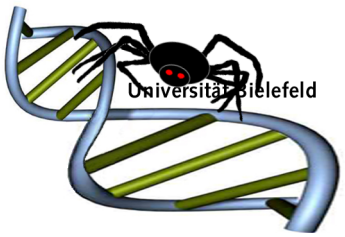

BiBiWS

Ein Framework für die Entwicklung asynchroner WebServices auf dem BiBiServ

Diplomarbeit von

Henning Mersch hmersch@techfak.uni-bielefeld.de

Betreut von
Robert Giegerich und Jan Krüger

15. Oktober 2004

Ein Dankeschön an

Robert Giegerich für das Ermöglichen der Diplomarbeit, der Teilnahme am HOBIT-Projekt und das Engagement.

Jan Krüger für die erstklassige Betreuung und Zusammenarbeit während der Diplomarbeit.

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Quellen verwendet habe.

Bielefeld, der 15. Oktober 2004,

(Henning Mersch)

Inhaltsverzeichnis

1 Motivation und Ziele

WebService ist die aktuelle Möglichkeit, um nicht nur in der Bioinformatik die Kommunikation zwischen Computern zu automatisieren.
Im Gegensatz zu anderen Ansätzen wie CORBA[CORBA], XMLRPC [XMLRPC] und JRMI [JRMI], ist einer der Hauptvorteile von WebServices die standardisierte Schnittstellendefinition, wodurch ein anderes Programm automatisiert auf diesen WebService zugreifen kann.
Aufgrund der großen Vielfalt von unterstützten Programmiersprachen ensteht so der problemlose, standardisierte Austausch von Daten. Dadurch ist eine Vernetzung der Programme über Server-, Hardware- und Softwaregrenzen hinweg möglich (siehe Kapitel 2.2).
Ferner existieren Ansätze, um WebServices automatisiert aufzufinden. Dafür werden die WebServices auf speziellen Servern katalogisiert, der anfragenden Programmen Auskunft erteilt (siehe Kapitel 2.7).

1.1 Der BiBiServ

Der *Bielefeld University Bioinformatics Server* (kurz: BiBiServ) [BIBISERV] ist ein im Rahmen des CeBiTec [CEBITEC] von der DFG [DFG] gefördertes Projekt und stellt Benutzern aus aller Welt zur Zeit mehr als 20 bioinformatische Anwendungen zur Verfügung.
Der BiBiServ soll die einfache Benutzung solcher bioinformatischer Programme ermöglichen, ohne daß der Benutzer diese lokal (und gegebenenfalls mit erheblichem Arbeitsaufwand) auf seinem eigenen Rechner installieren muß.

Das bisherige Konzept des BiBiServ verwendet HTML-Formulare als Eingabe, und gibt die Ergebnisse der Anwendung als HTML-Dokument oder Datei zurück, wie sie auch bei einer lokalen Installation entstehen würden.
Dafür füllt der Benutzer das HTML-Formular der Anwendung mit seinen Daten aus. Auf dem BiBiServ wird dadurch (lokal) die Anwendung mit den entsprechenden Daten und Parametern gestartet und benachrichtigt den Benutzer entweder innerhalb von fünf Minuten durch eine HTML Seite, oder der Benutzer bekommt eine E-Mail, falls der Benutzer diese übermittelt hat.

Bei dem bisherigen Konzept sind mittlerweile einige Nachteile aufgetreten, die durch den Einsatz von dem hier vorgestellten BiBiWS behoben werden.
Zum Einen werden sämtliche Anwendungen lokal auf dem Rechner *BiBiServ* gestartet, so daß dieser durch den stark gewachsenen Andrang von Benutzern überlastet ist

(siehe hierzu die Zugriffsstatistiken des BiBiServ: [BIBISERVSTATS]).
Um dieses Problem zu beheben, bietet BiBiWS die Möglichkeit, verschiedene Teile eines WebService Aufrufes über ein Computing Grid (*Sun Grid Engine*) zu verteilen. Dadurch erhält der BiBiServ Zugriff auf mehr Rechenleistung.
Zum Anderen bietet ein HTML Formular zwar eine benutzerfreundliche Möglichkeit des Zugriffs auf die Anwendungen, jedoch ist ein automatisierter Zugriff nahezu unmöglich, zumindest aber technisch schwierig und zeitaufwendig.
Gerade diesem Problem will auch das HOBIT-Projekt für verschiedene Bereiche der Bioinformatik mittels der WebService Technologie Abhilfe schaffen, indem standardisierte Ein- und Ausgaben der Programme entwickelt werden. Zudem ergibt sich hieraus noch die Möglichkeit, die WebServices automatisiert zu verschalten.
Auf Grund dieser gemeinsamen Interessen nimmt der BiBiServ an dem HOBIT-Projekt teil.

1.2 Das HOBIT-Projekt

Das *Helmholtz Open Bioinformatic Technologie* Projekt (kurz: HOBIT-Projekt) [HOBIT] ist vom Helmholtz Institute [HELMHOLTZ] im Jahr 2003 als Kooperation von 12 verschiedenen Instituten und Universitäten gegründet worden.

Im Rahmen des HOBIT-Projektes sollen bioinformatische Anwendungen als WebService bereitgestellt und vernetzt werden, so daß sich alleine aus der einfachen Kombination vorhandener Anwendungen neue Anwendungen und Ergebnisse erzielen lassen.
Ohne WebServices müssen derzeit die Ausgaben einer Anwendung mühsam per Hand in das jeweilige Eingabeformat einer anderen Anwendung überführt werden.
Diverse Projekte (z.B. HUPO-PSI [PSI], [HERM]) beschäftigen sich zur Zeit mit der Entwicklung von Standards, um biologische Informationen als XMLSchema zu repräsentieren. Daten in diesen universelllen Formaten können dann zum Austausch verwendet werden, so daß die Überführung entfällt und eine automatische Verschaltung von den Anwednungen mit minimalem Aufwand möglich wird.
Somit wird das Ziel der Vernetzung innerhalb des HOBIT-Projektes und dadrüber hinaus erreicht. Selbst wenn verschiedene Standards verwendet werden, können diese leichter als bisher durch XSLT-Skripte konvertiert werden.

BiBiWS ist die Grundlage für die Entwicklung von WebServices auf dem BiBiServ, wodurch letztlich das HOBIT-Projekt unterstützt wird.

1.3 Ziele

Aufgabe dieser Diplomarbeit ist es, ein Framework für den BiBiServ zu entwickeln, das es den Entwicklern von bioinformatischen Anwendungen ermöglicht, diese als

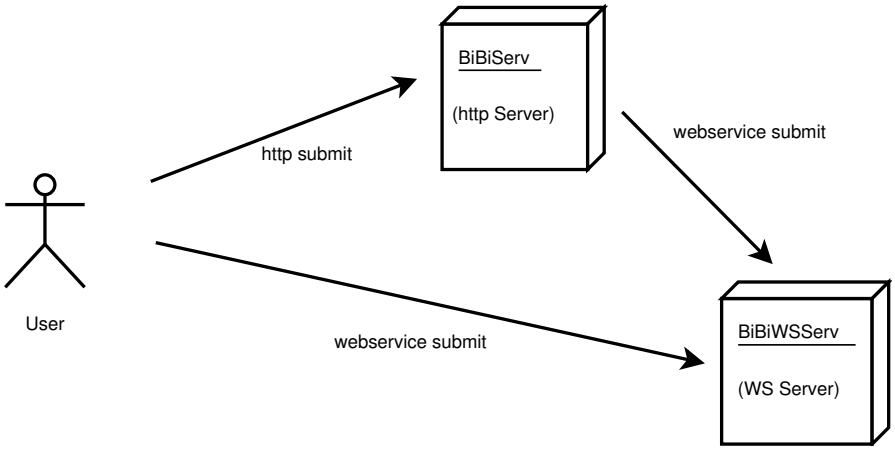

Abbildung 1.1: Zugriff auf die WebServices des BiBiServ

WebService anzubieten. Außerdem sollen Anwendungen über ein Computing Grid (*Sun Grid Engine*) verteilt werden, so daß der eigentliche Rechner *BiBiServ* entlastet wird.
Durch diese zusätzliche Abstraktionsebene, die BiBiWS darstellt, soll dem Entwickler möglichst viel Arbeit abgenommen werden und nur ein minimales Wissen abverlangt werden, so daß die Einarbeitungszeit gering gehalten wird.
Das HTML Interface des BiBiServ soll auch nach der Umstellung auf BiBiWS erhalten bleiben. Im Hintergrund wird der jetzige BiBiServ dann als WebService Client agieren. Damit unterscheidet er sich nicht von anderen Benutzern, die auf die WebServices direkt zugreifen (siehe Abbildung 1.1).
Verwendet wird als Programmiersprache für die zu erstellende Clientseite (den bisherigen BiBiServ) Perl [PERL] basierend auf dem vorhandenen Apache http Server [APACHE]. Auf Serverseite wird aufbauend auf einem Jakarta Tomcat Server [TOMCAT] die Entwicklung in Java [JAVA] realisiert.

1.4 Konventionen

An dieser Stelle werden einige hinweisende Worte über die in dieser Arbeit verwendeten Begriffe und Notationen gegeben.

Begriffsdefinitionen:

- Ein *Benutzer* ist entweder ein Benutzer eines WebServices oder des http-Interfaces.
- Ein *Entwickler* ist jemand, der ein bioinformatisches Programm mittels BiBiWS den Benutzern zur Verfügung stellt.
- Das *BiBiServ Administrator Team* ist zuständig für die Wartung der Server, Installation der WebServices auf den Produktionssystemen und gibt Hilfestellung bei

der Entwicklung von den Tools.

- Die Begriffe *Werkzeug*, *Anwendung*, *Projekt* und *Tool* werden synonym verwendet. Gemeint ist bei allen ein Programm, das ein Entwickler durch den BiBiServ erreichbar machen möchte.

Notationen:

- Zeilen oder Fragmente von Source Code werden folgendermaßen notiert:
  `Status status = new Status()`
  Gegebenenfalls werden Zeilennummern für den Bezug im Text vorrangestellt.
- Referenzen auf Dateien und Verzeichnisse sehen in dieser Arbeit wie folgt aus: `wss/build.xml`
- Kommandos oder Programme werden notiert wie folgt: `ant deploy_bibiwstest`
- <PN> und *TemplateWS* sind Synonyme für den jeweiligen Tools, die ein Entwickler realisiert. Sie werden in dieser Arbeit beispielhaft verwendet.
  Beides wird beim Erstellen eines neuen Projektes mittels eines Hilfsprogrammes durch den Projektnamen ersetzt (siehe Anhang B.4).

Außerdem möchte ich darauf hinweisen, daß die Diagramme in dieser Arbeit, soweit es sinnvoll erscheint, dem UML Standard zur Softwareentwicklung [OEST] entsprechen.
An einigen Stellen weiche ich aber davon ab, um die Übersichtlichkeit zu erhalten. Sequenzdiagramme sind laut UML zwar nur für Objekte und deren Ablauf definiert, aber auch auf die hier clientseitig verwendeten CGIs gut anzuwenden, um einen Ablauf zu veranschaulichen.

Insbesondere Anhang A, Anhang B und Anhang C sind auch für größere englischsprachige Personengruppen [1] interessant, deswegen liegen sie im Unterschied zu der restlichen Arbeit in englischer Sprache vor. Aus dem gleichen Grund sind sämtliche Diagramme in englisch gehalten.

1.5 Übersicht dieser Arbeit

Nach diesem Kapitel der einleitenden Worte beschäftigt sich Kapitel 2 mit den zugrunde liegenden Techniken, die in dieser Arbeit benutzt werden.
Im darauffolgenden Kapitel 3, werden die auftretenden Probleme aufgenommen und Lösungen vorgestellt bzw. Designentscheidungen erörtert.
Das Kapitel 4 erläutert nun ausführlich sowohl die Realisierung der Client- und Serverseite, als auch die Abhängigkeiten zwischen diesen Seiten.

[1] beispielsweise aus dem Graduiertenkoleg

Im Kapitel 5 werden die BiBiWS Libraries im Detail vorgestellt.
Nachdem bei der Entwicklung einige interessante Performancephänomene aufgetreten sind, werden diese in Kapitel 6 untersucht und dargestellt.
Abschließend zieht Kapitel 7 ein Fazit und gibt einen Ausblick auf Möglichkeiten, die sich aus BiBiWS ergeben.
Zusätzlich befindet sich im Anhang A „BiBiWS Development Guide“ eine Anleitung für die Benutzung von BiBiWS. Sie erleichtert Entwicklern den Umgang und die Realisierung ihrer bioinformatischen Werkzeuge als WebService.
Mit der Administration und Installation von BiBiWS beschäftigt sich Anhang B. Er soll den Administratoren des BiBiServ und gegebenenfalls anderen Betreibern ähnlicher Server ermöglichen, BiBiWS sowohl zu installieren und zu pflegen, als auch nötige Anpassungen an sich ändernde Umgebungen (wie den Servern) durchzuführen.
Anhang C gibt eine Übersicht über die Statuscodes, die BiBiWS als Standard vorgibt und verwendet (siehe dazu Kapitel 4, Anhang A und Anhang B).

2 Grundlagentechniken

BiBiWS baut auf einigen vorhandenen Techniken, Libraries und Programmen auf. Diese werden hier kurz vorgestellt, da sie deutlich machen, in welchem Rahmen und unter welchen Bedingungen BiBiWS läuft.

Ein Großteil der Programme stammt dabei aus dem Repertoire der *Apache Software Foundation* [APACHE].
Diese ist eine Gemeinschaft von Entwicklern, die viele *Open Source* Projekte fördert und unter unterschiedlichen Konditionen verbreiten (siehe [APALIZ]). Dazu gehören unter anderem die von BiBiWS verwendeten und hier vorgestellten Projekte Apache http Server, Tomcat, AXIS und Ant. Der gemeinsame große Vorteil von allen Projekten der *Apache Software Foundation* ist neben dem öffentlich zugänglichen Source Code, auch die große Entwickler- und Anwendergruppe. Diese bieten durch Mailinglisten einen optimalen Support, ohne daß Kosten oder Lizenzegebühren anfallen.
Gerade letzteres ist nicht nur beim Entwickeln von Programmen vorteilhaft, sondern auch für die Benutzung. So sind Projekte wie BiBiWS, die auf Apache Projekten aufbauen, ohne anfallende Lizenzen der darunter liegenden Programme, kostenlos für andere Anwender einsetzbar.
So kann BiBiWS an anderen Standorten installiert werden. Einige Besonderheiten wird es zwar zu beachten geben (beispielsweise wird nicht überall eine *Sun Grid Engine* verfügbar sein), aber prinzipiell ist eine einfache und schnelle Installation möglich (siehe Anhang B).

Dieses Kapitel stellt die benutzten Techniken vor und gibt dem Leser so einen Rahmen von BiBiWS. Dadurch können die folgenden Kapitel sich auf die Ideen und Implementierungsdetails konzentrieren.

2.1 WebServices

„*The World Wide Web is more and more used for application to application communication. The programmatic interfaces made available are referred to as Web services.*“ (von [WSDEF])

WebServices sind die Weiterentwicklung von *Remote Procedure Call (RPC)*. Sie ermöglichen es einem via Netzwerk Nachrichten zu übermitteln und eine Rückmeldung zu erhalten.
Dabei gibt es die Möglichkeit mittels der Publikation von sogenannten WSDL Definitionen einem Client alle Informationen zu geben, die nötig sind, um automatisiert

auf einen speziellen WebService zuzugreifen (siehe Kapitel 2.2). Eine Verwaltung und Katalogisierung kann ein sogenannter UDDI Server übernehmen, wodurch das Auffinden von WebService durch eine zentrale Instanz erleichtert wird (siehe Kapitel 2.7).

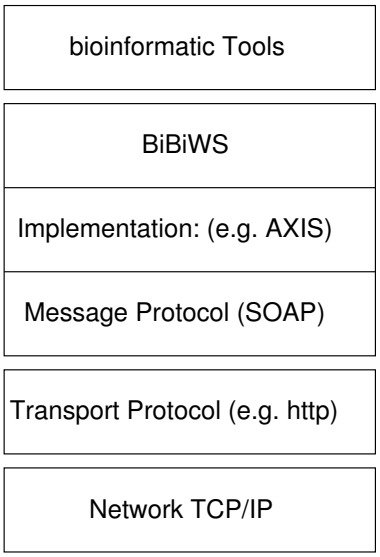

Abbildung 2.1: Ebenen eines BiBiWS WebServices

Wie in Abbildung 2.1 zu sehen ist, umfaßt der Begriff WebService eine ganze Reihe von Ebenen, zu denen es jeweils alternative Entwicklungen gibt, die meistens austauschbar sind.
Das Netzwerk auf TCP/IP stellt eine Basis für die Kommunikation zwischen Clients (dem Sender der Nachricht) und Servern (dem Empfänger) dar (siehe OSI-Layer Beschreibung in [TANE]).
Neben der Adresse (IP Adresse bzw. Name des Rechners), unter der der Server erreichbar ist, muß dem Client ebenso bekannt sein, wie die Verbindung aufgebaut werden kann.
Für die Kommunikation existieren unterschiedliche Protokolle, wobei das meistverbereitete und aktuellste *SOAP* [SOAP] ist. Das Akronym *SOAP* ist nicht eindeutig definiert. Am häufigsten werden *Simple Object Access Protocol* oder *Services Oriented Access Protocol* verwendet.
Eine zusammenfassende Definition der Aufgabe findet sich in der Spezifikation [SOAP]:

> „*SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.*"

Neben dieser Definition bleibt noch zu sagen, daß *SOAP* nicht auf ein spezielles Trans-

portprotokoll aufbaut. Vielmehr basiert die Version 1.1 auf dem http Protokoll [HTTP], ist jedoch nicht an dieses gebunden. Alternativen dazu befinden sich noch im Entwicklungsstadium (siehe auch Kapitel 2.5).
Durch die Spezifikation eines unabhängigen Nachrichtenprotokolls wird es möglich, daß Client und Server in verschiedenen Programmiersprachen geschrieben sein können – solange diese das Nachrichtenprotokoll unterstützt. BiBiWS beispielsweise baut auf Serverseite auf JAVA auf, während die CGIs auf Clientseite in Perl geschrieben sind. Nahezu alle gebräuchlichen Programmiersprachen bieten mittlerweile eine SOAP Library an, wodurch die mittels BiBiWS entwickelten WebServices für jeglicher Art von Programmen erreichbar sind.
SOAP Implementierungen existieren unter anderem für JAVA, Perl, C/C++, Python und PHP.
Eine SOAP Kommunikation wird in Kapitel 2.3 erläutert.

2.2 WSDL

Ein wichtiger Aspekt von WebServices im Allgemeinen ist eine maschinenles- und -auswertbare, XML basierende Schnittstellenbeschreibung. Sie erfolgt in dem WSDL-Format (WebService Description Language) [WSDL].
Die AXIS Library (wie auch andere) liefert Programme mit, die in der Lage sind, aus dem vorhandenen Source Code des WebServices eine WSDL Definitionen automatisch zu generieren.
Zu einer solchen Definition gehören:

- die URI des WebService
- die Liste der möglichen Funktionen
- die Spezifikation der Parameter.

Hier ist die WSDL Definition des Beispiel WebServices von BiBiWS *TemplateWS*, das die AXIS Library automatisch generiert hat: [1]
In Abbildung 2.2 werden nach der Definition der Namespaces die nicht elementaren Datentypen des WebServices definiert. Das Template bekommt ein Array von Parametern übergeben und die Methoden können einen `soap_fault` als Status liefern (siehe Kapitel 2.3).
Es folgt (Abbildung 2.3) die Definition der Messages, die Ein- und Ausgaben der Methoden `request()` und `response()`. Bei beide Methoden handelt es sich um *Request/Response* Methoden nach der SOAP Spezifikation, also Methoden, die eine Eingabe und Rückgabe haben.
Die `request()` Methode hat als Parameter den oben definierten Array und liefert die id, mit der die `response()` Methode später aufgerufen wird. Diese liefert das Resultat als String (siehe Kapitel 3.1.1).
In Abbildung 2.4 werden nun die Methoden mit den oben definierten Messages ver-

[1] zur besseren Darstellung sinnwahrend gekürzt; URI ist die URI des WebServices

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions ...SKIPPED...>
3   <wsdl:types>
4    <schema targetNamespace="URI" xmlns="http://www.w3.org/2001/XMLSchema">
5     <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
6     <complexType name="ArrayOf_tns1_anyType">
7      <complexContent>
8       <restriction base="soapenc:Array">
9        <attribute ref="soapenc:arrayType" wsdl:arrayType="tns1:anyType[]"/>
10      </restriction>
11     </complexContent>
12    </complexType>
13   </schema>
14   <schema targetNamespace="http://templatews.bibiserv.techfak.unibi.de"
       xmlns="http://www.w3.org/2001/XMLSchema">
15    <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
16    <complexType name="TEMPLATEWSException">
17     <sequence/>
18    </complexType>
19   </schema>
20  </wsdl:types>
```

Abbildung 2.2: Beispiel WSDL Definition eines WebServices (Teil 1)

```
21  <wsdl:message name="requestRequest">
22   <wsdl:part name="array_params" type="impl:ArrayOf_tns1_anyType"/>
23  </wsdl:message>
24  <wsdl:message name="requestResponse">
25   <wsdl:part name="requestReturn" type="xsd:string"/>
26  </wsdl:message>
27  <wsdl:message name="responseRequest">
28   <wsdl:part name="id" type="xsd:string"/>
29  </wsdl:message>
30  <wsdl:message name="responseResponse">
31   <wsdl:part name="responseReturn" type="xsd:string"/>
32  </wsdl:message>
33  <wsdl:message name="TEMPLATEWSException">
34   <wsdl:part name="fault" type="tns2:TEMPLATEWSException"/>
35  </wsdl:message>
```

Abbildung 2.3: Beispiel WSDL Definition eines WebServices (Teil 2)

knüpft. Zusätzlich wird die Verknüfung mit dem `soap_fault` hergestellt.

Folgende Zeilen in Abbildung 2.5 stellen die Bindung der Ein- und Ausgabe mit den ensprechenden Namespaces dar und sorgen so für die richtige *Kodierung* der Daten in SOAP.

Zum Abschluß wird in Abbildung 2.6 noch die Adresse (*URI*) des WebServices vermerkt.

```
36  <wsdl:portType name="TEMPLATEWSImplementation">
37   <wsdl:operation name="request" parameterOrder="array_params">
38    <wsdl:input message="impl:requestRequest" name="requestRequest"/>
39    <wsdl:output message="impl:requestResponse" name="requestResponse"/>
40    <wsdl:fault message="impl:TEMPLATEWSException" name="TEMPLATEWSException"/>
41   </wsdl:operation>
42   <wsdl:operation name="response" parameterOrder="id">
43    <wsdl:input message="impl:responseRequest" name="responseRequest"/>
44    <wsdl:output message="impl:responseResponse" name="responseResponse"/>
45    <wsdl:fault message="impl:TEMPLATEWSException" name="TEMPLATEWSException"/>
46   </wsdl:operation>
47  </wsdl:portType>
```

Abbildung 2.4: Beispiel WSDL Definition eines WebServices (Teil 3)

```
48  <wsdl:binding name="soapSoapBinding" type="impl:TEMPLATEWSImplementation">
49   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
50   <wsdl:operation name="request">
51   <wsdlsoap:operation soapAction=""/>
52    <wsdl:input name="requestRequest">
53     <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="URI"/>
54    </wsdl:input>
55    <wsdl:output name="requestResponse">
56     <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="URI"/>
57    </wsdl:output>
58    <wsdl:fault name="TEMPLATEWSException">
59     <wsdlsoap:fault encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         name="TEMPLATEWSException" namespace="URI"/>
60    </wsdl:fault>
61   </wsdl:operation>
62   <wsdl:operation name="response">
63    <wsdlsoap:operation soapAction=""/>
64    <wsdl:input name="responseRequest">
65     <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="URI"/>
66    </wsdl:input>
67    <wsdl:output name="responseResponse">
68     <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="URI"/>
69    </wsdl:output>
70    <wsdl:fault name="TEMPLATEWSException">
71     <wsdlsoap:fault encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
         name="TEMPLATEWSException" namespace="URI"/>
72    </wsdl:fault>
73   </wsdl:operation>
74  </wsdl:binding>
```

Abbildung 2.5: Beispiel WSDL Definition eines WebServices (Teil 4)

2.3 SOAP

SOAP ist das Nachrichtenprotokoll, was für BiBiWS die Kommunitkation zwischen Client und Server aufbauend auf http spezifiziert. Hier soll wieder am Beispiel des

```
75  <wsdl:service name="TEMPLATEWSImplementationService">
76   <wsdl:port binding="impl:soapSoapBinding" name="soap">
77    <wsdlsoap:address location="URI"/>
78   </wsdl:port>
79  </wsdl:service>
80 </wsdl:definitions>
```

Abbildung 2.6: Beispiel WSDL Definition eines WebServices (Teil 5)

TemplateWS WebServices von BiBiWS kurz gezeigt werden, wie SOAP Nachrichten aussehen. SOAP unterstützt nativ nur wenigen einfache Datentypen, die in der XML-Schema Spezifikation [XMLSCHEMA] definiert sind.
Komplexe Datentypen können jedoch in der WSDL Definition über XMLSchemas beschrieben werden, wie es auch das HOBIT-Projekt macht (siehe Kapitel 1.2).
Der interessierte Leser sei hier auf die Spezifikation verwiesen [SOAP].
Wie in dem oben aufgeführte WSDL Beispiel zu sehen, können SOAP Methoden entweder vom Typ „One-Way" sein (d.h. nur ein eine Richtung werden Daten übertragen) oder *Request/Response* (d.h. Daten werden beim Aufruf und als Rückgabe übertragen) sein.
SOAP sieht außerdem einen sogenannt `soap_fault` mit einer `soap_description` vor, falls die gewünschte Antwort nicht verfügbar ist.

Beispielhaft für eine SOAP Kommunikation sind *Request/Response* Methoden und ein `soap_fault` hier anhand des *TemplateWS* Beispiels in den Abbildungen 2.7, 2.8, 2.9, 2.10 und 2.11 dargestellt. [2]

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SOAP-ENV:Envelope ...SKIPPED...>
3   <SOAP-ENV:Body>
4    <namesp1:request xmlns:namesp1="URI">
5     <array_params xsi:type="impl:ArrayOf_tns1_anyType" SOAP-ENC:arrayType="xsd:ur-type[2]">
6      <item xsi:type="xsd:string">length</item>
7      <item xsi:type="xsd:int">23</item>
8     </array_params>
9    </namesp1:request>
10  </SOAP-ENV:Body>
11 </SOAP-ENV:Envelope>
```

Abbildung 2.7: Aufruf von `request()` des Clients auf dem Server

2.3.1 AXIS

Eine SOAP Implementierung des Apache Projektes ist AXIS [AXIS] und steht für *Apache extensible interaction system*.

