Einsatz von und Erfahrungen mit Web Service-Technologien

Arbeitspaket 3.3

Guido Scherp, Ludger Bischofs
OFFIS – Bereich Energie
Escherweg 2
26121 Oldenburg

Telefon: +49 441 9722-0
Telefax: +49 441 9722-102
E-Mail: guido.scherp@offis.de,
ludger.bischofs@offis.de
Internet: http://www.offis.de

Jochen Rehwinkel
meteocontrol GmbH
Spicherer Straße 48
86157 Augsburg

Telefon: +49 821 3466651
Telefax: +49 821 3466611
E-Mail: rehwinkel@meteocontrol.de
Internet: www.meteocontrol.de

1 Einleitung

Web Service-Technologien werden eingesetzt, um Dienste insbesondere für externe Nutzer anzubieten, die über einheitliche, standardisierte Schnittstellen und Protokolle genutzt werden können. Häufig wird dabei lediglich der Zugriff auf eine Altanwendung über einen Web Service „gewrapped“. Web Services ermöglichen zudem eine einfache Integration mehrerer Dienste, z.B. auf Basis von Workflows.

In dem Projekt WISENT gibt es zwei Anwendungen, für die entsprechende Web Service-Wrapper implementiert wurden. Dazu gehören ein Dienst zur Simulation von Photovoltaikanlagen (PV-Sim) sowie ein Dienst zur Kapselung des in Arbeitspaket 2.2 entwickelten MSG-Tailor-Systems (MSG-Tailor), die von meteocontrol (PV-Sim) sowie von DLR DFD (MSG-Tailor) angeboten werden.

Parallel dazu realisierte DLR-TT im EU-Projekt MESOR einen Webservice zur Extraktion von SOLEMI-Strahlungszeitserien mit Hilfe von JBoss und einer mit Eclipse erzeugten WSDL-Beschreibung (http://www.webservice-energy.org /services/solemi.html). Diese Technologie scheint sich im Rahmen des Global Earth Observation System of Systems (GEOSS) international als Element der Datenverteilarchitektur durchzusetzen.

Im Folgenden wird näher auf Umsetzung der PV-SIM und MSG-Tailor Web Services eingegangen, sowie von den Erfahrungen im Umgang mit diversen Web Service-Technologien, die dabei gemacht wurden.

2 Anbindung der Solarsimulation (PV-Sim) an einen Web-Service

Es handelt sich hierbei um die Anbindung der Simulation von Solaranlagen (AP 5.3) an andere Systeme. Es wurde eine Web-Service-Schnittstelle geschaffen, die es möglich macht, vom Internet aus Simulationen im Serverpark von meteocontrol anzustoßen und das Resultat abzufragen. Im Zuge dieser Tätigkeiten war es auch notwendig, die Simulation zu erweitern und an die neuen Anforderungen anzupassen. Aus diesem Grund wird dieses Dokument in zwei Teile gegliedert. Im ersten Teil wird die speziell für den Web-Service erstellte Simulation beschrieben. Der zweite Teil beleuchtet den Web-Service selbst und das dazugehörige Logging.

Durch den in AP 3.3 realisierten Solar-Web-Service wurde für meteocontrol die Voraussetzung für eine neue kommerzielle Dienstleistung geschaffen. Kunden können jetzt live erwartete Solaranlagenerträge abfragen und diese Abfrage in eigene Softwareprodukte einbinden. Ein konkret realisierter Fall ist die Einbindung in eine Angebotserstellung zum Solaranlagenbau. Der Vertrieb des Kunden kann jetzt vor Ort die erwartete erzeugte Energie für die gewünschte Anlagenkonfiguration berechnen.

2.1 Die Ertragssimulation

2.1.1 Projektplan zur Simulationsprogrammierung

Die Haupttätigkeit von meteocontrol im AP 5.3 ist die Erstellung und Parallelisierung einer eigenen Ertragssimulation von Solaranlagen. Die Entwicklung wird in drei Versionen vollzogen:

- Version 1: Erstellung einer Simulation, die den jährlichen erwarteten Ertrag einer noch nicht gebauten Solaranlage errechnet. Es wird ein Programm entwickelt, das auf Grund einer vorgegebenen Solaranlagenkonfiguration (Wechselrichter, Module, Verschaltung, etc.) und der meteocontrol-Wetterdaten (Einstrahlung, Temperatur) des geplanten Standorts die erwartete erzeugte Energie simuliert. Version 1 wird vor allem von Kunden von meteocontrol benötigt, es ist also notwendig eine Live-Schnittstelle zur Simulation zu bieten. Dies geschieht in Form eines Web-Service. Die Programmiertätigkeiten für Version 1 sind nicht genau auf AP 3.3 und AP 5.3 zu trennen. Aus diesem Grund wird in diesem Dokument ein Überblick über die gesamte Version 1 gegeben.
- Version 2: Die Simulation in Version 2 greift direkt auf Solaranlagenkonfigurationen im System von meteocontrol zu. Es handelt sich hierbei um die ca. 5000 Solaranlagen, die momentan überwacht werden. Ziel ist hier eine tägliche Simulation des erwarteten Ertrags des Vortages jeder Anlage im System. Im meteocontrol Internetportal Safer'Sun kann dann die Simulation mit den tatsächlich gemessenen Erträgen verglichen werden. Weicht die Simulation stark vom tatsächlichen Ertrag ab, so ist ein Defekt bei der Solaranlage zu vermuten, und es wird ein Alarm ausgelöst. Die Tätigkeiten innerhalb Version 1 und 2 überschneiden sich, da teilweise der gleiche Programmcode benutzt werden kann. Die Programmierung der Version 2 findet in AP 5.3 statt.
- Version 3: Die Programmierung von Version 3 zielt auf die Parallelität des Simulationsvorgangs ab und soll auf dem Rechencluster des WISENT-Partners OFFIS realisiert werden. Version 2 wird durch Erweiterung zu Version 3. Es wird aber auch Programmcode von Version 1 weiterverwendet werden. Die Tätigkeiten für Version 3 zählen auch zu AP 5.3.

2.1.2 Version 1 der Ertragssimulation

Überblick

Bei Version 1 ist die Schnittstelle zum Web-Service unbedingt notwendig, ansonsten gibt es keine Verwendung für diese Simulationsart. In der Regel sind Verkäufer von Solaranlagen direkt vor Ort beim Kunden und erstellen live im Internet ein spezielles auf die Kundenbedürfnisse angepasstes Angebot. Mit einer lauffähigen Version 1 ist es nun möglich, das Angebot um einen erwarteten Jahresertrag der geplanten Solaranlage zu erweitern.

Über den Web-Service-Client wird eine Anfrage über das Internet an den Web-Service im Serverpark von meteocontrol verschickt. Die notwendigen

Konfigurationsdaten der Solaranlage werden übermittelt. Die Simulation wird gestartet und greift dabei auf die Datenbank von meteocontrol zu. Es werden Wetterdaten sowie technische Informationen über verwendete Wechselrichter und Solarmodule ausgelesen. Der erwartete Jahresertrag wird abschließend dem Web-Service-Client übermittelt.

Im Zuge der Erstellung von Version 1 wurde auch eine Logging-Einheit programmiert, die es möglich macht, jede Funktion jedes Web-Services im meteocontrol-System zu überwachen und den Aufruf eines Kunden zu protokollieren. Die Kosten für den Kunden werden in die Datenbank geschrieben und können dann in Rechung gestellt werden. Dieser Logging-Mechanismus kann auch für weitere Web-Services, die im Rahmen von AP 3.3. erstellt werden, genutzt werden.

Die Web-Service-Schnittstelle kann vom Kunden in beliebigen Programmcode eingebunden werden. Es ist z.B. möglich, den Web-Service-Client direkt in Programme zur Angebotserstellung einzubetten.

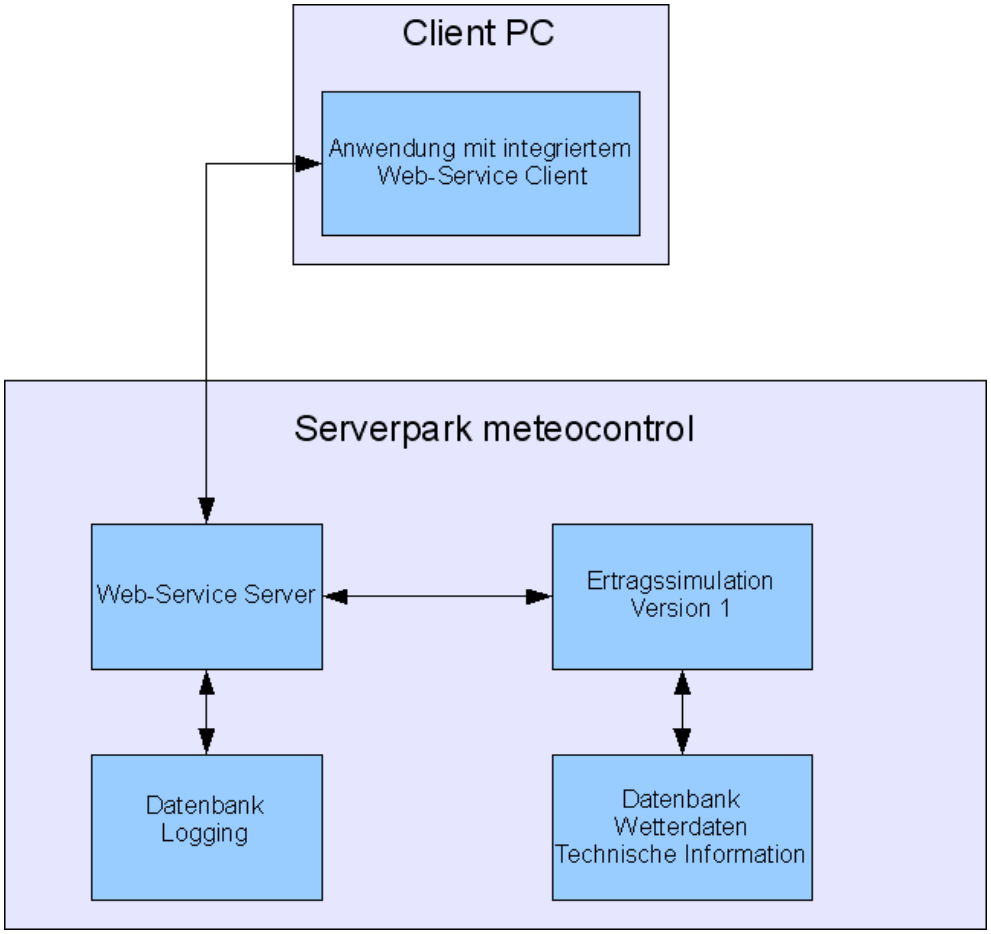


Abbildung 1 zeigt den schematischen Verlauf der Ertragssimulation. Endergebnis ist der zu erwartende Jahresertrag einer vom Kunden geplanten Solaranlage.

