Web Services
und
Service Oriented Architectures

Alfon Wenzler
Technische Universität Kaiserslautern

2. Juli 2004

Inhaltsverzeichnis

1 Motivation

Das Thema diese Seminars lautet *Service Oriented Architectures und Web Services – Grundlagen und Vergleich mit bestehenden Middleware-Technologien.* Web Services und serviceorientierte Architekturen sind derzeit sowohl im akademischen Umfeld als auch in der Industrie sehr populär. Bei Web Services handelt es sich um Bausteine zur Lösung von Software-Integrationsproblemen welche auf bestehende Ansätze in den Bereichen verteilte Systeme, Informationssysteme und Programmiersprachen aufbauen. Die Erwartungen an dieses neue Konzept und die damit verbundenen Anstrengungen auf diesem Gebiet sind sehr hoch. In Kapitel 2 wird das Konzept der serviceorientierten Architekturen beschrieben. Als Anwendung des SOA-Gedankens wird im Kapitel 3 auf den Begriff Web Services eingegangen. Die Basistechnologien für Web Services werden im Kapitel 4 behandelt. Abschnitt 4.1 beschreibt die SOAP-Spezifikation zum Nachrichtenaustausch. Die Beschreibung der Web Services durch die WSDL ist Thema des Abschnitts 4.2. Abschnitt 4.3 beinhaltet die UDDI-Spezifikation für Web-Service-Verzeichnisse. Auf alternative Ansätze zur Realisierung von Web Services wird in Kapitel 5 eingegangen. Behandelt werden die Ansätze XML-RPC sowie REST. Den Vergleich von Web Services mit der bestehenden Middleware Technologie CORBA ist Thema von Kapitel 6.

2 Service Oriented Architecture

In diesem Kapitel wird das Konzept der serviceorientierten Softwareentwicklung beschrieben. In Abschnitt 2.1 werden zunächst die Idee und die Zielsetzung von Service Oriented Architecture beschrieben, und die Notwendigkeit einer solchen Architektur wird aufgezeigt. Das eigentliche Model zur Umsetzung dieser Architektur wird in Abschnitt 2.2 formuliert und erklärt.

2.1 Motivation

Das Ziel von Softwareentwicklung ist es zuverlässige Software zu entwerfen, die leicht zu warten und wiederzuverwenden ist. In der Praxis sind diese Anforderungen jedoch kaum zu erreichen. Software wird meist zu komplex, um sie später zu warten oder weiterzuentwickeln, wenn sie zuverlässig

sein soll. Das Ziel des Konzepts einer serviceorientierte Architektur ist es daher, die Wiederverwendbarkeit von Software zu erleichtern und somit zuverlässige Software schneller und einfacher zu entwickeln. Diese Zielsetzung wird im Folgenden näher beschrieben.

2.1.1 Zielsetzung

Bei der Softwareentwicklung treten immer wieder dieselben Probleme auf, und große Teile einer Software finden auch in ähnlicher Form Verwendung in anderen Softwareprodukten. Daher ist es sinnvoll, nicht jede Software von Grund auf neu zu entwickeln, sondern andere, bewährte Software wiederzuverwenden. Dies reduziert den Entwicklungsaufwand erheblich, da sowohl Code wiederverwendet wird und somit nicht neu geschrieben werden muss als auch dadurch, dass die wiederverwandte Software bereits getestet und sich als zuverlässige erwiesen hat. Damit werden wiederholte Fehler vermieden. Um dieses Ziel zu erreichen, wurde das Konzept der Service orientierten Architektur entwickelt. Die Idee hierbei ist es, die Wiederverwendbarkeit von Software zu erleichtern, indem die Abhängigkeiten der einzelnen Softwaremodule auf das Notwendige reduziert werden.

Abhängigkeiten und Loose Coupling Es gibt zwei Arten von Abhängigkeiten, die unterschieden werden müssen. Zum einen gibt es die notwendigen Abhängigkeiten, welche die wiederzuverwendenden Teile der Softwareprodukte betreffen. Diese Abhängigkeiten sind gewollt, da diese die Wiederverwendung von Code ermöglichen. Auf der anderen Seite gibt es aber auch ungewollte Abhängigkeiten zwischen den einzelnen Softwareprodukten, welche die Zusammenarbeit erschweren. Somit ist es Ziel einer Service-orientierten Architektur, diese Abhängigkeiten auf ein Minimum zu reduzieren, da sich diese Abhängigkeiten nicht ganz vermeiden lassen.

He [11] vergleicht die Abhängigkeiten von Software mit der Abhängigkeit von Strom. Der Mensch braucht den Strom, um seine elektrischen Geräte zu betreiben, egal in welchem Land er sich befindet. Dies entspricht der direkten und gewollten Abhängigkeit. In jedem Land gibt es Strom, der genutzt werden kann. Allerdings gibt es auch noch die ungewollte Abhängigkeit von einem Adapter, um die Steckdosen in den einzelnen Ländern benutzen zu können. Dies ist lästig und erschwert die Nutzung des Stroms unnötig. Der Adapter entspricht hierbei einer unzureichenden Schnittstellenkompatibilität bei Software.

Wenn alle ungewollten, künstlichen Abhängigkeiten in einem System minimiert sind, spricht man von *Loose Coupling*. Dieser Zustand soll durch Service-Oriented Architecture erreicht werden, um die Softwareentwicklung zu vereinfachen, ohne Einbußen bei der Zuverlässigkeit zu erhalten.

2.2 Das Service Oriented Model

In diesem Abschnitt wird das Model zur Umsetzung einer Service orientierten Architektur beschrieben. Dazu müssen zunächst die Grundbegriffe in Abschnitt2.2.1 definiert werden. In dem Abschnitt 2.2.3 werden die Bedingungen, welche an eine Service orientierte Architektur gestellt werden, näher erläutert.

2.2.1 Definitionen

Zur Beschreibung der Service Oriented Architecture sind die Begriffe Aktion, Nachricht, Softwareagent und Service notwendig. Diese Komponenten werden in den folgenden Abschnitten definiert.

Aktionen und Nachrichten – Die serviceorientierte Architektur fokussiert die Interaktion verschiedener Softwareagenten. Eine Aktion in einem Service orientierten Modell ist hierbei jede Art von Tätigkeit, die ein Softwareagent ausführt. In Anlehnung an das W3C [8] lässt sich eine Aktion in einer serviceorientierten Architektur wie folgt definieren:

- Entweder das Senden einer Nachricht
- oder das Bearbeiten einer empfangenen Nachricht

- oder auch jede Art von Tätigkeit eines Agenten, die das System in den erwünschten Zielzustand überführt.

Eine Aktion wird von einem Softwareagenten auf Wunsch eines anderen ausgeführt, um einen geforderten Service zu leisten. In der serviceorientierten Architektur liegt der Fokus hierbei auf dem Senden und Empfangen von Nachrichten, um einen Service anzufordern beziehungsweise einen Service zu leisten.

Im folgenden ist zu definieren, was genau unter einem Softwareagenten verstanden wird.

Softwareagenten – Softwareagenten sind Programme, welche zum einen Services anfordern und welche zum anderen angeforderte Services bearbeiten. Laut W3C [8] lassen sich Softwareagenten im Kontext einer Service orientierten Architektur wie folgt definieren:

Softwareagenten sind Programme, welche

- einem Besitzer gehören,
- einen oder mehrere Services von anderen Agenten anfordern
- einen oder mehrere Services für andere Agenten leisten.

Es ist wichtig zu beachten, dass Softwareagenten im Auftrag eines Eigners agieren, da eine Aktion durchaus mit einem Ressourcen Austausch einhergehen können.

Service – Ein Service wird von einem Provideragenten für einen Kundenagenten bereitgestellt. Der Service ist im Besitz des Providers und wird vom Kunden angefordert. Ein Serviceleistung ist immer verbunden mit einer Aktion, die von den korrespondierenden Softwareagenten ausgeführt wird. Ein Service besteht aus einer oder mehreren Aufgaben, die von einem oder mehreren Serviceprovidern für den anfordernden Softwareagenten ausgeführt werden. Die Services werden über zuvor beschriebene Schnittstellen durch Softwareagenten ausgetauscht.