[2] zur besseren Darstellung sinnwahrend gekürzt; URI ist die URI des WebServices

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <soapenv:Envelope ...SKIPPED...>
3   <soapenv:Body>
4    <ns1:requestResponse ...SKIPPED...>
5     <requestReturn xsi:type="soapenc:string" ...SKIPPED...>
         bibiwstest_2004-10-10_142047_BYfQX</requestReturn>
6    </ns1:requestResponse>
7   </soapenv:Body>
8  </soapenv:Envelope>
```

Abbildung 2.8: Antwort von `request()` des Servers

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SOAP-ENV:Envelope ...SKIPPED...>
3   <SOAP-ENV:Body>
4    <namesp1:response ...SKIPPED...>
5     <id xsi:type="xsd:string">bibiwstest_2004-10-10_142047_BYfQX</id>
6    </namesp1:response>
7   </SOAP-ENV:Body>
8  </SOAP-ENV:Envelope>
```

Abbildung 2.9: Aufruf von `response()` des Clients auf dem Server

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <soap-env:Envelope ...SKIPPED...>
3   <soapenv:Body>
4    <ns1:responseResponse ...SKIPPED...>
5     <responseReturn xsi:type="soapenc:string" ...SKIPPED...>
6       ...SKIPPED...
7     </responseReturn>
8    </ns1:responseResponse>
9   </soapenv:Body>
10 </soapenv:Envelope>
```

Abbildung 2.10: Antwort von `response()` des Servers, wenn das Resulat berechnet ist

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <soap-env:Envelope ...SKIPPED...>
3   <soapenv:Body>
4    <soapenv:Fault>
5     <faultcode>604</faultcode>
6     <faultstring>Processing: Running</faultstring>
7    </soapenv:Fault>
8   </soapenv:Body>
9  </soapenv:Envelope>
```

Abbildung 2.11: Antwort von `response()` des Servers, wenn das Resulat nicht berechnet ist (`soap_fault`)

Nach dem ersten Versuch, der den einfachen Namen *SOAP* trug und die Weiterentwicklung von IBMs *SOAP4J* war, wurde beschlossen, ein komplettes Refactoring des

Source Codes durchzuführen, welches in dem Projekt AXIS geschah.

Auch wenn AXIS „nur" eine weitere SOAP Implementierung von vielen darstellt, ist sie doch eine der Umfassendsten und Verbreitetsten.
Das Entwickler Team empfiehlt selbst den Einsatz zusammen mit dem Tomcat Server, wenn ein *Servlet Container* gebraucht wird. So existiert ein großer Benutzerkreis, viele Dokumentationen und Hilfen zu Problemen, die bei der Entwicklung von WebServices unter genau diesen Umständen helfen.
Leider ist die Dokumentation an einigen Stellen sehr lückenhaft und ein Entwickler muß an vielen Stellen die mitgelieferten Beispiele konsultieren oder im Internet recherchieren. Immerhin finden sich in dem *AXIS User Manual* [AXISUM] entsprechende Hinweise wie diese zum Thema *Exceptions*:

„*This is an area which causes plenty of confusion, and indeed, the author of this section is not entirely sure how everything works,...*"

AXIS bietet mehr als eine einfache SOAP Implementierung der Spezifikation 1.1 (und teilweise 1.2).
So werden auch Hilfsprogramme angeboten, die aus einer vorhandenen WSDL Definition ein *Source Code Skeleton* erzeugen. Aber auch die andere Richtung ist möglich (und wird von BiBiWS benutzt): Erzeugen der WSDL Definition aus Source Code.
Zusätzliche Features wie der *TcpMon* zum Beobachten der übermittelten SOAP Nachrichten zu Entwicklungszwecken runden das Angebot ab. Weitere Erweiterungen erleichtern das Entwickeln von WebServices mittels AXIS, jedoch werden sie von BiBiWS nicht benötigt. Der geneigte Leser sei hier auf das *Axis User Manual* [AXISUM] verwiesen.

AXIS bietet zwei Möglichkeiten WebServices auf dem Tomcat zu *deploy*en, also anzumelden. Die einfache und „normale" Vorgehensweise ist, AXIS selbst als einen WebService bei dem Servlet Container anzumelden. Der Rest des Managements wird mittels der ebenfalls von AXIS mitgelieferten Hilfsprogramme bewerkstelligt. Aus Sicht des Tomcat Servers gibt es also nur einen WebService.
Hierdurch können sehr einfach WebService realisiert werden, im einfachsten Fall durch einfaches Kopieren des Source Codes an die entsprechende Stelle.
Die andere Möglichkeit ist, jedem Programm einen eigenständigen WebService zuzuordnen, was bei BiBiWS geschieht. Dabei wird dem WebService jeweils eine Datei mitgeliefert, die die Verbindung vom Tomcat zu AXIS herstellt (`web.xml`) und von AXIS zu der Implementation des Programms (`server-config.wsdd`).
Da beide Dateien automatisch generiert werden, muß ein Entwickler eines BiBiWS WebServices sich in diese Thematik nicht einarbeiten.
Neben diesen äußerst positiven Erweiterungen von AXIS, die nicht zu der SOAP Spezifikation gehören, bringen eigene Erweiterungen aber auch Nachteile mit sich. So kann AXIS nicht einfach durch eine andere SOAP Library ausgetauscht werden. Auch wenn BiBiWS WebServices innerhalb des Source Codes nur an einer Stelle (siehe Kapitel

3.2.1) AXIS spezifischen Code tragen, so ergibt sich doch eine Abhängigkeit.

BiBiWS wurde unter der Version 1.2beta von AXIS entwickelt, da erst diese einge benutze Zusatzprogramme mitliefert. Auf Grund eines robusten Designs und der Implementation von JAX-RPC und SAAJ Konformität (beides Interface Pakete) ist schon seit langer Zeit für einheitliche Interfaces gesorgt. Der Einsatz unter neueren Versionen wird also problemlos sein.

2.4 Apache Jakarta Tomcat

Der *Jakarta Tomcat* Server [TOMCAT] ist ein *Servlet Container* für *Java Servlets*. Er unterstützt in der Version 5.5 den offiziellen Standard 2.2 der *Java Servlet* Spezifikation [JSC], die durch den *Java Community Process* [JCP] entstanden ist.

WebServices, die auf AXIS aufbauen, können auch als Stand-alone Dienste laufen. Durch den Einsatz eines *Servlet Containers* bieten sich Möglichkeiten eines zentralen Managements der WebServices.
Der Servlet Container startet alle WebServices innerhalb einer gemeinsamen JavaVM, was sowohl Ressourcen schont, wie auch die Pflege der WebServices vereinfacht. Er reicht die Anfragen an die WebServices weiter, so daß aus Sicht des Clients kein Unterschied zu Stand-alone WebServices existiert. Berechtigte Personen können einen WebService abschalten und neu starten, was nach einer Änderung des WebServices nötig ist. Der Tomcat bietet auch Sicherheitsrichtlinien, die Restriktionen zulassen. Beispielsweise kann der Tomcat Bereiche des Filesystems für den Zugriff sperren, was die Arbeit des BiBiServ Administrator Teams wesentlich vereinfacht. Entwickler können z.B. Daten nur in dem Spoolverzeichnis ablegen und auch keine sensitiven Daten, die dem BiBiServ Administrator Team vorbehalten sind, auslesen. Auch wenn bösartige Absichten Keinem zu unterstellen sind, verhindert es den unbewußten Zugriff durch Programmierfehler.
Auch mit der eventuell gewünschten Unterstützung von verschlüsselter Datenkommunikation von Client und Server muß der Entwickler eines WebServices, der später auf dem Tomcat installiert wird, sich nicht kümmern.
Alle diese Möglichkeiten sind in einem Stand-alone WebServices selbst zu entwickeln und zu pflegen, was erhebliche Einarbeitungs- und Entwicklungszeit kostet.

Neben den hier vorgestellten Eigenschaften, die BiBiWS benötigt, kann der Tomcat als Ersatz für den Apache http Server benutzt werden und implementiert auch die JSP Spezifikation [JSP].
Da der BiBiServ aber seit seiner Gründung auf dem Apache http Server aufbaut, würde eine komplette Umstellung auf den Tomcat viel Zeit beanspruchen. Außerdem wären bei hoher Last auf dem Tomcat Server selbst die HTML Seiten des BiBiServ nicht erreichbar.

2.5 http

Wie oben schon erwähnt, ist SOAP nicht auf ein bestimmtes Transferprotokoll festgelegt. In der Spezifikation 1.1 ist SOAP jedoch als aufbauend auf http beschrieben [HTTP].

Dieses Protokoll hat eine weite Verbreitung im Internet gefunden und ist ursprünglich für das klassische WWW, das dem Internet den Durchbruch verschafft hat, entwickelt worden.
Es handelt sich um ein synchrones Protokoll, das heißt auf eine Anfrage erfolgt innerhalb von kurzer Zeit (typischerweise Sekunden) eine Antwort.
Beispielsweise stellt ein Browser eine Anfrage an einen Webserver, wie den Apache http Server, welcher eine HTML Seite zurückliefert.
Auf Grund der hohen Verwendung und allgemeinen Verfügbarkeit (http wird selten durch Firewalls geblockt und wird durch Proxys nicht behindert) findet dieses Protokoll nun auch Anwendung in anderen Bereichen, wie hier für WebServices.

Durch die Verwendung von http in BiBiWS entstehen auch Probleme, welche in Kapitel 3.2.3 beschrieben werden.

2.6 Apache http Server

Der *Apache http Server* [APACHEHTTP] ist wohl bei weitem das bekannteste Projekt der *Apache Software Foundation*, weswegen man meistens auch einfach nur von „Apache“ spricht, wenn der Apache http Server gemeint ist.
Dieses Projekt entwickelt den auf der Welt am meisten genutzten *http* Server. Im August 2003 belegte er laut [NETCRAFT] einen Marktanteil von circa 64 % der weltweit betriebenen Webservern. Seit seiner Gründung 1996 ist der BiBiServ selbst ein solcher Apache.
Er ist für die Beantwortung einkommender http Anfragen zuständig und liefert die entsprechenden HTML Seiten aus oder ruft CGI Programme auf, um deren Ausgabe auszuliefern.

BiBiWS setzt auf der Clientseite auf den Apache http Server. HTML Anfragen werden also als WebService Aufruf formuliert und an den Tomcat weitergegeben.
BiBiWS fügt sich in die existierende BiBiServ Umgebung nahtlos ein, für den Benutzer der HTML Oberfläche ist also nicht zu erkennen, ob ein angebotenes Programm als WebService intern aufgerufen wird, oder ob es sich um die herkömmliche Methode handelt, Programme auf dem BiBiServ zu präsentieren.

2.7 UDDI

Ein weiterer interessanter Aspekt der WebService Technologie ist *Universal Description, Discovery and Integration* (kurz: UDDI) [UDDI]. Mittels UDDI soll es möglich werden, WebServices aufzufinden. Dafür werden die WebServices registriert und katalogisiert – zentral auf einem UDDI-Server (sogenanntes *yellow paging*).
In Zukunft soll es auch möglich sein über diese Server automatisiert WebServices aufzufinden und zu benutzen – sofern die Konvertierung der Ein- und Ausgabe Daten automatisch gelingt, was eines der Hauptprobleme ist. Deswegen liegen auch hier die zentralen Aspekte des HOBIT-Projektes.

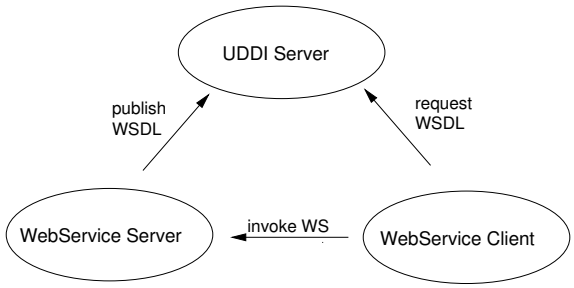

Abbildung 2.12: Zusammenspiel von UDDI und WSDL

Leider ist die Weiterentwicklung dieses interessanten Punktes problembehaftet, auf Grund von Patentstreitigkeiten und damit Unsicherheit bei einem Großteil der Interessierten gibt (see [UDDIPROB]).

BiBiWS hat deswegen in dieser Form keine Unterstützung für UDDI. Sobald das HOBIT-Projekt sich auf einen Standard zur Katalogisierung der angebotenen WebServices verständigt hat, wird es mit den bereits erzeugten und benutzten WSDL Definitionen der BiBiWS WebServices kein Problem sein, dieser Übereinkunft nachzukommen. Bis dahin werden die WSDL Definitionen der auf dem BiBiServ laufenden WebServices auf einer Webseite [BIBIWSDLS] bereitgestellt.

2.8 *Sun Grid Engine*

Die *Sun Grid Engine* ist eine von Sun entwickelte *Computing Grid* Umgebung, um eventuell rechenintensive Programme zu verteilen.
Bei dieser Grid Architektur gibt es drei unterschiedliche *Hosts*:

Submission Host von hier übermittelt der Benutzer die Jobs an den Master

Master Host empfängt die Jobs und übernimmt das Managment zur Verteilung auf den Execution Hosts

Execution Host führt die Jobs aus

Der *Master Host* berücksichtigt dabei auf den ihm zugeteilten Execution Hosts die Verfügbarkeit der Ressourcen und die aktuelle Last, so daß die *Sun Grid Engine* für

eine optimale Verteilung sorgt („Loadbalancing"). Dabei kann der Benutzer festlegen, welche Ressourcen gebraucht werden (beispielsweise Architektur oder Größe des Arbeitsspeichers).

Leider bietet Sun mit der Version 5 der *Sun Grid Engine* keine API für Java an, so daß hier das Aufrufen der Programme über die Kommandozeile mit anschließender Auswertung der Ausgabe derzeitig die einzige Möglichkeit darstellt.
Mit Erscheinen der Version 6 soll sich dieses ändern, da dann die allgemein für Computing Grid konzipierte Schnittstellendefinition *Distributed Resource Management Application API* – kurz DRMAA [DRMAA] unterstützt werden soll.
Ein weiterer Nachteil, der mit Version 6 hoffentlich behoben wird, besteht darin, daß derzeit keine aktive Rückmeldung über den Status eines Jobs möglich ist. Wieder ist es unumgänglich, die Kommandozeile zu bemühen und in regelmäßigen Abständen den Status eines Jobs zu überprüfen, also ein *Polling* durchzuführen.

BiBiWS kapselt die Anbindung der *Sun Grid Engine* soweit in einer Klasse, daß bei Erscheinen der Version 6 die Anbindung über DRMAA mit minimalem Aufwand realisierbar ist. Auch würden durch ein solches Interface die Integartion anderer Computing Grid Umgebungen möglich.

2.9 *make* und *ant*

Das klassische Werkzeug unter UNIX, um Arbeitsvorgänge auf der Shell zu automatisieren, ist das Programm `make`.
Auch BiBiWS setzt auf Clientseite auf dieses Programm, um dem Entwickler und dem BiBiServ Administrator Team die Installation der WebServices zu erleichtern. Durch die verschiedenen *targets* einer `Makefile`-Datei, die unterschiedliche Anweisungen zusammenfassen, ist es möglich, daß sowohl die Installationsanweisungen auf dem Testsystem, als auch auf dem Produktionssystem einheitlich sind, so daß sie sich nur im Zielverzeichnis unterscheiden.
Dadurch ist gewährleistet, daß eine erfolgreiche Installation auf dem Testsystem zu einer erfolgreichen Installation auf dem Produktionssystem führt.
`make` bietet neben den herkömmlichen Befehlen der Shell nur wenig Unterstützung für spezifische Programme. Es gibt auch keine Möglichkeit Erweiterungen einzubinden, mit denen sich projektspezifische Aufgaben erledigen lassen.
BiBiWS setzt `make` für die Installation der Clientseite ein, da es wesentlich schneller und einfacher zu begreifen ist als `ant`.

Für die Serverseite ist ein in und für JAVA geschriebenes Pendant sinnvoller: `ant`.
Es basiert im Gegensatz zu der Datei von `make`, auf einer XML Datei.
`ant` selbst bringt eine umfassende Unterstützung für JAVA mit, wie zum Beispiel kompilieren oder das Erstellen der *javadoc* Dokumentation eines Projektes. Dadurch, daß `ant` selbst ein JAVA Programm ist, muß nicht für jedes dieser *Targets* eine seperate

JavaVM gestartet werden, was bei größeren Projekten sogar zu einer Performancesteigerung gegenüber dem Einsatz von `make` bei diesen Aufgaben führt.
`ant` kann durch zusätzliche Libraries (JAR Libraries) erweitert werden, wie sie auch AXIS und Tomcat mitliefern. Durch die enthaltenen *target*s werden neue „Befehle" für den Benutzer ermöglicht, wie zum Beispiel das Installieren eines neuen WebServices durch das Target *catalina-deploy*. Weitere Beispiele befinden sich im Beispielprojekt *TemplateWS*.

3 Das Konzept von BiBiWS

BiBiWS stellt ein Framework für die Entwicklung von WebServices auf dem BiBiServ dar.
BiBiWS umfaßt auf der einen Seite Libraries, die die grundsätzliche benötigte Funktionalität bieten, auf der anderen Seite stellt es ein nicht unerheblicher Anteil die *Templates* dar.
Diese Templates sind Vorlagen, die einem Entwickler, der für den BiBiServ einen WebService entwickeln möchte, bereitgestellt werden. Sie werden bei der Erstellung eines neuen Projektverzeichnisses automatisch so abgeändert, daß an den passenden Stellen der Name des Projektes eingefügt wird. So präsentiert sich dem Entwickler ein voll funktionsfähiger WebService auf Client- und Serverseite. Durch dieses Vorgehen wird den Entwicklern eine Menge Einarbeitungszeit abgenommen, da die Vorgaben nur noch angepasst werden müssen (siehe Anhang B). Es wird also eine zusätzliche Abstraktionsebene eingeführt, die die Entwickler weitestgehend von der WebService Technologie entbindet.

Wie schon beschrieben, besteht BiBiWS aus einer Clientseite und einer Serverseite. Die Clientseite stellt ein HTML Interface für die klassische Präsentation der Werkzeuge auf dem BiBiServ dar. Damit erhalten Benutzer über HTML-Formulare transperent Zugriff auf die angebotenen Werkzeuge, ohne daß sie sich in die Installation des Programms oder in das Prinzip der WebServices einarbeiten müßten.
Die Serverseite ist in Abbildung 3.1 dargestellt. Vereinfacht führt ein Aufruf der `request()` Methode im Tomcat dazu, daß der mittels BiBiWS entwickelte WebService seine Processingklasse als Thread im Hintergrund startet und eine `id` zurückliefert. Die Processingklasse startet über die `SGECall` Klasse das eigentliche Programm auf ein Computing Grid (*Sun Grid Engine*). Danach werden die Ergebnisse in ein Spoolverzeichnis geschrieben, von wo aus die `response()` Methode Zugriff auf die Ergebnisse hat.

Wie in Abbildung 1.1 illustriert, können im Bedarfsfall auch Zugriffe direkt auf die WebServices erfolgen. Dafür muß der Benutzer seine Aufrufe an den BiBiServ WebService Server stellen.
Der Vorteil liegt auf der Hand: Über standardisierte Schnittstellen ist es möglich, automatisiert auf die WebServices zuzugreifen, so daß sogenannte *Chains* realisiert werden können. Chains sind hintereinander folgende Aufrufe von Programmen, hier WebServices.
Beispielsweise (wie von *e2g* [E2G] realisiert) kann erst ein WebService zur Datenbankabfrage von Sequenzen bemüht werden und das Resultat dann einem zweiten WebSer-

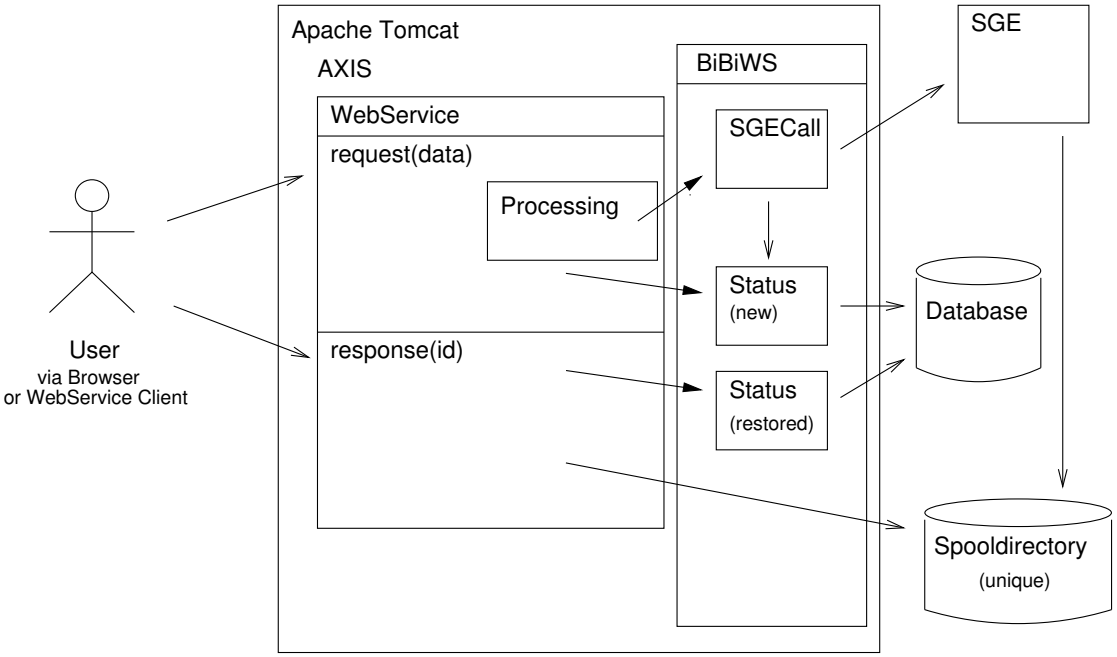

Abbildung 3.1: Vereinfachter BiBiWS Ablauf auf Serverseite

vice zur Alignmentbildung übergeben werden.
Nebenher ergibt sich für den BiBiServ Rechner noch der Vorteil, daß die Berechnungen nicht mehr auf dem Webserver lokal laufen. Über die Anbindung eines Computing Grids, das die Berechnungen auf andere Maschinen verteilt, stehen den Benutzern des BiBiServ nun ein Vielfaches an Rechenleistung zur Verfügung. Das Prinzip eines solchen Aufrufs ist unabhängig ob es sich um BiBiServ als Client oder einen Benutzer direkt handelt und in Abbildung 3.2 dargestellt.
Da WebServices auf (derzeit) synchronen Protokollen aufbauen, unterstützen sie keine Asynchronität. Diese ist jedoch nötig, da bioinformatische Programme im Normalfall keine direkte Antwort liefern können - ihre Rechenzeit ist dafür zu lang. BiBiWS implementiert deswegen die *Request and reply with polling* Technik, um eine Asynchronität zu erreichen.
Bei der *Request and reply with polling* Technik stellt der Client einen `request()` Aufruf an den WebServices Server und übermittelt die Daten, die zur Berechnung notwendig sind. Der Server empfängt die Daten, startet das Programm im Hintergrund und liefert eine `id` zurück. Zu einem späteren Zeitpunkt kann der Client mit dieser `id` nun über einen `response()` Aufruf das Ergebnis abfragen. Sollten die Berechnungen auf dem WebService Server noch nicht abgeschlossen sein, wird ein Statuscode [1] übermittelt, der den Benutzer über den aktuellen Status seiner Berechnungen informiert.

Im folgenden werden Entscheidungen und Alternativen diskutiert, die bei der Im-

[1] Statuscode 600 repräsentiert *finished*, siehe Anhang C

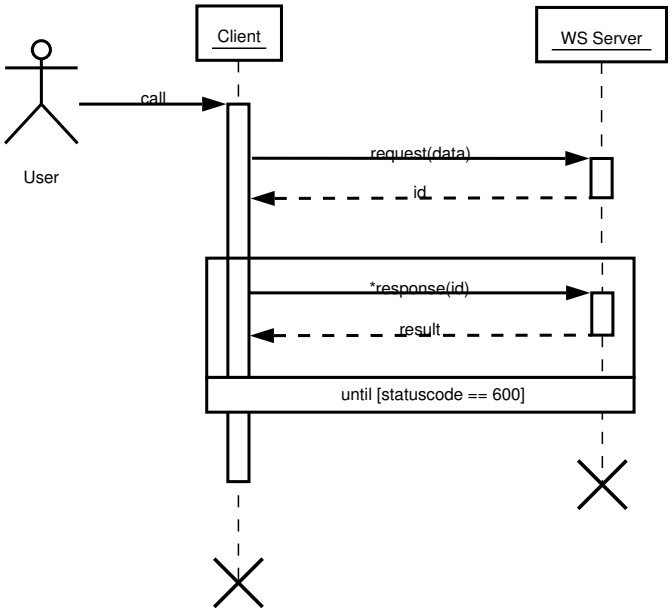

Abbildung 3.2: Grundsätzlicher Ablauf eines WebService Aufrufs mit BiBiWS.

plementation von BiBiWS getroffen werden mußten.
Dazu zählt die Abwägung der unterschiedlichen asynchronen Ansätze für WebServices und die Entscheidung, warum *RPC-styled* den *documented styled* WebServices vorzuziehen sind.
Die Wahl der benutzten, unterliegenden Software Pakete wird im nächsten Kapitel dargelegt. Insbesondere die Wahl, AXIS in Kombination mit Tomcat einzusetzen, ist hier wichtig.
Abschließend wird noch auf einige Probleme, Details und grundsätzliche Ideen, die bei der Implementierung von BiBiWS aufgetreten sind, eingegangen.

3.1 Entscheidungen

3.1.1 Benutzung von `soap_fault` und Kodierung der Daten

Bei der Kodierung der zu übermittelten Daten in SOAP stellen sich einige grunsätzliche Entscheidungen.
BiBiWS verwendet den `soap_fault` für alle Fälle, falls das Ergebis nicht verfügbar ist. Hierfür gibt es zwei Möglichkeiten: Entweder ist das Ergbnis noch nicht fertig, worauf-hin ein Statuscode `6xx` als `soap_fault` zurückgeliefert wird, oder es ist ein Fehler aufgetreten (Statuscode `7xx`). Zugehörig zu dem `soap_fault`, der die Meldung als Integer kodiert, wird noch eine Erklärung als `soap_description` übermittelt. Eine genauere Beschreibung der Statuscodes befindet sich in Kapitel 3.3.3 und Anhang C.

Da SOAP als einzigen Datentyen, der nicht elementar ist, ein Array spezifiziert, wird dieser verwendet, um die Parameter zu übergeben. Dabei wird ein Hash dargestellt, indem abwechselnd *key* und *value* in dem Array kodiert sind. So kann auf Serverseite aus dem Array ein Hash erzeugt werden. Zwar bieten sowohl die beiden verwendeten Libraries *AXIS* und *SOAP::Lite* auch eine Übertragung von Hashes, jedoch werden diese unterschiedlich kodiert, da es in der SOAP Spezifikation nicht vorgesehen ist. Eine Kommunikation über diese unterschiedlichen, implementationsabhängigen Erweiterungen nicht möglich.
Aus diesem Grund verwendet BiBiWS den spezifizierten Datentypen *Array*.

Leider ist es beim Erzeugen der WSDL Definitionen von WebServices zur Zeit in AXIS nicht möglich, als Ein- oder Ausgabe einer Methode ein XMLSchema zu definieren.
Da eine manuelle Erstellung der komplizierten WSDL Definitionen aber einem BiBiServ Entwickler nicht zugemutet werden kann, verwendet BiBiWS zum Übertragen von komplexen Datentypen ein serialisiertes DOM Dokument (siehe Kapitel A.5.2). Der Nachteil ist, daß in der WSDL Definition als Typ der Ein- oder Ausgabe einer Methode jediglich ein String definiert ist, der das serialisierte DOM Dokument enthält.
Auf der anderen Seite entfällt auf Sender- und Emfängerseite das aufwendige Entwicklen von Serializierern und Deserialisierern. Stattdessen können die für viele Programmiersprachen verfügbaren DOM Libraries verwendet werden (siehe Anhang A.5.2).