Verlauf der Ertragssimulation Version 1

Im Folgenden wird die innere Programmlogik der Ertragssimulation Version 1 dargestellt. Es handelt sich hierbei um den in PHP programmierten Rechenkern, der dann vom Web Service-Server über entsprechende Programmschnittstellen gesteuert werden kann.

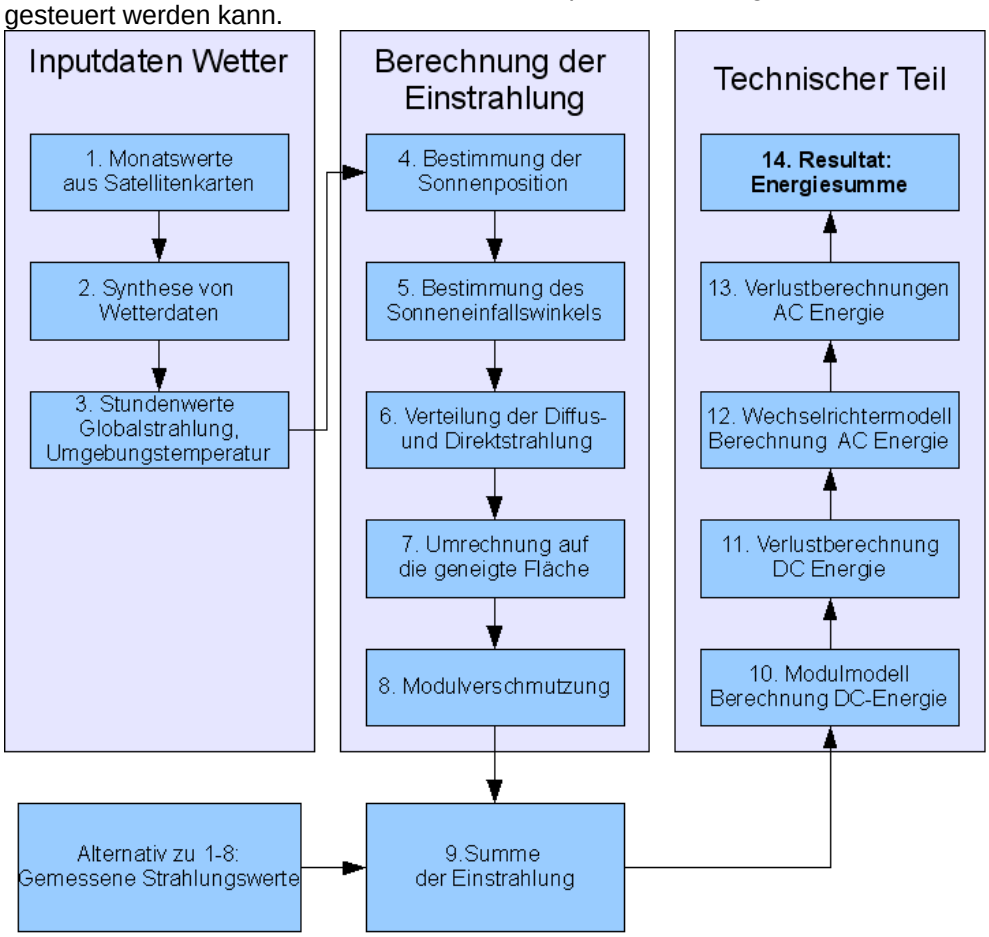


Abbildung 2: Schematische Darstellung der Programmlogik von Version 1

Die in Abbildung 2 dargestellten notwendigen Schritte zur Ertragssimulation werden im Folgenden näher erläutert. Ein Durchlauf von Schritt 1 bis 14 wird immer für einen Kalendertag durchgeführt. Soll ein ganzes Jahr berechnet werden, so wird die Simulation 365 mal durchgeführt. Es ist also generell möglich, den erwarteten Ertrag für einen beliebig großen Zeitraum zu berechnen

1. Schritt 1: Monatswerte aus Satellitenkarten: Es werden aus vorhandenen digitalen Satellitenkarten Wetterdaten für den Standort der geplanten Solaranlage ausgelesen. Es handelt sich hierbei um Globalstrahlung und Umgebungstemperatur. Es gibt nur einen Wert pro Monat.
2. Schritt 2: Berechnung von synthetischen Intervallwerten. Für die Simulation werden Intervallwerte im Stundenrhythmus benötigt. Aus diesem Grund wird eine Tagesverteilung der Satellitendaten auf Basis von statistischen Verfahren und physikalischen Zusammenhängen errechnet.
3. Schritt 3: Das Ergebnis sind Stundenwerte für die Globalstrahlung und die Umgebungstemperatur.
4. Schritt 4: Bestimmung der Sonnenposition für den zu berechnenden Tag
5. Schritt 5: Bestimmung des Sonneneinfallswinkels auf Grund des Standorts der Solaranlage
6. Schritt 6: Die gewürfelte Einstrahlung wird aufgeteilt in Diffus- und Direktstrahlung.
7. Schritt 7: Die berechnete Einstrahlung wird umgerechnet auf die Neigung der Solaranlagenmodule. Die Neigung ist ein Parameter der dem Web-Service übergeben wird.
8. Schritt 8: Eine simulierte Modulverschmutzung verringert die Einstrahlung auf die Solarmodule.
9. Schritt 1-8 alternativ: Bei vor Ort gemessener Einstrahlung kann sofort im Schritt 9 begonnen werden. Diese Alternative ist vor allem in Version 2 und 3 möglich. Es können hier die an bereits gebauten Solaranlagen befestigten Einstrahlungssensoren verwendet werden. Ist eine Anlage noch in Planung, besteht diese Möglichkeit nicht.
10. Schritt 9: Die erwartete Summe der Einstrahlung, die auf die Solarmodule trifft, wird auf Stundenbasis berechnet.
11. Schritt 10: Es wird die erwartete Energie auf der Gleichstromseite der Solaranlage berechnet. Hierzu werden die technischen Daten der verwendeten Solarmodule benötigt. Außerdem ist es erforderlich zu wissen, wie die Module miteinander verbunden sind (Strings). Es wird der so genannte modulabhängige Faktor pro Wechselrichtereingang bestimmt. In diesem Schritt fließen auch die ermittelten Temperaturwerte mit ein.
12. Schritt 11: Es werden Verluste (z. B. Kabelverluste) auf der Gleichstromseite berechnet.
13. Schritt 12: Es wird die erwartete Energie auf der Wechselstromseite berechnet. Hierzu wird das Wechselrichtermodell verwendet. Es werden technische Daten zum ausgewählten Wechselrichterhersteller aus der Datenbank ausgelesen.
14. Schritt 13: Es werden Verluste (z. B. Kabelverluste) auf der Wechselstromseite berechnet.
15. Schritt 14: Es wird die simulierte Energiesumme des Tages und letztendlich des Gesamtzeitraumes berechnet.

2.2 Der Solar-Web Service Version 1

2.2.1 Die Funktionen des Web-Service

Bei der Programmierung des Web-Service wurde die PHP-Bibliothek NUSOAP verwendet. NUSOAP bietet alle technischen Möglichkeiten, die eine

standardisierte Web-Service-Kommunikation erfordert. Der Web-Service bietet folgende Funktionen, die dann vom Web-Service-Client aufgerufen werden:

1. function doLogin
2. Übergabeparameter: Benutzerkennung, Passwort
3. Rückgabewert: Session-ID
4. Es wird die Benutzerkennung und das Passwort des Kunden übergeben. Username und Passwort sind in der generellen Web-Service Konfigurationsdatenbank abgespeichert. Es wird eine Session-ID zurückgegeben. Bei jedem Aufruf einer anderen Web-Service-Funktion wird die Session-ID mit übergeben. So kann jeder Aufruf einer Funktion dem Kunden zugeordnet werden und dementsprechend verrechnet werden.
5. function getAllModulTyp
6. Übergabeparameter: Session-ID
7. Rückgabewert: Alle für die Simulation verfügbaren Solarmodule und deren ID's
8. Der Web-Service-Client kann hier alle dem Kunden verfügbaren Solarmodule abfragen. Es werden der Hersteller und der Modelltyp übermittelt. Diese Information kann beim Client z. B. dafür benutzt werden, im Programm zur Angebotserstellung eine Auswahlliste von allen konfigurierbaren Solarmodultypen bereitzustellen.
9. function getAllWrTyp
10. Übergabeparameter: Session-ID
11. Rückgabewert: Alle für die Simulation verfügbaren Wechselrichtermodelle und deren ID's
12. Der Web-Service Client kann hier alle dem Kunden verfügbaren Wechselrichtermodelle abfragen. Es werden der Hersteller und der Modelltyp übermittelt. Diese Information kann beim Client z.B. dafür benutzt werden, im Programm zur Angebotserstellung eine Auswahlliste von den konfigurierbaren Wechselrichtern bereitzustellen.
13. function getSimErtrag
14. Übergabeparameter: Session-ID, Postleitzahl, Neigung, Ausrichtung, Wechselrichter-ID, Wechselrichteranzahl, Modul-ID, Modulanzahl
15. Rückgabewert: Erwarteter Jahresertrag in kWh
16. Der Simulation werden alle benötigten Eingangsparameter übergeben. Die Postleitzahl dient zur Ermittlung des Standorts mit Längen- und Breitengrad. Mit Neigung ist die Neigung der Solarmodule in Grad gemeint. Die Ausrichtung bezieht sich auf die Himmelsrichtung in die die Module zeigen, sie wird ebenfalls in Grad angegeben. Die Wechselrichter-ID sowie die Modul-ID dienen zur Identifikation der verwendeten Hardware der Solaranlage. Die ID's wurden in der Regel vorher vom Client über die Funktionen getAllWrTyp und getAllModulTyp beschafft. Zusätzlich wird vom Client noch übergeben, wie viele Module und Wechselrichter verwendet werden. Der Rückgabewert ist das eigentliche Resultat der Simulation. Es kann folgende Aussage getroffen werden: „Für die Solaranlage mit der übermittelten Konfiguration wurde ein erwarteter Jahresertrag von XXX,XX kWh simuliert."

2.2.2 Die Konfigurationsdatenbank und das Logging

Die Konfigurationsdatenbank von meteocontrol für die Web-Services ist so aufgebaut, dass jeder weitere Web-Service ebenfalls verwaltet werden kann. Die Web-Service Konfigurationsdatenbank enthält folgende Tabellen:

- T_KUNDE: Hier wird generelle Information zum Aufrufer des Web-Service gespeichert, sowie Passwort und Benutzerkennung.
- T_WEBSERVICE: Hier werden die Web-Services eingetragen. Die gesamte Solarsimulation Version 1 ist durch einen Eintrag in dieser Tabelle dargestellt.
- T_FUNKTION: Hier wird jede Funktion eines Web-Services hinterlegt
- T_KOSTEN: Hier wird jede Funktion eines Web-Services für jeden Kunden mit Kosten belegt. Es ist also möglich, eine genaue Preisdifferenzierung pro Kunde vorzunehmen.
- T_REl_KUNDEN_MODUL: Hier wird hinterlegt, welcher Kunde auf welche Solarmodule zugreifen darf. Die Funktion getAllModulTyp liefert alle Einträge eines Kunden.
- T_REL_KUNDEN_WR: Hier wird hinterlegt, welcher Kunde auf welche Wechselrichtertypen zugreifen darf. Die Funktion getAllWRTyp liefert alle Einträge eines Kunden.
- T_LOGGING: Hier wird hinterlegt, welcher Kunde zu welchem Zeitpunkt eine bestimmte Funktion eines Web-Service aufgerufen hat. Zusätzlich werden die dadurch entstandenen Kosten mitprotokolliert (siehe Abbildung 3).