2.2.2 Modellformulierung

Das Ziel einer Service orientierten Architektur ist es *Separation of Concerns* im Bereich Softwareentwicklung zu erreichen. Das heißt, nicht jede Software wird von Grund auf neu entwickelt, sondern bereits bestehende Software wird wiederverwendet, durch die Nutzung verschiedener Services. Die Nutzung eines externen Services ist dabei sowohl billiger als auch effektiver. Der Vorteil der Servicenutzung beruht dabei auf dem Prinzip der Arbeitsteilung. Jeder Service deckt einen speziellen Bereich ab, auf den die Entwickler dieser Services spezialisiert sind. Das heißt, durch den Einsatz von Services nutzt man für die jeweiligen Teile der Software Expertenwissen aus diesem Gebiet.

Um die Wiederverwendbarkeit einfach und realisierbar zu machen, muss *Loose Coupling* zwischen den einzelnen Softwareagenten sichergestellt werden. Dazu sind die folgenden architekturellen Bedingungen notwendig:

- Einfache Schnittstellen für die Softwareagenten
- Deskriptive und erweiterbare Nachrichten

Da diese Bedingen an die Architektur sehr wichtig sind, werden sie in dem folgenden Abschnitt näher erläutert.

2.2.3 Bedingungen

Die Anforderungen an eine Service-Oriented Architecture beziehen sich zum einen auf die Schnittstellenimplementierung und zum anderen auf den Aufbau der Nachrichten, die zwischen den einzelnen Softwareagenten ausgetauscht werden.

Interfaces – Eine gute Schnittstellenbeschreibung und -implementierung ist besonders wichtig, um künstliche Abhängigkeiten zu vermeiden und so *Loose Coupling* zu erreichen. Wenn die Schnittstellen nicht einfach und von überall zu erreichen sind, erschweren diese die Zusammenarbeit verschiedener Softwareagenten anstatt diese zu vereinfachen. Es ist daher sinnvoll, sich auf wenige generische Schnittstellen für alle Softwareprojekte zu beschränken.

Messages – Da nur wenige generische Schnittstellen zur Softwareentwicklung genutzt werden, werden Nachrichten benötigt, um anwendungsbezogene Semantik zu integrieren. Die Nachrichten werden von den Softwareagenten via der beschreibenden Schnittstellen ausgetauscht. Um einen reibungslosen Ablauf zu garantieren, müssen die ausgetauschten Nachrichten jedoch folgende Anforderungen erfüllen: Die Nachrichten müssen

- deskriptiv sein,
- in einem einheitlichen Format, einer gegebenen Struktur und einem bestimmten Alphabet vorliegen,
- erweiterbar sein.

Nachrichten müssen deskriptiv sein, denn der Einsatz von Services soll die Nutzung von Expertenwissen ermöglichen. Enthalten die Nachrichten jedoch Angaben, wie eine bestimmte Anfrage auszuführen ist, wird die Problemlösung vorgegeben und der Einsatz der Services ist sinnlos. Die Problemlösung muss dem Service Provider überlassen werden, da dieser über spezialisiertes Wissen dazu verfügt, welches durch Services für die Anwendung nutzbar gemacht wird. Daher dürfen die Nachrichten nur deskriptiv sein.

Die Nachrichten, welche zwischen den Softwareagenten ausgetauscht werden, müssen so sein, dass sie von allen beteiligten Agenten verstanden werden. Dazu müssen die Nachrichten in einem einheitlichen Format, einer gegebenen Struktur und einen bestimmten Alphabet sein. Die Nachricht ist umso einfacher für alle Agenten zu verstehen, je restriktiver die Grammatik und das genutzte Alphabet ist.

Da die Softwaresysteme einem ständigen Wandel unterliegen, müssen auch die Nachrichten erweiterbar sein. Je restriktiver jedoch die Grammatik und das benutzte Alphabet sind, desto schwieriger gestaltet sich die Erweiterbarkeit. Das Ziel eines guten Nachrichtendesigns muss es daher sein, diesen *Trade-Off* auszugleichen.

Eine weitere wichtige Anforderung an eine Service Oriented Architecture ist, dass der Service überhaupt nutzbar gemacht wird. Das heißt, der Servicekunde muss über Möglichkeiten verfügen, den Serviceprovider ausfindig zu machen.

Stateless- vs. Stateful Service – Man kann zwei Arten von Services unterscheiden: Stateless- und Stateful Services.

Bei einem Stateless Service wird gefordert, dass jede Nachricht vollständig ist. Das heißt, jede Nachricht muss alle notwendigen Informationen zur Bearbeitung bereitstellen. Das hat den Vorteil, dass der Service Provider keine Informationen zwischenspeichern muss, was die Skalierbarkeit erhöht. Auch die Fehlerbehandlung ist dadurch erheblich leichter, da keine Zwischenzustände zu berücksichtigen sind.

Stateless Services sind jedoch nicht immer realisierbar. Ein Beispiel dafür ist eine verschlüsselte Verbindung, welche die Sicherheit der Transaktion schützt, jedoch den Nachrichtenaustausch erschwert. In diesem Fall wird ein Stateful Service bevorzugt, um den Nachrichtenaustausch zu erleichtern. Diese erhöhte Kopplung von den Softwareagenten reduziert jedoch die Skalierbarkeit.

3 Web Services

Die Gedanken der serviceorientierten Architektur spiegeln sich bei Web Services wieder. In diesem Kapitel wird zuerst auf den Begriff Web Services eingegangen.

3.1 Was sind Web-Services?

3.1.1 Das Web – Heute und Morgen

Die Stärke des World Wide Web (WWW) ist heute insbesondere dadurch zu begründen, dass es die Möglichkeit bietet, interaktiv auf Dokumente und Applikationen zuzugreifen. Ein solcher Zugriff erfolgt, durch einen Anwender gesteuert, mittels Web-Browsern, Audio-Abspielgeräten, Peer2Peer-Clients oder anderer interaktiver Systeme. Das Web wird laut W3C [10] stark an Mächtigkeit und Umfang zunehmen wenn es um die Fähigkeit der Kommunikation zwischen Applikationen erweitert wird, welche die Mensch-Maschinen- durch eine Maschine-Maschine-Kommunikation ergänzt. Diesen Wandel beschreibt He [11] durch den Vergleich zwischen Web Services im B2B[1] mit Browsern im B2C[2]-Bereich.

Cerami [1] beschreibt das Web von heute als ein *Human Centric Web*, in dem Menschen die Hauptakteure sind, welche die meisten Web-Anfragen initiieren. Die Erweiterung um interagierende Applikationen führt zu einem *Application Centric Web*. In einem solchen Netz kommen Applikationen durch Web Services als Akteure immer mehr in den Vordergrund. Das Ziel dieser Erweiterung ist es, die Kommunikation zwischen Programmen so einfach zu gestalten wie die zwischen Browsern und Servern.

3.1.2 Beispiele

Die möglichen Einsatzgebiete von Web Services sind vielseitig: Zum einen gibt es die so genannten *Komponentendienste.* Beispiele hierfür wären Währungsumrechnung, Übersetzungen, Überprüfung von Kreditkarten, Börsenkurse, Frachtverfolgung, Flug- und Bahnpläne, Shopping-Bots oder Autorisierungsdienste. Des Weiteren gibt es so genannte *zusammengesetzte Dienste.* Diese verbinden die Funktionalität verschiedener Komponenten, um einen größeren, zusammengesetzten Dienst zu erfüllen. Hierunter fallen Reklamationsbearbeitung oder Reisebuchungen.