3.1.2 Asynchrone Ansätze

WebServices bieten laut Spezifikation von SOAP derzeit keine Möglichkeit für asynchrone Kommunikation. Es wird also immer eine Antwort auf eine Anfrage erwartet und im Nachhinein ist keine Referenzierung auf diese Kommunikation möglich. Dieses basiert auf dem Umstand, daß SOAP in der Version 1.1 auf dem synchronen Protokoll http basiert.
Es gibt nun mehrere Möglichkeiten, dieses Problem zu bewältigen. Neben der einfachsten Lösung, nämlich ein asynchrones Protokoll zu Grunde zu legen, kann mit `ids` zur Referenzierung gearbeitet werden. Da bisher keine asynchronen Protokolle marktreif sind und sie so auch keine Akzeptanz besitzen, sind sie zur Zeit noch nicht einsetzbar.
[ADA1] und [ADA2] bieten eine gute Übersicht der Möglichkeiten der asynchronen WebService Realisierung, die an dieser Stelle kurz zusammengefaßt werden soll:

1. **Vorschlag: One-way and notification operations** Verwendet werden *One-Way* Methoden (siehe Kapitel 2.3) in den WSDL Definitionen von Client **und** Server zum Senden der Anfrage und Zurücksenden der Antwort. Der Client ist verantwortlich für die Generierung der `id` und setzen einer *reply-to* Adresse. Dieser Vorschlag ist sicherlich der am Einfachsten zu verstehende. Allerdings wird eine Verbindung vom Server zum Client aufgebaut, was in der Realität für den BiBiWS Anwendungsfall nicht vorausgesetzt werden darf (Abbildung 3.3).

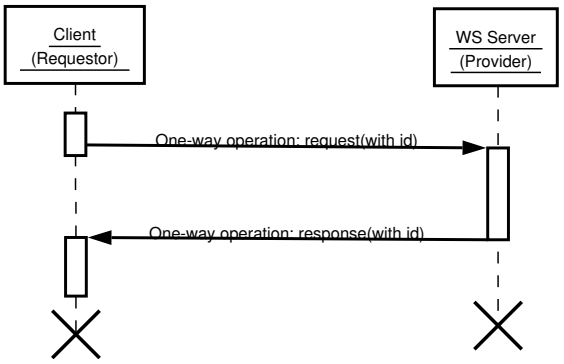

Abbildung 3.3: Asynchrone WebServices, Vorschlag 1: *One-way and notification operations*

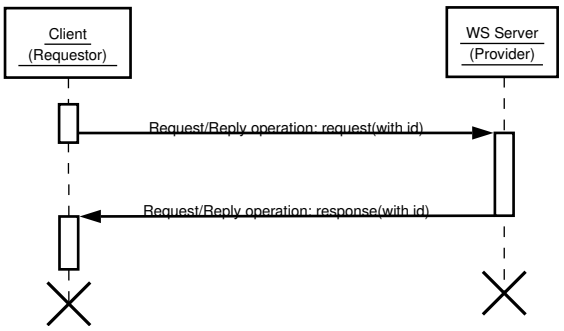

Abbildung 3.4: Asynchrone WebServices, Vorschlag 2: *Request/Reply operation*

2. Vorschlag: Request/Reply operation Hier wird eine einzelne Operation in zwei Transportnachrichten geteilt; referenziert wird wieder über eine `id`, die auf der Clientseite generiert wird. Da wiederum eine Verbindung von Server zu Client aufgebaut wird, wie im ersten Vorschlag, ist sie für den BiBiWS Anwendungsfall nicht akzeptabel. (Abbildung 3.4).

3. Vorschlag: Request/Reply operation with polling In diesem Vorschlag sendet der Server als Antwort auf die Anfrage eine `id`, mittels derer sich der Client später per *Polling* das Ergebnis abholen kann. Da nicht bekannt ist, wann das Ergebnis fertig ist, muß der Client gegebenenfalls häufiger anfragen (Abbildung 3.5).

4. Vorschlag: Request/Reply operations with posting Wie im dritten Vorschlag generiert der Server eine `id`, jedoch sendet der Server die Antwort zum Client, welcher den Empfang bestätigt. Aus dem gleichen Grund, wie bei den ersten beiden Vorschläge, ist diese Methode für BiBiWS nicht realisierbar (Abbildung 3.6).

BiBiWS basiert auf dem dritten Vorschlag: *Request/Reply operation with polling*. Dieser nimmt dem Client, der gegebenenfalls von einem unerfahrenen Programmierer geschrieben wird, möglichst viel Arbeit ab (das Generieren der `ids`) und ist mit dem

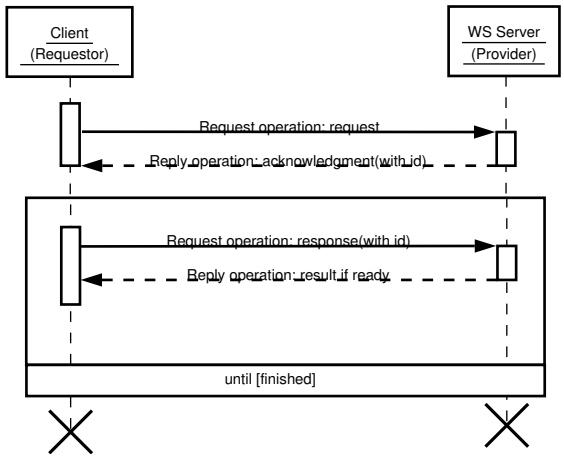

Abbildung 3.5: Asynchrone WebServices, Vorschlag 3: *Request/Reply operation with polling*

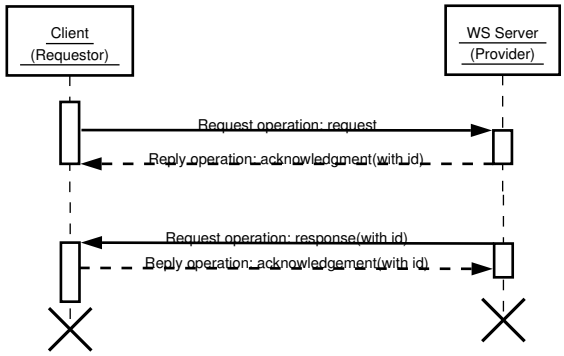

Abbildung 3.6: Asynchrone WebServices, Vorschlag 4: *Request/Reply operations with posting*

gebräuchlichen http Protokoll vereinbar. Das Generieren und Verwalten der `ids` passiert selbständig durch BiBiWS, so daß dem Entwickler auf Seiten des BiBiServ diese Arbeit abgenommen wird.

3.1.3 Documented vs. RPC-styled WebServices

WebServices unterstützen zwei grundlegend verschiedene Arten der Datenübermittlung.
Bei *documented-styled* WebServices wird ein XML Dokument in der DOM Spezifikation des W3C [DOM] übermittelt und ebenso ein Dokument zurückgeliefert.
Bei *RPC-styled* WebServices wird eine Methode über ein Netzwerk aufgerufen.
Zusammengefaßt und auf den Punkt gebracht wurde dies auf [RPCVSDOC] folgendermaßen:

„*it's the choice between send(document) vs someOperation(parameters[])*"

Beide Arten bieten keine grundsätzlichen Vorteile oder einen Mehrwert. Jedoch sind *RPC-styled* WebServices intuitiver zu verstehen und einfacher zu begreifen, da sie dem normalen Programmierverständnis entsprechen. Es werden Methoden aufgerufen, ähnlich zu lokalen Methoden.
Außerdem reicht meistens als Eingabe für die WebServices am BiBiServ nicht ein Dokument aus, wie es bei *documented-styled* WebServices vorgesehen ist. BiBiWS WebServices erhalten und liefern ihre Daten als Dokument entsprechend einem XML-Schema (wie in Kapitel 1.2 beschrieben). Zusätzliche Parameter werden getrennt von diesem Dokument übergeben. Würde ein Dokument auch diese Parameter enthalten, wäre ein validierendes XMLSchema auf eine Anwendung speziallisiert und damit könnten die Ziele des HOBIT-Projektes nicht erreicht werden.

Aus diesen Gründen unterstützt BiBiWS *RPC-styled* WebServices. Zusätzlich entspricht ein Aufruf einer Methode auf einem entfernten Rechner eher der Anwendung auf dem BiBiServ. Im Endeffekt werden *Documented-styled* WebServices aber durch BiBiWS ebenso möglich zu entwickeln, die Beispielanwendungen jedoch beschränken sich auf die *RPC-styled* WebServices.

3.2 Software Pakete

3.2.1 AXIS als SOAP Library

Die AXIS Library als SOAP Implementierung nach der W3C „Standard“ [2] [SOAP] stellt eine der Hauptfunktionen dar, mittels der BiBiWS die nahezu völlige Abstraktion von der WebService Technologie vollzieht.
AXIS ist in der Lage die WSDL Definition der WebServices aus dem Source Code zu erzeugen. Außerdem existiert die Möglichkeit, unabhängige WebServices zu erzeugen, wie in Kapitel 2.3.1 beschrieben. Zudem ist AXIS mit dem Konzept der Interfaces seit einigen Versionen stabil. Es implementiert die Interface Pakete JAX-RPC und SAAJ, so daß schon während der Entwicklung das Update von Version 1.1 zu 1.2beta problemlos vollzogen wurde.

Im Verhältnis zu anderen SOAP Implementationen ist der auf AXIS aufbauende Source Code relativ unabhängig von AXIS.
Beispielsweise ist BiBiWS lediglich bei der Umsetzung der `soap_faults` direkt von der AXIS Implementierung abhängig. Die AXIS Library liefert dann an den Client einen `soap_fault` zurück, wenn eine Exception aufgetreten ist, die die AXIS Exception `AxisFault` erweitert. Dieses ist ein sehr elegantes Verfahren, da so das Konzept der Java Exceptions beibehalten wird, jedoch ist hierdurch der geschrieben Code an die Axis Library gebunden.
Da für die Umsetzung des Standards durch die unterschiedlichen Paradigmen der

[2] genauer: „Recommendation“, da der W3C keine Standards setzt, sondern diese selbst nur als Empfehlung bezeichnet. Jedoch wird im Normalfall von Spezifikation gesprochen.

einzelnen Programmiersprachen naturgemäß unterschiedliche Realisierungen auftauchen, läßt sich eine solche Abhängigkeit kaum vermeiden.
Leider wirft die AXIS Library auch Probleme auf:
Zum einen scheint ein Performanceproblem zu immensem Speicherbedarf beim Verpacken von Daten in das Protokoll zu entstehen (siehe Kapitel 6).
Zum anderen war es aus unerklärlichen Gründen nicht möglich, die oben beschriebene Exception in die BiBiWS Library selbst zu integrieren. So muß jedes Werkzeug auf dem BiBiServ eine eigene Exceptionklasse beinhalten. Dies stellt kein wirkliches Problem dar, schließlich muß der Entwickler an dieser Klasse nichts ändern, jedoch ist es grundsätzlich wünschenswert, auch Exceptions aus Libraries zu benutzen.
Die Entwickler des AXIS Projektes wurden über diese beiden Probleme informiert.

Auf Grund der weiten Verbreitung, der mitgelieferten Hilfsprogrammen und der optimalen und empfohlenen Kombination mit dem Tomcat Server, gibt es für BiBiWS keine andere Wahl als die AXIS Library.

3.2.2 Jakarta Tomcat als Servlet Container

Für die Verwendung des *Apache Jakarta Tomcat* Projekt (kurz: *Tomcat*) der Apache Software Foundation sprechen einige Punkte.
Zum einen gibt es keine wirkliche Alternative zu diesem Produkt. Der Tomcat hat mehr Features als BiBiWS nutzt, jedoch stören diese Erweiterungen in keiner Weise. Der Tomcat ist, wie auch die Serverseite von BiBiWS, in Java geschrieben, ergänzen sich diese auf vorbildliche Weise.
Zum anderen ist die schon mehrfach erwähnte Verwandschaft mit der AXIS Library wichtig. Hier wird durch eine breite Benutzer- und Entwickler Gruppe die Kompatibilität der beiden Programme sichergestellt. So empfiehlt das AXIS Projekt ausdrücklich den Einsatz eines Tomcat Servers.
Ein nicht unwesentlicher Grund ist auch die Unterstützung der Firma Sun Microsystems für den Tomcat und die damit leichtere Administration von Tomcat durch die Rechnerbetriebgruppe der *Technischen Fakultät der Universität Bielefeld*.
Sun Microsystems liefert mit der Version 10 ihres Betriebssystems den Tomcat zusammen mit einer Eigenentwicklung als Alternative aus. Auch diese Alternative, der Sun WebService Server, wurde untersucht, aber die Kombination Tomcat mit AXIS ist ausgereifter und vor allem gebräuchlicher und besser dokumentiert.

BiBiWS ist wiederum nicht an den Tomcat gebunden - jeder andere WebService Server ist ebenso geeignet, da die mit BiBiWS entwickelten WebServices nur einen Java Servlet Container nach der Spezifikation [JSC] benötigen.

3.2.3 http als Transferprotokoll

http [HTTP] ist zur Zeit das einzige in der Praxis einsetzbare Protokoll für WebServices. Dieses zieht einige Nachteile nach sich. Zum Beispiel die fehlende Unterstützung von Asynchronität und der „Mißbrauch" eines Protokolls, das eigentlich für das WWW konzipiert war.
Diese Nachteile werden jedoch durch die allgemeine Verfügbarkeit von http mehr als wieder wettgemacht, da es von den meisten Firewalls nicht geblockt wird, was automatisch einen Großteil der möglichen Anwender ausschließen würde.
Außerdem bauen sämtliche Dokumentationen, die man im Internet zu WebServices findet, wie auch die aktuelle Spezifikation von SOAP, auf http auf.

In Zukunft werden andere Protokolle auf den Markt kommen, beispielhaft sei hier BEEP [ROSE] genannt, das sich im Entwicklungsstadium befindet.
Sollte eines dieser Protokolle wesentliche Vorteile bringen und sich durchsetzen, vielleicht sogar die SOAP Spezifikation entsprechenden Erweiterungen beinhalten, so muß über einen Austausch von http nachgedacht werden.
Da BiBiWS aber nicht direkt auf http aufbaut, sondern die Vermittlung von AXIS übernommen wird, kann ein möglicher Austausch ohne Änderungen an BiBiWS erfolgen.

3.2.4 Apache http Server als Webserver

Zu diesem renommierten Webserver gibt es aus zwei Gründen keine Alternative für BiBiWS:
Zum einen ist er, wie schon in Kapitel 2.6 beschrieben, der am meisten eingesetzte und am weitesten verbreitetste Webserver, womit er auch neue Standards und ein schneller Support bei Sicherheitsproblemen und allgemeinen Problemen gegeben sind.
Zum anderen basiert der BiBiServ seit Gründung auf diesem Webserver. BiBiWS fügt sich in diese Umgebung ein, womit auch einhergeht, daß die Clientseite von BiBiWS auf dem Apache http Server läuft.

Prinzipiell ist BiBiWS jedoch nicht genau an diesen Webserver gebunden. Sobald ein Webserver CGIs in Perl ausführen kann und entsprechende Perl Module installiert sind (siehe Anhang B), sind die Anforderungen von BiBiWS erfüllt.

3.2.5 Die *Sun Grid Engine*

Immer häufiger überschreiten die Anforderungen im Bereich der Bioinformatik die Leistungen einzellner Rechner. In Bereichen, wie dem BiBiServ, wo unabhängige Aufrufe von Programmen erhöhte Rechenzeit benötigen, bietet ein Verteilen der einzellnen Programmaufrufe sich an.
Alternativen im Bereich *Grid Computing* gibt es einige.
Allerdings ist dieses Thema sehr betriebssystemnah, und die Firma *Sun Microsystems*,

von der die an den BiBiServ angeschlossenen Computer sind, unterstützt und entwickelt die *Sun Grid Engine* [SGE], welche auch frei verfügbar ist.
Auch wenn diese einige Probleme bereitet, wie die bisher fehlende Java API, so ist die *Sun Grid Engine* ein zuverlässiges Produkt im Bereich des Grid Computing. Die in der Technischen Fakultät der Universität Bielefeld installierte *Sun Grid Engine* wird außerdem noch von anderen Benutzern als dem BiBiServ genutzt.

Vor allem die Rechnerbetriebsgruppe hat eine Entscheidung zu Gunsten der *Sun Grid Engine* schon vor längerer Zeit aufgrund der guten Verknüpfung zu dem Betriebssystem *Solaris* getroffen, so daß sich BiBiWS an dieser Stelle anpassen mußte.
Allerdings kapselt BiBiWS das Starten externer Programme in eine einzige Klasse `SGECall`, so daß nur eine Alternative zu dieser Klasse geschrieben werden können, um andere Grid Engines zu unterstützen.
So existiert zum Beispiel bereits eine Klasse `LocalCall`, die transparent zu `SGECall` genutzt werden kann, um Programmaufrufe direkt auf dem WebService Server zu starten. Diese wurde während der Entwicklung von BiBiWS benötigt und sollte **nicht** benutzt werde, weil die daraus folgende Last auf dem WebService Server zu Problemen führen kann.

3.3 Probleme, Details und grundsätzliche Ideen von BiBiWS

BiBiWS basiert auf den oben vorgestellten Softwarepaketen, aber auch auf den Erfahrungen, die das BiBiServ Administrator Team in den letzten Jahren gesammelt hat.
Außerdem bedingen einige Details, wie das Fehlen von Asynchronität in WebServices und das Unterstützen der Standards des HOBIT-Projektes, nach besonderen Lösungen.
Im Folgenden wird auf genau solche Details eingegangen.

3.3.1 Properties Dateien

Alle für sämtliche BiBiWS WebServices gleichen Parameter, wie beispielsweise der Ort des Spoolverzeichnisses, wurden in eine Properties Datei (`bibi.properties`) ausgelagert. Diese kann einfach geändert werden und wird von allen WebServices gemeinsam genutzt, was den administrative Aaufwand reduziert. Außerdem enthält diese Datei Definitionen, an die sich alle BiBiWS WebServices halten, wie zum Beispiel die Statuscodes (siehe Anhang C) mit ihren Beschreibungen (detailiert in Anhang B.2.1).
Außerdem hat jeder WebService für sich eine Properties Datei (`tool.properties`), die BiBiWS in die Lage versetzt, sich Zugriff auf benötigte Informationen des WebServices zu erhalten, um ihre Dienste spezifisch für einen WebService zu erledigen.
Eine Alternative wäre gewesen, diese Informationen in dem Konstruktor der BiBiWS Klassen zu übergeben, was aber wesentlich umständlicher wäre (detailiert in Kapitel A.5.1).
Properties Dateien haben außerdem den Vorteil, daß sie nicht zum Zeitpunkt des

Kompilierens eingebunden werden. Es ist also möglich, Änderungen der Properties Dateien ohne erneutes Kompilieren des WebServices zu übernehmen.
Auf beide Properties Dateien bietet die Klasse `WSSTools` einen einfachen Zugriff.

3.3.2 Datenbank

BiBiWS benutzt eine Postgresql Datenbank zum Speichern der Zustände der WebServices, druch die die `response()` Methode zu jeder Zeit Informationen über einen Status abfragen kann.
Im Prinzip kann jedoch jede SQL Datenbank benutzt werden, da nur *SELECT* und *UPDATE* Befehle einer Tabelle ausgeführt werden. In dieser Tabelle sind pro Aufruf eines WebServices die folgenden Informationen gespeichert:

id Die eindeutige `id` zum Referenzieren des Aufrufs

toolname Name des Werkzeugs

statuscode Statuscode des Aufrufs als Integer

description Beschreibung des Statuscodes

internalDescription Interne Beschreibung des Statuscodes

sgeId Die JobID des *Sun Grid Engine* Aufrufs

lastMod Zeitpunkt der letzten Modifikation des Statuses

createDate Zeitpunkt an dem der `request()` Aufruf geschah

3.3.3 Statuscodes

Die Statuscodes stellen den Zustand von einem Aufruf eines WebServices dar.
In Anhang C befindet sich eine Erklärung und Übersicht der vordefinierten Statuscodes.
Diese werden mit weiteren Informationen in Status Objekten gehalten, welche auf Client- und Serverseiten existieren.
Auf Serverseite erzeugt der Konstruktor eines Status Objektes beim Aufruf von `request()` eine eindeutige `id`, die als ein Eintrag in der Datenbank gespeichert werden. Sobald Änderungen an dem Status Objekt geschehen, werden auch diese in der Datenbank gespeichert. Ein Aufruf der `response()` Methode kann ab diesem Zeitpunkt Informationen über den Status des Vorschritts geben. Der Status wird mittels der `id` referenziert und aus der Datenbank geladen.
Auf Clientseite stellt der Statuscode die Antwort oder die Rückgabe der `request()` Methode (liefert `id`) oder `response()` Methode (liefert `statuscode`, `description`) dar.
Sollten clientseitig Fehler auftreten, werden auch diese in dem Status Objekt vermerkt. Der `statuscode` beginnt in diesem Fall mit einen `c`.

3.3.4 Spoolverzeichnis

Wie schon bei den klassischen Werkzeugen auf dem BiBiServ bekommt jeder Aufruf eines Werkzeugs bei der BiBiWS Realisierung ein eigenes Spoolverzeichnis. Dadurch wird der Entwickler von der Arbeit entbunden, sich um Phänomene zu kümmern, die auftreten, wenn mehrere Benutzer gleichzeitig ein Werkzeug benutzen. Hierbei könnte es dazu kommen, daß sich gleichzeitig aufgerufene Programminstanzen gegenseitig Dateien überschreiben. Solange ein Programm alle seine Dateien in dieses speziell angelegte Verzeichnis schreibt, ist dieses Problem mit seperaten, eindeutigen Spoolverzeichnissen gelößt.

3.3.5 Result Caching auf Clientseite

Der BiBiWS Client überprüft, ob das Resultat schon von dem WebService Server geholt wurde und auf dem Apache http Server gespeichert wurde. In diesem Fall zeigt es in diesem Fall direkt an. Durch diesen Art *Result Caching* Mechanismus wird Last von dem WebService Server genommen, wenn ein Benutzer die Resultate eines Aufruf mehrmals über das HTML Interface abruft.

3.3.6 WebService Methoden `_orig`

Die klassische Ein- und Ausgabe der Werkzeuge auf dem BiBiServ sind in dem jeweiligen Ein- und Ausgabeformaten der Kommandozeilenaufrufe gehalten. Da in dem HOBIT-Projekt WebServices auch untereinander automatisiert verknüpft werden sollen, reichen diese Ausgaben nicht aus. Die Eingabe der `request()` Methoden und die Rückgabe der `response()` Methoden erfolgt deswegen in einem Werkzeug unabhängigen Format. Die Daten müssen dabei konvertiert werden, damit die Programme nicht geändert werden müssen, was in der Pre- und Postprocessing Phase geschieht.
Auf die Standards für diese Datenrepräsentation einigt sich das HOBIT-Projekt. Dabei wird auf XMLSchemas zurückgegriffen, die durch andere Projekte entwickelt wurden oder noch werden. Es wird versucht diese XMLSchemas so zu selektieren, daß auf der einen Seite sämtliche Daten der Programme repräsentiert werden, jedoch trotzdem keine Überspezifikation erfolgt und die XMLSchemas allgemein genug gehalten sind um mehreren Programmen als Ein- und Ausgabe zu dienen.
Durch dieses Vorgehen entsteht naturgemäß das Problem, daß nur die Schnittmenge zwischen Ausgabe des einen Programms, dem XMLSchema und der Eingabe des anderen Programms übermittelt werden kann.

Es gibt einige Programme, die die Ausgabe eines anderen Programms direkt als Eingabe akzeptieren. Hierbei sollte natürlich eine zwischenzeitige Repräsentation der Daten in einem XMLSchema entfallen.
Für diesen Fall schlägt BiBiWS zusätzliche Methoden `request_orig()` und

response_orig() vor, die prinzipiell das gleiche darstellen, allerdings keine Konvertierung machen, also die Daten „original“ dem Programm übergeben/zurückliefern.

3.3.7 Implementation der Asynchronität mittels *Request/Reply operation with polling*

Die request() Methoden liefern, wie schon beschrieben und von dem Ansatz *Request/Reply operation with polling* verlangt, eine id, mittels derer der aktuelle Status der Berechnungen oder das Resultat abgefragt werden kann bzw. zurückliefern.

Wenn ein Benutzer einen WebService direkt benutzt, wird er dieses Polling mittels einer While Schleife (oder Ähnlichem) in seinem Client realisieren. Ein entsprechendes Code Fragment ist in Abbildung 3.7 zu sehen.
Für die Clientseite, das http Interface zu einem BiBiWS WebService, wird nun ein

```
1  while($statuscode != 600) {
2    $result = SOAP::Lite->service($URI)
3                        ->on_fault(sub {$statuscode = $_[0]->faultcode()})
4                        ->response($id);
5    wait 5; #for no DoS Attack against WS Server
6  }
```

Abbildung 3.7: Perl Code Fragment für Abfrage eines WebServices per Polling

Mechanismus benötigt, der zum einen regelmäßig den Status abfragt und zum anderen den Benutzer in dessen Browser auch über die Fortschritte informiert.
Dieses realisiert BiBiWS mittels eines HTML-redirect Befehls.
Der Browser des Benutzers wird hierbei dazu gebracht, das selbe CGI in regelmäßigen Abständen aufzurufen. Dieses CGI (<PN>_redirect) fragt bei jedem Aufruf nun den Status des Aufrufs bei dem WebService Server ab, um diesen dem Benutzer zurückzuliefern. Sollte der Statuscode 600 sein und damit das Resultat vorliegen, wird statt des HTML-redirect Befehls das Resultat angezeigt.

3.3.8 *Notification by e-mail*

Zwar sind Aufrufe auf dem BiBiServ auf 24 Stunden Rechenzeit begrenzt, jedoch ist auch dieses gelegentlich zu lange, um einen Browser offen zu halten (beim Benutzen des Web Frontends) oder um ein Polling laufen zu lassen (bei direktem Zugriff auf die WebServices). Dafür unterstützen die WebServices eine *Notification by email* Technik. Der Benutzer kann optional eine E-Mail Adresse angeben und bekommt eine Benachrichtigung, sobald die Berechnung abgeschlossen ist. In dieser Mail ist sowohl ein Hyperlink zum Abfragen der Ergebnisses über das HTML-Interface des BiBiServ, wie auch die id um die Ergebnisses direkt abzufragen.

Dadurch wird das Polling über mehrere Stunden hinweg überflüssig. So muß ein Benutzer des http Interfaces das Browserfenster mit dem HTML-redirect nicht geöffnet halten und ein Benutzer, der die WebServices direkt anspricht, müß keine While Schleife benutzen um das Polling zu realisieren.

4 BiBiWS

Die BiBiWS Library besteht aus Server und Clientseite, mit jeweils mehreren Klassen. In diesem Kapitel werden die Zusammenarbeit und der Ablauf dieser Klassen von einem BiBiWS WebService erläutert.