←T→	ID_LOGGING	TIMESTAMP ▲	ID_KUNDE	ID_FUNKTION	SESSIONID	KOSTEN
	4	2004-06-22 14:53:44	1	4	1087908816_1	144
	8	2004-06-22 14:55:14	1	4	1087908910_1	144
	12	2004-06-22 14:55:30	1	4	1087908918_1	144
	16	2004-06-22 14:56:37	1	4	1087908991_1	144
	20	2004-06-22 14:59:38	1	4	1087909172_1	144
	24	2004-06-22 14:59:47	1	4	1087909181_1	144
	28	2004-06-22 15:23:56	1	4	1087910631_1	144
	32	2004-06-24 12:02:00	1	4	1088071305_1	144
	36	2004-06-24 12:05:43	1	4	1088071538_1	144
	40	2004-06-24 12:07:25	1	4	1088071641_1	144
	59	2004-06-24 15:34:08	1	4	1088084038_1	144
	63	2004-06-24 15:34:20	1	4	1088084056_1	144

Abbildung 3: Auszug aus der Web-Service Tabelle T_LOGGING

```
//--------------------------------------------------
$server->register('getAllModulTyp',                 // method name
    array('sSessionId' => 'xsd:string'),            // input parameters
    array('sModulTyp'  => 'xsd:string'),            // output parameters
          'urn:anlagenertrag',                      // namespace
          'urn:anlagenertrag#getModulTyp',          // soapaction
          'rpc',                                    // style
          'encoded',                                // use
          'Returns available Modules'               // documentation
);

//--------------------------------------------------
$server->register('getAllWrTyp',                    // method name
    array('sSessionId' => 'xsd:string'),            // input parameters
    array('sWrTyp'     => 'xsd:string'),            // output parameters
          'urn:anlagenertrag',                      // namespace
          'urn:anlagenertrag#getWrTyp',             // soapaction
          'rpc',                                    // style
          'encoded',                                // use
          'Returns available Converters'            // documentation
);
//--------------------------------------------------
$server->register('getSimulation',                  // method name
    array('sSessionId'   => 'xsd:string',
          'iPlz'         => 'xsd:int',
          'iNeigung'     => 'xsd:int',
          'iAusrichtung' => 'xsd:int',
          'iIdModul'     => 'xsd:int',
          'iIdWr'        => 'xsd:int',
          'iModulAnzahl' => 'xsd:int',
          'iWRAnzahl'    => 'xsd:int'),            // input parameters
    array('iSimErtrag'   => 'xsd:double'),         // output parameters
          'urn:anlagenertrag',                      // namespace
          'urn:anlagenertrag#getWrTyp',             // soapaction
          'rpc',                                    // style
          'encoded',                                // use
          'Returns simualated energy yield'         // documentation
);
```

Abbildung 4: Programmcode (Auszug) Web-Service

2.3 Ausblick

Meteocontrol möchte unterschiedliche Zugriffsmöglichkeiten auf bereitgestellte Web-Services anbieten können, so dass ein einfacher Zugriff über bekannte Programmiersprachen unter Zuhilfenahme vorgefertigter Bibliotheken möglich ist. Ein vielversprechender Lösungsansatz dazu ist die automatische Generierung von Web-Service-Client-Stubs auf Basis von WSDL-Dateien (WSDL = Web Service Description Language), um Bibliotheken für den Zugriff auf Web-Services in unterschiedlichen Programmiersprachen mit geringem Aufwand anbieten zu können. Eine automatische Generierung von Web Service-Clients auf Grund der Web Service-Struktur würde eine erhebliche Arbeitserleichterung darstellen.

Zusätzlich ist es sehr interessant für meteocontrol, künftig Kunden auf Knopfdruck Clients in unterschiedlichen Programmiersprachen (z.B. Java, C++) bereitzustellen. Erfahrungen zur automatischen Generierung von Web-Service-Clients werden im Kapitel zu „Erfahrungen mit Web-Service-Technologien“ beschrieben. Eine weitere Tätigkeit im Arbeitspaket 3.3 ist die Anbindung der Wetterdaten an einen Web-Service. Es soll für den Client möglich sein, standortbezogene Wetterdaten direkt vom Server von meteocontrol abzufragen. Dieser Web-Service kann dann auch als Schnittstelle für die in AP 5.3. entstehende Solaranlagensimulation dienen.

3 MSG-Tailor-Web-Service

3.1 Einleitung

Für die Erzeugung der Datenprodukte der Meteosat First Generation wird derzeit ein Tailor (siehe Abbildung 5) eingesetzt, der die in einer Cube-Struktur abgelegten Rohdaten aus dem Archiv extrahiert und für die weitere Berechnung von Wolkenindex- und Strahlungsprodukten bereitstellt.

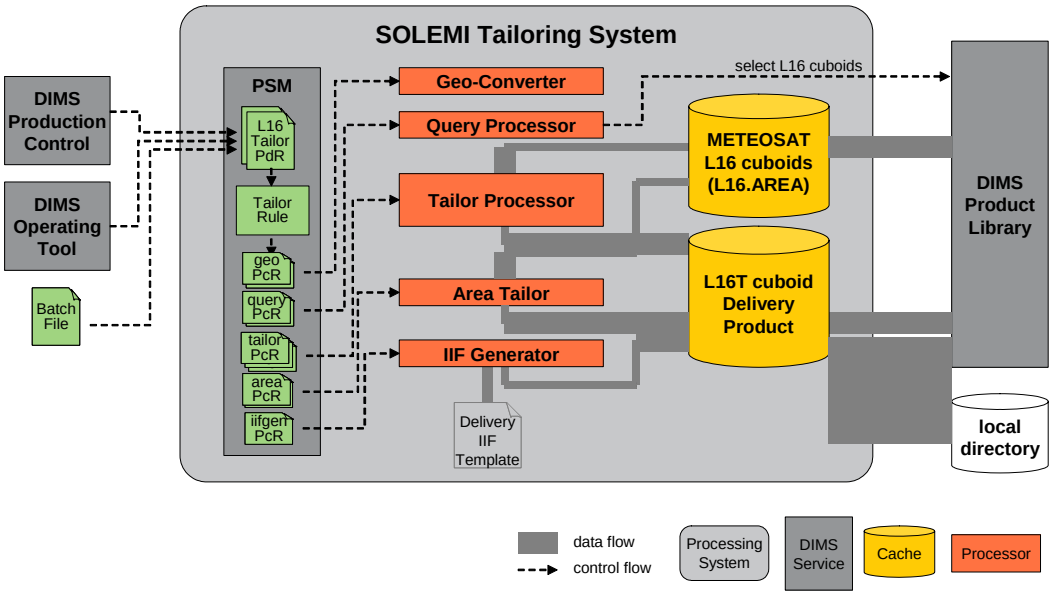


Abbildung 5: Solemi Tailor Processing System für Meteosat First Generation Rohdaten

Mit der neuen Satellitengeneration Meteosat Second Generation (MSG) verändern sich nun die Rahmenbedingungen. Die Rohdaten werden vom Satelllitenbetreiber EUMETSAT nicht mehr Vollscheiben-weise sondern in 8 Segmente unterteilt geliefert. Für die Rohdatenextraktion existiert bereits das umfangreiche Programmpaket SCENES/APOLLO, das Daten aus dem Archiv bestellt, diese

passend für die gewünschte Region ausschneidet und Wolken-, Schnee- und Wasserdampfprodukte erstellt. I.A. geschieht dies auf großen Ausschnitten bzw. der vollen MSG-Scheibe.

Diese Produkte werden anschließend in einer den Rohdaten der Meteosat First Generation vergleichbaren Cube-Struktur ins Archiv abgelegt. Dasselbe ist in Zukunft für die vom Kooperationspartner Ecole des Mines gelieferten Strahlungsprodukte geplant.

Daher muss der Tailor nun statt Rohdaten bereits weiter verarbeitete Produktstufen extrahieren und nach Nutzerwunsch passend raum-zeitlich ausschneiden. Eine entsprechende Software-Adaption wurde parallel in WISENT AP 3.2 vorgenommen.

Für diese Tailor-Prozessoren soll in WISENT AP3.3 nun ein webbasierter Zugang für Nutzer außerhalb des DLR geschaffen werden. Diese sollen zukünftig auch ins WDC integriert werden.

3.2 Realisierung

Aufgrund der strengen IT-Sicherheitsregeln beim DLR ist es Firewall-technisch (siehe Abbildung 6) nicht möglich, den MSG-Tailor-Service direkt an das eigentliche MSG-Tailor-System anzukoppeln. Das MSG-Tailor-System befindet sich innerhalb des internen DLR-Netzwerks und ein darauf eingerichteter Web Service wäre somit von außen nicht erreichbar. Daher wurde der Web Service auf einem Rechner (eridanus.caf.dlr.de) in der DMZ eingerichtet, der bereits zur Einrichtung der im Rahmen der D-Grid-Sonderinvestition angeschafften Speichereinheit verwendet wird.