Web Services beschränken sich aber nicht nur auf reine Applikationsinteraktionen, sondern beziehen sich auch auf die klassische Kommunikationsform zwischen Anwendern und Servern mittels Webseiten. Web Services können zu einer starken Entkopplung von Präsentation und Inhalten auf Internetseiten führen. Eine starke Ausprägung dessen wäre es, Internetseiten nur als Container zu betrachten. Die eigentlichen Seiteninhalte würden von der Geschäftslogik via parametrisierter Web-Service-Aufrufe angefordert. Somit können sowohl Anwendungen als auch Anwender einen Service nutzen.

Web Services sind nichts von Grund auf Neues. Seit Jahren werden zur Interaktion zwischen Programmen z.B. CGI-Skripte oder Java Servlets eingesetzt. Hierbei handelt es sich aber laut Cerami [1] eher um Ad-hoc-Lösungen. Web Services bieten hier eine Standardisierung, die die Integration verschiedener Applikationen erleichtert.

3.1.3 Definition

Für den Begriff Web Service gibt es keine einheitliche, allgemein anerkannte Definition. Cerami [1] versteht unter Web Services ein Paradigma zum Erstellen von verteilten Web-Applikationen. Er definiert Web Services wie folgt:

> A web service is a service that is available over the Internet, uses a standardized XML messaging system, and is not tied to any one operating system or programming language.

[1] Business-to-Business
[2] Business-to-Customer

Coyle [2] bezeichnet Web Services folgendermaßen:

> Web Services is a technology and process for discovery and connection.

Eine ausführlichere Definition von Web Service gibt IBM [13]:

> Web services are a new breed of Web application. They are self-contained, self-describing, modular applications that can be published, located, and invoked across the Web. Web services perform functions that can be anything from simple requests to complicated business processes. [...] Once a Web service is deployed, other applications (and other Web services) can discover and invoke the deployed service.

Die einzelnen Definitionsansätze werden im Folgenden näher beschrieben und diskutiert.

Web Services sind laut der ersten Definition über das Internet nutzbar, wohingegen die darauf folgenden zwei Definition das Einsatzgebiet von Web Services nicht nur auf das Internet beschränken. Web Services sind danach in allen auf Internettechnologien aufbauenden Netzen, wie beispielsweise auch dem Intranet, einsetzbar. Das Internet wird, nach Vasudevan [15], bevorzugt als Kommunikationsmedium für Web Service Applikationen genutzt, was Vasudevan [15]durch Einfachheit und Allgegenwärtigkeit des Webs begründet. Mit Einfachheit ist hierbei die geringe Anzahl an Protokollen und deren einfacher Funktionalität gemeint. Die wichtigsten Protokolle im Internet sind `HTTP`, `SMTP` und `FTP`. Der Funktionenumfang von `HTTP` beispielswise, kann durch die drei Hauptfunktione `GET`, `POST` und `PUT` beschrieben werden. Nutzt man das Internet als Plattform für den Einsatz von Web Services, kann man also eine vorhandene Infrastruktur mit bekannten und bewährten Protokollen nutzen. Der zweite große Vorteil des Internets als Kommunikationsmedium für Web Services bezieht sich auf die Verbreitung oder auch Allgegenwärtigkeit des Internets. Das Internets ist von nahezu überall zugänglich. Firewalls und Proxyserver, welche für gewöhnlich Internet-Clients von internen Servern separieren, sind für Web Services dank der zugrundeliegenden Standard-Protokolle keine große Hürde, weshalb auch spezielle Anforderungen an die Sicherheit von Web Services sicherzustellen sind.

Laut der ersten Definition von Web Services eignet sich XML besonders gut als Datenstruktur für Web Services aufgrund der Trennung von Struktur, Inhalt und Form sowie der Erweiterbarkeit von XML. Desweiteren ist XML weit verbreitet und bietet den Vorteil, dass sowohl die Wohlgeformtheit, als auch die Validität des Dokument leicht überprüft werden können. Des weiteren gibt es bereits viele Programmiersprachen für den Einsatz von XML, wie beispielsweise XSLT zur Transformierung von XML Dokumenten oder XPath und XQuery als Abfragesprachen.

Die Unabhängigkeit von Betriebssystemen und Programmiersprachen ist ein wesentlicher Bestandteil für die erste Definition von Web Services. Agenten, die Web Services nutzen, können Handys, PDAs, PCs und Legacysysteme sein, aber auch andere, exotischere Plattformen wie z.B. Parkuhren oder Cola-Automaten. Daher ist die Forderung der Unabhängigkeit für Web Services unabdingbar.

Wie bereits in Abschnitt 3.1.2 beschrieben, handelt es sich bei Web Services um nichts von Grund auf Neues. Vergleichbare Funktionalität der Services wurden und werden durch Middleware-Technologien wie RMI, JINI, CORBA, DCOM oder EJB zur Verfügung gestellt. Ihre Stärken liegen in einer gut administrierten aber abgeschlossenen Server-to-Server Kommunikation. Das Hauptproblem dieser Middleware-Technologien liegt an der schwer erreichbaren Interoperabilität. Hinzu kommt eine große Anzahl an Programmiersprachen, -paradigmen und Plattformen was eine Orchestrierung und Integration dieser Services weiter erschwert. Die Benutzung eines Web Service ist dagegen sehr viel transparenter und verhält er sich aus Anwendersicht wie eine Black Box.

Die Definition von IBM hebt hervor, dass Web Services selbstbeschreibend sind. Die Beschreibung eines Web Services enthält Angaben zu allen öffentlich verfügbaren Interfaces zu einem bestimmten Service, die damit verbundenen Datentypen, das zu verwendende Transportprotokoll sowie die Adresse, wo der Service aufrufbar ist. In der Regel erfolgt eine solche Beschreibung via WSDL (siehe Kapitel 4.2).

Nach der Definition von IBM haben Web Services einen modularen Aufbau. Dieser Aufbau hat viele Vorteile, zum Beispiel die Zusammensetzung einer Dienstleistungen aus mehreren Komponenten. So könnte beispielsweise ein Verlag wie Springer online auf spiegel.de verschiedene Artikel oder Dossiers anbieten. Zur Authentifizierung der Benutzer sowie der Abrechnung diverser Gebühren müsste dieser aber kein eigenes System entwickeln, sondern könnte hierzu auf bereits entwickelte und bewährte Module anderer Anbieter wie beispielsweise Microsofts Passport oder FirstGate zurückgreifen. Die Vorteile eines modularen Aufbaus wie Code-Wiederverwendung, zentrale Änderungen, *Validität* oder *Separation of Concerns* sind aus dem Bereich Software Engineering bekannt und finden hier Anwendung.

Ein weitere wichtiger Bestandteil der Web Service-Architektur sind logisch zentrale, verteilte Verzeichnisse, anhand welcher ein bestimmter Service gefunden werden kann. Ist ein passender Service gefunden, muss eine Beschreibung vorhanden sein, wie er aufzurufen ist. Dies setzt eine Registrierung des Services bei einem solchen Verzeichnis voraus. Hierfür wird hauptsächlich UDDI (siehe Kapitel 4.3) eingesetzt.

3.1.4 Vision

Die Vision von Web Services ist ein Automated Web, das die Integration von Web Services automatisiert. Grundlage hierfür sind leicht auffindbare, selbstbeschreibende und sich an Standards haltende Services. In der Industrie wird diese Vision auch als *Just In Time Application Integration* bezeichnet.