Sämtliche Diagramme und Beschreibungen orientieren sich an dem Beispiel WebService *TemplateWS* und sind teilweise abstrahiert, da eine detaillierte Beschreibung hier vom Umfang her nicht möglich ist. Insbesondere das Verhalten bei Fehlern ist hier nicht aufgeführt, da es prinzipiell immer gleich abläuft (setzen des Statuscodes auf einen Wert mit `7xx`) und die Erklärung stark verkomplizieren würde.
Eine Erklärung, die durch den Source Code führt, findet sich im *Developers Guide* (Anhang A), wodurch eine einfachere Entwicklung ermöglicht wird. Dort wird natürlich auch jeder einzelne Schritt beschrieben.

Ein BiBiWS WebService unterscheidet zwei Möglichkeiten Daten beim Aufruf zu verschicken und beim Abruf des Ergebnisses zu erhalten. Entweder befinden sich die Daten in dem Ein- bzw. Ausgabeformat des Tools (`_orig` Methoden; siehe Kapitel 3.3.6), oder sie werden in ein anderes dem HOBIT-Projekt Standard entsprechendes Format konvertiert.
Die hier beschriebenen Schritte mit einer solchen Konvertierung fallen gegebenenfalls also einfach weg. Ansonsten sind die Abläufe gleich.

Eine Übersicht mit genauer Beschreibung und Funktionsumfang der BiBiWS Library kann in Kapitel 5 nachgelesen werden.
Die verwendeten Statuscodes sind als Übersicht tabellarisch in Anhang C dokumentiert.

4.1 Der Ablauf auf Serverseite

Die Clientseite ruft als erstes die `request()` Methode auf, um dann mittels eines *Pollings* die `response()` Methode aufzurufen, bis das Ergebnis vorliegt.
Somit sind die beiden serverseitigen Methoden unabhängig zu betrachten.

4.1.1 Ein typischer `request()` Ablauf

Das Sequenzdiagramm Abbildung 4.1 zeigt den Ablauf eines `request()` Aufrufs von einem BiBiWS WebService.

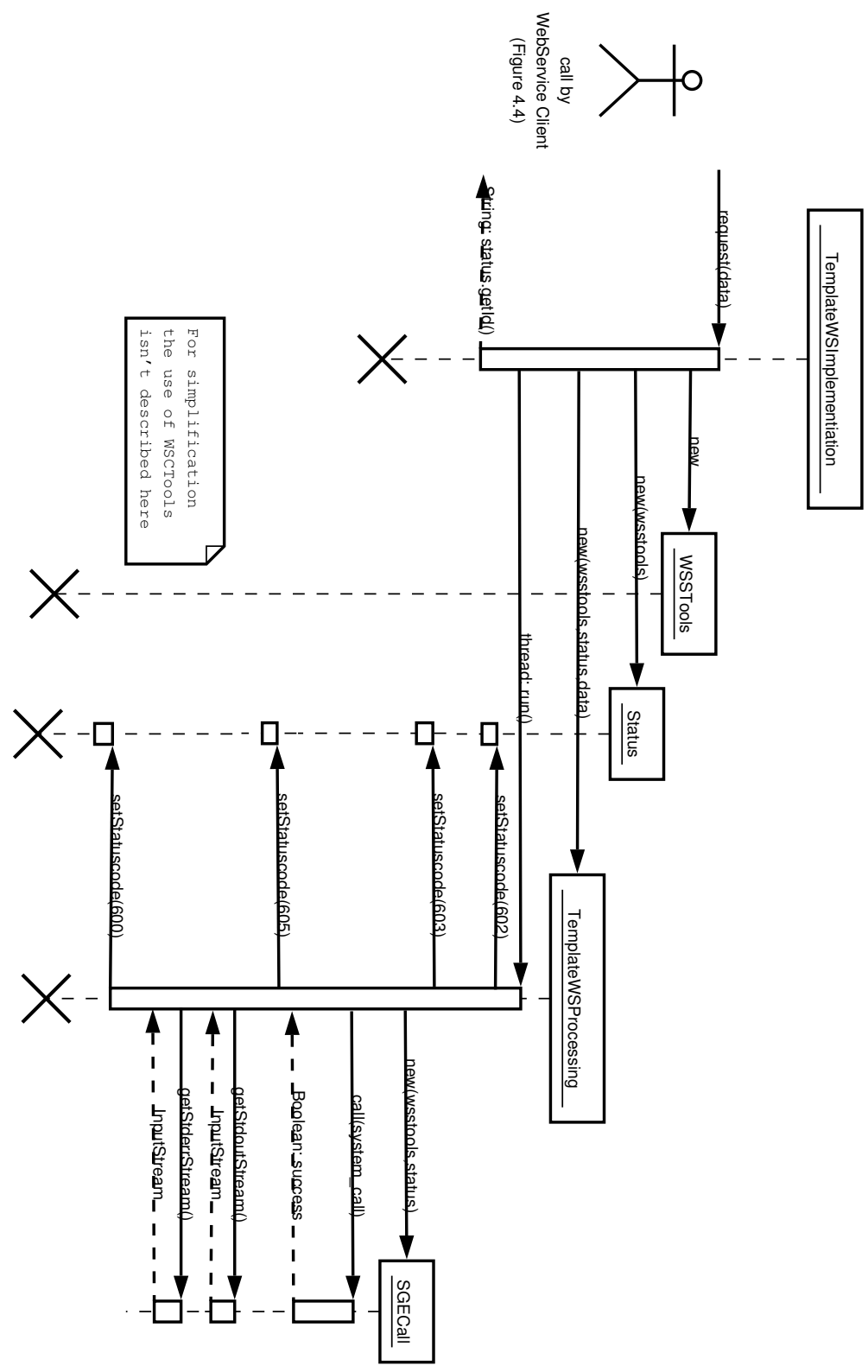

Abbildung 4.1: Ablauf eines Aufrufs von `request()`

Die Implementationsklasse des WebService beinhaltet die `request()` Methode. Als erstes erzeugt sie jeweils ein `WSSTools` und ein `Status` Objekt der BiBiWS Library. Sollte der Benutzer eine E-Mail Adresse angegeben haben, kann nun mittels `checkemail()` des `WSSTools` Objektes eine erste Überprüfung erfolgen, ob die Domain der E-Mail Adresse existiert.
Durch den Konstruktor des `Status` Objektes wird ein neuer WebService Aufruf eröffnet und in die Datenbank eingetragen. Nachdem `request()` ein Objekt der Processingklasse erzeugt hat und diese als Thread im Hintergrund gestartet hat, kann es die generierte `id` des Status Objektes zurückliefern, über die der Benutzer später das Ergebnis abfragen kann.
Den eigentlichen Ablauf ist in der Processingklasse:
Sie durchläuft die drei Zustände eines BiBiWS WebService Aufrufes: *Preprocessing*, *Processing*, *Postprocessing*.

Nach dem Setzen des entsprechenden Statuscodes (602) in dem Status Objekt (und der damit verbundenen Aktualisierung der Datenbank) kann nun eine gegebenenfalls nötige Vorverarbeitung (*Preprocessing*) vor dem eigentlichen Programmaufruf erfolgen.
Im Gegensatz zu `request_orig()` gehört bei der Methode `request()` auch das Konvertieren der Eingabedaten aus dem übermittelten Format in das Eingabeformat des Programms. Außerdem können in diesem Schritt die Eingabedaten mittels der Methode `writeSpoolFile()` aus `WSSTools` in das Spoolverzeichnis geschrieben werden, falls das Programm auf Daten aus einer Datei zugreift.
Das Parsen der einkommenden Daten wird durch die Methoden `makeNiceSeq()` und `seqInfo()` des `WSSTools` Objektes unterstützt.
Das gesammte *Preprocessing* sollte so schnell und klein wie möglich gestaltet werden, da es Rechenleistung und Speicher auf dem WebService Server belegt.
Für komplexere Aufgaben kann hier ein Aufruf über das *Computing Grids* erfolgen. Dies benötigt allerdings ein eigenständiges Programm, so aber nicht auf dem WebService Server läuft, sondern verteilt wird.
Der nächste Schritt in dem Ablauf der Processingklasse trägt den Statuscode 603 (*Pending*) bzw 604 (*Running*). Das Setzen der Statuscodes wird hier von der `SGECall` Klasse übernommen. Unter diesen Statuscodes wird der eigentliche Programmaufruf über das Computing Grid *Sun Grid Engine* gestartet. Eine Computing Grid, wie die *Sun Grid Engine*, verteilt die eingehenden Jobs auf den ihr angeschlossenen Rechnern und sorgt so für eine minimale Laufzeit.
Dafür legt die Processingklasse ein `SGECall` Objekt an, welches durch den Aufruf von `call()` den Programmaufruf der *Sun Grid Engine* übergibt. `call()` beendet sich erst, wenn die Berechnungen beendet sind. So kann ein einfacher sequentieller Ablauf der drei Schritte erfolgen.
Das *Postprocessing* mit Statuscode 605 ist nun dafür zuständig, die `InputStreams` des `SGECall` Objektes weiter zu verarbeiten und das Resultat in einem geeigneten Format in das Spoolverzeichnis zu schreiben. Alle Daten, die später bei einem `response()` Aufruf zurückgeliefert werden sollen, müssen in das Spoolverzeichnis geschrieben

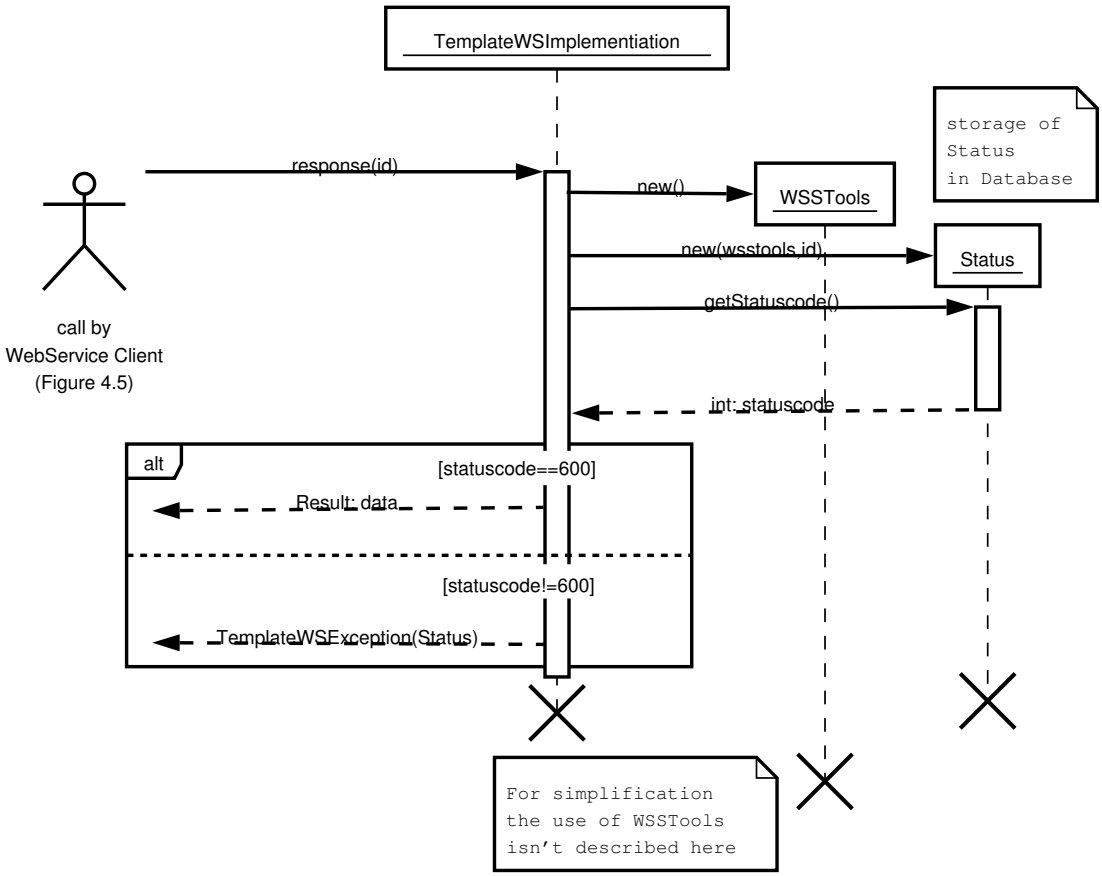

Abbildung 4.2: Ablauf eines Aufrufs von `response()`

werden, da `response()` sonst keinen Zugriff auf diese Daten hat.
Dafür stellt das `WSSTools` Objekt einige nützliche Methoden bereit, wie das schon erwähnte `writeSpoolFile()`.
Abschließend wird der Statuscode auf `600` gesetzt, was symbolisiert, daß die Berechnungen abgeschlossen sind und eine Anfrage des Benutzers zur Rückgabe des Ergebnisses führen soll. Sollte der Benutzer eine E-Mail Adresse angegeben haben, so kann er eine Benachrichtigung über den Abschluß der Berechnungen erhalten. Auch hierfür stellt `WSSTools` die Methode `mail()` bereit.

4.1.2 Ein typischer `response()` Ablauf

Im Sequenzdiagramm Abbildung 4.2 wird der typische Ablauf einer Ergebnisanfrage veranschaulicht.
Dabei fragt der Benutzer mittels der von `request()` zurückgelieferten `id` nach einem Ergebnis und bekommt entweder einen Statuscode oder das Resultat geliefert.

Dafür erzeugt ein Aufruf der `response()` Methode als erstes ein `WSSTools` Objekt gefolgt von einem `Status` Objekt. Im Gegensatz zu dem `request()` Aufruf wird diesmal allerdings im Konstruktor zusätzlich die `id` übermittelt, woraufhin `Status` den Status aus der Datenbank lädt.
Sollte der Statuscode `600` (*finished*) entsprechen, kann das Resultat mittels den von `WSSTools` bereitgestellten Methoden `readSpoolFile()` oder `readAllSpoolFiles()` gelesen werden. Diese Daten müssen nun noch gegebenenfalls in das gewünschte Format konvertiert werden. Das Ergebnis kann nun dem Benutzer mitgeteilt werden.
Sollte der Statuscode nicht `600` sein, ist entweder die Berechnung noch nicht abgeschlossen (Statuscode ist `6xx`) oder es ist ein Fehler aufgetreten (Statuscode ist `7xx`). In beiden Fällen wird eine Exception zurückgeliefert, die die AXIS Library dazu veranlaßt, einen `soap_fault` mit Beschreibung (`soap_description`) zu erzeugen und dem Benutzer als Antwort zu liefern.
Dadurch ist der Benutzer über den Stand der Dinge informiert und kann entscheiden, ob eine spätere Anfrage Sinn hat (Statuscode `6xx`) oder nicht (Statuscode `7xx`).

4.2 Der Ablauf auf Clientseite

Auf Clientseite wird durch das Abschicken des `submission.html` Formulars das entsprechende `<PN>_submit` CGI des Projektes aufgerufen. Dieses startet den WebService auf der Serverseite mittels Aufruf von `request()` und sorgt mit Hilfe des `redirect` CGIs in regelmäßigen Abständen für eine neue `response()` Anfrage bis das Resultat vorliegt.

4.2.1 Ein typischer Ablauf des `submit` CGIs

Die Abbildung 4.4 zeigt, was ein Aufruf über die `submission.html` auslöst.
Als erstes werden dabei Plausibilitätsüberprüfungen stattfinden, ob die Eingabe des Benutzers Sinn macht. Gegebenenfalls kann so ein Aufruf der Serverseite vermieden werden. Allerdings müssen diese Überprüfungen sehr einfach gehalten werden, da sie auf dem Webserver Resourcen verbrauchen.
Da die WebServices auch direkt von außen aufgerufen werden können, macht eine detaillierte Überprüfung auf Webserverseite auch keinen Sinn – auf WebService Serverseite müssen sämtliche einkommenden Daten auf jeden Fall überprüft werden.

Nach dem Anlegen der drei Objekte `WSCTools`, `WSCLayout` und `Status` wird das toolspezifische Objekt des Paketes `WSC::<PN>` geladen, welches den WebService Server kontaktiert.
Dieses Objekt ruft nun unter Verwendung des Perl Moduls `SOAP::Lite` die serverseitige `request()` Methode auf und startet so den WebService Aufruf.
Die dafür benötigte WSDL Definition wird von dem Webserver geladen.
Die zurückgelieferte `id` wird in dem Status Objekt gespeichert und der Aufruf der

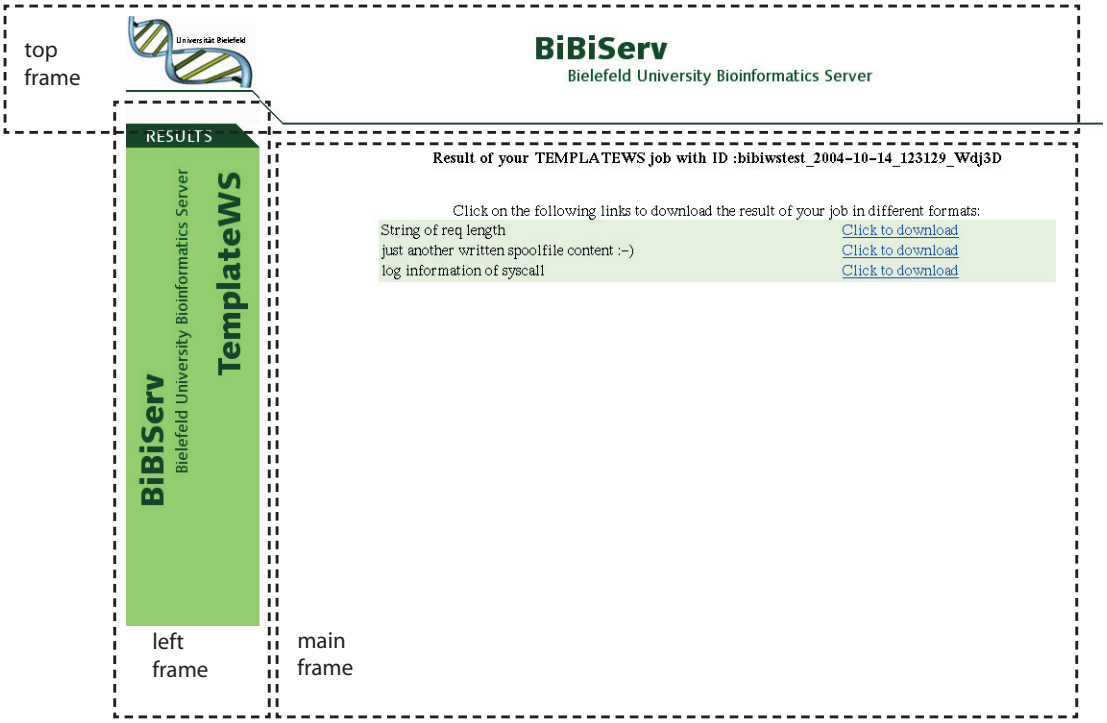

Abbildung 4.3: Eine BiBiServ Ergebnisseite

Methode `mainFrame()` des Objektes `WSClayout` sorgt für die Anzeige einer Frameumgebung im Browser des Benutzers.
Dabei werden der obere Frame (*top frame*) und linke Frame (*left frame*) direkt von der Methode erzeugt, der Hauptframe (*main fram*) jedoch entsteht durch den Aufruf des CGIs `<PN>_redirect` (siehe Abbildung 4.3).

4.2.2 Ein typischer Ablauf des `redirect` CGIs

Nachdem das CGI `<PN>_submit` den WebService gestartet und den grundsätzlichen Rahmen an den Benutzer geliefert hat, sorgt das `<PN>_redirect` CGI (Abbildung 4.5) für ein *Polling* des WebServices, bis das Ergebnis vorliegt oder ein Fehler gemeldet wird.
Dafür erzeugt es als erstes ein Objekt vom Typ `WSCTools`, bevor es ein Status Objekt erzeugt und die `id` des Aufrufs in diesem speichert.
Vor der eigentlichen Abfrage des aktuellen Status mittels des projektspezifischen Objektes `WSC::<PN>` wird noch ein Objekt `WSCLayout` benötigt sowie überprüft, ob das Ergebnis nicht schon in dem entsprechenden Spoolverzeichnis gespeichert ist. Dies ist eine Art *Result Caching* Funktion, die bei mehrmaligem Abrufen des Ergebnisses vom Benutzer verhindert, unnötigerweise mehrfach das Resultat vom WebService Server

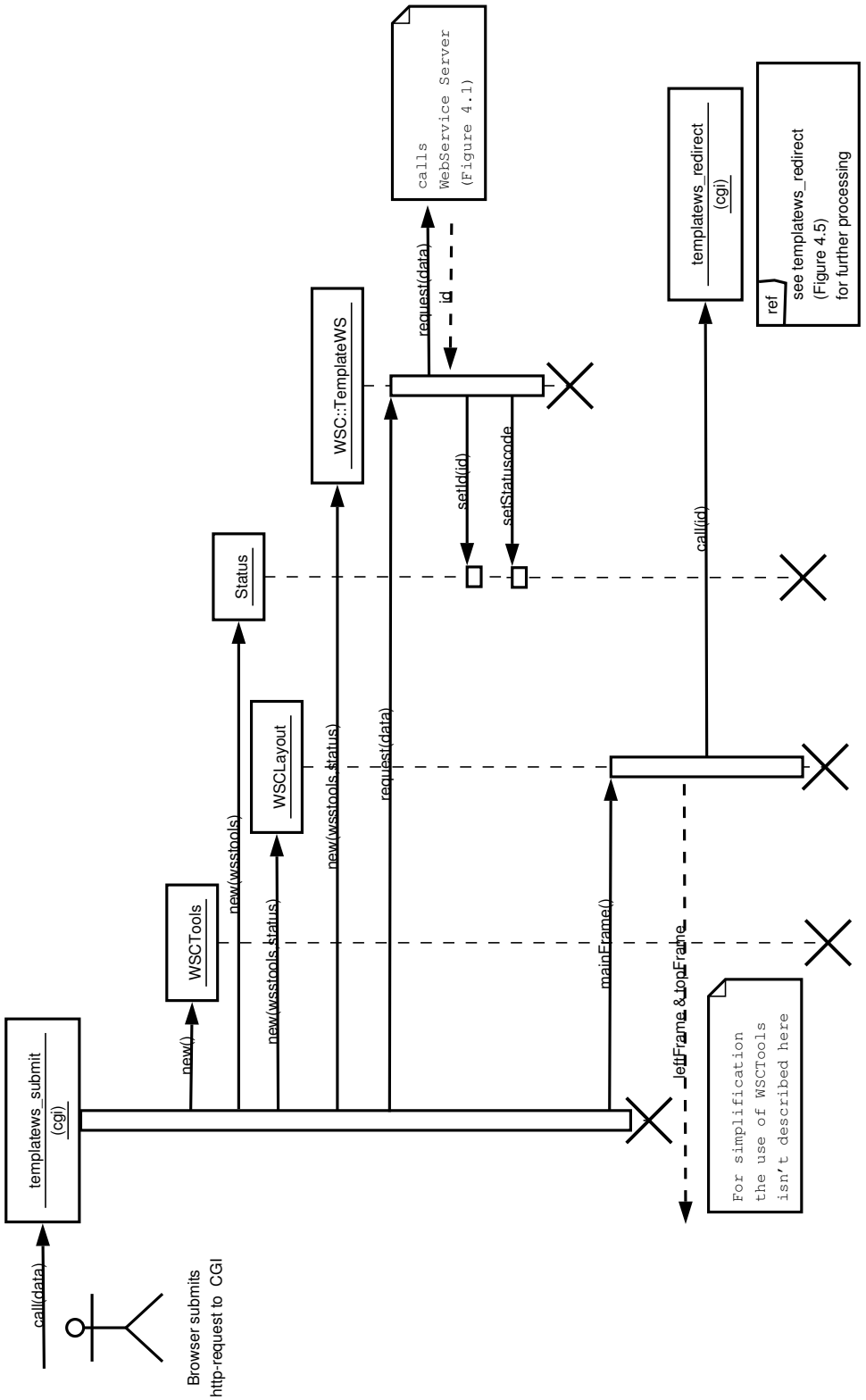

Abbildung 4.4: Ablauf eines Aufrufs von <PN>_submit

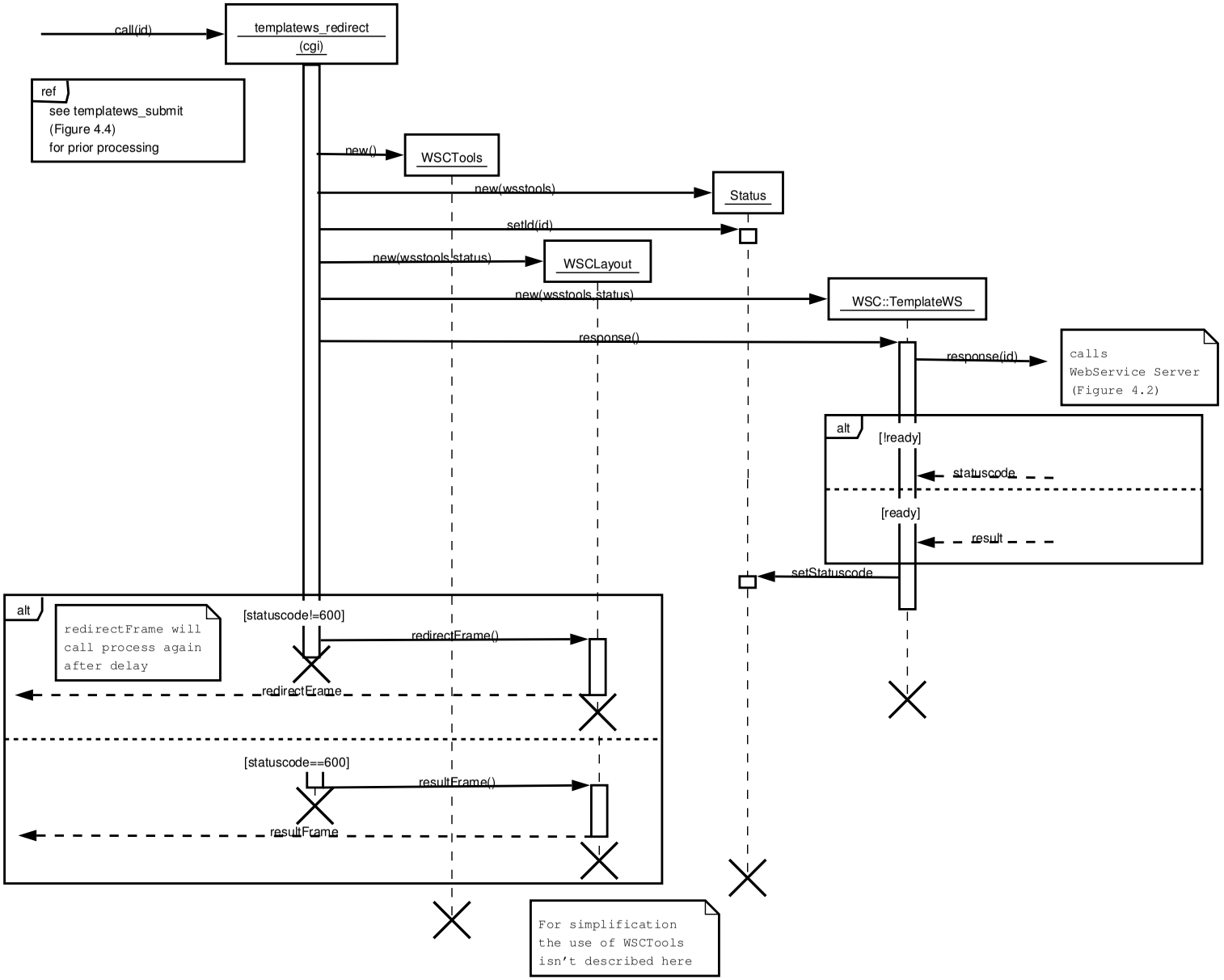

Abbildung 4.5: Ablauf eines Aufrufs von `<PN>_redirect`

zu holen.[1]
Der Aufruf der `response()` Methode mit `id` kann nun zwei Antworten liefern: Entweder antwortet der WebService Server mit einem `soap_fault`. Dies ist der Fall, wenn der WebService noch nicht beendet oder ein Fehler aufgetreten ist. Die weitere Möglichkeit ist, falls der WebService seine Berechnungen abgeschlossen hat, das Resultat zurückzuliefern, woraufhin der Statuscode auf `600` gesetzt wird.
Für den ersten Fall wird nun unterschieden, ob der Statuscode `6xx` entspricht, was zu einer Anzeige eines *redirect Frames* führt, oder ob er `7xx` entspricht, was dazu führt, daß der Benutzer den Fehler mit einer Beschreibung angezeigt bekommt. Damit der Benutzer visuell deutlich gezeigt bekommt, daß ein Fehler aufgetreten ist, wird das linke Frame in Orange einfärbt.
Im Fall, daß das Resultat von dem WebService Server zurückgeliefert wurde, werden die Daten nun in das clientseitige Spoolverzeichnis geschrieben. Eine vordefinierte Methode `resultFrame()` in dem `WSCTools` Objekt kann benutzt werden um eine einfachen Hash von Datei (`key`) und Beschreibung (`value`) anzuzeigen.
Bei vielen Programmen wird es aber Sinn machen, eine eigene Methode zu entwickeln, die das Resultat ansprechender visualisiert.