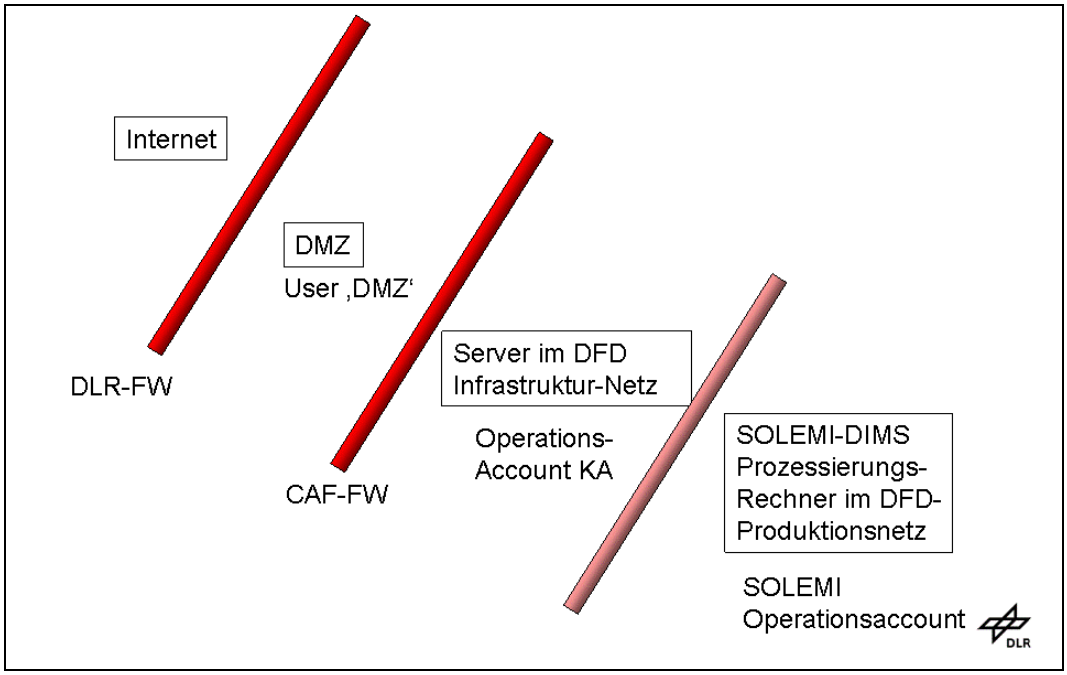


Abbildung 6: Firewall-Struktur zwischen DMZ, Infrastruktur- und Produktionsnetz im DFD

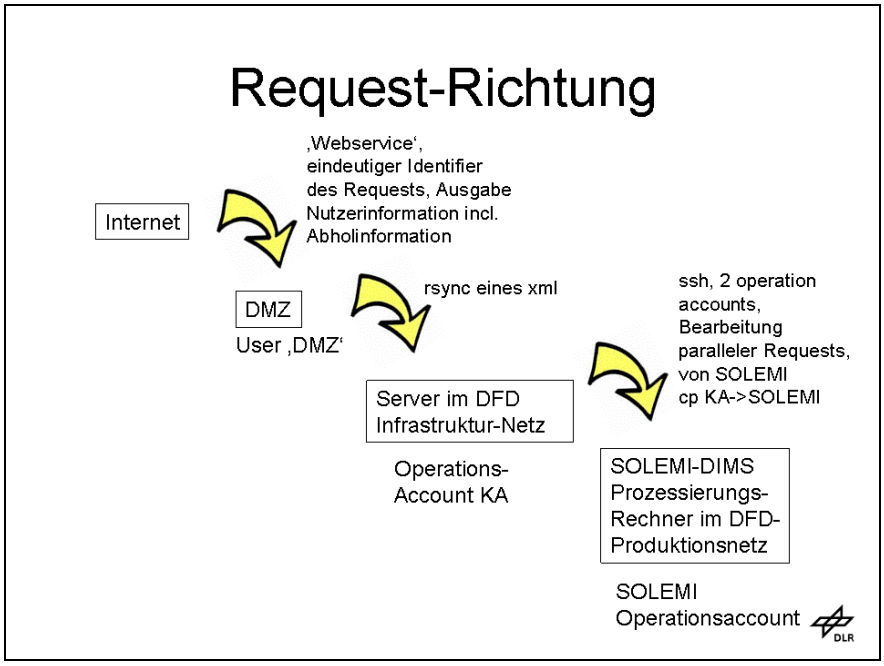
Auf dem Dateisystem von eridanus gibt es zwei Bereiche, die über „rsync“ mit der eigentlichen Speichereinheit, die sich im internen DLR-Netzwerk befindet, periodisch synchronisiert werden. Der erste Bereich wird dabei von eridanus auf die Speichereinheit synchronisiert und der zweite Bereich in die umgekehrte Richtung. D.h. Daten können aus dem internen DLR-Netzwerk in die DMZ übertragen werden, um sie beispielsweise externen Nutzern zur Verfügung zu stellen („Pick-up point“) und umgekehrt. Detailliertere Informationen zur Firewallproblematik und dem Betrieb der Speichereinheit befinden sich in dem Dokument „Bericht zu den Auswirkungen der DLR-IT-Sicherheitsstrukturen beim Datentransfer mit Partnern außerhalb des DLR“. Darauf aufbauend sind mit dem MSG-Tailor-Web Service folgende Prozessschritte verbunden, die auch in den Abbildungen 8 und 9 grafisch dargestellt werden.

1. Der Nutzer schickt die Anfrage an der MSG-Tailor-Web Service.
2. Der MSG-Tailor-Web Service generiert eine XML-Datei (siehe Abbildung 7), die in dem Speicherbereich abgelegt wird, der auf die interne Speichereinheit synchronisiert wird. Der Client bekommt ein Verzeichnis (wird temporär angelegt) zurück, auf das per GridFTP zugegriffen werden kann und in dem sich später die gewünschten Daten befinden.
3. Nach der Synchronisation wird intern (das Verzeichnis wird periodisch abgefragt) ein Prozess basierend auf den Informationen in der XML-Datei im MSG-Tailor-System gestart, der die gewünschten Daten extrahiert.

4. Die extrahierten Daten werden in den Speicherbereich der internen Speichereinheit kopiert, der auf eridanus synchronisiert wird. Das Verzeichnis entspricht dem Verzeichnis, das der Nutzer nach seiner Anfrage erhalten hat.
5. Nach der Synchronisation kann der Nutzer die Daten per GridFTP oder FTP abholen.

```
<psm-batch>
  <ProductionRequest type="MSG-Tailor">
     <ProcessingParameter key="mission">MSG</ProcessingParameter>
     <ProcessingParameter
key="productParameter">mask</ProcessingParameter>
     <ProcessingParameter key="startTime">2004-06-
03T11:00:00</ProcessingParameter>
     <ProcessingParameter key="stopTime">2004-06-
03T19:30:00</ProcessingParameter>
     <ProcessingParameter key="outputFormat">hdf4</ProcessingParameter>
     <ProcessingParameter key="regionType">pixel</ProcessingParameter>
     <ProcessingParameter key="region">
       <GeoPoint latitude="1600"   longitude="560"/>
       <GeoPoint latitude="1850"   longitude="850"/>
     </ProcessingParameter>
     <OutputProduct  type="MSG.SEVIRI.L2T"
                          id="((KP:sensor:SEVIRI),(KP:code:L2T))"
                          path="/export/home/example" />
    </ProductionRequest>
</psm-batch>
```

Abbildung 7: Beispiel-Parametersatz für eine Zeitreihenextraktion mit Pixelkoordinaten



Abbbildung 8: Struktur eines Tailor-Auftrags bis ins Produktionsnetz hinein

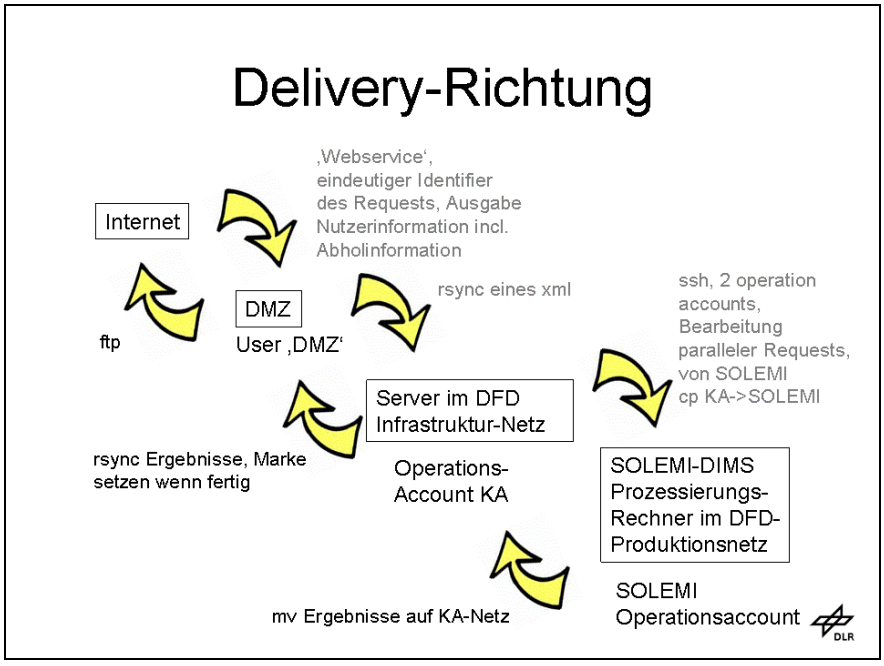


Abbildung 9: Struktur eines Tailor-Delivery-Prozesses bis zum Endnutzer

Demnach bietet der MSG-Tailor-Web Service nur eine Funktion an, um das MSG-Tailor-System im Hintergrund anzusprechen. Der Nutzer erhält ein Verzeichnis, aus dem die Daten später abgeholt werden können. Eine Überwachung des Fortschritts auf Web Service-Ebene wird zurzeit nicht umgesetzt. Der Nutzer muss durch wiederholtes Abfragen (Polling) des Verzeichnisses per GridFTP schauen, ob alle Daten vorhanden sind, was durch eine entsprechende Datei markiert wird, die bei der Synchronisation grundsätzlich zuletzt geschrieben wird.

Technisch gesehen wurde der MSG-Tailor-Web Service auf 2 Arten umgesetzt, mit Perl und mit Axis2C, wobei jeweils Apache2 als HTTP-Server verwendet wurde. Version 1 des MSG-Tailor-Web Service ist auf Basis von Perl entstanden, weil damit sehr schnell und einfach ein funktionierender Web Service implementiert werden kann, lediglich ein paar Perl-Pakete installiert werden müssen und nichts zu konfigurieren ist und die SOAP-Engine von Perl auf CGI basiert und somit einfach mit jedem Web Server wie Apache zusammenarbeitet. Leider gab es einige Probleme mit der Client-Generierung, was dazu geführt hat, dass Version 2 in Axis2C implementiert wurde. Eine detaillierte Beschreibung dazu gibt es in Abschnitt 3.

4 Erfahrungen mit Web Service-Technologien

Dieser Abschnitt beinhaltet die Erfahrungen mit diversen Web Service-Technologien, die im Laufe des Projektes WISENT gemacht wurden. Sowohl zur Implementierung von Web Services selbst, als auch zur Generierung von so genannten Client-Stubs. Zum Teil gibt es eine kurze Anleitung, wie mit der verwendeten Technologie ein Web Service implementiert werden kann.

4.1 Axis2CPP

4.1.1 Installation

Mit AxisCPP (http://ws.apache.org/axis/cpp/) können Web Services bzw. entsprechende Clients auf Basis von C++ implementiert werden. Allerdings wird AxisCPP seit März 2006 nicht weiterentwickelt. Für die Installation von AxisCPP wird ein entsprechendes Apache-Modul verwendet, somit ist es an diesen Web Server gebunden. Des Weiteren ist AxisCPP nicht so einfach lauffähig zu bekommen. Zur Installation wurden sowohl die Binary- als auch die Source-Version verwendet. Im Folgenden wird aufgeführt, welche Probleme aufgetreten sind bzw. was für manche Schritte wichtig ist, damit AxisCPP läuft. Auf dem Testsystem war Debian-Linux (sid) installiert.