Am Anfang der JIT-Idee steht das Abfragen von Service-Registern. Eine Automation setzt das Verstehen der Ergebnisse voraus und eine darauf aufbauende Auswahl der passenden Dienstleistungen. Neben einer Vielzahl von Programmen für diese Aufgabe ist immer noch ein gewisser Grad an menschlicher Interaktion nötig. Das automatische Aufrufen der ausgewählten Services stellt aufgrund einer großen Softwarepalette und einheitlicher Standards kein Problem dar. Auch wenn die beschriebenen Schritte völlig automatisiert ablaufen, existieren noch keine Mechanismen um Geschäftsprozesse völlig zu automatisieren. Gegenwärtige Servicebeschreibungen enthalten keine Aussagen über Preis, Lieferbedingungen wie zum Beispiel Zeit sowie rechtliche Aspekte, beispielsweise bei Nichterfüllung der Lieferung. Auch über den Entwicklungsgrad des Services und dessen Verfügbarkeit fehlen Angaben. Diese Aufgaben sind nicht einfach zu lösen und dadurch auch nicht einfach zu automatisieren. Cerami [1] behauptet aufgrund dieser Probleme, dass eine JIT Application Integration nie realisiert werden wird. Dennoch bringen uns Web Service Technologien einen Schritt näher an die Vision des Automated Webs.

3.2 Web-Service-Architektur

Es gibt zwei Sichtweisen um die Web Service Architektur näher zu betrachten: Zum einen die Web Service Roles und zum anderen den Web Service Protocol Stack.

3.2.1 Web Service Roles

Innerhalb der Web-Service-Architektur gibt es drei Hauptrollen: Service Provider, Service Requestor und Service Registry. In Abbildung 1 sind die beschriebenen Rollen und deren Interaktion veranschaulicht.

Service Provider – Service Provider sind die Anbieter des Services. Sie implementieren den Service und veröffentlichen dessen Beschreibung und dessen Schnittstellen über das Internet.

Service Requestor – Service Requestors sind alle möglichen Kunden des Services. Der Kunde benutzt den Service indem er eine Netzwerkverbindung aufbaut, einen evtl. parametrisierten Request sendet und den erwünschten Service als Antwort darauf erhält.

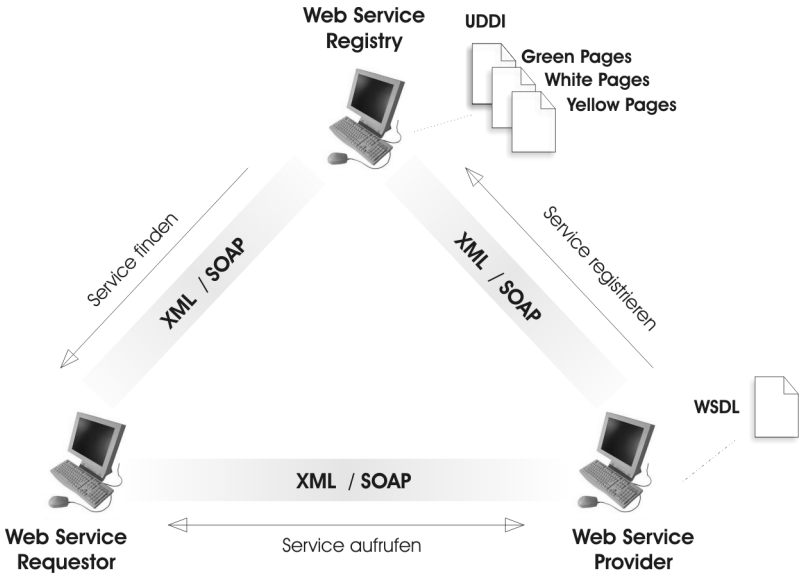

Abbildung 1: Web Service Rollen

Service Registry / Broker – Bei einer Service Registry handelt es sich um ein zentrales, logisches Verzeichnis von Services. Die Registry ist eine zentraler Punkt, via welcher Entwickler ihre Services veröffentlichen können und Kunden existierende auffinden können.

3.2.2 Web Service Protocol Stack

Die Hauptschichten des Web Service Protocol Stack sind Service Transport, XML Messaging, Service Description sowie Service Discovery (siehe Abbildung 2). Nachfolgend werden diese Begriffe kurz erläutert. In den nachfolgenden Abschnitten werden diese dann ausführlicher behandelt.

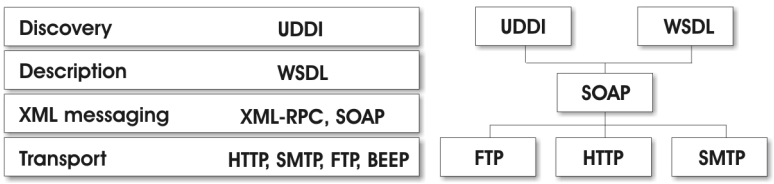

Abbildung 2: Web Service Protocol Stack

Service Transport – Die Service-Transport-Schicht ist für den Transport der Nachrichten zwischen den Applikationen verantwortlich. Wie bereits beschrieben nutzen Web Services die vorhandene Infrastruktur des Internets. Hierbei finden Protokolle wie `HTTP`, `SMTP` und `FTP`, aber auch neuere Protokolle wie z.B. das Blocks Extensible Exchange Protocol (`BEEP`) Anwendung.

XML Messaging – In der XML-Messaging-Schicht werden die Nachrichten in ein gebräuchliches, von beiden Seiten verstandenes `XML`-Format transformiert. Die gegenwärtigen Technologien hierfür sind `SOAP` und `XML-RPC`.

Service Description – Die Beschreibung der öffentlichen Schnittstelle zu einem bestimmten Web Service ist Aufgabe der Service Description Schicht. Die Beschreibung erfolgt durch die Web

Service Description Language (WSDL).

Service Discovery – Die Unterstützung der Veröffentlichung und des Auffindens von Web Services innerhalb eines zentralen Verzeichnisses ist Aufgabe der Service- Discovery-Schicht. Das hier am verbreitetsten Protokoll ist Universal Description, Discovery and Integration Protocol (UDDI).

Im folgenden Kapitel wird auf die Basistechnologien für Web Services eingegangen. In Abschnitt 4.1 wird auf die SOAP Paradigma näher beschrieben. Die Service-Beschreibung anhand von WSDL ist Thema von Abschnitt 4.2. UDDI als logisch zentrales, verteiltes Verzeichnis für Web Services wird in Abschnitt 4.3 behandelt.

4 Basistechnologien

Ziel diese Kapitels ist es die Basistechnologien für Web Services näher zu beschreiben. Es behandelt SOAP (4.1) als Austauschformat für Nachrichten, WSDL (4.2) zur Schnittstellenbeschreibung und UDDI (4.3) als logisch zentrales, verteiltes Verzeichnis für Web Services. Im Anhang (A) sind fernen einige Beispiellistings zu den einzelnen Abschnitten vorhanden.

4.1 SOAP

In diesem Abschnitt wird SOAP näher betrachtet. Er ist gegliedert in einen kurzen Überblick und die Spezifikation. Im Anhang ist ferner ein Beispiellisting einer SOAP-Nachricht über HTTP beigefügt. Die Darstellung dieses Abschnitts erfolgt in Anlehnung an das SOAP-Primer des W3C [9], Coyle [2], Deßloch [3] sowie Cerami [1].

4.1.1 SOAP – Überblick

Die SOAP[3]-Spezifikation [9], welche momentan in der Version 1.2 vorliegt, beschreibt abstrakt gesehen eine auf XML basierende, strukturierte und typisierte Information. Diese kann zum Austausch zwischen Teilnehmern in einer dezentralen, verteilten Umgebung benutzt werden. Bei SOAP handelt es sich eigentlich um ein zustandsloses, One-way-Paradigma zum Nachrichtenaustausch. Komplexere Interaktionsformen, wie zum Beispiel request-response oder request-multiple-responses, können durch eine Kombination der einfachen Nachrichten mit protokollspezifischen Eigenheiten bzw. Anwendungslogik erreicht werden. Die Spezifikation macht keine Aussagen über die Semantik der zu transportierenden Daten, das Routing der Nachrichten, Verlässlichkeit des Transfers oder der Überwindung von Firewalls. SOAP stellt ein Framework zum Nachrichtenaustausch aufbauend auf vorhandene Protokolle wie zum Beispiel HTTP, SMTP oder FTP dar. Beschrieben werden auch die Aktionen, die ein SOAP-Knoten bei Empfang einer Nachricht auszuführen hat.

