.950
64	4.796	605.622	630.618	5.827	6.200	9.097	0.408	600.535	609.869	25.939	629.379	651.247
128	1.904	602.661	656.006	2.418	2.570	51.952	0.400	600.617	608.162	13.810	614.345	1044.730
256	2.140	600.382	715.687	2.840	37.604	76.872	0.401	3.196	142.745	21.751	622.678	1122.830
512	1.709	600.766	981.813	7.577	12.740	63.667	0.150	600.174	855.747	69.390	699.608	1156.450
1024	2.364	602.937	2034.117	20.646	10.169	215.864	0.141	600.160	1346.140	627.633	709.981	2209.910

D.2.2 Testreihe MIX Kapazität 200

Testreihe MIX: Webservice mit Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	4.456	80.584	93.379	1.402	212.236	211.085	2.075	2.754	16.192	6.062	605.828	615.753
16	1.687	77.838	103.000	1.599	205.093	200.385	0.000	0.926	20.370	5.241	605.422	617.985
32	1.726	78.252	118.125	2.185	201.785	194.626	0.000	0.667	29.692	8.704	608.229	632.840
64	1.516	77.478	164.916	1.971	200.153	185.917	0.078	0.657	30.621	8.924	609.914	680.038
128	1.050	77.203	217.054	1.610	199.071	182.069	0.062	0.530	29.753	10.508	606.433	724.141
256	0.776	76.854	314.846	1.056	198.686	187.231	0.000	0.137	22.924	7.645	608.891	829.062
512	0.717	76.764	595.901	0.958	198.622	277.877	0.000	0.538	30.952	8.224	607.621	1091.270

Testreihe MIX: Webservice ohne Input/Output Operationen CPU 2 Capacity 200

	Mittelwert (empirisch)			Standardabweichung			MinimalWert			Maximalwert		
# Instanzen	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL	SETUP	REAL	WALL
8	3.241	79.940	81.171	1.586	211.748	211.550	1.422	2.145	2.822	5.643	603.938	604.651
16	2.471	78.484	79.913	1.936	205.753	205.382	0.464	1.181	2.213	6.354	606.768	607.103
32	1.588	77.828	79.207	1.446	201.643	201.356	0.000	0.764	1.267	5.129	604.060	604.793
64	1.242	77.166	80.527	1.246	199.948	199.272	0.000	0.575	1.286	5.406	604.219	608.048
128	0.754	76.529	78.504	1.033	199.157	198.640	0.320	0.640	1.353	7.515	606.244	606.833
256	0.616	76.488	78.219	0.735	198.731	198.254	0.310	0.594	1.427	7.375	609.315	610.059
512	0.593	76.533	78.511	0.646	198.454	198.055	0.000	0.531	1.409	5.246	605.294	610.480

Appendix E

XML Schema für SOAP Subsystem XML-Kommando

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="OPERAWEBSERVICEREQUEST">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="WSDLLOCATION"/>
				<xs:element ref="SERVICE"/>
				<xs:element ref="PORT"/>
				<xs:element ref="OPERATION"/>
				<xs:element ref="XMLMESSAGE"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="XMLCOLLECTOR">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="WSDLLOCATION"/>
				<xs:element ref="SERVICE"/>
				<xs:element ref="PORT"/>
				<xs:element ref="OPERATION"/>
				<xs:element ref="PARAMETERS"/>
				<xs:element ref="OUTPARAM"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="XMLDIVERTER">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="XMLMESSAGE"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="NAME">
		<xs:simpleType>
			<xs:restriction base="xs:string">
```

```
                                <xs:enumeration value="echo"/>
                                <xs:enumeration value="xml"/>
                        </xs:restriction>
                </xs:simpleType>
        </xs:element>
        <xs:element name="NAMESPACE">
                <xs:complexType/>
        </xs:element>
        <xs:element name="OPERATION" type="xs:string"/>
        <xs:element name="OUTPARAM">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="NAME"/>
                                <xs:element ref="NAMESPACE"/>
                                <xs:element ref="TARGETNAMESPACE"/>
                                <xs:element ref="TYPE"/>
                                <xs:element ref="VALUE"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="PARAM">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="NAME"/>
                                <xs:element ref="NAMESPACE"/>
                                <xs:element ref="TARGETNAMESPACE"/>
                                <xs:element ref="TYPE"/>
                                <xs:element ref="VALUE"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="PARAMETERS">
                <xs:complexType>
                        <xs:sequence minOccurs="0" maxOccurs="unbounded">
                                <xs:element ref="PARAM"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="PORT" type="xs:string"/>
        <xs:element name="SERVICE" type="xs:string"/>
        <xs:element name="TARGETNAMESPACE">
                <xs:complexType/>
        </xs:element>
        <xs:element name="TYPE" type="xs:string"/>
        <xs:element name="VALUE" type="xs:string"/>
        <xs:element name="WSDLLOCATION" type="xs:anyURI"/>
        <xs:element name="XMLMESSAGE" type="xs:string"/>
</xs:schema>
```

Appendix F

Sourcecode Axis Test Web Service

```
import org.apache.axis.MessageContext;
import org.apache.axis.transport.http.HTTPConstants;

import javax.servlet.http.HttpServletRequest;
import java.util.Enumeration;
import java.util.ArrayList;
import java.util.Iterator;