- Laut Webseite werden für AxisCPP Apache 2.0.* (oder Apache 1.3.*) und Xerces-C 2.20 benötigt. Tatsächlich ist das Apache2-Modul auf eine entsprechende Version angewiesen (aktuell ist zurzeit allerdings Apache 2.2.*). In diesem Test wurde Apache bzw. httpd 2.0.63 verwendet (kann über die offizielle Webseite http://archive.apache.org/dist/httpd/ bezogen

werden). Die Xerces-C-Version kann ruhig neuer sein, im Test wurde 2.8.0 verwendet.

- Für Xerces-C müssen noch folgende Links in `/usr/lib` eingetragen werden
  - `libxerces-c.so -> libxerces-c.so.28` (wird von der Source-Version benötigt)
  - `libxerces-c.so.20 -> libxerces-c.so.28` (wird von der Binary-Version benötigt)
- Zur Sicherheit sollten folgende Umgebungsvariablen gesetzt sein. Bis auf `XERCES_HOME` entsprechen aber alle Umgebungsvariablen den eigentlichen Standardwerten).
  - `APACHE2_HOME=/usr/local/apache2/`: Verzeichnis, in dem die entsprechende Apache-Version installiert ist.
  - `AXISCPP_DEPLOY=/usr/local/axiscpp_deploy`: Verzeichnis, in das AxisCPP „deployed“ warten soll. `AXISCPP_HOME=/work/AxisCPP/axis-c-1.6b-Linux-trace-(src|bin)`: Verzeichnis, in das das Archiv (Binary|Source) entpackt wurde.
  - `XERCES_HOME=/usr/`: Eigentlich das Installationsverzeichnis von XERCES-C, aber es wird lediglich die Bibliothek `lib/libxerces-c.so` (Source) bzw. `lib/libxerces-c.so.20` (Binary) benötigt. Daher muss XERCES-C nicht zwingend installiert werden, es reicht daher der Pfad auf `/usr` aus.
  - `LD_LIBRARY_PATH=$XERCES_HOME/lib:$AXISCPP_DEPLOY/lib`
- Das Deploy-Verzeichnis (`$AXISCPP_DEPLOY`) muss immer manuell gefüllt werden, d.h. sowohl in der Binary als auch in der Source-Version muss der Inhalt des in dem jeweiligen Archivs enthaltenen, vorgefertigten Deploy-Verzeichnisses (`deploy/`) nach `$AXISCPP_DEPLOY` kopiert werden. Alternativ kann natürlich auch ein entsprechender Link gesetzt, bzw. die Variable `$AXISCPP_DEPLOY` passend gesetzt werden.

- In der Source-Version gibt es in der Konfigurationsdatei `$AXISCPP_HOME/build/build.common.properties` die Property `dir.obj`, die auf das Verzeichnis gesetzt ist, in dem die Ergebnisse des Build-Prozesses kopiert werden sollen. Per Default ist diese Property seltsamerweise `${basedir}/../../../obj`, wobei `${basedir}` in dem dazugehörigen ANT-Build-Skript für das aktuelle Verzeichnis steht, in dem der Build-Prozess gestartet wird. In dem hier beschriebenen Testfall ergab sich daraus das Verzeichnis /obj/.

4.1.2 Beispiel-Service

Ein Beispiel-Service (Calculator) ist unter http://ws.apache.org/axis/cpp/arch/End-2-End-Sample.html aufgeführt. Dieses und weitere Beispiele befinden in dem

Verzeichnis `$AXISXPP_HOME/samples`. An dieser Stelle wird nur anhand des Calculator-Service auf die größten Fehler des Beispiel-Services eingegangen.

- Als WSDL wird empfohlen `$AXICPP_DEPLOY/wsdls/calculator.wsdl` zu nehmen und gleich anzupassen. Und zwar muss am Ende der Adresse das `calculator` klein geschrieben werden. Die Services in AxisCPP werden immer in der Kleinschreibung der Namen angeboten.

```
<wsdlsoap:address location="http://localhost/axis/calculator"/>
```

- Der Befehl zur Kompilierung der Quelldateien ist falsch. Der richtige Befehl lautet.

Binary:

```
g++ *.cpp -shared -I$AXISCPP_HOME/include -L$AXISCPP_HOME/lib/axis/ -
laxis_server -o Calculator.so
```

Source:

```
g++ *.cpp -shared -I$AXISCPP_HOME/include -L/obj/bin -laxis_server -o
Calculator.so
```

- Das gleiche gilt für den Befehl, um den Client zu kompilieren. Der richtige Befehl lautet.

Binary:

```
g++ *.cpp -I$AXISCPP_HOME/include -L$AXISCPP_HOME/lib/axis/ -laxis_client -o
Calculator
```

Source:

```
g++ *.cpp -I$AXISCPP_HOME/include -L/obj/bin/ -laxis_client -o Calculator
```

- Beim „Deployment“ (Einfügen in `$AXISCPP_DEPLOY/etc/server.wsdd`) muss der Name der Services immer klein geschrieben werden, s.o. Sonst sind die URLs für die WSDL-Publizierung und den Service unterschiedlich, d.h. `Calculator?WSDL` für die WSDL und `calculator` für den eigentlichen Service am Ende der URL.

```
...
<service name="calculator" ...
```

```
  ...
</service>
...
```

4.1.3 Aufgetretene Probleme

Im Folgenden sind die wesentlichen Nachteile von AxisCPP aufgeführt.

- Wird seit März 2006 nicht mehr weiterentwickelt.
- Die Dokumentation ist miserabel und es hat sehr lange gedauert, um einen lauffähigen Service umzusetzen.
- Ausgangspunkt für die Web-Service-Entwicklung ist eine manuell erstellte WSDL-Datei. Etwas Analoges zu `Java2WSDL` auf Basis einer Java-Schnittstelle, z.B. `CPP2WSDL`, gibt es nicht.
- Nach außen wird lediglich die selbst erstelle WSDL-Datei publiziert (`?WSDL` am Ende der URL) und nicht automatisch erzeugt (wie bei Axis2/Java).

4.2 Axis2/C

4.2.1 Installation

Axis2/C (http://ws.apache.org/axis2/c/) ist der C-Ableger von Axis2/Java und basiert analog zu AxisCPP auf einem Apache-Modul. Die Installation und Konfiguration mit einem Apache2-Web Server ist mit der vorgegebenen Installationsanleitung (http://ws.apache.org/axis2/c/docs/installationguide.html) recht einfach. Im Folgenden wird angenommen, das die Umgebungsvariable `$AXIS2C_HOME` richtig gesetzt ist, z.B. auf `/usr/local/axis2c`, sowie das Axis2C über die URL `http://localhost/axis2` erreichbar ist.

4.2.2 Beispiel-Service

Als Beispiel wird der auf der Axis2-Webseite aufgeführte Hello-Service (http://ws.apache.org/axis2/c/docs/axis2c_manual.html#quick_start) verwendet. Der Code der Web-Service-Implementierung sieht allerdings sehr unübersichtlich aus, weil alles in einer Datei gespeichert ist. Insbesondere bestimmte Axis-Funktionen werden vom Anwender nie angefasst. Mit dem Skript `WSDL2C.sh` (aus `$AXIS2_HOME/bin/`) kann aus einer WSDL-Datei sowohl Client- als auch Server-Code generiert werden, der besser strukturiert ist. `WSDL2C.sh` ruft allerdings Java-Bibliotheken auf, die nur in Axis2/Java vorhanden sind. Daher muss Axis2/Java installiert werden, sowie dessen Installationsverzeichnis in das Skript `WSDL2C.sh` eingetragen werden. Des weiteren ist das Skript im MS-DOS-Encoding gespeichert und muss unter Linux noch im entsprechenden UNIX-Encoding abgespeichert werden.

Es folgt die Implementierung des Hello-Service (`hello_svc.c`).

```
#include <axis2_svc_skeleton.h>
#include <axutil_log_default.h>
#include <axutil_error_default.h>
#include <axutil_array_list.h>
#include <axiom_text.h>
#include <axiom_node.h>
#include <axiom_element.h>
#include <stdio.h>

axiom_node_t *axis2_hello_greet(const axutil_env_t *env,
        axiom_node_t *node);

int AXIS2_CALL
hello_free(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env);

axiom_node_t* AXIS2_CALL
hello_invoke(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env,
        axiom_node_t *node,
        axis2_msg_ctx_t *msg_ctx);


int AXIS2_CALL
hello_init(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env);

axiom_node_t* AXIS2_CALL
hello_on_fault(axis2_svc_skeleton_t *svc_skeli,
        const axutil_env_t *env, axiom_node_t *node);


axiom_node_t *
build_greeting_response(const axutil_env_t *env,
        axis2_char_t *greeting);



axiom_node_t *
build_greeting_response(const axutil_env_t *env, axis2_char_t *greeting)
{
    axiom_node_t* greeting_om_node = NULL;
    axiom_element_t * greeting_om_ele = NULL;

    greeting_om_ele = axiom_element_create(env, NULL, "greetResponse", NULL,
&greeting_om_node);

    axiom_element_set_text(greeting_om_ele, env, greeting,
greeting_om_node);

    return greeting_om_node;
}

static const axis2_svc_skeleton_ops_t hello_svc_skeleton_ops_var = {
    hello_init,
    hello_invoke,
    hello_on_fault,
    hello_free
};

axis2_svc_skeleton_t *
```

```
axis2_hello_create(const axutil_env_t *env)
{
    axis2_svc_skeleton_t *svc_skeleton = NULL;
    svc_skeleton = AXIS2_MALLOC(env->allocator,
            sizeof(axis2_svc_skeleton_t));

    svc_skeleton->ops = &hello_svc_skeleton_ops_var;

    svc_skeleton->func_array = NULL;

    return svc_skeleton;
}

int AXIS2_CALL
hello_init(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env)
{
    svc_skeleton->func_array = axutil_array_list_create(env, 0);
    axutil_array_list_add(svc_skeleton->func_array, env, "helloString");
    return AXIS2_SUCCESS;
}

axiom_node_t* AXIS2_CALL
hello_invoke(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env,
        axiom_node_t *node,
        axis2_msg_ctx_t *msg_ctx)
{
    return axis2_hello_greet(env, node);
}

axiom_node_t* AXIS2_CALL
hello_on_fault(axis2_svc_skeleton_t *svc_skeli,
        const axutil_env_t *env, axiom_node_t *node)
{
    axiom_node_t *error_node = NULL;
    axiom_node_t* text_node = NULL;
    axiom_element_t *error_ele = NULL;
    error_ele = axiom_element_create(env, node, "EchoServiceError", NULL,
            &error_node);
    axiom_element_set_text(error_ele, env, "Echo service failed ",
            text_node);
    return error_node;
}

int AXIS2_CALL
hello_free(axis2_svc_skeleton_t *svc_skeleton,
        const axutil_env_t *env)
{
    if (svc_skeleton->func_array)
    {
        axutil_array_list_free(svc_skeleton->func_array, env);
        svc_skeleton->func_array = NULL;
    }

    if (svc_skeleton)
    {
        AXIS2_FREE(env->allocator, svc_skeleton);
        svc_skeleton = NULL;
    }
```

```
    return AXIS2_SUCCESS;
}


AXIS2_EXPORT int
axis2_get_instance(axis2_svc_skeleton_t **inst,
        const axutil_env_t *env)
{
    *inst = axis2_hello_create(env);
    if (!(*inst))
    {
        return AXIS2_FAILURE;
    }

    return AXIS2_SUCCESS;
}

AXIS2_EXPORT int
axis2_remove_instance(axis2_svc_skeleton_t *inst,
        const axutil_env_t *env)
{
    axis2_status_t status = AXIS2_FAILURE;
    if (inst)
    {
        status = AXIS2_SVC_SKELETON_FREE(inst, env);
    }
    return status;
}


axiom_node_t *
axis2_hello_greet(const axutil_env_t *env, axiom_node_t *node)
{
    axiom_node_t *client_greeting_node = NULL;
    axiom_node_t *return_node = NULL;

    AXIS2_ENV_CHECK(env, NULL);

    if (node)
    {
        client_greeting_node = axiom_node_get_first_child(node, env);
        if (client_greeting_node &&
                axiom_node_get_node_type(client_greeting_node, env) ==
AXIOM_TEXT)
        {
            axiom_text_t *greeting = (axiom_text_t
*)axiom_node_get_data_element(client_greeting_node, env);
            if (greeting && axiom_text_get_value(greeting , env))
            {
                const axis2_char_t *greeting_str =
axiom_text_get_value(greeting, env);
                printf("Client greeted saying \"%s\" \n", greeting_str);
                return_node = build_greeting_response(env, "Hello Client!");
            }
        }
    }
    else
    {
        AXIS2_ERROR_SET(env->error,
AXIS2_ERROR_SVC_SKEL_INVALID_XML_FORMAT_IN_REQUEST, AXIS2_FAILURE);
        printf("ERROR: invalid XML in request\n");
```

```
        return_node = build_greeting_response(env, "Client! Who are you?");
    }

    return return_node;
}
```