4.1.2 SOAP – Spezifikation

Nachfolgend möchte ich kurz auf die Begriffe SOAP-Nachricht, SOAP Header und SOAP-Body sowie SOAP via HTTP eingehen.

SOAP-Nachrichten Das SOAP Envelope stellt das Wurzelelement einer SOAP-Nachricht dar. Es besteht aus den beiden Teilen header und body. Die Inhalte dieser Elemente sind applikationsspezifisch. Es werden in der Spezifikation nur Aussagen darüber gemacht, wie diese Elemente aufgebaut sind. Abbildung 3 zeigt den Aufbau einer SOAP-Nachricht.

[3]Die Abkürzung SOAP stand ursprünglich für Simple Object Access Protocol. Das W3C sah aufgrund des Fehlens von Objekten in der Spezifikation die Notwendigkeit zu einer Umbenennung. Aufgrund der Popularität des Begriffs wurde aber darauf verzichtet; die Abkürzung wird aber nicht mehr als solche betrachtet: SOAP ist mittlerweile nur ein Begriff.

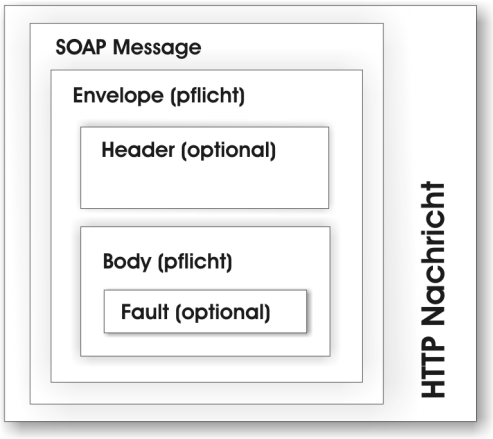

Abbildung 3: SOAP-Envelope – Aufbau

SOAP-Header – Das `header`-Element ist optional. Es wird hauptsächlich dazu verwendet, Kontrollnachrichten zu transportieren. Diese Kontrollnachrichten können Verarbeitungshinweise oder Kontextinformationen der Nachricht beinhalten. Die einzelnen im `header` enthaltenen Elemente werden als `headerBlocks` bezeichnet. Als Anwendung des `header`-Elements seien kurz komplexere Services zu nennen. Bei diesen Value-Added-Services gibt es neben dem eigentlichen Client und ersten Server mehrere Zwischenschritte. Diese Intermediaries erweitern den Service um eigene Teilservices, welche zusammengesetzt den eigentlich angeforderten Service ergeben. Die Koordination dieser Intermediaries wird über das `header`-Element gesteuert, indem die Knoten `headerBlocks` interpretieren, hinzufügen oder löschen. Für den `header` gibt die Spezifikation noch zwei Attribute an: Das `actor`-Attribut legt den Empfänger der Nachricht innerhalb einer Verarbeitungskette fest. Ob ein `header`-Element verstanden und verarbeitet werden muß legt das `mustUnderstand` Attribut fest

SOAP-Body – Das `body`-Element hingegen ist Pflicht. Es enthält die eigentlichen Nachrichten für die Ende-zu-Ende-Kommunikation meist in Form eines RPC request oder response, verpackt in eine XML-Syntax. Diese Inhalte sind wiederum Anwendungsspezifisch und werden deshalb nicht weiter von der SOAP-Spezifikation festgelegt. Als Erweiterung der `body`-Elemente können `Fault`-Elemente angegeben werden. Diese regeln, wie von wem auf Fehler zu reagieren ist.

SOAP via HTTP Wie bereits weiter oben in diesem Kapitel erwähnt ist SOAP nicht an ein spezielles Transportprotokoll gebunden. Am verbreitetsten ist wohl die Kombination von SOAP und `HTTP`; es sind aber auch Kombinationen mit `SMTP`, `FTP`, `XMPP`, IBMs `MQSeries` oder `MSMQ` von Microsoft möglich. Im Folgenden möchte ich kurz auf SOAP via `HTTP` eingehen.

SOAP requests werden via eines `HTTP` requests und die SOAP responses entsprechend mit einer `HTTP` response übertragen. Hierbei können sowohl `HTTP GET` als auch `HTTP POST` verwendet werden. Aufgrund der Beschränkung der Zeichenanzahl bei der `GET`-Variante wird von den meisten Servern `POST` bevorzugt. Clients müssen (Version 1.1) bzw. können (Version 1.2) einen zusätzlichen SOAP Action Header angeben. Dabei handelt es sich um eine serverspezifische URI. Damit ist es möglich schnell und einfach die Absicht des requests zu ermitteln, ohne die SOAP-Nachricht zu untersuchen. In der Praxis wird es dazu verwendet, SOAP request abzublocken bzw. an den richtigen Server intern weiterzuleiten.

Um den Aufbau einer SOAP-Nachricht via `HTTP` zu verdeutlichen habe ich im Anhang A.1.1 ein ausführliches Listing beigefügt.

4.2 WSDL

Im folgenden Abschnitt wird ein kurzer Überblick über die Web Service Description Language in Anlehnung an Cerami [1], Coyle [2], Deßloch [3], Vasudevan [15] sowie der WSDL-Spezifikation [7] gegeben .

4.2.1 WSDL–Überblick

WSDL steht für Web Service Description Language. Es ist eine Spezifikation, in welcher festgelegt ist wie ein Web Service in einer XML Grammatik zu beschreiben ist. Zusammenfassend gesehen beinhaltet eine solche Beschreibung Angaben zu allen öffentlich verfügbaren Schnittstellen zu einem bestimmten Service, zu den damit verbundenen Datentypen, zum verwendende Transportprotokoll sowie zur Adresse, unter welcher der Service aufrufbar ist. Cerami [1] vergleicht WSDL-Elemente mit Java Interfaces. Der Vorteil der WSDL-Elemente liegt in ihrer Plattform- und Sprachunabhängigkeit. WSDL ist eine Plattform zur Beschreibung von Services und deren damit verbunden (eventuell vollautomatischen) Integration und stellt damit einen wesentlichen Eckpunkt der Web-Service-Architektur dar.

4.2.2 WSDL–Spezifikation

In der WSDL-Spezifikation werden sechs Hauptelemente beschrieben: `definitions`, `types`, `message`, `portType`, `binding` und `service`. Ergänzend werden die Elemente `documentation` und `import` genannt. Abbildung 4 gibt einen groben Überblick über deren Zusammenhang.

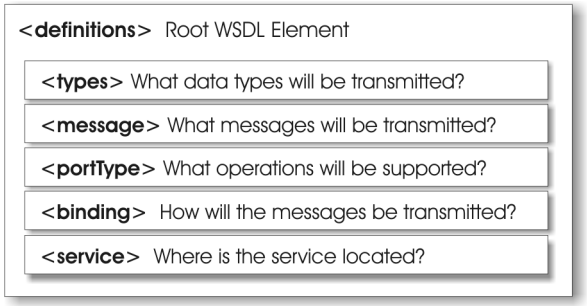

Abbildung 4: WSDL Spezifikation

Nachfolgend werden diese Elemente kurz beschrieben.

definitions – *Root WSDL Element* – Das `definitions`-Element ist das Wurzelelement eines jeden WSDL-Dokuments. Es beinhaltet den Namen des Web Services, alle verwendeten *Namespaces* und enthält die folgenden Elemente:

types – *What data types will be transmitted?* – Anhand der `types`-Elemente werden alle Datentypen, die beim Austausch zwischen dem Client und dem Server verwendet werden, beschrieben. Hierbei ist WSDL nicht an ein bestimmtes Typensystem gebunden. Als *default* wird das Typensystem des W3C XML Schema verwendet.

message – *What messages will be transmitted?* – Die `message`-Elemente beschreiben eine einfache Nachricht, also entweder ein einzelner *request* oder eine einzelne *response.* Sie beinhalten den Namen der Nachricht als auch Parameter bzw. Rückgabewerte (jeweils Name und Typ).