[1] nicht in der Abbildung dargestellt

5 BiBiWS im Detail

Nachdem im Kapitel 4 neben der Idee von BiBiWS die Zusammenhänge und die Verwendung der Libraries erläutert wurden, enthält dieses Kapitel eine Übersicht aller Klassen von BiBiWS.
Zu weitergehenden Informationen über die Verwendung der Libraries sei hier auf das vorherige Kapitel und das *Developers Guide* in Anhang A hingewiesen.
Einige Funktionen, die nützlich und hilfreich sind, wurden bisher nicht erklärt. Sie sind für das Verständnis nicht erforderlich, um den Fokus auf essientiellen Askepten zu belassen oder werden in der Beispielanwendung nicht gebraucht.

5.1 Die BiBiWS Server Library

Serverseitig sind die Klassen in dem JAVA Package `de.unibi.techfak.bibiserv`. Alle für den Einsatz von BiBiWS auf dem Apache Tomcat nötigen Klassen und Funktionen sind hier zusammengefaßt und werden bereitgestellt. BiBiWS wird neben den benötigten anderen Packages mit auf dem WebService Server installiert (siehe Anhang B).

5.1.1 Die Klasse `de.unibi.techfak.bibiserv.WSSTools`

Beschreibung:
`WSSTools` stellt zum einen grundlegende Funktionen bereit (wie z.B. Zugriff auf die Properties Dateien und das Spoolverzeichnis) zum anderen werden Funktionen angeboten, die im Anwendungskontext sehr praktisch sind (beispielsweise Überprüfen einer E-Mail Adresse).
Konstruktor:

`WSSTools(propStream)` `propStream` ist vom Typ `java.io.InputStream` der Datei `tool.properties`, aus der die Werkzeug spezifischen Konfigurationen geladen werden. Die `bibi.properties` werden durch einen Eintrag in der `tool.properties` Datei referenziert.

Methoden:

`getProperty(propertyitem)` Liefert ein Propertyeintrag aus `tool.properties`.

`getBibiProperty(propertyitem)` Liefert ein Propertyeintrag der globalen `bibi.properties`.

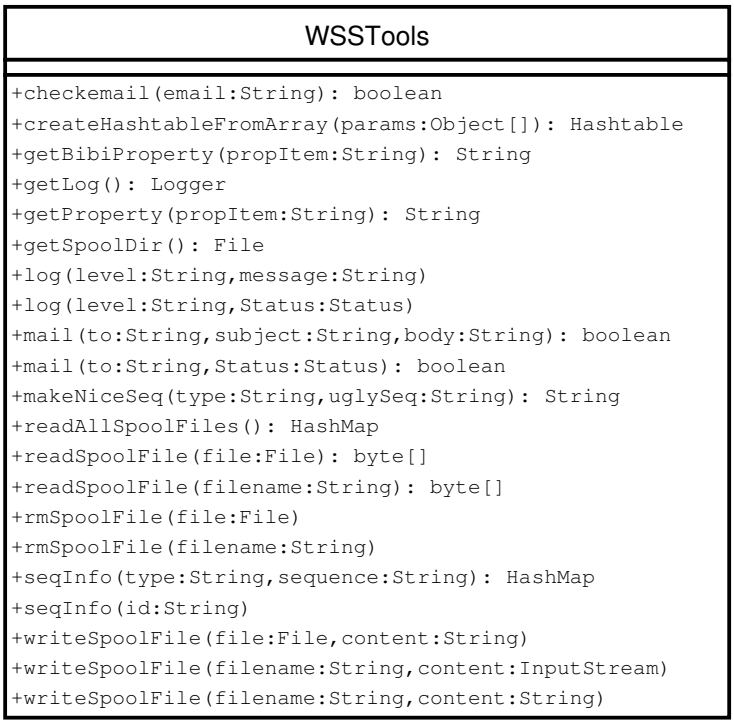

Abbildung 5.1: `WSSTools` Klasse

`getLog()` Liefert ein Log4J Objekt für das logging – Alternative zu der `log()` Methode.

`getSpoolDir()` Liefert das aktuelle Spoolverzeichnis.

`log(level,msg)` Logged mit `level` die gegebene Nachricht `msg`; Statt `msg` kann auch `status` übergeben werden, wodurch dann `status.toString()` gelogged wird. Zusätzlich zu den Log4J `level` s gibt es „mailfatal", was nach „fatal" logged und das BiBiServ Administrator Team informiert. Siehe [LOG4J].

`checkemail(emailadr)` Überprüft, ob `emailadr` dem Format `a@b.c` entspricht, wobei für die Domain `b.c` ein MX-Record existieren muß.

`createHashtableFromArray(params)` Erzeugt einen Hashtable aus einem `Object[]` `params`.

`mail(to,subject,body)` Schickt eine E-Mail an `to` zur Benachrichtigung des Benutzers oder um das BiBiServ Administrator Team über Fehler zu informieren (loggen nach `mailfatal`). Statt `subject` und `body` kann auch ein Status Objekt übergeben werden, was dann eine standardisierte E-Mail verschickt.

`makeNiceSeq(typ,sequenz)` Versucht eine `sequenz` so zu konvertieren, daß sie `typ` entspricht.

`seqInfo(typ,sequenz)` Liefert Informationen über die `sequenz` als `HashMap`.

`writeSpoolFile(datei,inhalt)` Schreibt eine Datei. `datei` ist entweder Dateiname oder `java.io.File` und `inhalt` ist entweder `String` oder `java.io.InputStream`.

`readSpoolFile(datei)` Liest eine Datei `datei` (Dateiname ist `String` oder `java.io.File`).

`readAllSpoolFiles()` Liest alle vorhandenen Dateien im Spoolverzeichnis und liefert sie in einer `HashMap`, wobei der Schlüssel der Dateiname und der Wert der Inhalt der Datei darstellt.

`rmSpoolFile(datei)` Löscht eine Datei `datei` aus dem Spoolverzeichnis (Dateiname oder `java.io.File`).

5.1.2 Die Klasse `de.unibi.techfak.bibiserv.Status`

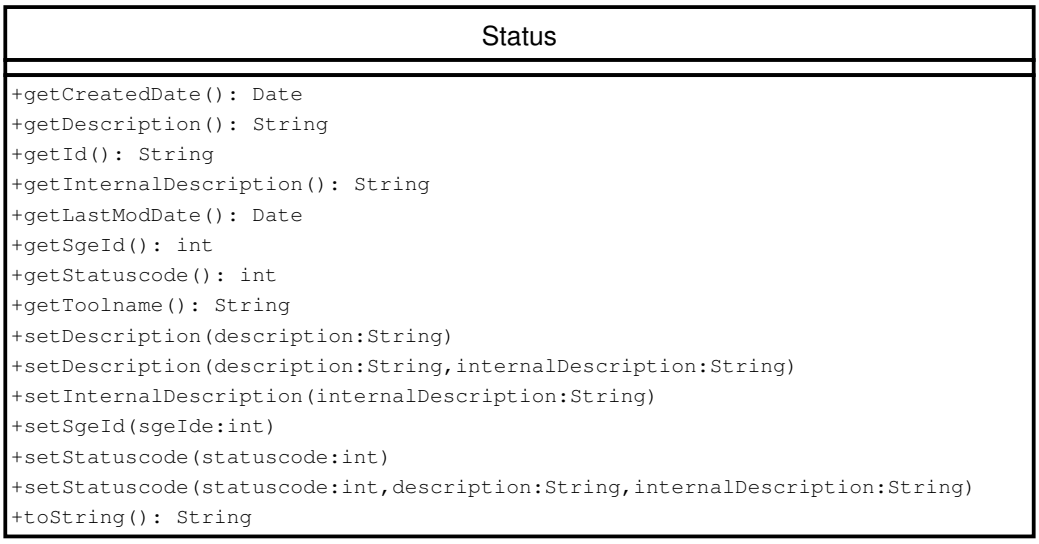

Abbildung 5.2: `Status` Klasse

Beschreibung:
Die Klasse `Status` spiegelt den aktuellen Zustand des WebService Aufrufs wieder. Bei einem `request()` wird ein neuer Status in der Datenbank angelegt und sämtliche Veränderungen sofort persistent in der Datenbank gespeichert, so daß ein zwischenzeitiger `response()` den aktuellen Status erfragen kann, der über die im Konstruktor gelieferte `id` referenziert wird. Mehr zu der Verwaltung von WebService Aufrufs in Kapitel 4. Eine Auflistung der Statuscodes befindet sich in Anhang C
Konstruktor:

`Status(wsstools)` Generiert einen neuen Status inklusive neuer `id`.

`Status(wsstools,id)` Lädt einen Status mittels `id` aus der Datenbank.

Methoden:

`getId()` Liefert die `id` des WebService Aufrufs.

`getStatuscode()` Liefert den aktuellen Statuscode.

`setStatuscode(statuscode,beschreibung,interne_beschreibung)` Setzt den `statuscode`. Optional auch die `beschreibung` und `interne_beschreibung`, die ansonsten aus den Property Dateien geladen werden.

`getDescription()` Liefert die aktuelle Beschreibung.

`getInternalDescription()` Liefert die aktuelle interne Beschreibung.

`setDescription(beschreibung, interne_beschreibung)` Setzt die `beschreibung` und (optional) die `interne_beschreibung` manuell, überschreibt die Standardbeschreibungen für den `statuscode`.

`setInternalDescription(interne_beschreibung)` Setzt die `interne_beschreibung` manuell, überschreibt die interne Standardbeschreibungen für den `statuscode`.

`getToolname()` Liefert den Toolnamen des WebService Aufrufes.

`getSgeId()` Liefert die JobId des aktuellen SGE Aufrufes.

`setSgeId(Id)` Setzt die JobId des SGE Aufrufes.

`getCreatedDate()` Liefert den Zeitpunkt, an dem der WebService Aufruf gestartet wurde.

`getLastModDate()` Liefert den Zeitpunkt, an dem zum letzten Mal der Status geändert wurde.

`toString()` Liefert Informationen über den aktuellen Status als String (beispielsweise zum loggen).

5.1.3 Die Klasse `de.unibi.techfak.bibiserv.SGECall`

Beschreibung:
`SGECall` startet einen `command` durch die Sun Grid Engine. Die Methode `call()` wartet dabei, bis der Job beendet ist. Dabei wird die SgeId im Status gespeichert. Ein Job kann maximal 24 Stunden laufen, danach wird dieser abgebrochen (siehe dazu Anhang A).
Der *Sun Grid Engine* kann einige Paramter übergeben bekommen. Dabei setzt das BiBiServ Administrator Team einige von diesen fest (`BiBiSgeParams`), einige können vom Entwickler beeinflußt werden (`UserSgeParams`). Eine ausführliche Beschreibung findet sich im Handbuch der *Sun Grid Engine* [SGEUM].

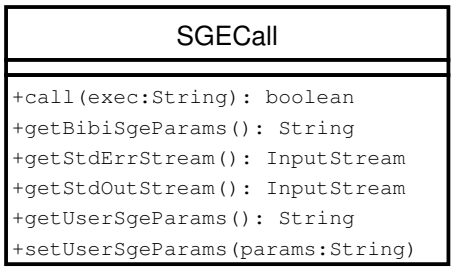

Abbildung 5.3: `SGECall` Klasse

Transparent dazu ist die Klasse `LocalCall`, die einen `command` Aufruf lokal auf dem Rechner ausführt.

Konstruktor:

`SGECall(wsstools,status)` Erzeugt ein neues Objekt `SGECall`.

Methoden:

`call(syscall)` Führt das in `syscall` angegebene Programm aus und blockiert, bis dieser beendet ist.

`getBiBiSgeParams()` Liefert die unveränderbaren Parameter beim Start des `command` Aufrufes.

`getStdErrStream()` Liefert einen `java.io.InputStream` mit der STDERR Ausgabe des `command` Aufrufes.

`getStdOutStream()` Liefert einen `java.io.InputStream` mit der STDOUT Ausgabe des `command` Aufrufes.

`getUserSgeParams()` Liefert die vom Entwickler veränderbaren Parameter beim Start des `command` Aufrufes.

`setUserSgeParams()` Setzt die vom Entwickler veränderbaren Parameter beim Start des `command` Aufrufes.

5.1.4 Die Klasse `de.unibi.techfak.bibiserv.LocalCall`

Beschreibung:

`LocalCall` startet einen `command` Aufruf lokal auf dem Rechner.
Die Methode `call()` wartet dabei, bis der Job beendet ist.
Diese Klasse ist beim Entwickeln von BiBiWS entstanden, sollte **nicht** benutzt werden, wenn eine *Sun Grid Engine* bereit steht, da ansonsten der Rechner, der auch den Tomcat beheimatet, die gesamte Rechenleitung erbringen muß. Außerdem werden Jobs nicht nach 24 Stunden beendet, wie es bei `SGECall` ist.
Die Klasse ist transparent zu `SGECall`, die die `command` Aufrufe verteilt.

Konstruktor:

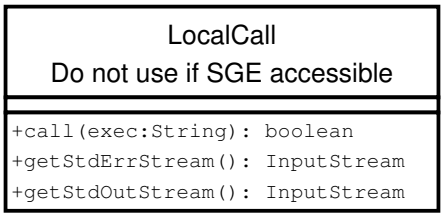

Abbildung 5.4: `LocalCall` Klasse

`LocalCall(wsstools,status)` Erzeugt ein neues Objekt `LocalCall`.

Methoden:

`call(syscall)` Führt `syscall` als Programm aus und blockiert, bis dieser beendet ist.

`getStdErrStream()` Liefert einen `java.io.InputStream` mit der STDERR Ausgabe des `command` Aufrufes.

`getStdOutStream()` Liefert einen `java.io.InputStream` mit der STDOUT Ausgabe des `command` Aufrufes.

5.1.5 Die Klasse `de.unibi.techfak.bibiserv.BiBiProcessing`

Abbildung 5.5: `BiBiProcessing` Klasse

Beschreibung:
`BiBiProcessing` ist eine einfach gehaltete Klasse, um einen `command` Aufruf auszuführen. Im Normalfall wird ein BiBiWS WebService eine eigene Processingklasse haben, um ein Pre- und Postprocessing durchführen zu können.
Sollte aber ein Tool nur einen `command` Aufruf machen, bietet `BiBiProcessing` eine Standardmöglichkeit:
Sie kann als Thread aus der Implementierungsklasse aufgerufen werden.
Neben den vom `command` Aufruf erzeugten Dateien finden sich die Ausgabe nach STDOUT im Spoolverzeichnis unter `stdout-log.txt` und nach STDERR unter `stderr-log.txt`.
Konstruktor:

`BiBiProcessing(wsstools,status,email,syscall)` Erzeugt ein neues BiBiProcessing Objekt, was `syscall` beim Aufruf von `run()` ausführt und eine Benachrichtigung an `email` schickt.

Funktion:

`run()` Startet den Thread.

5.2 Die BiBiWS Client Library

Die clientseitigen BiBiWS Implementierungen finden sich in dem PERL-Package `BiBiServ::`. Das Package wird mit dem Apache http Server mitinstalliert, so daß es neben anderen Perl Packages zur Verfügung steht (siehe Anhang B).

5.2.1 Die Klasse `BiBiServ::WSCTools`

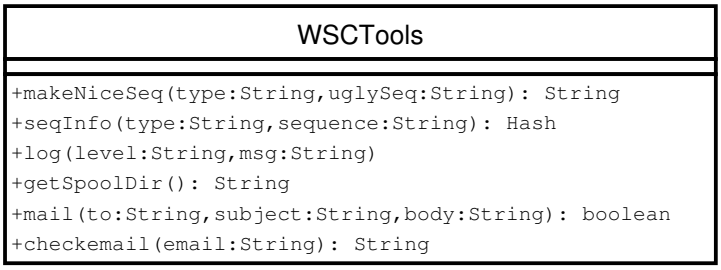

Abbildung 5.6: BiBiServ::WSCTools

Beschreibung:
`WSCTools` ist (wie `WSSTools`) eine Ansammlung von benötigten und nützlichen Funktionen. Diese werden auf der Clientseite verwendet, um einen reibungslosen Ablauf der BiBiWS WebService zu ermöglichen.
Eine größtmögliche Ähnlichkeit zwischen den beiden Libraries ist beabsichtigt, um eine einfache Einarbeitung zu gewährleisten.
Einige Funktionen stehen sowohl auf Server-, wie auch auf Clientseite bereit, um schon beim clientseitigen `<PN>_submit` Aufruf vorzeitig Fehler zu entdecken und unnötige WebService Aufrufe zu minimieren (z.B. `makeNiceSeq()`).
Konstruktor:

`WSCTools(toolname=>TOOLNAME)` Erzeugt ein neues `WSSTools` Objekt, was durch `TOOLNAME` die passende Properties Datei lädt.

Methoden:

`getSpoolDir()` Liefert das aktuelle Spoolverzeichnis.

`log(level,msg)` Logged mit `level` die gegebene Nachricht `msg`. Statt `msg` kann `status` übergeben werden, wodurch dann `status.toString()` gelogged wird. Zusätzlich zu den Log4Perl `levels` gibt es noch „mailfatal“, was nach „fatal“ logged und das BiBiServ Administrator Team informiert (siehe [LOG4P]).

`mail(to,subject,body)` Schickt eine E-Mail an `to` zur Benachrichtigung des Benutzers oder um das BiBiServ Administrator Team über Fehler zu informieren.

`checkemail(emailadr)` Überprüft, ob `emailadr` dem Format `a@b.c` entspricht, wobei für die Domain `b.c` ein MX-Record existiert muß.

`makeNiceSeq(typ,sequenz)` Versucht eine `sequenz` zu konvertieren, daß sie `typ` entspricht.

`seqInfo(typ,sequenz)` Liefert Informationen über die `sequenz` als Hashreferenz.

5.2.2 Die Klasse `BiBiServ::Status`

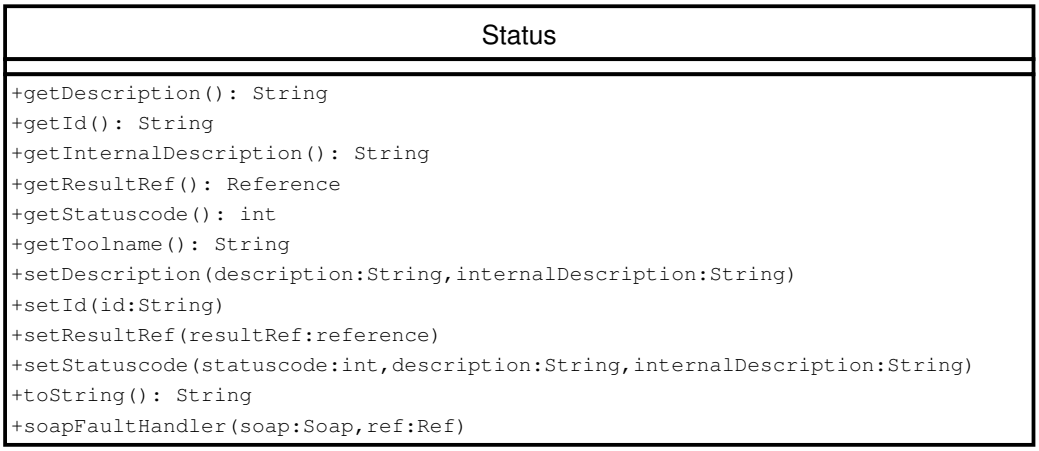

Abbildung 5.7: BiBiServ::Status

Beschreibung:
`Status` stellt den aktuellen Status aus Sicht der Clientseite dar. Der Statuscode und Beschreibung kommen zum einen von den WebService Aufrufen, zum anderen können lokale Fehler auf Clientseite repräsentiert werden (`statuscode` beginnt mit `c`).
Eine größtmögliche Ähnlichkeit zwischen den Status Objekten auf Client- und Serverseite ist wiederum beabsichtigt.
Konstruktor:

`Status(WSCTools=>WSCTOOLS)` Erzeugt einen neuen Status. `WSCTOOLS` ist eine Referenz auf ein `WSCTools` Objekt. Bei Referenzierung auf eine `id` sollte diese mittels `setId()` gesetzt werden.

Methoden:

`getId()` Liefert die `id` des WebService Aufrufs.

`setId()` Setzt `id` des WebService Aufrufs.

`getStatuscode()` Liefert den aktuellen Statuscode.

`setStatuscode(statuscode,beschreibung,interne_beschreibung)` Setzt den `statuscode`. Optional ist die `beschreibung` und `interne_beschreibung`, die sonst aus den Property Daiteien geladen wird.

`getDescription()` Liefert die aktuelle Beschreibung.

`setDescription(beschreibung, interne_beschreibung)` Setzt die `beschreibung` und `interne_beschreibung` manuell, überschreibt die Standardbeschreibungen für den `statuscode`.

`getInternalDescription` Liefert die aktuelle interne clientseitige Beschreibung.

`getToolname()` Liefert den Toolnamen des WebServices.

`getResultRef()` Liefert eine Referenz auf das WebService Resultat.

`setResultRef(referenz)` Setzt eine `referenz` auf das WebService Resultat.

`soapFaultHandler(soap,res)` Verarbeitet bei einem `on_fault` von `SOAP::Lite` die Fehlermeldung und aktuallisiert den Status entsprechend.

`toString()` Liefert Informationen über den aktuellen Status als String (beispielsweise zum Loggen).

5.2.3 Die Klasse `BiBiServ::WSCLayout`

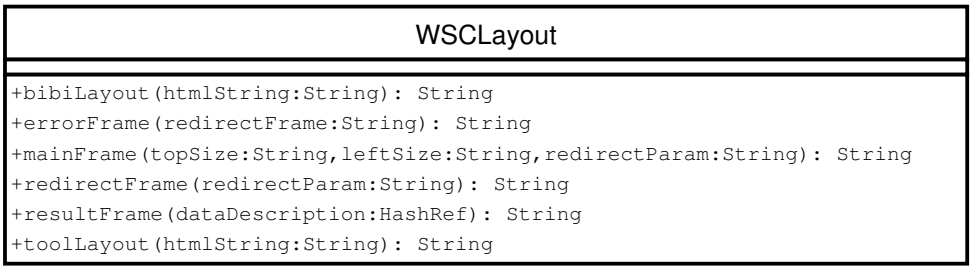

Abbildung 5.8: BiBiServ::WSCLayout

Beschreibung:
`WSCLayout` ist eine Klasse, um das BiBiServ Corporate Identity zu erzeugen. Sie benhaltet Funktionen, um das Layout so einfach wie möglich dem Entwickler zur Verfügung zu stellen.
Einige Funktionen sind nur Vorschläge für den Entwickler, gegebenenfalls sollten sie nicht verwendet werden, um (beispielsweise bei `resultFrame()`) einem Tool ein besser passendes Aussehen zu verleihen (siehe Abbildung 4.3).
Konstruktor:

`WSCLayout(WSCTools=>WSCTOOLS,Status=>STATUS)` Erzeugt ein WSCLayout Objekt. `WSCTOOLS` und `STATUS` sind Referenzen auf entsprechende Objekte.

Methoden:

`mainFrame(topSize,leftSize,redirectTarget)` Erzeugt das BiBiServ Frameenvironment mit oberem Frame (*top frame*) und linkem Frame (*left frame*). Für das Hauptframe wird das `_redirect` CGI aufgerufen oder ein anderes, wenn `redirectTarget` definiert ist. Linkes und oberes Frame können dabei `big` (default) oder `small` sein, um dem Layout des Resultats angepaßt zu werden. Zusätzlich kann `topFrame` mittels `no` auch ausgeschaltet werden.

`redirectFrame(target)` Erzeugt ein Redirectframe, um das Resultat des gegebenen CGI Aufrufs. Es muß nur bei mehrstufigen WebServices benutzt werden.

`resultFrame(resultdata)` Erzeugt ein Frame mit einer Tabelle aus `resultdata`, welches eine Referenz auf einen Hash ist. Dabei ist der *key* ein Dateiname und der *value* eine Beschreibung auf Dateien, die im Spoolverzeichnis liegen.

`errorFrame(target)` Erzeugt einen (orangen) Fehlerframe, der den Benutzer über einen Fehler informiert. Bei nicht-permanenten Fehlern, kann ein `target` angegeben werden, so daß der Benutzer bei serverseitigen Fehlern zu einem späteren Zeitpunkt seinen WebService Aufruf noch einmal wiederholen kann.

`bibiLayout(html)` Erzeugt einen Frame mit dem BiBiServ Layout um gegebenes `html` Seite.

`toolLayout(html)` Erzeugt einen Frame mit dem Tool Layout um gegebenes `html` Seite.

6 Performanceuntersuchungen

Bei der Entwicklung von BiBiWS ist ein Phänomene aufgetreten, das hier kurz diskutiert wird: Es trat eine `java.lang.OutOfMemoryError` Exception auf.
Als Testfall wurde der Beispielwebservice, der auch den neu erstellten BiBiServ WebService Projekten beiliegt, untersucht.
Dieser liefert beim Aufruf der `response()` Methode einen String der Länge des Wertes des Integers, der bei `request()` übergeben wurde.

Ich erwarte folgenden Speicherverbrauch:

$O(2 * n)$ (n Länge des gelieferten Strings)

Zum einen muß die einkommende SOAP Message empfangen werden, zum anderen müssen die Daten (der String) aus der SOAP Message entpackt dem Programm bereitgestellt werden.