Die zentrale Methode ist `axis2_hello_greet`, in der die Anwortnachricht zusammengesetzt wird. Als nächstes wird die Datei `hello_svc.c` kompiliert.

```
gcc -shared -olibhello.so -I$AXIS2C_HOME/include/axis2-1.2/ \ -
L$AXIS2C_HOME/lib -laxutil -laxis2_axiom \
-laxis2_parser -laxis2_engine -lpthread -laxis2_http_sender \ -
laxis2_http_receiver hello_svc.c
```

Die dabei erzeugte Bibliothek `libhello.so` wird zusammen mit dem Folgenden Service Descriptor `services.xml` in das Verzeichnis `$AXIS2C_HOME/services/hello` kopiert (das Verzeichnis `hello` muss vorher angelegt werden). Danach ist noch ein Neustart des Apache-Web Servers notwendig.

```
<service name="hello">
   <parameter name="ServiceClass" locked="xsd:false">hello</parameter>
   <description>
        Quick start guide hello service sample.
   </description>
   <operation name="greet"/>
</service>
```

Jetzt steht der Hello-Service zur Verfügung und sollte unter `http://localhost/axis2` angezeigt werden. Ein entsprechender Client `hello.c` sieht wie folgt aus.

```
#include <stdio.h>
#include <axiom.h>
#include <axis2_util.h>
#include <axiom_soap.h>
#include <axis2_client.h>

axiom_node_t *
build_om_request(const axutil_env_t *env);

const axis2_char_t *
process_om_response(const axutil_env_t *env,
        axiom_node_t *node);

int main(int argc, char** argv)
{
    const axutil_env_t *env = NULL;
    const axis2_char_t *address = NULL;
    axis2_endpoint_ref_t* endpoint_ref = NULL;
    axis2_options_t *options = NULL;
    const axis2_char_t *client_home = NULL;
```

```
    axis2_svc_client_t* svc_client = NULL;
    axiom_node_t *payload = NULL;
    axiom_node_t *ret_node = NULL;

    env = axutil_env_create_all("hello_client.log", AXIS2_LOG_LEVEL_TRACE);

    options = axis2_options_create(env);

    address = "http://localhost:9090/axis2/services/hello";
    if (argc > 1)
        address = argv[1];
    if (axutil_strcmp(address, "-h") == 0)
    {
        printf("Usage : %s [endpoint_url]\n", argv[0]);
        printf("use -h for help\n");
        return 0;
    }
    printf("Using endpoint : %s\n", address);
    endpoint_ref = axis2_endpoint_ref_create(env, address);
    axis2_options_set_to(options, env, endpoint_ref);

    client_home = AXIS2_GETENV("AXIS2C_HOME");
    if (!client_home && !strcmp(client_home, ""))
        client_home = "../..";

    svc_client = axis2_svc_client_create(env, client_home);
    if (!svc_client)
    {
        printf("Error creating service client\n");
        AXIS2_LOG_ERROR(env->log, AXIS2_LOG_SI, "Stub invoke FAILED: Error
code:"
                " %d :: %s", env->error->error_number,
                AXIS2_ERROR_GET_MESSAGE(env->error));
        return -1;
    }

    axis2_svc_client_set_options(svc_client, env, options);

    payload = build_om_request(env);

    ret_node = axis2_svc_client_send_receive(svc_client, env, payload);

    if (ret_node)
    {
        const axis2_char_t *greeting = process_om_response(env, ret_node);
        if (greeting)
            printf("\nReceived greeting: \"%s\" from service\n", greeting);

        axiom_node_free_tree(ret_node, env);
        ret_node = NULL;
    }
    else
    {
        AXIS2_LOG_ERROR(env->log, AXIS2_LOG_SI, "Stub invoke FAILED: Error
code:"
                " %d :: %s", env->error->error_number,
                AXIS2_ERROR_GET_MESSAGE(env->error));
        printf("hello client invoke FAILED!\n");
    }

    if (payload)
```

```
    {
        axiom_node_free_tree(payload, env);
        payload = NULL;
    }

    if (svc_client)
    {
        axis2_svc_client_free(svc_client, env);
        svc_client = NULL;
    }

    if (env)
    {
        axutil_env_free((axutil_env_t *) env);
        env = NULL;
    }

    return 0;
}

axiom_node_t *
build_om_request(const axutil_env_t *env)
{
    axiom_node_t* greet_om_node = NULL;
    axiom_element_t * greet_om_ele = NULL;

    greet_om_ele = axiom_element_create(env, NULL, "greet", NULL,
&greet_om_node);
    axiom_element_set_text(greet_om_ele, env, "Hello Server!",
greet_om_node);

    return greet_om_node;
}

const axis2_char_t *
process_om_response(const axutil_env_t *env,
        axiom_node_t *node)
{
    axiom_node_t *service_greeting_node = NULL;
    axiom_node_t *return_node = NULL;

    if (node)
    {
        service_greeting_node = axiom_node_get_first_child(node, env);
        if (service_greeting_node &&
                axiom_node_get_node_type(service_greeting_node, env) ==
AXIOM_TEXT)
        {
            axiom_text_t *greeting = (axiom_text_t
*)axiom_node_get_data_element(service_greeting_node, env);
            if (greeting && axiom_text_get_value(greeting , env))
            {
                return axiom_text_get_value(greeting, env);
            }
        }
    }
    return NULL;
}
```

Dieser Client wird dann wie folgt kompiliert.

```
gcc -o hello -I$AXIS2C_HOME/include/axis2-1.2/ \
-L$AXIS2C_HOME/lib -laxutil -laxis2_axiom -laxis2_parser \
-laxis2_engine -lpthread -laxis2_http_sender \
-laxis2_http_receiver hello.c -ldl -Wl,--rpath \
-Wl,$AXIS2C_HOME/lib
```

Ein Aufruf des Client sieht dann wie folgt aus.

```
./hello http://localhost/axis2/services/hello
Using endpoint : http://localhost/axis2/services/hello

Received greeting: "Hello Client!" from service
```

4.2.3 Aufgetretene Probleme

Im Folgenden sind die wesentlichen Nachteile von Axis2/C aufgeführt.

- WSDL2C unterstützt kein `encoded`-Stil, daher werden keine Perl-Web Services unterstützt (siehe auch Abschnitt 3.5).

- Ausgangspunkt für die Web-Service-Entwicklung ist eine manuell erstellte WSDL-Datei. Etwas Analoges zu `Java2WSDL` auf Basis einer Java-Schnittstelle, z.B. `C2WSDL`, gibt es nicht.

- Es werden keine WSDL-Dateien publiziert. Beim Aufruf der URL `http://localhost/axis2/services/echo?wsdl` kommt folgende Fehlermeldung.

```
XML Parsing Error: syntax error
Location: http://localhost/axis2/services/echo?wsdl
Line Number 1, Column 1:Unable to retreive wsdl for this service
^
```

- Es wird kein Hot-Deployment unterstützt, daher ist bei Änderungen an bestehenden bzw. neuen Web Services der Neustart des Apache-Web Servers notwendig.

4.3 Apache Axis2/Java 1.4

Die Open-Source-Software Apache Axis2 (http://ws.apache.org/axis2/) beinhaltet Werkzeuge für die automatische Generierung von Client-Stubs aus WSDL. Einschränkungen von Apache Axis2 liegen in den unterstützten WSDL-Styles, da Axis2 keine "encoded Styles" unterstützt, sondern nur RPC/Literal und Document/Literal. Weitere Infos zu den WSDL-Styles sind u. a. zu finden unter http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/.

Die Generierung von Java-Client-Stubs mit Hilfe von Axis2 und Eclipse gestaltet sich aufgrund der verfügbaren Axis2-Plugins für Eclipse als komfortabel. Eine einfache Anleitung dazu ist z.B. zu finden unter http://wso2.org/library/1719 und

unter http://ws.apache.org/axis2/tools/1_3/eclipse/wsdl2java-plugin.html. Die von Axis2 gelieferten Fehlermeldungen bei Problemen während der Generierung von Client-Stubs auf Basis von WSDL sind jedoch nicht immer aussagekräftig. Das Kommandozeilentool WSDL2Java (siehe http://ws.apache.org/axis2/1_3/reference.html) gibt im Falle von Fehlern die geworfenen Java-Exceptions auf der Kommandozeile aus. Diese müssen entsprechend interpretiert werden, was mit hohem Aufwand verbunden ist. Eine vernünftige Dokumentation der Funktionsweise des Werkzeugs und der möglichen auftretenden Fehler ist nicht vorhanden. Stattdessen muss in Foren oder im Quelltext nach Lösungswegen gesucht werden.