portType – *What operations (functions) will be supported?* – `portType`-Elemente verbinden mehrere `message`-Elemente und bilden somit eine bestimmte Operationen oder Funktion. Ein `portType` kann mehrere Operation beinhalten. WSDL unterstützt hierbei vier Basis-Patterns

für die Kommunikation: `one-way`, `request-response`, `solicit-response` und `notification`. Bei der `one-way`-Kommunikation handelt es sich um eine Nachricht von einem Client an einen Server. Dem gegenüber steht die `notification` als Nachricht von einem Server an einen Client. `request-response` beinhaltet eine Anfrage eines Clients sowie eine entsprechende Antwort eines Servers. Bei `solicit-response` sind die Rollen vertauscht. Die letzten beiden Pattern unterstützten auch eine `fault`-Nachricht, um auf Fehler zu reagieren.

binding – *How will the messages be transmitted on the wire? What SOAP-specific details are there?* – Wie die Informationen im Netz übermittelt werden, wird im `binding`-Element beschrieben. Mögliche `bindings` wären `HTTP-PUT`, `HTTP-POST` oder `SOAP`. Für eine `portType` können mehrere `bindings` angegeben werden. Die `bindings` selber sind protokollspezifisch, sie sind also nicht mehr Bestandteil von WSDL.

service – *Where is the service located?* – Das `service`-Element beschreibt, wo der Service aufgerufen werden kann. Diese Informationen sind in `port`-Elementen gekapselt. Jedes `port`-Element beschreibt die Netzwerkadresse des Endpunktes, wo der Service angeboten wird. Das `service`-Element stellt somit eine Sammlung der möglichen Netzwerkendpunkte für einen bestimmten Service dar.

documentation – Anhand des `documentation`-Elements soll eine Beschreibung in einem menschenlesbaren Format zur Verfügung gestellt werden. Dieses Element kann in jedem der anderen beschriebenen Elemente verwendet werden.

import – Anhand des `import`-Elements können andere WSDL-Dokumente oder XML-Schemata eingebunden werden. Mit ihrer Hilfe lassen sich modulare WSDL-Dokumente erstellen.

Im Anhang A.2.1 ist zur Veranschaulichung der aufgelisteten Element ein Beispiel abgebildet.

4.3 UDDI

Dieser Abschnitt soll einen kurzen Überblick über UDDI liefern. Behandelt werden der konzeptionelle Aufbau sowie die Architektur von UDDI. Im Anhang sind ferner mehrere Beispiellistings zur Veranschaulichung beigefügt. Die Darstellung dieses Abschnitts erfolgt in Anlehnung an die Texte von Vasudevan [15], Cerami [1], Coyle [2], Deßloch [3], JavaMagazin [14] sowie der UDDI-Spezifikation [4].

4.3.1 UDDI–Überblick

Die Abkürzung UDDI steht für Universal Description, Discovery and Integration. UDDI besteht im Grunde aus zwei Teilen: Zum einen ist UDDI eine technische Spezifikation eines verteilten Verzeichnisses für Web Services. Die Daten sind hierzu in einem bestimmten XML-Format abzuspeichern. Für das Veröffentlichen und Suchen enthält die UDDI-Spezifikation eine API. Zum anderen ist die UDDI Business Registry, auch unter dem Namen Cloud Service bekannt, eine voll funktionsfähige Implementierung der UDDI-Spezifikation. Die UDDI Business Registry wurde von IBM und Microsoft im Mai 2001 gestartet. Hierbei handelt es sich um ein globales, öffentliches Onlineverzeichnis zur Beschreibung des Services und dessen Schnittstelle als auch zum Auffinden bestimmter Web Services. Die Hauptfunktionen, die UDDI anbietet sind `publish`, `find` und `bind`. Vasudevan [15] vergleicht `UDDI` mit `Corba` Tradern oder einem `DNS` Service für Geschäftsanwendungen.

Die für UDDI relevanten Daten lassen sich in drei Kategorien einteilen:

White Pages – *Who am I?* – Diese Kategorie enthält allgemeine Informationen über ein bestimmtes Unternehmen. Darin können zum Beispiel Name, Anschrift, Telefonnummern, Internetadresse und weitere Kontaktinformationen sowie eine kurze, evtuell mehrsprachige Beschreibung der Unternehmung und deren Ziele enthalten sein. Diese Informationen werden in einem `businessEntity`-Objekt (siehe Abschnitt 4.3.2) abgespeichert.

Yellow Pages – *What do I offer?* – Yellow Pages stellen einen Klassifikation der Unternehmen oder der angebotenen Services dar. Dabei können Informationen wie die Branche, Produkte oder geographische Codes enthalten sein. Auch hierfür existiert eine Datenstruktur, nämlich die `businessService`-Objekte (siehe Abschnitt 4.3.2).

Green Pages – How to do business with me? – Hierbei handelt es sich um die technischen Informationen eines Web Services. Enthalten sind ein Verweis auf eine (externe) Spezifikation sowie eine Adresse über welche der Service aufgerufen werden kann. Die Details der Service Spezifikation können über `tModels` (siehe Abschnitt 4.3.2) abgefragt werden. Dies sind Metadaten über die verschiedenen Web Services. Nähere Informationen über die Adresse enthält das `bindingTemplate` (siehe Abschnitt 4.3.2).

UDDI ist nicht auf Web Services beschränkt, die auf SOAP basieren. Es kann dazu benutzt werden, jegliche Art von Service, angefangen von einer einfachen Webseite oder eMail Adresse bis hin zu CORBA und Java RMI Services zu erfassen.

4.3.2 UDDI–Architektur

Die UDDI-Architektur besteht aus den drei Teilen UDDI Data Model, UDDI API und UDDI Cloud Service.

UDDI-Datenmodell UDDI beinhaltet ein XML-Schema welches die vier Kernklassen an Information beschreibt: `businessEntity`, `businessService`, `bindingTemplate` und `tModel`. Abbildung 5 gibt einen Überblick über das UDDI Data Model. Die einzelnen Elemente werden im Folgenden kurz erläutert.

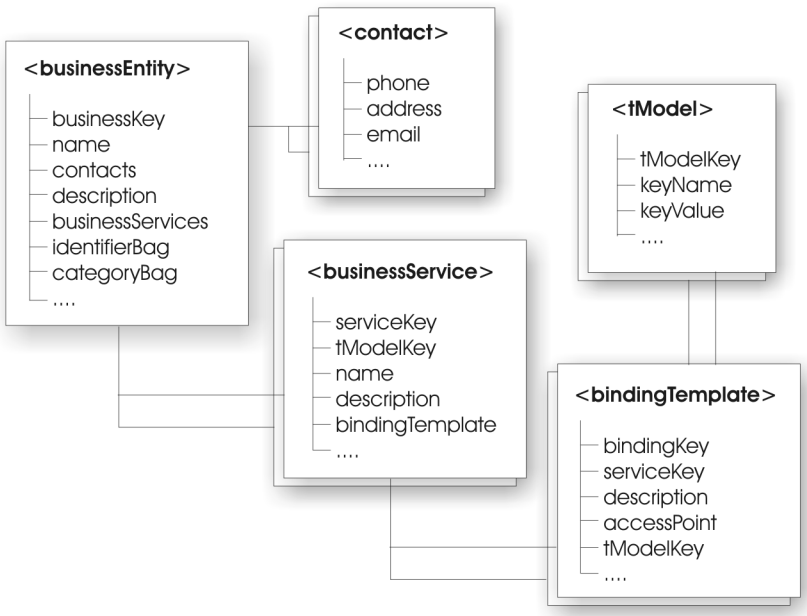

Abbildung 5: UDDI Data Mode

businessEntity – Das `businessEntity` Element enthält Informationen über ein bestimmtes Unternehmen. Die beinhalteten Informationen fallen unter die Kategorie White Pages (siehe Abschnitt 4.3.1). Diese beinhalten den Namen der Unternehmung, eine Beschreibung, Adresse und weitere Kontaktinformationen wie Telefon, eMail oder URLs.