Da der Tomcat Server keine direkte Messung des Arbeitsspeichers zuläßt, wurde das Verhalten auf Clientseite untersucht.
Um einen Vergleich zu erhalten wurden Clients in den folgenden Programmiersprachen entwickelt:

Perl benutzte Library: *SOAP::Lite* Version 0.55 (verfügbar auf [CPAN])

Java benutzte Library: *AXIS* Version 1.2beta [AXIS]

C++ benutzte Library: *gSoap* Version 2.5 [GSOAP]

Alle Clients rufen die `response()` Methode auf, empfangen die Daten und messen anschliessend die Länge des gelieferten Strings, damit eine Garbage Collection den String nicht zu früh entfernt.
Die in der *Technischen Fakultät der Universität Bielefeld* installierte Versionen von `memtime`, das zur Messung der Zeit und des Arbeitsspeicherverbrauchs herrangezogen wurde, arbeiten nur mit 32-Bit Programmen, so daß Java auch nicht in der 64-Bit Version in diese Messungen eingehen konnte. Außerdem kann auch das installierte Perl maximal zwei Gigabyte Arbeitsspeicher nutzen. So ergeben sich auf Clientseite unterschiedliche Grenzen, jedoch sind die Phänomene auch so schon sehr deutlich zu erkennen.
Sowohl der Tomcat als auch der jeweilige Client liefen auf dem gleichen Rechner, einem *Sun v880* mit 8 UltraSPARC III Prozessoren und 64 Gigabyte Arbeitsspeicher, bei keiner nennenswerten Last. Hierdurch wird sichergestellt, daß Seiteneffekte durch das Netzwerk oder andere Programme vermieden werden.

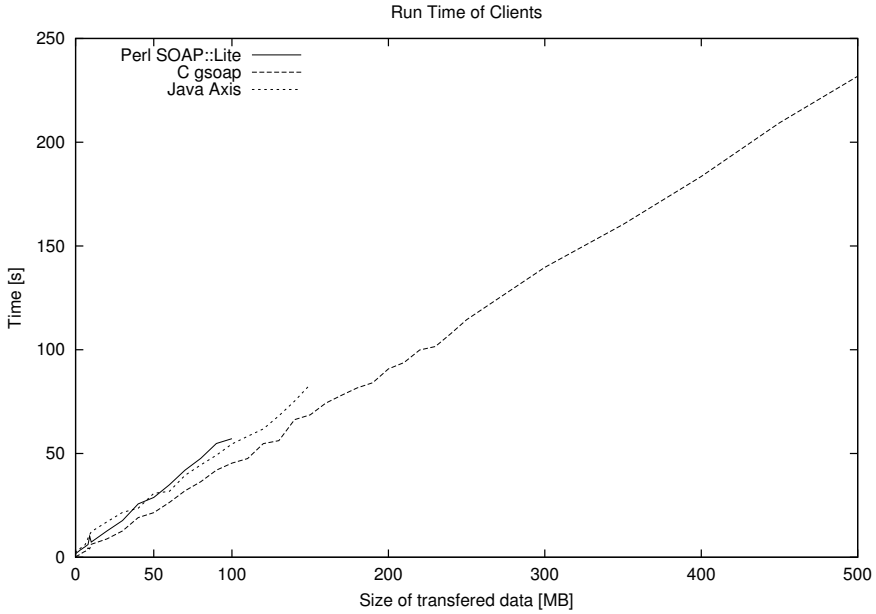

Abbildung 6.1: benötigte Zeit über Länge des übertragenen Strings

6.1 Ergebnisse

Die Resourcen wurde mit dem UNIX Programm `memtime` gemessen, was sowohl die Zeit als auch den Speicherverbrauch nach Beenden eines Prozesses ausgibt.
Folgende Größen wurden als String emfangen um die Messungen durchzuführen:
1, 10, 100, 1000, 10000, 100000 Byte
1-10 MB, 20, 30 . . . 250 Megabyte
300, 350, 400, 450 und 500 Megabyte
Als Ergbnis sind Mittel von jeweils drei Durchläufen pro Client dargestellt.

Zeit

In Abbildung 6.1 ist die benötigte Zeit (y-Achse) der Clients über der Länge des empfangenen Strings (x-Achse) dargestellt.
Wie zu erkennen ist, unterscheidet sich die benötigte Zeit der Clients nicht signifikant.

Arbeitsspeicher

In Abbildung 6.2 ist der benötigte Arbeitsspeicher der Clients über der Länge des empfangenen Strings (x-Achse) dargestellt.
Leicht ist zu erkennen, daß der *gSoap* Client am wenigsten Arbeitsspeicher verbraucht, und somit auch für große Strings noch getestet werden kann. Dieser Client scheint sich an meine Erwartung zu halten, was belegt, daß diese realisierbar sind.

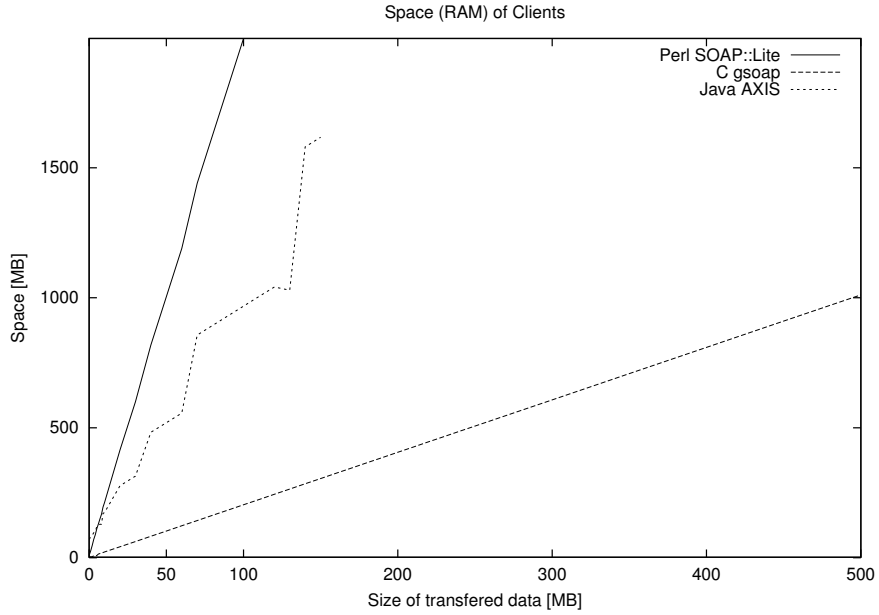

Abbildung 6.2: benötigter Arbeitsspeicher (RAM) über Länge des übertragenen Strings

Die beiden anderen Clients, die die *AXIS* und *SOAP::Lite* Libraries benutzen, liegen deutlich über der Erwartung.
Es handelt sich anscheind um einen linearen Verlauf.
Meine Abschätzung des Ergebnisses für *AXIS* ist:

$$O(10 * n)$$ (n Länge des gelieferten Strings)

Die Stufen sind durch die dynamische Allokierung des Arbeitsspeichers der JavaVM zu erklären. Perl scheint hier in kleineren Stufen vorzugehen. Relativiert wird dieses Ergebnis durch die interne Repräsentation der Daten in Java. Durch die Kodierung in UTF-16 ist klar, daß ein Zeichen in Java mit zwei Bytes dargestellt werden muß.

Als Abschätzung für *SOAP::Lite* muß folgendes angenommen werden:

$$O(15 * n)$$ (n Länge des gelieferten Strings)

Anscheinend brauchen sowohl *AXIS* als auch *SOAP::Lite* zum „Entpacken" der Daten aus der SOAP Message mehr Speicher als erwartet.
Ich vermute, daß in diesen Libraries die SOAP Message intern nicht durch *call-by-referenz* sondern durch *call-by-value* zwischen unterschiedlichen Teilen der Libraries weitergereicht werden, was jeweils zu einer Kopie der Daten führt.
Da WebServices bisher nicht zu den Alltagsanwendungen gehören und bei den heute bereits laufenden Anwendungen hauptsächlich kleine Datenmengen übermittelt werden, ist dieses Problem vermutlich bisher nicht aufgetaucht.

Beide Projekte wurden auf dieses Ergebnisse aufmerksam gemacht und haben Interesse an einer weitergehenden Analyse bekundet.

7 Fazit

BiBiWS ist ein Framework mit dessen Hilfe ein Entwickler für den BiBiServ, nahezu ohne Kenntnis von WebService Techniken, WebServices für den BiBiServ entwickeln kann.
Dabei besteht BiBiWS aus einem serverseitigen und einen clientseitigen Teil.

Der serverseitige Teil ist das eigentliche Framework, um einen WebService anzubieten.
Es wird die im bioinformatischen Bereich benötigte Asynchronität mittels der leicht verständlichen *Request and reply with polling* Technik unterstützt. Auch das gesamte Management der Resultate wird dem Entwickler abgenommen. Dabei stehen dem Entwickler eine Vielzahl von nützlichen Methoden zur Verfügung, um sein Programm, was er als WebService anbieten möchte, zu realisieren.
Außerdem wird eine Anbindung zu einem Computing Grid mittels *Sun Grid Engine* zur Verfügung gestellt, so daß dem Benutzer des BiBiServ wesentlich mehr Rechenleistung bereit steht.

Der clientseitige Teil läuft auf dem bisherigen BiBiServ als Webfrontend für die WebService Server.
Dabei unterscheiden sich die Webfrontends der klassischen Werkzeuge und der BiBiWS Werkzeuge für den Benutzer nicht. Dieser findet ein dem einheitliches Layout des BiBiServ vor.
Das Polling, das die Clientseite durch die *Request and reply with polling* Technik übernehmen muß, wird durch HTML-redirects ausgelöst. Dabei bekommt der Benutzer des Webfrontends ebenso wie der Benutzer des WebServices direkt eine Rückmeldung über den Status seines Aufrufs (normalerweise *Preprocessing, Processing, Postprocessing* und *finished*). Somit wird die Benutzerfreundlichkeit erhöhnt.

Die Programmierung der Werkzeuge wird außerdem durch die Bereitstellung vom Beispiel WebServices *TemplateWS* erleichtert. Ein neues Projekt wird vom BiBiServ Administrator Team eingerichtet und enthält alle notwendigen Dateien, so daß ein voll funktionstüchtiger WebService vorliegt. Diesen muß der Entwickler nun nur noch seinen eigenen Anforderungen des Programms anpassen. Insbesondere bei existierenden Kommandozeilenprogrammen ist es sehr einfach, diese als WebService mittels BiBiWS bereitzustellen.
Der Arbeitsaufwand hängt selbstverständlich sehr von den Programmiererfahrungen des Entwicklers und der Komplexität der Ein- und Ausgabe ab, da diese in die XML-Schema konformen Dokumente des HOBIT-Projekt überführt werden müssen, um das

HOBIT-Projekt zu unterstützen.
Einfache WebServices sind für versierte Entwickler oder unter Anleitung innerhalb von ein bis zwei Arbeitstagen realisierbar.

BiBiWS ist voll funktionsfähig und im Produktionsbetrieb.
Den bisherigen Erfahrungen zufolge ist es zuverlässig in die Umgebung des BiBiServ integriert. Schon jetzt bauen die BiBiServ Projekte *e2g* ([E2G]; implementiert von Jan Krüger) und *reputer* ([REPUTER]; implementiert von Georg Sauthoff) auf BiBiWS auf.

7.1 Ausblick

Einige Details sollten für die Zukunft beachtet werden:

Dazu gehört das in Kapitel 6 betrachtete Performance Problem.
Der enorme Verbrauch an Arbeitsspeicher, um Daten in eine SOAP Message zu verpacken, sollte und kann auch verringert werden, wie der gSoap Client zeigt. Im Rahmen dieser Arbeit konnte das Ausfindig machen von evtl. vorhandenen Fehlern in den *AXIS* und *SOAP::Lite* Libraries leider nicht mehr behandelt werden.
Zwar sind bioinformatische Anwendungen auch heute schon vorstellbar, bei denen große Datenmengen zur Übermittlung anfallen, jedoch sind die bisherigen Programme des BiBiServ nicht in einem kritischen Bereich.

Eine der nächsten Aufgaben im Rahmen des BiBiServ wird die Umstellung von existierenden Programmen mit klassischen Webinterface auf das BiBiWS Framework sein. Durch diese Umstellung wird zum einen das Programm als WebService den Benutzern verfügbar gemacht, womit ein Benutzer automatisiert auf dieses Programm zugreifen kann. Zum anderen wird durch die Verteilung der stark belastete Rechner des BiBiServ entlastet.

Die Anbindung an die *Sun Grid Engine* bringt zur Zeit leider keine Java API mit, so daß externe Programme bemüht werden müssen, um den Status der Jobs abzufragen. Mit der Version 6 der *Sun Grid Engine* soll ein Interface *DRMAA* [DRMAA] implementiert werden, daß die Anbindung performanter machen soll.

BiBiWS scheibt bzw. liest die Ergebnisse der Aufrufe zur Zeit aus einem Spoolverzeichnis.
Eine mögliche Erweiterung besteht darin, diese von den Programmen in diese Spoolverzeichnis geschrieben Ergebnisse, stattdessen in eine Datenbank abzulegen. Dieses würde den Zugriff bei der Abfrage mittels `response()` beschleunigen.

BiBiWS selbst ist für den BiBiServ und seine Umgebung entwickelt worden. Durch die verwendeten Techniken, die allesamt auf vielen aktuellen Systemen verfügbar sind, ist es jedoch auch möglich, BiBiWS an anderen Standorten zu installieren und betrei-

ben.
Die Serverseite wird ohne Probleme installierbar sein, da sie selbstständig ist.
Die Clientseite hingegen ist stark angepaßt an das *Corporate Identiy* des BiBiServ. Eine Installation auf anderen Rechnern wäre nur für die nicht layoutspezifischen Teile sinnvoll.

Zur Diskussion steht auch, ob eine Anwendung der sich nun entwickelnden asynchronen Protokolle als Alternative zu dem synchronen http in Frage kommt.
Diese könnten das *Polling* überflüssig machen. Jedoch sind diese Protokolle alle noch in der Entwicklungsphase. Die verfrühte Anwendung würde zu einer wesentlichen Verkomplizierung des Zugriffs auf die WebServices hinauslaufen. Dadurch müßten Entwickler und Benutzer erhebliches Vorwissen beziehungsweise Einarbeitungszeit aufbringen.

A BiBiWS Development Guide

This *Development Guide* will show developers the steps to develop a WebService on BiBiServ using the BiBiWS framework.
First, basic ideas and functions of a BiBiWS WebService are presented briefly, which are neccessary for understanding the concept. Afterwards the development environment at the BiBiServ will be presented .
For getting into the idea of BiBiWS WebServices, a very brief introduction will clarify the workflow of development and how to get it working.
They are followed by a detailed description of the source code of the default WebService for doing complex things.
Finally there are some hints for testing and debugging WebServices.

Just for clarification:
A *developer* in this document is a person, who develops a bioinformatic tool and would like to provide a world-wide accessible WebService for this tool. A *user* is a person, who uses such a WebService. The BiBiServ Administrator Team 's task is to provide help on the one hand for developing WebServices and on the other hand answer questions coming from users.
Users and developers can contact BiBiServ Administrator Team via email:
bibi-help@techfak.uni-bielefeld.de
As notation `<PN>` (project name) and `TemplateWS` is used as name of the created project in filenames as well as at source code and URLs.

A.1 Basics of a WebService

All of the following explanations are mostly specific for BiBiWS WebServices at the BiBiServ environment.
For general information about WebServices have a look at [CHAP] and [RAY].

A WebService contains two sides: Client and server. In general a WebService is a synchronous data exchange service of two points connected via a network. This means after a short time (less than the http timeout, which normally is 300 seconds) an answer of the WebService server is required.

Bioinformatic programs normally run much longer than five minutes. To avoid problems with timeout and changes of IP-Address client, BiBiWS uses a technique called *Request and reply with polling*: First the client side submits a job with the required data (program parameters and data etc.) and immediately gets an `id` after the job was

started which normally takes some seconds (name convention `request()`).
Afterwards the client side is able to get the result using this `id`. If the computation of the program is not finished, the client side gets back a code with enhanced information. Otherwise the result of the call is returned (name convention `response()`).

This polling technique completely avoids problems with timeouts. The user can even request the results hours or days later – or from another host, just with knowledge of the *id*. Although there are asynchronous WebService projects, they have hard restrictions. Further informations about possible solutions are described in Chapter 3.1.2 (German).
As mentioned in Chapter 1.2 a BiBiWS WebService should follow some naming conventions for the provided methods. The methods `request()` and `response()` should follow the HOBIT-Project specific input/output formats (see Chapter A.5.2). For some tools it may be useful also to accept/return data in tool specific formats, because then other tools might be able to accept them directly. Thus a double conversion can be avoided. For these WebService methods BiBiWS suggests using names `request_orig()` and `response_orig()`.

A.2 Introduction to BiBiServ environment

The BiBiServ itself is only one of four servers involved.
BiBiServ is the production system of the http interface. BiBiTest is the corresponding test environment, where developers can install their tools for testing. Both are Apache http servers, which are configured in the same way, and acting as WebService clients to the WebService Servers.
BiBiServ WebService Server and BiBiTest WebService Server are servlet containers servers (Jakarta Tomcat) for providing the WebService server side.
A developer will develop and test it's project by installing on test environment (BiBiTest and BiBiTest WebService Server).
Publishing an installation of the tools is done by the BiBiServ Administrator Team on the production systems (BiBiServ and BiBiServ WebService Server).

This feature of separating test and production sites gives best results in stability and performance. Also the BiBiServ Administrator Team has a certain amount of control and verification before projects are released on the production site.

A.3 Description of the project directory

Here are described the three basic subdirectories of the created default project.
Also the WebService global installation is described to get the default WebService to run. Understanding this is important to get the basic ideas at the following brief introduction.

install_test	(default) install client and server side on test systems
reinstall_test	reinstall client and server side on test systems
clean_test	clean client and server side on test systems

Table A.1: BiBiServ Developer commands of BiBiWS project

```
wss/
  <PN>.wsdd          for generating the WSDL of the WebService and for deploying on the tomcat
  WEB-INF/           for including to WAR File
  build.xml          ant build file for building the web service
  config/            properties files
  dist/              distribution WAR and WSDL files
  doc/               documentation of project server side
  doc/api            automatical generated Java API
  lib/               additional libraries
  src/               source code
```

Figure A.1: Overview of wss/ directory

A.3.1 Project directory

The BiBiServ Administrator Team will set up a project on /vol/bibidev/<PN>/, which already includes a running example of a BiBiWS WebService.
It produces a simple string of given length, so a pretty simple example, which the developer can modify to his or her own requirements.
All files and information have to be located in this directory and will be installed from this directory.
For installing client and server side a tool-wide Makefile file exists within the project directory (see Table A.1 for commands).
There are three subdirectories:

wss/ – WebService server side implementation

Subdirectory wss/ contains all required files for compiling and installing the server side of a WebService. Figure A.1 gives an overview of the wss/ directory. We will discuss the individual elements later in Chapter A.5.1

cmd/ – command line WebService client side

Subdirectory cmd/ represents a simple command line client for the WebService server side. On the one hand this can be used for debugging and testing the WebService server side, which may be much easier than testing BiBiServ client side and server side together. On the other hand the developer may want to distribute an example command line client – so this is the starting point.

```
cmd/
  <PN>-request.pl   start a web service call and get an id.
  <PN>-response.pl   get a response of a submitted id.
  *.properties      properties files for *.pl
  wsc.log           log file of *.pl
```

Figure A.2: Overview of `cmd/` directory

```
wsc/
   Makefile          make file for installing web service client side
   cgi-bin/          CGIs of client side
   data/             HTML files like welcome.html and submission.html
   framework.xml     Configure File for BiBiServ layout framework
   vol/<PN>/etc/     Properties files
   vol/<PN>/lib/     web service client side Perl module and others.
```

Figure A.3: Overview of `wsc/` directory

Figure A.2 shows the contents of this directory. Discussion in detail will follow in Chapter A.5.3

`wsc/` – WebService client side implementation

Subdirectory `wsc/` contains files that are required for the client http side on BiBiServ of the WebService. Figure A.3 gives a short explanation of the contents. Discussion in detail will follow in Chapter A.5.4

A.3.2 Limitations on BiBiServ and BiBiServ WebService Server

Due to the fact that BiBiServ offers computation time to the whole world, we have to follow certain policies. See [BIBISERVPOLICIES] for details.

As a result, the BiBiServ Administrator Team limits CPU time to 24 hours per call on WebService server side. The developer has to be careful about other possible problems like hard disc space. Computation time on client side (http side of WebService) should be as low as possible.

For not running out of hard disc space, the spooldirectory is tidied up every day. Data older than 3 days will be deleted, but the `ids` are stored at the database for 30 days.

While we provide direct access to the WebService, it makes no sense to do complete input checking on the client side – server side has to check anyway.
So we suggest to check only simple things – like whether combination of parameters make sense on client side.
Never call programs on client side !

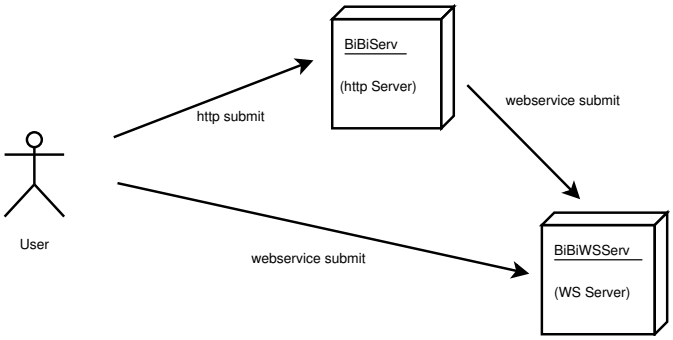

Figure A.4: Access of BiBiServ WebServices

While BiBiServ provides neither encryption nor authentication, all results are possibly available for everybody, who knows or guesses the `id`. BiBiServ Administrator Team is not responsible for sending results to anyone.

BiBiServ has also the policy *Tool release*, which includes to check performance and correctness of the installed tools at the production systems. So please talk to the BiBiServ Administrator Team for discussion.

A.4 Brief introduction to the BiBiWS WebServices

This is a very basic and rudimentary description of how a BiBiWS WebService works. It is neither complete nor describes the whole source code, which is described later (Chapter A.5). It is just a short description of the most important ideas of a BiBiWS WebService.
While a new project is created by the BiBiServ Administrator Team, an example is included. This WebService is a running example application, which only has to be modified by the developer.

The example WebService gets an `Integer` „length“ as input and creates a `String` of `x`'s of the given length, which is returned to the user.

WebServices located on the BiBiServ are accessible by two ways (see Figure A.4).
The first calls the WebService directly, like the usal way to access WebServices.
The second is the main task of BiBiServ since founding: To provide easy access to complex bioinformatic tools. This is done by an HTML form and CGIs, which call the WebService server as client.
So a project on BiBiServ always has two sides: the Server side and the client side (HTML form and CGIs).
The server side is written in Java and the client side CGIs in Perl.

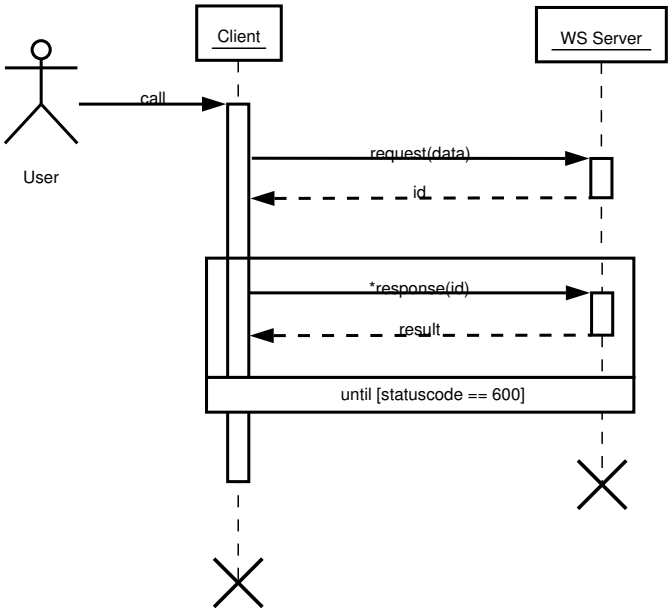

Figure A.5: Basic flow of an WebService call of BiBiWS.

Figure A.5 shows the basic flow of a call at the created WebService server side. After „starting" the program by calling `request()` with the data to progress and getting an `id`, the client can *poll* for the result by calling the `response()` method with this `id`. Statuscodes (see Appendix C) with reason phrases are used to represent the current status, if result is not available. [1]

A.4.1 Installing and running the default WebService of a new project

Very litte work has to be done for getting the default BiBiWS WebService of a new created project to run:

Step 1: Let the BiBiServ Administrator Team create a project.

Step 2: Go to the project directory and type `make`. The Client and Server side will be installed completly. Your default WebService is ready !

Step 3a: Test with command line client: Requesting a string of given length by going to the `cmd/` subdirectory and type `<PN>_request.pl <your_email_adr> 23` and get an `id`.

Step 3b: Same directorty, type `<PN>_response.pl <id of request>` and get the string of requested length (or get a status, so you have to try again later).

[1] statuscode 600 indicates *successfully finished*

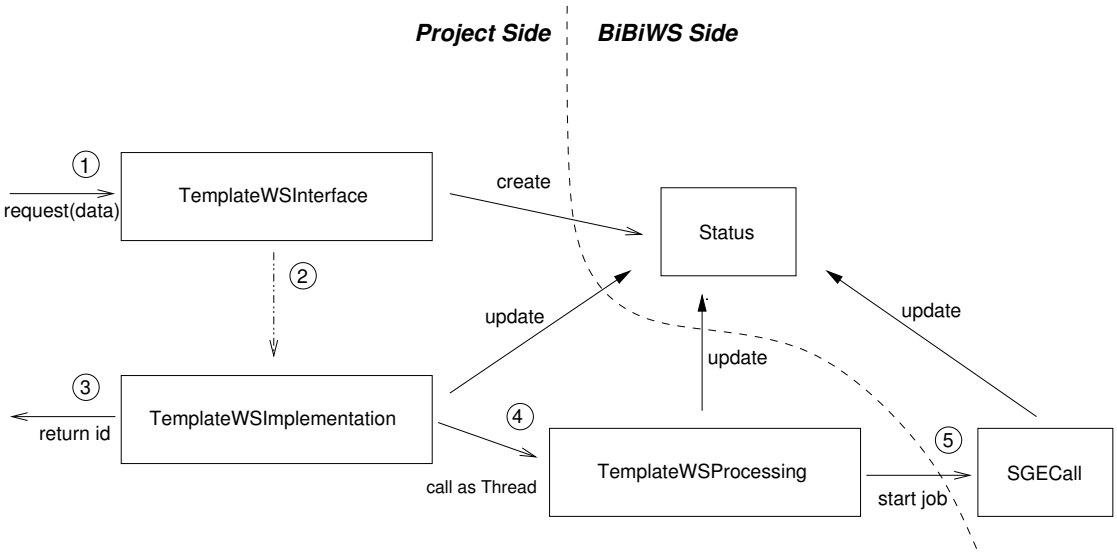

Figure A.6: Basic flow of `request()` call on server side.