Ein einfacher Client, der eine über Axis2 generierte Stub-Klasse für den WebService-Zugriff verwendet, sieht beispielsweise wie folgt aus:

```
import de.meteocontrol.ws.service.kd5x6.SampleServiceStub;

import java.rmi.RemoteException;

import org.apache.axis2.AxisFault;
import org.apache.axis2.transport.http.HTTPConstants;
import
org.apache.axis2.transport.http.HttpTransportProperties.Authenticator;
import org.apache.log4j.BasicConfigurator;

/**
 * Web service test client
 */
public class TestClient
{
  public static void main( String[] args )
  {
    // Set up a simple configuration for log4j that logs on the console.
    BasicConfigurator.configure();

    try
    {
      SampleServiceStub stub = new SampleServiceStub(
        "https://ws.abc.de/index.php?service=xyz" );

      Authenticator authenticator = new Authenticator();
      authenticator.setUsername( "xxx" );
      authenticator.setPassword( "yyy" );
      stub._getServiceClient().getOptions().setProperty(
        HTTPConstants.AUTHENTICATE, authenticator );

      SampleServiceStub.HelloWorldResponse response = stub.helloWorld(
        new SampleServiceStub.HelloWorld() );
      System.out.println( "Result (helloWorld): " + response.get_return() );

      SampleServiceStub.Hello hello = new SampleServiceStub.Hello();
      hello.setSName( "Dies ist ein Text!" );
      SampleServiceStub.HelloResponse response2 = stub.hello( hello );
      System.out.println( "Result (hello): " + response2.get_return() );

      SampleServiceStub.GetFaultResponse response3 = stub.getFault(
        new SampleServiceStub.GetFault() );
      System.out.println( "Result (getFault): " + response3.get_return() );
    }
    catch( AxisFault e )
```

```
    {
      e.printStackTrace();
    }
    catch( RemoteException e )
    {
      e.printStackTrace();
    }
  }
}
```

Der Zugriff auf von meteocontrol bereitgestellte Web-Services verlief problemlos über eine verschlüsselte HTTPS-Verbindung mit Benutzername/Passwort-Authentifizierung. Allerdings musste die vorliegende WSDL-Datei leicht angepasst werden, um den Java-Client-Stub generieren zu können, da die WSDL-Datei „literal|encoded“ als Style verwendete. Dies führte bei der Generierung zu Problemen, die in dem verwendeten Beispiel mittels einer Umbenennung zu „literal“ gelöst werden konnten, so dass ein Client-Stub generiert werden konnte. Allerdings besteht weiterhin das Problem für komplexere WSDL-Dateien, das diese in Verbindung mit Axis2/Java nicht im „encoded style“ vorliegen dürfen, da eine Generierung für diese Encodierung nicht unterstützt wird.

4.4 Apache CXF 2.1

Das Open-Source-Framework Apache CXF (http://cxf.apache.org/) beinhaltet wie Apache Axis2 Werkzeuge zur Generierung von Client-Stubs auf Basis von WSDL. Im Gegensatz zu Axis2 werden jedoch Fehlermeldungen vom Werkzeug WSDL2Java geliefert, die besser nachvollziehbar sind. Eine Beschreibung der Nutzung von Apache CXF in Verbindung mit Eclipse ist verfügbar unter http://www-128.ibm.com/developerworks/edu/os-dw-os-eclipse-soatp.html. Apache CXF unterstützt wie Apache Axis2 keine "encoded Styles".

Apache CXF bietet nicht nur die Generierung von Java-Client-Stubs auf Basis von WSDL, sondern bietet die folgenden weiteren Tools zur Generierung von Stubs in unterschiedlichen Programmiersprachen:

- wsdl2corba (zur Generierung der Interface Description Language (IDL))
- wdl2java (Java)
- wsdl2js (JavaScript)
- wsdl2service erstellt eine neue WSDL-Datei, die eine HTTP oder JMS Service-Definition enthält.
- wsdl2soap erstellt eine neue WSDL-Datei mit einem SOAP-Binding.
- wsdl2xml erstellt eine neue WSDL-Datei mit einem XML-Binding.

Zusätzlich bietet Apache CXF einen WSDL-Validierer (wsdlvalidator) zur Überprüfung von WSDL-Dateien.

4.5 Apache Axis (Java) 1.4

Im Gegensatz zu Apache Axis2 unterstützt Apache Axis (http://ws.apache.org/axis/) auch "encoded Styles". Seit April 2006 gab es jedoch keine neuen Versionen zu Axis. Es ist davon auszugehen, dass sich die zukünftigen Entwicklungsarbeiten auf Apache Axis2 konzentrieren werden. Für eine WSDL-Datei im encoded Style konnte mit Hilfe von Apache Axis ein Client-Stub in der Programmiersprache Java generiert werden. Der Zugriff auf den von meteocontrol angebotenen Testservice über einen auf dem Client-Stub basierenden Testclient war jedoch nicht erfolgreich. Da Apache Axis voraussichtlich nicht mehr weiterentwickelt wird, wurde auf weitergehende Tests verzichtet.

4.6 Perl

4.6.1 Installation

Damit Web-Services in Perl implementiert werden können, müssen zunächst folgende Perl-Pakete auf dem System installiert sein:

- XML-Parser (http://search.cpan.org/~msergeant/XML-Parser-2.36/)
- XML-Writer (http://search.cpan.org/~josephw/XML-Writer-0.603/)
- XML-XPath (http://search.cpan.org/~msergeant/XML-XPath-1.13/)
- SOAP-Lite (http://search.cpan.org/~byrne/SOAP-Lite-0.60a/)
- POD-WSDL (http://search.cpan.org/~tareka/Pod-WSDL-0.05/)
- CGI-Lite (http://search.cpan.org/~smylers/CGI-Lite-2.02/)

Eine Möglichkeit, diese Pakete zu installieren, ist, diese herunterzuladen, zu entpacken und folgende Befehle in dem entpackten Verzeichnis auszuführen.

```
perl Makefile.PL
make
make install
```

4.6.2 Beispiel-Service

Es wird davon ausgegangen, dass ein Web-Server wie Apache mit einem CGI-Verzeichnis, z.B `/usr/lib/cgi`, vorhanden ist. Darin das Verzeichnis `echoService` anlegen und folgende Dateien für einen einfachen Echo-Web Service erstellen.

- `MyService.pm`: Dies ist die Implementierung des Web Services. Es gibt die Methode `echo`, die einfach den übergebenen String zurückgibt.

```
package MyService;

use strict;
```

```
use warnings;

sub echo
{
    shift;
    my $parameter=shift;
    return $parameter;
}

1;
```

- `proxy.pl`: Dies ist ein SOAP-Proxy, der Client-Anfragen an die Web Service-Implementierung weiterleitet. In dem Beispiel wird dazu eine `dispatch_to`-Methode verwendet, die nur den Aufruf der `echo`-Methode zulässt, auch wenn `MyService.pm` mehrere Methoden hätte.

```
#!/usr/bin/perl

use strict;
use warnings;
use MyService;

use SOAP::Transport::HTTP;

SOAP::Transport::HTTP::CGI
    -> dispatch_to(
        'MyService::echo'
    )
    -> handle;

1;
```

- `client.pl`: Dies ist der Client, der die `echo`-Methode des Web Service über den SOAP-Proxy aufruft. Der Aufruf `perl client.pl` sollte den String `Hello World!` ausgeben.

```
#!/usr/bin/perl

use strict;
use warnings;

use SOAP::Lite;
#use SOAP::Lite +trace => 'debug';

my $result = SOAP::Lite
    -> uri('http://127.0.0.1/MyService')
    -> proxy('http://127.0.0.1/cgi-bin/echoService/proxy.pl')
    -> echo('Hello World!')
    -> result;

print $result."\n"
```

Wird die Zeile „`use SOAP::Lite +trace => 'debug';`“ auskommentiert, so werden die übermittelten SOAP-Nachrichten mit ausgegeben.

Als nächstes wird eine WSDL-Datei erzeugt, die zur Generierung von Client-Stubs in anderen Programmiersprachen verwendet werden kann. Die Web Service-Implementierung `MyService.pm` wird dazu um so genannte POD-Annotationen (POD = plain old documentation) erweitert. In diesem Beispiel wird die Methode `echo` entsprechend annotiert.

```
package MyService;

use strict;
use warnings;

=begin WSDL
    _IN in @string
    _RETURN @string
=cut
sub echo
{
    shift;
    my $parameter=shift;
    return $parameter;
}

1;
```

Die Annotationen sind eigentlich selbsterklärend, weitere Informationen dazu gibt es unter http://search.cpan.org/~tareka/Pod-WSDL-0.05/lib/Pod/WSDL.pm. Eine WSDL-Datei kann jetzt über ein weiteres Perl-Skript, hier `genWSDL.pl` erzeugt werden.

```
use Pod::WSDL;

my $pod = new Pod::WSDL(source => 'MyService',
    location => 'http://127.0.0.1/cgi-bin/echoService/proxy.pl',
    pretty => 1,
    withDocumentation => 1);

print $pod->WSDL
```

Der Parameter `source` gibt das einzulesende Perl-Modul an gibt, `location` die URL des SOAP-Proxies, `pretty` auf 1 gesetzt bewirkt eine schön formatierte XML-Ausgabe (sonst alles in einer Zeile) und `withDocumentation` auf 1 gesetzt erzeugt auch Kommentare in der WSDL-Datei. Der Aufruf `perl genWSDL.pl` erzeugt jetzt eine entsprechende WSDL-Datei und gibt sie auf `STDOUT` aus.

Üblicherweise wird über die URL eines Web Service die WSDL-Datei ausgegeben, wenn man den Zusatz „`?wsdl`“ anhängt. Damit das funktioniert, wird der SOAP-Proxy `proxy.pm` wie folgt geändert.

```
#!/usr/bin/perl

use strict;
```

```
#use warnings # parse_form_data wirft sonst Warnungen beim Parsen von SOAP-
Nachrichten;

use CGI::Lite;

my $cgi = new CGI::Lite;
my %form = $cgi->parse_form_data;

my $wsdl="false";
while ((my $key, my $value) = each(%form)){
    if ($key eq "wsdl") {
        $wsdl="true";
        last;
    }
}


if ($wsdl eq "true") {
    print "Content-type: text/plain", "\n\n";
    use Pod::WSDL;

    my $pod = new Pod::WSDL(source => 'MyService',
        location => 'http://127.0.0.1/cgi-bin/echoService/proxy.pl',
        pretty => 1,
        withDocumentation => 1);

    print $pod->WSDL;
} else {
    use MyService;
    use SOAP::Transport::HTTP;
    SOAP::Transport::HTTP::CGI
    -> dispatch_to(
        'MyService::echo'
    )
    -> handle;
}

1;
```

Das Skript macht eigentlich nichts weiter, als zu überprüfen, ob in der URL der Parameter `wsdl` gesetzt wurde. Wenn ja, dann verhält sich der SOAP-Proxy als normales CGI-Skript und erzeugt die WSDL-Datei analog zu `genWSDL.pl`. Ansonsten wird ein SOAP-Aufruf erwartet und entsprechend weitergeleitet. Mit einem Browser sollte die URL `http://127.0.0.1/cgi-bin/echoService/proxy.pl?wsdl` aufrufbar sein und WSDL-Code angezeigt werden. Diese WSDL-Beschreibung kann zur Generierung von Client-Stubs genutzt werden.

4.6.3 Aufgetretene Probleme

Im Folgenden sind die wesentlichen Nachteile von Web Services mit Perl aufgeführt.