/**
 * Test class for performance testing
 */
public class BioOperaPerfTest {

    /**
     * demo message context stuff
     * @return list of request headers
     */
    public String[] list() {
        HttpServletRequest request = getRequest();
        Enumeration headers=request.getHeaderNames();
        ArrayList list=new ArrayList();
        while (headers.hasMoreElements()) {
            String h = (String) headers.nextElement();
            String header=h+':'+request.getHeader(h);
            list.add(header);
        }
        String[] results=new String[list.size()];
        for(int i=0;i<list.size();i++) {
            results[i]=(String) list.get(i);
        }
        return results;
    }

    /**
```

```
     * get the caller; may involve reverse DNS
     * @return
     */
    public String whoami() {
        HttpServletRequest request = getRequest();
        String remote=request.getRemoteHost();
        return "Hello caller from "+remote;
    }

    /**
     * very simple method to echo the param.
     * @param param
     * @return
     */
    public String echo(String param) {
     System.out.println("Thread " + Thread.currentThread().getName()
     + " going asleep@ "
     + new java.util.Date().toString());
     try {
     Thread.currentThread().sleep(600000);
     }
     catch (InterruptedException e)
     {
     System.out.println("Thread " + Thread.currentThread().getName()
     + " interupted@ "
     + new java.util.Date().toString());
}

     System.out.println("Thread " + Thread.currentThread().getName()
     + " woken up@ "
     + new java.util.Date().toString());

        return param + "\n" + param + "\n" + "\n" + param;
    }

    /**
     * helper
     * @return
     */
    private HttpServletRequest getRequest() {
        MessageContext context = MessageContext.getCurrentContext();
        HttpServletRequest req = (HttpServletRequest)
        context.getProperty(HTTPConstants.MC_HTTP_SERVLETREQUEST);
        return req;
    }

}
```

Bibliography

[1] Win Bausch, Cesare Pautasso, Martin Grüter, Christian Rupp, and Markus Haller. *The BioOpera Manual — Process Support for Bioinformatics*. ETH Zürich, Institut für Informations- und Kommunikationssysteme, Zürich, Switzerland, 2003.

[2] Michael Champion, Chris Ferris, Eric Newcomer, and David Orchard. Web Services Architecture. http://www.w3.org/TR/ws-arch/.

[3] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol 1.1 Specification. http://www.w3.org/TR/SOAP.

[4] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[5] Dave Ehnebuske, Barbara McKee, Dan Rogers, and Claus von Riegen. UDDI Version 2 Specifications. http://www.oasis-open.org/committees/uddi-spec/tcspecs.shtml#uddiv2.

[6] Christian Rupp. Building an Application Server for BioOpera. Diplomarbeit, ETH Zürich, Institut für Informations- und Kommunikationssysteme, 2003.

[7] C. M. Sperberg-McQueen and Henry Thompson. XML Schema, April 2000. http://www.w3.org/XML/Schema.

[8] *Apache Axis 1.1*. http://ws.apache.org/axis.

[9] Nilo Mitra. SOAP Version 1.2 Part 0: Primer, 2002. http://www.w3.org/TR/2002/CR-soap12-part0-20021219/.

[10] Web Services Overview, June 2002. http://otn.oracle.com/tech/webservices/ htdocs/wsvsm/wsdlover.html.

[11] William A. Nagy, Franciso Curbera, and Sanjiva Weerawaranna. The Advertisement and Discovery of Services (ADS) protocol for Web services. hhttp://www-106.ibm.com/developerworks/library/ws-ads.html.

[12] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web Services Inspection Language (WS-Inspection) 1.0. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-inspection.asp.

[13] David Ehnebuske, Dan Rogers, and Claus von Riegen. UDDI Version 2 Data Structure. http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm.

[14] David Ehnebuske, Dan Rogers, and Claus von Riegen. UDDI Version 2 Data Structure. http://uddi.org/schema/uddi_v2.xsd.

[15] Bilal Siddiqui. Using UDDI as a Search Engine. http://www. webservicesarchitect.com/content/articles/siddiqui01.asp.

[16] Liang-Jie Zhang and Qun Zhou. Aggregate UDDI searches with Business Explorer for Web services. http://www-106.ibm.com/developerworks/library/ws-be4ws/.

[17] Toufic Boubez and Luc Clément. UDDI tModels: Classification Schemes, Taxonomies, Identifier Systems, and Relationships, Version 2.04. http://uddi.org/taxonomies/UDDI_Taxonomy_tModels.htm.

[18] ebXML. http://www.ebxml.org/.

[19] Kim Haase. The JavaTM Web Services Tutorial: Java API for XML Registries. http://java.sun.com/webservices/downloads/webservicestutorial.html.

[20] Java API for XML Registries (JAXR). http://java.sun.com/xml/jaxr.

[21] CSV-Format. http://www.vip7.de/info/develop/csv-format.htm.

[22] Axis User's Guide. http://ws.apache.org/axis.

[23] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. http://www.w3.org/TR/REC-xml.

[24] Jean-Jacques Moreau and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 1.2: Bindings. http://www.w3.org/TR/wsdl12-bindings/.

[25] The NetBeans Project. http://www.netbeans.org.

[26] Rahul Sharma et al. Java API for XML-based RPC - JAX-RPC 1.0 Specification. http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec1.

[27] Uddi homepage. http://www.uddi.org/about.html.

[28] The netbeans project. http://www.netbeans.org.

[29] *Java Web Services Developer Pack 1.0*. http://java.sun.com/webservices/webservicespack.html.