Step 4: Test the Client http side: go to http://bibitest.techfak.uni-bielefeld.de/templatews, click on `submission` and submit this HTML form

Step 5: As you see – everything works fine. You are welcome to start modifying this default project to support your own bioinformatic tool

A.4.2 Server Side

Figure A.6 shows the basic flow of objects, if a `request()` call comes in.
The „project side" represents classes, which have to be modified by the developer where the BiBiWS side is the connection to the BiBiWS library.
The Status object is representing the current status while processing, so every class can modify this.

The source code is located at the subdirectory `wss/src/`.

Here is biefley described, what has to be changed for getting your own tool running.

① The Interface class defines the types of incoming and outgoing data.
Change the parameters of the `request()` method to your requirements.

② The Implementation class of the Interface. Again the parameters of `request()` have to be changed. Also parameters have to be checked.

③ `request()` returns the `id` of the WebService call.

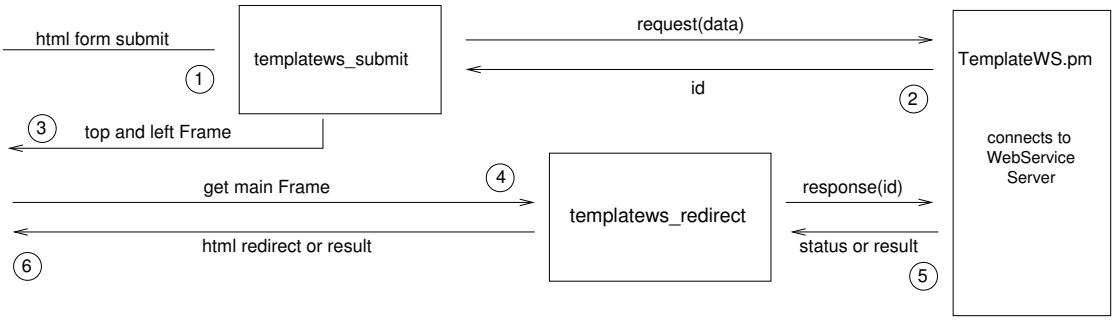

Figure A.7: Basic flow of `request()` call on client side.

④ Before returning the `id` the processing class is started as thread. Change the constructor for matching your parameters. Within the `run()` method of the processing class prepare the call of your tool by writing data to spooldirectory and convert parameters to command line parameters.

⑤ Also at the Processing class is the useage of `SGECall`, which executes your tool. This has to be modified to your personal tool.

The only modification at the `response()` method within Interface and Implementation classes are: Change the returned data type and read the result from spooldirectory for returning.

A.4.3 Client Side

Figure A.7 shows the process on client side while communicating with users.

The CGIs are located at the subdirectory `wsc/cgi-bin`, the `TemplateWS.pm` at `wsc/vol/templatews/lib/` and the HTML files at `wsc/data/templatews/`.

Again, here is briefly described what has to be changed for accessing your WebService server side from BiBiServ.

① Change `submission.html` form to your tool parameters. A Click on the submit will call the `templatews_submit` CGI.

② The WebService server side is called by `request()` of `TemplateWS.pm`, where the data of the submission page has to be submitted as parameters. This sets the `id` at the status object (not shown at figure).

③ A frameset of three frames is returned. The top and left frame are set by `templatews_submit`.

④ The third frame of the frameset is a link to the `templatews_redirect` CGI, which is called with the `id`.

⑤ `templatews_redirect` calls `response()` with the `id` at the `TemplateWS.pm`, which gets the status or result from the WebService server side.

⑥ If a status is returned, this means, the computation is not done yet and an HTML redirection to itself will be returned to the user. Otherwise, the returned result data has to be presented to the user. By default this is only a list of files, maybe the developer would like to write a more conveniet HTML result page.

That's all on client side, have a look at the working default example for getting a „feeling" of how it works.

A.4.4 Install your WebService

After modifying the default WebService application, it has to be installed.

step 1: Server side: call `ant` within the `wss/` directory. This will compile and build WebService, build the WSDL Specification, install it on the WebService server and copy the WSDL to the webserver.

step 2: Client side: call `make` within the `wsc/` directory. This will install the CGIs and `TemplateWS.pm`. Also the HTML files are processed and installed on the webserver.

A.4.5 Testing

For testing server side the `cmd/` directory contains simple Perl scripts, which uses the `TemplateWS.pm` of WebService client side.
The developer has to modify the paramaters of the `request()` methods and return type of the `response()` method only.

A.5 Detailed introduction to the BiBiWS WebServices

This section explains all files and directories of the default project, which is created by the BiBiServ Administrator Team for the developer. It gives a developer a detailed overview of the source code of the default project.
After the brief introduction at Chapter A.4, this gives the possibility to do more complex processing for development of a BiBiWS webservice.

While explaining the source code, unimportant lines like `use` statements, log messages and code comments are skipped for a less confusing explanation.

Additional BiBiWS libraries are documented in Chapter 5 (German). The location of server side API is [BIBIWSAPI-WSS] and client side API is [BIBIWSAPI-WSC].

A.5.1 Server side of a BiBiWS tool

The `wss/` directory contains several files and subdirectories, which are explained here. (see Figure A.1 for overview)

`<PN>.wsdd`

The *Web Service Deployment descriptor* (short WSDD) is required to generate the Web-Service WAR File. Basically it just describes the class Axis has to bind to and methods to export.[2] Normally developers don't have to change anything here, except one would like to add methods or change names of methods.

`WEB-INF/`

This directory will be part of the WAR file to release. If specific jars or other files are neccessary for the tool – place them here in the `lib/` subdirectory.
Two libraries are already included in every WebService, because of tomcat problems otherwise.
Normally developer don't have to change anything here.

`build.xml`

On the server side development `ant` is used for processing. There are some `ant` targets for developers to use, described in Table A.2. For installation on the BiBiServ WebService Server a member of the BiBiServ Administrator Team has to be informed.

`config/`

`config/` includes two properties files.
First there is `tool.properties` (see Figure A.8). there are some properties configured

```
toolname=TEMPLATEWS
max.submit=100
statuscode.701=Input Error - Tool spec explanation
statuscode.701.internal=Input Error - Tool spec internal information
```

Figure A.8: default `tool.properties` file

on creation of the default project:

[2] replace methods to * for exporting all methodes

reinstall_bibiwstest	(default) Will uninstall and install WebService on BiBiTest WebService Server also releasing WSDL on BiBiTest (use this after modifications of WebService)
deploy_bibiwstest	Will install and start WebService on BiBiTest WebService Server also releasing WSDL on BiBiTest (use this at 1st time installation)
undeploy_bibiwstest	Will remove and uninstall WebService from BiBiTest WebService Server and remove WSDL from BiBiTest
reload_bibiwstest	Will reload WebService on BiBiTest WebService Server
clean_dist	Will tidy up the project directory
api	Will generate the Java-Doc API to `doc/api/`

Table A.2: BiBiServ Developer commands of `wss/` subdirectory

`toolname` is your `<PN>`

`max.submit` is maximum submission characters over all parameters accepted by WebService

`statuscode.701` is an example of how to set a client side statuscode. This will override the description of the default BiBiStatuscodes if defined there. (see Table C.2 for complete listing of predefined BiBiStatuscodes)

`statuscode.701.internal` is the corresponding internal description, which will go to the log files additionally, but never occur outside.

Second there is the `log4j.properties`, which configures logging to `$CATALINA_HOME/spool/<PN>/wss.log` [3]. See log4j manual for details [LOG4J].

`src/`

All classes belonging to a WebService are included at the package `de/unibi/techfak/bibiserv/<PN>`.
Here are the basic constraints of the classes:
AXIS requires an interface to the implementation of the methods called by WebService. The implementation class creates an `id` and calls the processing class as a thread with checking for maximum submitting characters before. After starting the thread, it returns the `id` to the client side. The processing class itself calls the tool after preprocessing and before postprocessing by developer's implementation.

[3] `$CATALINA_HOME/` is basedirectory of Tomcat

Following is the detailed description of classes:

<PN>Interface.java The interface definition in Figure A.9 shows, that every

```
1  abstract String request_orig(String email, Object[] params) throws TEMPLATEWSException;
2  abstract String request(String email, Object[] params) throws TEMPLATEWSException;
3  abstract HashMap response_orig(String id) throws TEMPLATEWSException;
4  abstract String response(String id) throws TEMPLATEWSException;
```

Figure A.9: client to call `request()` on WebService

method to provide for a WebService call has to be defined here.
The two `request()` methods will return an `id` and get data for starting the WebService call. `request()` would take the data as specified by HOBIT-Project and `request_orig()` as the normal tool would like to have it. This example does not get any data, but a length as parameter.
The string `email` is the email notification address – possibly an empty string.
The two `response()` methods get an `id` and return results either in XMLSchema specified format or original tool format.

<PN>Implementation.java The methods `request()` and `request_orig()` (same for `response()` and `response_orig()`) are mainly the same, except for the mapping from and to the HOBIT-Project standard. The documentation concentrates on the `request()` and `response()` methods and mentions where to skip for `_orig` methods.
Figure A.10 shows the `request()` method for submitting a job to WebService. After

```
1  public String request(String email, Object[] array_params) throws TEMPLATEWSException {
2    WSSTools wsstools = new WSSTools(this.getClass().getResourceAsStream("/tool.properties"));
3    Status status = new Status(wsstools);
4    if (email != null && !email.equals("") && !wsstools.checkemail(email)) {
5      status.setStatuscode(708);
6      throw new TEMPLATEWSException(status);
7    };
8    Hashtable params = wsstools.getHashtableFromArray(array_params);
9    int submitsize = ((params.get("length")).toString()).length();
10   if(submitsize > Integer.parseInt(wsstools.props.getProperty("max.submit"))) {
11     status.setStatuscode(702);
12     throw new TEMPLATEWSException(status);
13   }
14   //warp input from HOBIT Standards to toolinput
15   TEMPLATEWSProcessing proc = new TEMPLATEWSProcessing(wsstools,status,
       email,(new Integer(params.get("length").toString())));
16   Thread t = new Thread(proc);
17   t.start();
18   return status.getId();
19 }
```

Figure A.10: Lines from <PN>Implementation.java

creating a new `status` object (line 3) the email address is checked – if submitted (line 4-7). While transfering a hash as array, this has to be converted (line 9). At line 10 the length of input is calculated, followed by a check whether this is smaller than the limit set in `tool.properties`. The input has to be transformed from HOBIT-Projcet input format to a format which is accepted by the tool. (line 14) (this step could be left out if called with `request_orig()`, because input is already in format accepted by the tool.) How to convert data is described in Chapter A.5.2. Afterwards the processing class is called as a thread. (line 15-17) `request()` returns an `id` (line 17) for calling `response()` after a while by the client side.
The `response()` method just requests the current status of the submitted id (line 3 at

```
1  public String response(String id) throws TEMPLATEWSException {
2    WSSTools wsstools = new WSSTools(this.getClass().getResourceAsStream("/tool.properties"));
3    Status status = new Status(wsstools,id);
4    if(status.getStatuscode() == 600) { // ready
5      String returnString = new String(wsstools.readSpoolFile("stdout-log.txt"));
6      //warp output to HOBIT Standards
7      return returnString;
8    } else {
9      throw new TEMPLATEWSException(status);
10   }
11 }
```

Figure A.11: Lines from `<PN>Implementation.java`

Figure A.11). If statuscode is 600 the job has finished and the result can be read (line 5). Afterwards the result has to be converted to HOBIT-Project standard for returning (line 6/7). How to convert your data is described in Chapter A.5.2. (Again this has to be skipped for `response_orig()` method.) If statuscode is different, `response()` just informs the client side by returning an Axis fault (throwing a `<PN>Exception`, which is converted by the AXIS library).

`<PN>Processing.java` After setting to statuscode 602 the preprocessing can be done, if any is required (seeFigure A.12 – line 2). Statuscode 603 means main processing, so the `command` execution is done (line 5), which returns after the job has finished or failed. Following line 9 there is the phase of postprocessing. In this example the STDOUT and STDERR streams of the execution call are written to the spooldirectory, because the example application prints the result to STDOUT. After all this is done, the statuscode is set to „finished" – 600 (line 12), which indicates a possible return of the result, if the client asks for it. At last the user is informed by email – `mail()` sends an email about status, if email is not empty. (line 13)

`<PN>Exception.java` Due to the fact that AXIS does not accept a general `BiBiException` class, which is mapped to a `soap_fault` on error, every WebService has to have its own exception to throw.
In Figure A.13 there are two constructors to create this exception. Either a status

```
1  public void run() {
2    status.setStatuscode(602);
3    status.setStatuscode(603);
4    SGECall call = new SGECall(wsstools, status);
5    boolean success = call.call("SYS CALL");
6    if(!success) {
7      return;
8    }
9    status.setStatuscode(605);
10   wsstools.writeSpoolFile("stdout-log.txt",call.getStdOutStream());
11   wsstools.writeSpoolFile("stderr-log.txt",call.getStdErrStream());
12   status.setStatuscode(600);
13   wsstools.mail(email,ststus);
14 }
```

Figure A.12: Lines from <PN>Processing.java

```
1   public TEMPLATEWSException(Status status)
2   public TEMPLATEWSException(int statuscode, String description)
```

Figure A.13: client to call request() on WebService

object can be given (line 1, will take statuscode and description from status object) or soap_fault and soap_description is set manually (line 2). So, normally, the developer has nothing to change here.

PerformanceTest.java This class is just a standalone script, which is called by the default WebService as an example. So it is not documented here and the developer should replace the execution of this class to their own tool's execution.

A.5.2 Converting result from/to specified XMLSchemas

The HOBIT-Projekt uses different existing XMLSchemas for representing data. This XMLSchemas describe formats to represent biological data in XML. If a program accepts or returns data in one of these standards, another tool can handle this data without converting. This way tools can be combined as chains.
So a BiBiWS WebService supporting HOBIT standards needs to accept and to return data within one of these XMLSchemas. It's up to the developer's decision which are best matching the tool's in- and output.
For a list of XMLSchemas have a look at the HOBIT Webpage [HOBIT].
Following is a description how to convert data.

Converting data according to a XMLSchema to tool's input format.

While the example WebService just takes a parameter, but no further input data, converting from XMLSchema conform data to tool input is not demonstrated.
Figure A.14 shows how to retrieving incoming XML. One just has to create a DOM

```
1       String xmlString = incoming_data;
2       DocumentBuilder docBuilder = factory.newDocumentBuilder();
3       Document doc = docBuilder.parse(new InputSource(new StringReader(xmlString)));
4       String stringOfLength = doc.getElementsByTagName("stringOfLength").item(1).getNodeValue();
```

Figure A.14: create a DOM object and access data

object from the incomming XML string and access the included data.

See [MCL] for further informations on the DOM library.

Converting tool's result format according to a XMLSchema

```
1     String stringOflength = new String(wsstools.readSpoolFile("stdout-log.txt"));
2     String logmsg = new String(wsstools.readSpoolFile("stderr-log.txt"));
3     //Creating DOM
4     DOMImplementationImpl impl = new DOMImplementationImpl();
5     Document domDocument =
        impl.createDocument("de:unibi:techfak:bibiserv:templateexample", "dataTypeTest", null);
6     Element dataTypeTestElement = domDocument.getDocumentElement();
7     dataTypeTestElement.setAttribute("xmlns", "de:unibi:techfak:bibiserv:templateexample");
8     dataTypeTestElement.setAttribute("xmlns:xsi", "http://www.w3.org/2001/XMLSchema-instance");
9     dataTypeTestElement.setAttribute("xsi:schemaLocation",
        "de:unibi:techfak:bibiserv:templateexample
        http://bibiserv.techfak.uni-bielefeld.de/xsd/TemplateExample.xsd");
10    Element stringOfLengthElement = domDocument.createElement("stringOfLength");
11    dataTypeTestElement.appendChild(stringOfLengthElement);
12    stringOfLengthElement.appendChild(domDocument.createTextNode(stringOflength));
13    Element logmsgElement = domDocument.createElement("logmsg");
14    dataTypeTestElement.appendChild(logmsgElement);
15    logmsgElement.appendChild(domDocument.createTextNode(logmsg));
```

Figure A.15: create a DOM object and fill with result data

Figure A.5.2 shows a basic conversion of result data to XML according to a XMLSchema. At this WebService, we have an example XMLSchema `http://bibiserv.techfak.uni-bielefeld.de/xsd/TemplateExample.xsd` to follow.
Line 1 and 2 gets the data from the spooldirectory.
Developers have to create a DOM object, including namespaces (lines 4-9), which is serializeable simple to return as string.
Lines 10 to 12 appends the returned string and lines 13 to 15 the log message.
Afterwards the serialization is done, which does not have to be modified. The generated string is returned to the client of the WebService call.

See [MCL] for further informations on the DOM library.

install_bibitest	(default) Will install the client side on BiBiTest
clean_bibitest	Will tidy up the project on BiBiTest

Table A.3: BiBiServ Developer commands of `wsc/` subdirectory

A.5.3 Command line client for testing server side

`cmd/` contains several simple Perl programs for testing the WebService side. Also, this could be the starting point for developing WebService clients for distribution to users, who would like to call WebService from remote machines.
For each WebService method (`request()`, `request_orig()`, `response()`,

```
1  my $WSCTools = new BiBiServ::WSCTools(toolprops=>"tool.properties");
2  my $status = new BiBiServ::Status(WSCTools=>$WSCTools);
3  my $TEMPLATEWS = new BiBiServ::WSC::TEMPLATEWS(WSCTools=>$WSCTools,Status=>$status);
4  my @params = ("length",$ARGV[0]);
5  $TEMPLATEWS->request($ARGV[1],params);
```

Figure A.16: client to call `request()` on WebService

`response_orig()`) there are separate programs. All of them look pretty simple, while using the tool Perl module `BiBiServ::WSC::<PN>.pm` (see Figure A.16).
After creating required objects `$WSCTools`, `$status` and the WebService specific `BiBiServ::WSC::<PN>`, the method `request()` is called and returns the `id` (line 5). This way all (four) default scripts work.

A.5.4 Client side of a BiBiWS tool

The `wsc/` directory contains several files and subdirectories, which are explained here. (see Figure A.3 for overview)

`Makefile`

While calling `make` at the `wsc/` directory the developer can install the client side of his/her WebService on BiBiTest. There are two different targets for developers described in Table A.3. These targets will use `install_data` and `clean_data`, which the developer may has to modify if adding files or directories to the WebService client side. For installation on BiBiServ a member of the BiBiServ Administrator Team has to be informed.

`cgi-bin/`

This directory contains the CGIs for submitting a WebService call on the server side and presenting the result to the user of BiBiServ. This documentation will explain step by step (code line by code line), what is done.

<PN>_submit will be called by the submission form to submit a WebService call.
In Figure A.17 the three objects from the BiBiWS client side package are created.

```
1  my $WSCTools=new BiBiServ::WSCTools(toolname=>$TOOLNAME);
2  my $status=new BiBiServ::Status(WSCTools=>$WSCTools);
3  my $WSCLayout=new BiBiServ::WSCLayout(WSCTools=>$WSCTools,Status=>$status);
```

Figure A.17: Lines from <PN>_submit

$WSCTools provides useful functions and access to the properties files.
$status is used for representing the current status of the WebService call.
Last, $WSCLayout gives access to the layout functions, for generating layout matching the BiBiServ corporate identity.
Figure A.18 loads the maximum submission size from properties. Error handling fol-

```
4  $CGI::POST_MAX = $WSCTools->props->getProperty("max.submit");
```

Figure A.18: Lines from <PN>_submit

lows below.
Figure A.19 shows the next lines. submission.html calls this CGI. It first checks the

```
5  if ($cgi->param('Action') eq "Submit") {
6    print $cgi->header();
7    if(!($cgi->param("inputfield") =~ /^\ d+$/)) {
8      $status->setStatuscode("c701");
9      print $WSCLayout->mainFrame(); #give $topSize,$leftSize here !
10     exit;
11   } elsif ($cgi->param("email") ne "" && $WSCTools->checkemail($cgi->param("email")) ne "") {
12     $status->setStatuscode("c708");
13     print $WSCLayout->mainFrame();
14     exit;
15   } else { #call WSS !!!
16     my $TEMPLATEWS =
           new BiBiServ::WSC::TEMPLATEWS(WSCTools=>$WSCTools,Status=>$status);
17     my @params = ("length",$cgi->param("inputfield"));
18     $TEMPLATEWS->request($cgi->param("email"),params);
19     print $WSCLayout->mainFrame(); #give $topSize,$leftSize here !
20 }
```

Figure A.19: Lines from <PN>_submit

size of submitted data (line 5). At this example the WebService accepts only integers – so if input is not an integer, the status is set to error and $WSCLayout->mainFrame() will give an error message to the user (line 7-9). Next check is validating the submitted email, if it is not empty (line 11-13). Before calling the WebService, we have to construct the array of parameters (line 17). Otherwise the CGI calls $TEMPLATEWS->request() for submitting the WebService (line 18).
$WSCLayout->mainFrame() will show a redirect frame to <PN>_redirect in this case

(line 19). If the result will be too long in any direction, one can manipulate the size of top and left result Frame. See Chapter 5.2 for details.

Lines from Figure A.20 are for retrieving result after email notification. If the user

```
21 } elsif (defined $cgi->param('bibiid')) {
22   print $cgi->header();
23   $status->setId($cgi->param('bibiid'));
24   print $WSCLayout->mainFrame();
```

Figure A.20: Lines from <PN>_submit

closes the browser and gets the notification email , a link is send, which calls the submit CGI with the id, which simply calls the mainFrame() method with the corresponding id of the job.

Figure A.21 shows on how to react if max.submit limit is exceeded. (line 25)

```
25 } elsif ($cgi->cgi_error()) { #upload size problem or other
26   print $cgi->header();
27   my $error = $cgi->cgi_error();
28   if($error && $error =~ /413/) {
29     $status->setStatuscode("c702");
30   } else {
31     $status->setStatuscode("c700");
32   }
33   print $WSCLayout->mainFrame();
```

Figure A.21: Lines from <PN>_submit

413 is the HTML error set by CGI.pm (line 28) , where the CGI just sets the statuscode and the $WSCLayout->mainFrame() will present an error message to the user (line 33).

If the call of CGI does not came from submission.html (line 34 at Figure A.22) pos-

```
34 } else {
35   print $cgi->redirect("/$TOOLNAME/submission.html");
36 }
```

Figure A.22: Lines from <PN>_submit

sibly a user has bookmarked the CGI and a redirection to submission.html is done (line 35).

<PN>_redirect will be called by user's browser (using an HTML redirect) until the response returned the result or a permanent error.

Figure A.23: redirect starts with checking whether an error exists to give back to user (line 1). If so, the statuscode is set (line 2) and an error frame will be shown. (line 4).

Figure A.24: If an id was submitted (line 5), the CGI tests if the result is already fetched from WebService server (line 8/9) and creates a default result page.

```
1  if($cgi->param("soap_code")) {
2    $status->setStatuscode($cgi->param("soap_code"),$cgi->param("soap_description"),"");
3    print $cgi->header();
4    print $WSCLayout->errorFrame();
```

Figure A.23: Lines from <PN>_redirect

```
5  } elsif ($cgi->param("bibiid")) {
6    $status->setId($cgi->param("bibiid"));
7    print $cgi->header();
8    my ($resultData,$spool) = $WSCTools->existsResult();
9    if ($resultData) {
10     print $WSCLayout->resultFrame($resultData,$spool);
```

Figure A.24: Lines from <PN>_redirect

Figure A.25: Otherwise the CGI has to fetch result from the WebService server

```
11    } else {
12     $TEMPLATEWS = new BiBiServ::WSC::TEMPLATEWS(WSCTools=>$WSCTools,Status=>$status);
13     $TEMPLATEWS->response_orig();
14     if($status->getStatuscode() eq "c600") { #finished !
15       my $data = $status->getResultRef();
16       my $dataDescription = {};
17       my $spool = $WSCTools->writeResult($data,$dataDescription);
18       if (defined $spool) {
19         print $WSCLayout->resultFrame($dataDescription,$spool);
20       } else {
21         $status->setStatuscode("c723");
23         print $WSCLayout->errorFrame();
24       }
25     } elsif ($status->getStatuscode() =~ /^ 6\ d\ d/){ #no error, so waiting
26       print $WSCLayout->redirectFrame();
27     } elsif ($status->getStatuscode() =~ /^ 7\ d\ d/){ #error !
28       print $WSCLayout->errorFrame();
29     } else { #should never occur !
30       print $WSCLayout->errorFrame();
31     }
```

Figure A.25: Lines from <PN>_redirect

by creating a BiBiServ::WSC::TEMPLATEWS object (see Chapter A.5.4) and calling response_orig() (line 12/13). If this call returns a code c600, result was fetched and CGI will write it to spooldirectory (line 17) and give back a default result frame (line 19).
This default result frame hasn't to be used. While it is a simple listing of file vs. description, the developer is welcome to present another return frame.
If another statuscode beginning with 6 is returned by WebService server, WebService call isn't finished yet, so will redirect again.
If code starts with 7 or anything else an error has occurred and CGI will present an

error frame (line 27/29).
Figure A.26: Like TEMPLATEWS_submit CGI will redirect to submission page, if called

```
32 } else {
33   print $cgi->redirect("/$TOOLNAME/submission.html");
34 }
```

Figure A.26: Lines from <PN>_redirect

without any parameters.

data/<PN>/

This directory contains all the static HTML pages of your project.
Installing these files using the Makefile will process these files. Like one can see at the Figure A.27 the BiBiServ layout (*left frame* and *top frame*) will be added.
This is done by some XSLT scripts of BiBiServ Administrator Team, so all files have to follow the XHTML Standard of W3C [XHTML].

framework.xml

For generating your tool navigation (*tool frame* Figure A.27) this configuration file is used by some scripts.
Generating a menu entry is quite simple like mentioned in Figure A.28. Just the text of the button and the link to the HTML <url> file has to be configured. External links are also possible by setting <url> to begin with http://.

vol/<PN>/etc/

vol/<PN>/etc/ includes two properties files.
The first is tool.properties (see Figure A.29).
There are some basic properties:

toolname is your <PN>

max.submit is maximum submission size for your submit CGI.

statuscode.c701 is an example of how to set a client side statuscode. This will override the description of the default BiBiStatuscodes, if defined. (see Table C.3 for complete listing of predefined BiBiStatuscodes)

statuscode.c701.internal is the corresponding internal description, which will go to the log files additionally, but never occur outside.

Second there is the log4perl.properties, which configures logging to /vol/bibitest/www/userlogs/templatews/wsc.log. See log4perl manual for details [LOG4P].