- Es gibt im Prinzip kein WSDL (außer POD-WSDL) bei Perl-Web Services, daher fehlen auch die entsprechenden Werkzeuge wie `Java2WSDL` und `WSDL2Java`, um Client-Stubs bzw. Server-Skeletons zu erzeugen.

- Mit dem Paket POD-WSDL werden derzeit nur WSDL-Dokumente mit dem Stil "`RPC/encoded`" erzeugt, insbesondere weil SOAP-Lite nur den Stil "`encoded`" unterstützt (weitere Informationen zu Stilen siehe http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/. Leider wird dieser Stil nicht von Axis2/Java und Axis2/C unterstützt, so dass damit keine Clients generiert werden können. Mit dem älteren Axis (quasi Axis1/Java) sowie AxisCPP (nicht überprüft aber theoretisch möglich) ist eine Client-Generierung mit diesem Stil möglich.

4.7 HTTPS-Authentifizierung mit Grid-Zertifikaten

Um die Nutzung von Web Services abzusichern, gibt es verschiedene Möglichkeiten. Beispielsweise könnte die Angabe von einem Benutzername und Passwort beim Aufruf der Funktion zwecks Authentifizierung verlangt werden. Des Weiteren könnte die Kommunikation über einen verschlüsselten HTTPS-Kanal abgesichert werden, wobei die dabei eingesetzten X.509-Zertifikate auch zur Authentifizierung herangezogen werden können. Im (D-)Grid-Kontext bzw. bei den eingesetzten Grid-Middleware gLite, Globus Toolkit 4 und UNICORE ist es üblich, dass eine wechselseitige Authentifizierung über so genannte Grid-Zertifikate (X.509-Zertifikate) durchgeführt wird, d.h. sowohl Nutzer als auch der Server besitzen ein Zertifikat, welches von einer CA signiert wurde, der beide Parteien vertrauen. Globus Toolkit 4 und UNICORE 6 bauen dabei HTTPS-Verbindungen auf. Da es sich bei Grid-Zertifikaten um Standard-X.509-Zertifikate handelt, können diese auch für eine HTTPS-Authentifizierung mit einem Web Server verwendet werden. Im Folgenden wird beschrieben, wie der Web Server Apache2 konfiguriert werden muss, um eine Authentifizierung mit Grid-Zertifikaten zu unterstützen und wie ein mit Axis2c generierter Web Service-Client ebenfalls ein Grid-Zertifikat verwendet, um sich damit gegenüber dem mit einer HTTPS-Authenfizierung gesicherten Web Service zu authentisieren.

4.7.1 Konfiguration des Apache2 Web Servers

Im Folgenden wird beschrieben, wie der HTTPS/SSL-Web-Server von Apache2 so eingerichtet werden kann, dass eine wechselseitige Authentifizierung auf Basis von (D-)Grid-Zertifikaten möglich ist. Da es sich bei Grid-Zertifikaten auch um X.509-Zertifikate handelt, kann diese Anleitung auch auf Anwendungen außerhalb des Grid-Kontextes übertragen werden.

Es wird davon ausgegangen, dass der Apache2 Web-Server bereits installiert und lauffähig ist, sowie sich dieser auf einem System mit einem gültigen Server-Grid-Zertifikat befindet bzw. das Verzeichnis `/etc/grid-security/` existiert. Die Konfigurationsdateien von Apache2 befinden sich in `/etc/apache2`. Um einen SSL-Server einzurichten, sofern noch nicht vorhanden, wird in

/etc/apache2/vhosts.d/ eine entsprechende Konfigurationsdatei angelegt, z.B. vhost-ssl. Der Inhalt dieser Datei kann beispielsweise wie folgt aussehen.

```
<IfDefine SSL>
<IfDefine !NOSSL>

<VirtualHost srvgrid01.offis.uni-oldenburg.de:443>

    DocumentRoot "/var/www/"
    ServerName srvgrid01.offis.uni-oldenburg.de
    ServerAdmin dgrid-admin@offis.de
    ErrorLog /var/log/apache2/ssl_error.log
    TransferLog /var/log/apache2/ssl_access.log

    SSLEngine on
    SSLCipherSuite
ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL
    SSLCertificateFile  /etc/grid-security/hostcert.pem
    SSLCertificateKeyFile /etc/grid-security/hostkey.pem
    SetEnvIf User-Agent ".*MSIE.*" nokeepalive ssl-unclean-shutdown
downgrade-1.0 force-response-1.0

</VirtualHost>

</IfDefine>
</IfDefine>
```

Die meisten Einstellungen enthalten dabei Standardwerte, lediglich SSLCertificateFile und SSLCertificateKeyFile sind an dieser Stelle interessant, weil dort ein so genanntes (Grid-)Server-Zertifikat und der dazugehörige private Schlüssel verwendet wurden. Mit dieser Konfiguration kann ein Browser eine HTTPS-Verbindung zu https://srvgrid01.offis.uni-oldenburg.de aufbauen. Da per Default die Grid-CA des DFN. die dieses Server-Zertifikat signiert hat, nicht als vertrauenswürdige CA in jedem Browser eingetragen ist, wird es einen entsprechenden Warnhinweis geben. In der jetzigen Konfiguration überprüft nur der Browser die Identität des Web-Servers bzw. ob er der CA vertraut, die dessen Zertifikat signiert hat, aber nicht umgekehrt.

Im nächsten Schritt geht es darum, den Zugriff auf den HTTPS-Server einzuschränken, so dass nicht jeder Benutzer darauf zugreifen kann. Es wird davon ausgegangen, dass ein Browser vorhanden ist, in dem ein so genanntes (Grid-)User-Zertifikat importiert wurde. In dem Beispiel soll nicht der gesamte Server, sondern lediglich das cgi-bin-Verzeichnis (in dem sich ein Perl-Web Service befindet) abgesichert werden. Folgendes muss dazu zu der obigen Konfiguration hinzugefügt werden.

```
<IfDefine SSL>
<IfDefine !NOSSL>

<VirtualHost srvgrid01.offis.uni-oldenburg.de:443>

...
```

```
ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
    AllowOverride None
    Options ExecCGI -MultiViews +SymLinksIfOwnerMatch
    Order allow,deny
    Allow from all
    SSLRequireSSL
    SSLVerifyClient require
    SSLVerifyDepth  10
    SSLCACertificatePath /etc/grid-security/certificates/
    SSLOptions +StdEnvVars
</Directory>

</VirtualHost>

</IfDefine>
</IfDefine>
```

Auch in diesem Fall sind fast nur Standardeinstellungen übernommen worden. Mit SSLCACertificatePath wird ein Verzeichnis angegeben, in dem sich Zertifikate von vertrauenswürdigen CAs befinden. In dem Beispiel wurde /etc/grid-security/certificates/ verwendet, in dem sich die Zertifikate aller vertrauenswürdigen Grid-CAs. z.B. die des DFN, befinden. Die Einstellung SSLOptions +StdEnvVars bewirkt, dass vor der Ausführung eines CGI-Skriptes etliche SSL-Umgebungsvariablen hinzugefügt werden. In SSL_CLIENT_S_DN beispielsweise steht der DN (*distinguished name*) des Benutzers, was im CGI-Skript zur weiteren Einschränkung von Benutzerrechten verwendet werden kann.

Neben der Einschränkung auf vertrauenswürdige CAs kann bereits in der Apache2-Konfiguration der Zugriff auch auf Benutzerebene anhand des DN des Benutzers weiter eingeschränkt werden. Die obige Konfiguration wird dann wie folgt erweitert.

```
...

ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
<Directory "/usr/lib/cgi-bin">
    ...
    SSLOptions +StdEnvVars +FakeBasicAuth
    AuthName "Secured Area"
    AuthType Basic
    AuthUserFile /usr/lib/cgi-bin/.passwd
    Require valid-user
</Directory>

...
```

Die zusätzliche SSL-Option FakeBasicAuth bewirkt, dass die Authentifizierungsmöglichkeit 'Basic' (normalerweise ein Popup-Fenster mit Feldern für Benutzername und Passwort) im SSL-Kontext verwendet werden kann. Die Datei /usr/lib/cgi-bin/.passwd in AuthUserFile enthält alle zugelassenen Benutzer (inklusive Passwort) und kann mit dem Programm htpasswd erstellt werden. Dort werden als Benutzernamen die DNs der

zugelassenen Benutzer und als Passwort immer 'password' (ist eine Konvention) eingetragen.

```
htpasswd -c .passwd "/C=DE/O=GridGermany/OU=OFFIS e.V./OU=Grid-RA/CN=Guido
Scherp"
New password: password
Re-type new password: password
Adding password for user /C=DE/O=GridGermany/OU=OFFIS e.V./OU=Grid-
RA/CN=Guido Scherp
```

Die Option '-c' bewirkt, dass die Datei neu erzeugt wird und kann beim Hinzufügen weiterer Benutzer weggelassen werden. Bei jedem Zugriff auf einen Pfad unterhalb von `https://srvgrid01.offis.uni-oldenburg.de/cgi-bin/`, muss der Nutzer ein Zertifikat vorweisen, welches von einer CA signiert wurde, der der Apache2-Web Server vertraut, z.B. der vom DFN .

4.7.2 Konfiguration von Axis2C

Um eine HTTPS-Authentifizierung auf Basis von (Grid-)Zertifikaten zu ermöglichen, muss die zentrale Konfigurationsdatei `$AXIS2C_HOME/axis2.xml` angepasst werden (siehe auch in der offizielle Dokumentation unter `http://ws.apache.org/axis2/c/docs/axis2c_manual.html#ssl_client`). Jeder Web Service-Client auf Basis von Axis2C nutzt (per Default) diese Konfigurationsdatei.

```
<transportSender name="https" class="axis2_http_sender">
    <parameter name="PROTOCOL" locked="false">
    HTTP/1.1
    </parameter>
</transportSender>

<parameter name="SERVER_CERT">
/etc/grid-security/certificates/c25b265c.0
</parameter>

<parameter name="KEY_FILE">
$USER_HOME/userkeyandcert.pem
</parameter>

<parameter name="SSL_PASSPHRASE">geheim</parameter>
```

Der Parameter SERVER_CERT ist das SSL-Zertifikat, welches der Zielrechner verwendet, in diesem Fall das Grid-Server-Zertifikat vom Apache2-Web Server. Es kann keine CA oder ein CA-Verzeichnis angegeben werden, sondern lediglich direkt das Zertifikat, dem vertraut wird. Hinter dem Parameter KEY_FILE verbirgt sich eine Datei, in der sich sowohl das Grid-Benutzer-Zertifikat befinden, als auch der dazugehörige private Schlüssel (einfach zu erzeugen durch `cat userkey.pem usercert.pem > userkeyandcert.pem`). Dies wird vom Web Service-Client zur Authentisierung gegenüber dem HTTPS-Web Server Apache2

verwendet. Der Parameter SSL_PASSPHRASE beinhaltet das Passwort, mit dem der private Schlüssel in KEY_FILE verschlüsselt wurde.