Jedes `businessEntity`-Objekt erhält eine eindeutige ID, den `businessKey`. Dadurch werden verschiedene Services an eine Unternehmung gebunden. Zusätzlich zu den oben genannten, normalen Identifikatoren kann ein `businessEntity`-Objekt weitere Elemente zur Identifikation wie zum Beispiel eine Dun-&-Bradstreet-Nummer[4] oder eine Thomas Registry Supplier ID[5] enthalten. Eine Unternehmung kann sich auch in verschiedene Geschäftskategorien einordnen. Hierbei werden Standardklassifikationen wie NAICS[6], UNSPSC[7] und ISO 3166[8] von UDDI unterstützt.

Im Anhang A.3.1 ist exemplarisch ein solches `businessEntity` aufgelistet.

businessService – Das `businessService`-Object repräsentiert Daten eines einzelnen oder einer Gruppe verwandter Web Services aus den Yellow Pages (siehe Abschnitt 4.3.1). Sie beinhalten den Namen des Services, eine Beschreibung und eine Liste von `bindingTemplates` sowie zur Zuordnung zu dem anbietenden Unternehmen die ID des entsprechenden `businessEntities`. Wie die `businessEntity`-Objekte haben auch diese eine eindeutige ID.

Auch für die `businessService`-Objekte ist im Anhang A.3.2 ein Beispiel aufgeführt.

bindingTemplate – Die Informationen wo und wie man einen bestimmten Web Service Aufrufen kann erhält man über `bindingTemplate`-Objekte. Sie sind Teil der `businessService`-Objekte. Diese `bindings` sind nicht auf `HTTP`-basierte Services beschränkt, sondern unterstützten auch eMail-, Fax-, `FTP`- und Telefonbasierte Services oder einfach eine Homepage. Das Wie wird durch die referenzierten `tModels` näher spezifiziert.

tModel – Die `tModels` (technical Model) werden als Zeiger auf externe technische Spezifikationen eingesetzt. Diese Spezifikationen enthalten hauptsächlich die Details, wie man mit einem bestimmten Web Service kommuniziert. Meistens dient hierfür eine WSDL Datei (siehe Kapitel 4.2), aber auch eine normale Homepage als Anleitung ist ausreichend. Die Formulierung *hauptsächlich* deshalb, weil `tModels` neben den Kommunikationsdetails auf jegliche externe Spezifikationen und auf sich selbst verweisen können. Darunter fällt zum Beispiel auch die weiter oben beschriebene Duns-&-Bradstreet-Nummer.

Ein Beispiel für ein solches `tModel` ist im Anhang A.3.3 abgebildet.

UDDI-API Die UDDI-API deckt die Funktionalität zum Veröffentlichen und Suchen von Einträgen ab. Sie ist SOAP-basiert (siehe Kapitel 4.1) und kann über Web-Interfaces[9] [10] oder Programme[11] [12] [13] benutzt werden. Die Tabelle 1 im Anhang gibt einen kurzen Überblick über die zur Verfügung gestellten Funktionen.

UDDI Cloud Service Bei den UDDI Cloud Services handelt es sich um Vermittlungsseiten als Implementierung der UDDI-Spezifikation. Sie werden von Microsoft und IBM angeboten. Die Cloud Services stellen ein logisch zentrales aber physisch verteiltes Verzeichnis für Web Services dar. Dies bedeutet, dass Daten die bei einem Knoten registriert werden in regelmäßigen Abständen auf allen weiteren Knoten repliziert werden.

Unternehmen haben intern die Möglichkeit, ein privates UDDI Verzeichnis mit firmeninternen Web Services aufzubauen. Diese werden nicht mit den Öffentlichen synchronisiert und sind somit nicht Teil der UDDI Cloud.

[4] Dun & Bradstreet D-U-N-S Nummer (Data Universal Numbering System). Verzeichnis mit über 62 Millionen registrierten Unternehmen (und Filialen). http://www.dnb.com/US/duns_update/index.html

[5] Eindeutige ID für US Amerikanische und Kanadische Unternehmen. http://www.thomasregister.com

[6] North American Industry Classification System. http://www.naics.com

[7] Universal Standard Products and Service Classification. ECCMA Standard. http://www.unspsc.org

[8] International Organization for Standardization. http://www.din.de/gremien/nas/nabd/iso3166ma/index.html

[9] Microsoft UDDI site: http://uddi.microsoft.com

[10] IBM UDDI site: http://www-3.ibm.com/services/uddi

[11] Java: uddi4j. http://oss.software.ibm.com/developerworks/projects/uddi4j

[12] Microsoft.com: http://uddi.microsoft.com/developer/

[13] Perl: http://www.soaplite.com

5 Alternative Ansätze für Web Service

Bisher wurden Web Services auf Basis von SOAP, WSDL und UDDI vorgestellt. In diesem Kapitel möchte ich kurz auf zwei alternative Ansätze für Web Services eingehen: XML-RPC und REST.

5.1 XML-RPC

In diesem Abschnitt wird, in Anlehnung an die XML-RPC Spezifikation [16], das XML-RPC-Konzept vorgestellt. Im Anhang A.4 ist ein Listing eines kompletten Beispiels aufgeführt.

XML-RPC ermöglicht Programmen einen Funktions- oder Methodenaufruf über ein Netzwerk auszuführen. Als Transportprotokoll zwischen Client und Server wird das `HTTP` Protokoll verwendet. Die Anfragen und Antworten werden mit Hilfe eines kleinen XML-Vokabulars kodiert. Die Clients geben lediglich einen Methodennamen und eine Liste mit Parametern an. Die Antwort des Servers ist entweder eine Liste von Ergebniswerten im Erfolgsfall oder ansonsten eine Fehlernachricht. Bei den Parametern bzw. Ergebniswerten handelt es sich um einfache Typ-Wert-Paare. Die unterstützten Typen sind beschränkt auf `int`, `double`, `boolean`, `string`, `dateTime.iso8601` sowie `base64`[14]. Des weiteren können diese Basistypen in `structs`[15] und `arrays`[16] angeordnet werden. XML-RPC unterstützt weder Objekte noch ist eine Erweiterung des Vokabulars um weitere Informationen möglich.

Bei XML-RPC handelt es sich um ein einfaches Konzept mit beschränkten Möglichkeiten. Diese Einfachheit und Beschränktheit machen jedoch die Attraktivität von XML-RPC aus. Die Verwendung von XML-RPC erlaubt es den Programmierern, sich auf die Schnittstellenimplementierung zu konzentrieren und nicht auf das zu verwendende Protokoll. Auch der Aufwand für Testen und Dokumentation lässt sich dadurch reduzieren.

5.2 REST

Neben SOAP und XML-RPC gibt es eine weitere Alternative für die Realisierung von Web Services. Thomas Roy Fielding beschreibt in seiner Dissertation [5] einen Architekturstil, den er REpresentational State Transfer oder kurz REST nennt. In diesem Abschnitt wird REST kurz in Anlehnung an OIO [12] dargestellt.

Bei REST handelt es sich nicht um ein Produkt oder einen Standard, sondern um einen Architekturstiel für Web Services. Hierbei orientiert sich REST an den Prinzipien des World Wide Web. Im Grunde stellt das WWW selbst eine gigantische REST-Anwendung dar.

Die so genannte Welt von REST besteht aus Ressourcen, Repräsentationen, Methoden und Nachrichten, welche im folgenden beschrieben werden:

Ressourcen – Ressourcen können zum Beispiel Webseiten, Bilder oder Servlets sein und werden über URIs adressiert und angesprochen. Eine Webanwendung wird hierbei als eine Ansammlung von Ressourcen gesehen. Mit `HTTP` können Nachrichten an die Ressourcen gesendet werden. Ressourcen können nur indirekt über deren URI manipuliert werden.