Figure A.27: A BiBiServ HTML page

```
<menue>
  <content>Welcome</content>
  <url target="_self">welcome.html</url>
</menue>
```

Figure A.28: Example entry of `framework.xml`

```
toolname=TEMPLATEWS
max.submit=1000000
statuscode.c701=Input Error - Tool spec explanation
statuscode.c701.internal=Input Error - Tool spec internal information
```

Figure A.29: default `tool.properties` file

`vol/<PN>/lib/`

This directory represents the tool specific Perl modules.
By default, only `BiBiServ::WSC::<PN>.pm` is located here. This module is used by the WebService client side on BiBiServ and also by the command line clients located in `cmd/`.
If your tool needs other perl modules on client side, the developer is welcome to install here.
But remember that there should not be any time/space consuming tasks on the client

side !
BiBiServ::WSC::<PN>.pm is a simple Perl module for accessing WebService server.
On creation (Figure A.30) this object needs a reference to a $WSCTools and a $status

```
$TEMPLATEWS = new BiBiServ::WSC::TEMPLATEWS(WSCTools=>$WSCTools,Status=>$status)
```

Figure A.30: Creation of a BiBiServ::WSC::TEMPLATEWS object

object, which will be manipulated to represent the returned status of call.
Figure A.31 shows one of the four default methods of the tool

```
1  sub request_orig {
2    my $self = shift;
3    my $email = shift;
4    my $params = shift;
5    $self->Status->setId(SOAP::Lite->service($self->URI)
6                                    ->on_fault(sub {$self->{Status}->soapFaultHandler(@_)})
7                                    ->request_orig($email,$params));
8    if($self->{Status}->getStatuscode() ne "") { #error occurrence and Soap::Lite will return
9      $self->{Status}->setId("error while calling");
10   }
11 }
```

Figure A.31: One method of BiBiServ::WSC::TEMPLATEWS

specific BiBiServ::WSC:<PN>.pm.
The parameters to call are submitted to this method (line 3/4) and passed to the request_orig() method. One would like to make a check of the parameters here.
On error the $status object has a special handling method for processing the SOAP::Lite's error objects. (line 5)
Similar methods exist for every method of the WebService with this Perl module.

A.5.5 Enhancements for a multi-step WebService

A multi-step WebService is a WebService, which requires user interaction after one step.
This is not calling more than one program, but if some input depends on previous calculation this is the way to do it.
The BiBiServ Administrator Team will create a two step WebService for the developer, which runs a default application like a one step project.
If a developer would like to have more than two steps, it should be pretty simple to enhance the two step version.
This describes the differences to a „normal“ one-step WebService.

server side

As the created example shows, there are additional methods to call with suffix _2ndstep. The request_2ndstep() gets in addition to the submitted data of second step the id of the WebService call for refering to results of the first step.
The rest is straight forward from the normal one-step WebService.

client side

Again the example two-step WebService shows the differences:
<PN>_redirect returns a second submission form instead of presenting a result. This submission calls the CGI <PN>_submit_2ndstep, which starts the second step of the WebService on the server side. <PN>_redirect_2ndstep is responsible for redirecting until the result is returned and presented to the user.

A.6 Debugging BiBiWS WebService

For debugging the starting point should look at log files of the WebService, which will be created on client and server sides. Also one should have a look at the written files in the spooldirectories. If the log level at log4perl.properties and log4j.properties is set to „DEBUG“ even the library give output of their process state.

/vol/bibitest/www/userlogs//<PN>/wsc.log Log file of the testing client side

/vol/bibitest/www/data/spool//<PN>/ Spooldirectory of the testing client side

$CATALINA_HOME/spool/<PN>/wss.log Log file of the testing server side

$CATALINA_HOME/spool/<PN>/ Spooldirectory of the testing server side

There are also client and server wide log files, which are more confusing due to the fact, that they are system wide. But they include more general debug messages of the server, like not start/interpret a CGI or class:

/vol/bibitest/www/logs/error Log file of the testing client side

$CATALINA_HOME/logs/catalina.out Log file of the testing server side

Corresponding log files of production system are not available for developers, due to the BiBiServ policies.

A.6.1 AXIS Tool: Tcpmon

If there are error occouring while calling your WebService and you changed the interface of a method or you are working on Serializers/Deserialziers, try the TcpMon of the AXIS Library.

This is a monitoring and forwarding programm for your SOAP Messages.
So your client has to connect to TcpMon, which will show you the SOAP Message and forward it to the Tomcat. The returned SOAP Message is also shown and forwarded back to your client.
So, what you have to do is this:

- Comment in at `wss/build.xml` within the `generate_wsdl` target: Line of server setting.
- Modify it to your favorite Server/Port where to run TcpMon
- Generate and publish on BiBiTest the WSDL pointing to your TcpMon by calling `ant`
- Start TcpMon on your favorite Server like this:
  `java org.apache.axis.utils.tcpmon [listenPort targetHost targetPort]`
- Let the Client call the WebService over TcpMon
- TcpMon Window will show you the submitted SOAP Messages.

A.6.2 ask for help

If you are in deep trouble, don't hasitate to contact the BiBiServ Administrator Team via email: bibi-help@techfak.uni-bielefeld.de

B BiBiWS Administration Guide

This guide documents configuration and installation of the BiBiWS framework.

Just for clarification:
A *developer* in this document is a person, who develops a bioinformatic tool and would like to provide a world-wide accessible WebService for this tool. A *user* is a person, who uses such a WebService. The BiBiServ Administrator Team's task is on the one hand to provide help for developing WebServices, and on the other hand answering questions coming from users. Users and developers can contact BiBiServ Administrator Team via email: bibi-help@techfak.uni-bielefeld.de
Overall there are three main programs that the project BiBiWS relies on:

Apache http Server Presenting HTML interface to WebService server side (acting as WebService client side) The servers present a user friendly HTML-form interface to the server side of WebServices.

Apache Tomcat Servlet container for the WebService server side.

Postgresql Database For server side to manage the WebService calls by `ids`.

B.1 Setting up Apache http Server

BiBiServ and BiBiTest are Apache http Servers accessible at [BIBISERV] and [BIBITEST]. The Apache http Server was set up by the Support Group of *Technical Faculty of the University Bielefeld*, version 1.3.29. BiBiWS does not use any specific extensions of this version, so every (newer) version should work.
While BiBiTest is hosting a test environment system, it is only internally accessible at the *Technical Faculty of the University Bielefeld*. Assuming a running Apache http Server environment, this section will explain installation requirements for getting BiBiWS to run.
Basically the http server has to provide Perl-CGIs. CGIs of BiBiWS require the following packages to be installed, which are all accessible from CPAN [CPAN]:

CGI for CGI support from Perl side

SOAP::Lite connecting to the WebService server side

Data::Dumper stores and prints out Perl objects

Log::Log4perl proper logging facility

Net::DNS checking MX record of an email address

Net::SMTP sending mails

File::Path simple accessing of files

B.1.1 Installation of the client side BiBiWS Library

The supporting client side library of BiBiWS is the `BiBiServ::` module at `/vol/bibidev/bibi/wsc/vol/bibi/lib` directory.
It will be installed during the installation of the whole `bibi` project, which also does general installation like generating the layout of BiBiServ (not part of this document).
The API documentation of the classes is located at [BIBIWSAPI-WSC]. For generating these APIs, call special target `make api` at `/vol/bibidev/bibi/wsc/` directory.
There is a general properties file `/vol/bibidev/bibi/wsc/vol/bibi/etc/<SERVER>.properties`, which will be installed on `<SERVER>`. [1] This properties file contains the following elements:

server the server name *bibitest* or *bibiserv*

mailfatal.email email address for log level *mailfatal*

mail.from sender email address of server sent emails

frame.bibiserv Location of general bibiserv environment layout

redirect.time time in seconds to wait before requesting results of WebService again (HTML redirect)

base.spooldir location of the spooldirectory

statuscode.cXXX default description of the statuscode generated on client side

statuscode.cXXX.internal default internal description of the statuscode generated on client side

statuscode.undef description of unknown statuscodes

statuscode.undef.internal internal description of unknown statuscodes

A complete list of default statuscodes can be found at Table C.3.

[1] `<SERVER>` is either BiBiTest or BiBiServ

B.2 Setting up Apache Jakarta Tomcat

BiBiServ WebService Server and BiBiTest WebService Server are Apache Jakarta Tomcat servers. The development of BiBiWS is based on Version 5.0.18, but higher versions should work, too.
Their only job is to provide the server side WebService access of the tools running on BiBiServ. Like the http BiBiTest, the BiBiTest WebService Server also is only internally accessible at the *Technical Faculty of the University Bielefeld*.
Again assuming a running Tomcat environment, this section will explain installation requirements for getting BiBiWS running.
Beside Tomcat's own libraries and requirements, BiBiWS needs some special libraries, which have to go to `$CATALINA_HOME/shared/lib/` [2]:

log4j-1.2.8.jar proper logging

mail.jar sending emails

postgresql.jar accessing database, where *id*s are managed (using JDBC Feature of Tomcat, this library must be located at `$CATALINA_HOME/common/lib/`)

The following libraries are also used by BiBiWS, but must be located at every WebService (`WEB-INF/lib/` directory) due to Tomcat restrictions on (un)deploy:

axis.jar AXIS library

commons-discovery.jar General Apache Library

Axis depends on the following libraries, which also have to be installed at the Tomcat server at `$CATALINA_HOME/shared/lib/`:

jaxrpc.jar Axis implements this interface API

saaj.jar Axis implements this interface API

wsdl4j.jar Generating WSDL and java source code

xercesImpl.jar Implementation of the Xerces XML parser

xml-apis.jar API of XML parser, belonging to Xerces

After installation of a new version of any library, the server has to be restarted for reloading the libraries! All this is done by installing the `bibi` project.

Tomcat supports a feature called *JDBC*, which gives all WebServices of a Tomcat Server connection to a database. This is less performance consuming than making own connections by every WebServices.
BiBiWS uses this for storing the statuscodes of a WebService call, so this database connection has to be configured. Just include the code of Figure B.1 to the file `$CATALINA_HOME/conf/server.xml` at the `<Host>` element.

[2] `$CATALINA_HOME/` is basedirectory of the Tomcat Server

```
<DefaultContext>
  <Resource name="jdbc/postgres" type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/postgres">
    <parameter>
      <name>maxWait</name>
      <value>5000</value>
    </parameter>
    <parameter>
      <name>maxActive</name>
      <value>4</value>
    </parameter>
    <parameter>
      <name>password</name>
      <value>YOUR_PASSWORD</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:postgresql://YOUR_HOST/bibiwstest</value>
    </parameter>
    <parameter>
      <name>driverClassName</name>
      <value>org.postgresql.Driver</value>
    </parameter>
    <parameter>
      <name>maxIdle</name>
      <value>2</value>
    </parameter>
    <parameter>
      <name>username</name>
      <value>YOUR_USER</value>
    </parameter>
  </ResourceParams>
</DefaultContext>
```

Figure B.1: JDBC Connector of Tomcat to access statuscode database

B.2.1 Installation of the server side BiBiWS Library

BiBiWS server side library is the package `de.unibi.techfak.bibiserv`, which is located in the `/vol/bibidev/bibi/wss/` directory.
For installation, there is a `build.xml`, which has several targets:

The API documentation of the classes is located at [BIBIWSAPI-WSS].

As on the client side there is a properties file for the BiBiWS server side library at `/vol/bibidev/bibi/wss/config/<WSSERVER>.properties` [3], which will be installed at the corresponding server.
This properties file contains the following elements:

[3] `<WSSERVER>` is either bibiwstest or bibiwsserv

`install_(bibiwstest|bibiwsserv)`	(default: `install_bibiwstest`) installs the required libraries and the BiBiWS server library with properties file to the server
`clean_(bibiwstest|bibiwsserv)`	removes the library from the server
`start_(bibiwstest|bibiwsserv)`	starts the Apache Tomcat server (for use at the host machine)
`stop_(bibiwstest|bibiwsserv)`	stops the Apache Tomcat server (for use at the host machine)
`api`	for generating the BiBiWS server side API

Table B.1: Commands for installation of BiBiWS Serverside

server the server name either *bibiwstest* or *bibiwsserv*

spooldir.base base directory of spool dir

mail.smtp.host smtp host for sending mails

mailfatal.email email address for log level *mailfatal*

mail.from sender email address of server sent emails

sge.root SGE_ROOT environment variable for all calls of SGE

sge.cell SGE_CELL environment variable for all calls of SGE

sge.qsub location of `qsub`

sge.qstat location of `qstat`

sge.batchfile.sh shell of the generated shell scripts for SGE job start

sge.qsub.params.user parameters of qsub (can be changed by tool)

sge.qsub.params.bibi parameters of qsub (can NOT be changed by tool)

sge.qstat.finishedparams `qstat` parameters to return list of finished jobs

sge.qstat.pendingparams `qstat` parameters to return list of pending jobs

sge.waittime time to wait until looking next if SGE finished job

chmod location of `chmod`

chmodparams.spooldir parameters for `chmod` for spooldirectory creation

chmodparams.batchfile parameters for `chmod` for generated SGE shell scripts

statuscode.*XXX* default description of this statuscode *XXX* generated on the server side

statuscode.*XXX*.internal default internal description of statuscode *XXX* enerated on the server side

statuscode.undef description of unknown statuscodes

statuscode.undef.internal internal description of unknown statuscodes

A complete list of default statuscodes can be found at Table C.2.

For installing and reloading a BiBiWS WebService using the management interface of Apache Tomcat, some properties are required. They are stored in the file `/vol/bibidev/bibi/wss/config/<WSSERVER>-admin.properties`.

manager.url URI of manager interface of Apache Tomcat

manager.username username for log in

manager.passwd password for log in

server.url URL of server to access (used for creating WSDL files)

The corresponding production system file is readable only for BiBiServ Administrator Team, so only they can (re)install/reload and generate WSDL Definition of a WebService for the production system.

B.3 Setting up database

For managing the `ids`, BiBiWS uses a SQL database. This does not has to be a Postgresql, but this is recommended, because development of BiBiWS based on Postgresql Version 7.2.4. Every database wich includes a JDBC library should be fine. The library creates one row per WebService call and updates the statuscode, if required. So an incoming `response()` could look up the status of the WebService call, which is identified by its `id`.
Assuming the database server is installed and the tomcat is set up for the database connection, Figure B.2 shows the installation commands in SQL.
A database with one table for managing the `ids` has to be created like described in

```
1  CREATE DATABASE wssstatus;
2  CREATE TABLE wssstatus (id varchar, toolname varchar, statuscode int,\
       description varchar, internaldescription varchar, sgeid int,\
       created timestamp, lastmod timestamp);
3  CREATE INDEX idindex ON wssstatus(id);
4  ANALYZE wssstatus;
```

Figure B.2: Installing database for BiBiWS (Postgresql syntax)

Figure B.2:

id id for identifying WebService

toolname name of tool

statuscode current statuscode of WebService call

description description of statuscode

internaldescription internal description of statuscode

sgeid if submitted to SGE, the Job Id of currently running job

created timestamp of submitting WebService call

lastmod timestamp of last modification

B.3.1 Cleaning up database and spooldirectroy

`cleanWSSSide` is a tool for cleaning up the database and spooldirectory to avoid overflows of the spooldirectory on testing and production WebService server side. It is located at `/vol/bibidev/bibi/wss/cleanWSSSide/` and has to get some properties from `/vol/bibidev/bibi/wss/cleanWSSSide/cleanWSSSide.properties`:

bibiwstest.properties location of `bibi.properties` of test WebService server

bibiwsserv.properties location of `bibi.properties` of production WebService server

cleantime delete spooldir of WebService call (days) and mark `id` as cleaned in database

deletetime delete `id` from database

It should be called regularly (possibly daily as cron job) and provides logging to `/vol/bibidev/bibi/wss/cleanWSSSide/cleanWSSSide.log`.

B.4 Creating WebService projects

`makeProjectWS <PN>` creates a new project with included example WebService. It is located at `/vol/bibiadm/bin/`. Overall this creates a running WebService from the project *templatews*.
While copying directory `/vol/bibidev/templatews/` including subdirectories and all files, this tool replaces any occurrence of the word `TEMPLATEWS` by `<PN>` in the filenames and content. `templatews` is replaced by lower case of `<PN>`.
For generating a two step WebService there is the program `makeProjectWS2step`, which does the same from `/vol/bibidev/templatews2step/` with keywords *TEMPLATEWS2step* and *templatews2step*.
The generated example WebService should be easy to expand to more than two steps upon request.

whole WebService:	/vol/bibidev/<PN>/
make install_serv	install client and server side on production systems
make clean_serv	clean client and server side on production systems
client side:	wsc/ subdirectory
make install_bibiserv	install client side
make clean_bibiserv	clean client side
server side:	wss/ subdirectory
ant redeploy_bibiwsserv	uninstall and install server side and release WSDL)
ant deploy_bibiwsserv	install server side and release WSDL (use for 1st install)
ant reload_bibiwsserv	install server side, reload and release WSDL
ant undeploy_bibiwsserv	undeploy server side and remove WSDL

Table B.2: BiBiServ Administrator Team commands of BiBiWS projects

B.5 Installation of WebService projects on BiBiServ and BiBiServ WebService Server

Upon request of the developer, the BiBiServ Administrator Team will install the tool on the production system. The files Makefile (client side) and build.xml (server side) contain special targets to call for installing on production system (see Table B.2). These targets perform the same like installation on the development system, so BiBiServ Administrator Team does not have to worry about tool specific file and directories extensions: If installation by the developer on development systems works, this will work on the production system, too.

C BiBiWS - Predefined statuscodes

Like the (for now) underlaying protocol http, BiBiWS returns statuscodes with descriptive messages to inform the user if the requested data isn't returned.
For not getting confused with http statuscodes, BiBiWS uses statuscodes beginning with `6` and `7`, while http uses statuscodes beginning with `1` to `4` (see RFC 2616 [RFC2616] for details).

This chapter gives an overview of the predefined statuscodes by BiBiWS.
They can be overridden by the tool specific statuscodes, simply by giving same numbers at the tool specific properties file.
There are some general conventions to follow explained in Table C.1, which are required, because BiBiWS decides on these rules whether an error occurred, the WebService finished or user has to wait.
Some statuscodes can be used for several reasons and it might be hard for the developer to decide, where it came from. So all statuscodes may have a internal description (`.internal` at the `properties` file) , which never leaves the server. It is just for debugging and development of the tools on server side. It can be set individually to enhance the normal description, which is presented to the user.

Predefined statuscodes occurring on server side are explained in Table C.2 and the client side ones are described in Table C.3. Their symbolic names begin in general with a `c`, so the statuscodes can be as equal as possible on both sides.

Statuscode	Description
600	WebService call is finished successfully, result ready.
6xx (beginning with 6)	WebService call is not yet finished. `xx` gives more information.
700	WebService call is in general error state. This is a fall-back, should be avoided by developers to give enhanced information.
70x (beginning with 70)	User errors. Submitted data is NOT processed by the WebService call.
72x (beginning with 72)	Errors relaying to BiBiServ Administrator Team or developer of the tool.

Table C.1: General BiBiWS conventions for statuscodes

Statuscode	Description
600	Finished OK
601	Submitted
602	Preprocessing
603	Processing: Pending
604	Processing: Running
605	Postprocessing
700	General Error
701	Input Format Error
702	Input Size Error (submitted data to large)
703	Exec Error (executable gives enhanced errormsg)
704	RAM Size Error
705	CPU Time Error
706	ID unknown (or older than 30 days)
707	ID data deleted (older than 3 days)
708	Mail Check Failed (notification email is not valid)
720	WSS Server Busy
721	Internal Resource Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (Grid)
722	Internal Resource Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (DB)
723	Internal Resource Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (HDDfull)
724	Internal Resource Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (WS-Error)
725	Internal Resource Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (BiBiWSS-Lib Error)
731	Resource Busy *internal description:* Resource Busy (Grid)
732	Resource Busy *internal description:* Resource Busy (DB)
733	Resource Busy *internal description:* Resource Busy (HDDfull)

Table C.2: Predefined BiBiWS statuscodes on server side

Statuscode	Description
c600	Ready - got result
c601	Submitted
c700	General unknown error
c701	Input Error
c702	Input size too large
c708	Mail Check Failed (notification email is not valid)
c723	Internal Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (HDDfull)
c724	Internal Error - BiBiServ Team is informed, please try again later. *internal description:* Internal Resource Error (WS-Error)

Table C.3: Predefined BiBiWS statuscodes on client side

Literaturverzeichnis

[ADA1] Adams, H.: *Asynchronous operations and WebService, Part 1: A primer on asynchronous transactions.*
12.10.2004 <http://www-106.ibm.com/developerworks/library/wsasynch1/ >

[ADA2] Adams, H.: *Asynchronous operations and WebService, Part 2: Programming patterns to build asynchronous WebService.*
12.10.2004 <http://www-106.ibm.com/developerworks/library/wsasynch2/ >

[APACHE] *The Apache Software Foundation*
12.10.2004 <http://www.apache.org/ >

[APACHEHTTP] The Apache Software Foundation: *Apache http Server*
12.10.2004 <http://httpd.apache.org/http/ >

[APALIZ] *The Apache Software Foundation - Licenses*
12.10.2004 <http://www.apache.org/licenses/ >

[AXIS] The Apache Software Foundation: *AXIS*
12.10.2004 <http://ws.apache.org/axis/

[AXISUM] The Apache Software Foundation: *AXIS User's Manual*
12.10.2004 <http://ws.apache.org/axis/java/user-guide.html >

[BIBISERVSTATS] *Statistiken des BiBiServ*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/statistics/>

[BIBISERVPOLICIES] *Policies of BiBiServ* 12.10.2004
<http://bibiserv.techfak.uni-bielefeld.de/bibi/Administration_Policies.html >

[BIBISERV] *Technische Fakultät der Universität Bielefeld*: *The Bielefeld University Bioinformatics Server (BiBiServ)*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/ >

[BIBISERV] *Der BiBiServ (production system)*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/ >

[BIBITEST] *Der BiBiTest (development system)*
12.10.2004 <http://bibitest.techfak.uni-bielefeld.de/ >

[BIBIWSAPI-WSS] *BiBiWS server side API*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/hobit/wss-api/ >

[BIBIWSAPI-WSC] *BiBiWS client side API*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/hobit/wsc-api.html >

[BIBIWSDLS] *WSDL Definitionen der auf BiBiServ angeboteten WebServices*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/wsdl/>

[CEBITEC] *Center for Biotechnology (CeBiTec)*
12.10.2004 <http://www.cebitec.uni-bielefeld.de/ >

[CHAP] Chappell D. and Jewell, T.:
Java Web Services O'Reilly, 2002

[CORBA] Sun Microsystems: *Introduction to CORBA*
12.10.2004 <http://java.sun.com/developer/onlineTraining/corba/corba.html >

[CPAN] *CPAN - Comprehensive Perl Archive Network*
12.10.2004 <http://www.cpan.org/ >

[DFG] *Deutsche Forschungsgemeinschaft*
12.10.2004 <http://www.dfg.de/ >

[DOM] W3C: *Document Object Model (DOM)*
12.10.2004 <http://www.w3.org/DOM/ >

[DRMAA] *Distributed Resource Management Application API Working Group*
12.10.2004 <http://www.drmaa.org/ >

[E2G] J. Krger, A. Sczyrba, S. Kurtz, R. Giegerich : *e2g - An Interactive Web-Based Server for Efficiently Mapping large EST and cDNA sets to Genomic Sequences*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/e2g/ >

[GSOAP] *gSOAP: C/C++ Web Services and Clients*
12.10.2004 <http://gsoap2.sourceforge.net/ >

[HELMHOLTZ] *Helmholtz Association of National Research Centres*
12.10.2004 <http://www.helmholtz.de/ >

[HERM] Hermjakob, Henning et. al.:
The HUPO PSI Molecular Interaction Format - A community standard for the representation of protein interaction data Nature Biotechnology, 2004

[HOBIT] *HOBIT-Projekt*
12.10.2004 <http://mips.gsf.de/proj/hobit/ >

[HTTP] W3C: *Hypertext Transfer Protocol – HTTP/1.1*
12.10.2004 <http://www.w3.org/Protocols/rfc2616/rfc2616.html >

[JAVA] Sun Microsystems: *Java*
12.10.2004 <http://www.java.com/ >

[JCP] *Java Comunity Process - Homepage*
12.10.2004 <http://jcp.org/en/home/index/ >

[JRMI] David Reilly: *Introduction to Java RMI*
12.10.2004 <http://www.javacoffeebreak.com/articles/javarmi/javarmi.html >

[JSP] *JavaServer Pages Technology*
12.10.2004 <http://java.sun.com/products/jsp/ >

[JSC] *Java Servlet 2.4 Specification*
12.10.2004 <http://www.jcp.org/aboutJava/communityprocess/final/jsr154/ >

[LOG4P] *The log4perl project*
12.10.2004 <http://log4perl.sourceforge.net/ >

[LOG4J] Apache Software Foundation: *Logging Services*
12.10.2004 <http://logging.apache.org/log4j/ >

[MCL] McLaughlim, B.:
Java and XML O'Reilly, 2001

[NETCRAFT] Netcraft: *Web Server Survey* 12.10.2004
<http://news.netcraft.com/archives/2003/08/01/august_2003_web_server_survey.html
>

[OEST] Oestereich, B.:
Objektorientierte Softwareentwicklung: Analyse und Design mit der UML Oldenbourg, 2004

[PERL] O'Reilly: *Perl - The Source for Perl*
12.10.2004 <http://www.perl.com/ >

[PSI] HUPO: *Proteomics Standards Initiative*
12.10.2004 <http://psidev.sf.net/ >

[RAY] Ray, R. J. and Kulchenko, P.:
Programming Web Services with Perl O'Reilly, 2002

[REPUTER] S. Kurtz, C. Schleiermacher: *REPuter: Fast Computation of Maximal Repeats in Complete Genomes*
12.10.2004 <http://bibiserv.techfak.uni-bielefeld.de/reputer/ >

[RFC2616] *RFC 2616: Hypertext Transfer Protocol - HTTP/1.1*
12.10.2004 <http://www.w3.org/Protocols/rfc2616/rfc2616.html >

[ROSE] Rose, Marshall T.:
Beep - The Definition Guide O'Reilly, 2002

[RPCVSDOC] *RPC vs. Document WSDL encoding*
12.10.2004 <http://www.rassoc.com/gregr/weblog/archive.aspx?post=465 >

[SGE] Sun Mircosystems: *Grid Engine*
12.10.2004 <http://gridengine.sunsource.net/ >

[SGEUM] Sun Mircosystems: *Sun grid engine user manual*
12.10.2004 <http://gridengine.sunsource.net/project/gridengine-download/SGE53AdminUserDoc.pdf?content-type=application/pdf>

[SOAP] W3C: *SOAP Version 1.2*
12.10.2004 <http://www.w3.org/TR/soap12-part1/ >

[TANE] Tanenbaum, A. S.:
Computer Networks Prentice Hall, 1996

[TOMCAT] The Apache Software Foundation: *Apache Jakarta Tomcat*
12.10.2004 <http://jakarta.apache.org/tomcat/ >

[UDDI] *OASIS UDDI*
12.10.2004 <http://www.uddi.org/ >

[UDDIPROB] *UDDI Specifications*
12.10.2004 <http://www.oasis-open.org/committees/uddi-spec/ipr.php >

[WSDEF] *Web Services Activity*
12.10.2004 <http://www.w3.org/2002/ws/ >

[WSDL] W3C: *Web Services Description Language (WSDL) 1.1*
12.10.2004 <http://www.w3.org/TR/wsdl >

[XHTML] W3C: *XHTML 1.0 - The Extensible HyperText Markup Language*
12.10.2004 <http://www.w3.org/TR/xhtml1/ >

[XML] W3C: *Extensible Markup Language (XML)*
12.10.2004 <http://www.w3c.org/XML/ >

[XMLRPC] *XML-RPC Home Page*
12.10.2004 <http://www.xmlrpc.com/ >

[XMLSCHEMA] W3C: *XML Schema*
12.10.2004 <http://www.w3.org/XML/Schema >

Abbildungsverzeichnis

Index