Repräsentationen – Unter der Repräsentation einer Ressource versteht man die Form, in welcher sie vom Server bereit gestellt wird. Hierbei kann es sich um Webseiten, Bilder oder um strukturierten Text in Form von XML handeln. Sie kann auf weitere Ressourcen verweisen. Folgt ein Client einem Link in einer Repräsentation, so gelangt er von einem Zustand in einen anderen.

Methoden – Die Semantik der `HTTP`-Methoden `GET`, `PUT`, `POST` und `DELETE` wurden von REST übernommen. Sie stellen die *Verben* dar, die auf die *Hauptwörter* bzw. die *Ressourcen* angewandt werden können. Durch diese vier Methoden besitzt jede REST-Ressource eine generische Schnittstelle, mit welcher alle Anwendungsfälle abgedeckt werden müssen. Durch

[14] Binärinformation, siehe RFC 2045

[15] `structs` sind Name-Typ-Werte-Paare und somit vergleichbar mit HashTables

[16] hierbei werden auch gemischte und mehrdimensionale Arrays unterstützt.

dieses generische Schnittstelle müssen bei REST keine Protokoll-Konventionen bekannt sein, damit Client und Server sich verständigen können. Die Bedeutung der einzelnen Methoden soll folgend kurz beschrieben werden:

- `GET` fragt die Repräsentation einer Ressource ab. Solche Anfragen sind frei von Seiteneffekten.
- `POST` fügt einer Ressource etwas hinzu. Im Gegensatz zu `PUT` ist `POST` mit Seiteneffekten behaftet.
- `PUT` erzeugt neue Ressourcen bzw. ersetzt den Inhalt bereits bestehender Ressourcen.
- `DELETE` löscht Ressourcen.

Diese Schnittstelle ist mit den aus SQL bekannten generischen Methoden `SELECT`, `INSERT`, `UPDATE` und `DELETE` vergleichbar.

Nachrichten – Für die Anwendung von REST muss kein neues Nachrichtenformat erlernt werden: via REST können sämtliche Datenformate übertragen werden. Für die Interpretation einer REST-Nachricht ist serverseitig kein Kontext nötig. Alle vom Server benötigten Informationen zur Verarbeitung einer Nachricht müssen in dieser enthalten sein. Dies hängt damit zusammen, dass der Server nur seinen eigenen Zustand kennt. Auch Sessions werden vom Server nicht unterstützt. Somit ist der Client selber für seinen Status und die Reihenfolge seiner Funktionsaufrufe verantwortlich. Geht es um die Authentifizierung von Clients bei einem Server, so wird auf bereits vorhandene Webtechnologien wie `HTTP` oder `HTTPS` zurückgegriffen.

Abschließend wird zur Veranschaulichung ein kurzes, vereinfachtes Beispiel einer REST basierten Online-Buchhandlung aus Sicht eines Kunden gegeben.
Unser Kunde interessiert sich für Java Bücher des Verlags Prentice Hall. Die Suchanfrage könnte lauten:

```
GET /book/byPublisher/PrenticeHall
```

Als Ergebnis erhält er eine XML-Nachricht aller Java-Bücher des Verlags. Sein Interesse gilt dem Buch *Java - Volume II Advanced Features* mit der Artikelnummer 0-13-092738-4. Durch die Anfrage

```
GET /book/byID/0-13-092738-4
```

erhält er die Details. Er ist begeistert und möchte sich das Buch sofort bestellen. Dazu muss er sich zunächst einen Warenkorb einrichten:

```
PUT /Warenkorb
```

Die Antwort enthält die ID seines Warenkorbs, in unserem Beispiel die 78554. Als nächstes muss das Buch in den Warenkorb gelegt werden:

```
POST /Warenkorb/78554
book/0-13-092738-4
```

Zu guter letzt fehlt nur noch die Bestellung:

```
PUT /order/78554
```

6 CORBA vs. Web Services

Im letzten Kapitel werden Web Services (SOAP, WSDL, UDDI) mit der verbreiteten Middlewaretechnologie CORBA[17] verglichen. Der Vergleich ist an das Paper *Reinventing the Wheel? Corba vs. Web Services* [6] angelehnt. In dem Abschnitt Vergleich sollen sowohl die Gemeinsamkeiten und Unterschiede im Verarbeitungsmodell als auch die Eigenschaften der einzelnen Technologien vorgestellt werden. Im Abschnitt Einsatzgebiete 6.2 werden die Stärken und Schwächen der einzelnen Technologien in Bezug auf bestimmte Einsatzgebiete vorgestellt.

[17] CORBA – Commen Object Request Brocker Architecture

6.1 Vergleich

Der Hauptunterschied zwischen beiden Ansätzen ist, dass CORBA eine objektorientierte Komponentenarchitektur zur Verfügung stellt wohingegen SOAP in erster Linie nachrichtenbasiert ist. Des Weiteren ist CORBA von Haus aus mit Services wie Events, Naming oder Tradern ausgestattet, was den Entwicklern ermöglicht, sich mehr auf die Geschäftslogik als auf die Details der Kommunikationsinfrastruktur zu konzentrieren. Die weiteren Unterschiede oder Ähnlichkeiten sind im Folgenden untergliedert in das Verarbeitungsmodell und die Eigenschaften der Technologien. Anhand von Abbildung 6 soll ein kurzer Vergleich zwischen den beiden Technologien gegeben werden.

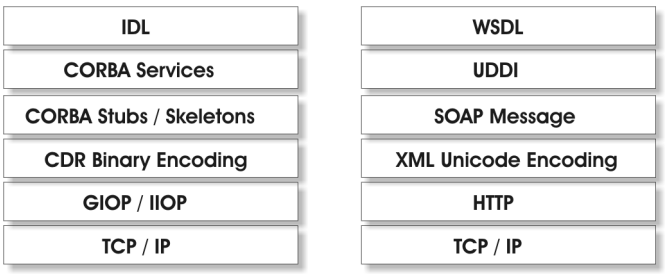

Abbildung 6: CORBA und Web Service Technologie Stack

6.1.1 Verarbeitungsmodell

Datenmodell – CORBA und Web Services unterscheiden sich hierbei in der Kopplung zwischen Kommunikationspartnern sowie der Verbindung. Bei CORBA besteht eine enge Kopplung zwischen Client und Server. Beide Seiten verfügen über die gleichen Schnittstellen (auf der Clientseite ein Stub und auf der Serverseite ein entsprechendes Skeleton) und auf beiden Seiten muss eine ORB[18] am Laufen sein. Was Verbindungen zwischen CORBA Knoten betrifft, so müssen nach der ersten Nachricht für alle flgenden keine neuen Verbindungen mehr aufgebaut werden. Im Gegensatz dazu handelt es sich bei Web Services um einen entkoppelten Ansatz. Es werden hier keine Objekte, sondern nur Nachrichten übermittelt. Für jede Nachricht ist eine neue Vermittlung notwendig.

Aufruf-Semantik – Um Datenintegrität zu gewährleisten, sollte ein Server die *At-most-once*-Semantik unterstützen. Dies verhindert die mehrmalige Verarbeitung einer Anfrage eines Clients. CORBA ORBs unterstützen eine solche Semantik. Bei Web Services hängt eine solche Semantik vom zugrundeliegenden Protokoll ab. Im Fall von `HTTP` wird dies nicht unterstützt und erfordert somit einen zustandsorientierten Service, bei welchem der Anwendungsserver diese Funktionalität übernehmen muss. Ein weiterer Kritikpunkt bei Web Services ist die Fehlerbehandlung. Im Falle eines Fehlers kann eine `SOAP Error Message` versendet werden, welche aber gleichzeitig einen `HTTP-500`-Fehlerstatus enthalten muss. Hierbei wird das Schichtenmodell verletzt.

Skalierbarkeit – CORBA ist durch seine Lastbalancierung skalierbar. Bei Web Services ist dies wiederum nicht Bestandteil des Standards und wird somit den implementierenden Anwendungsservern überlassen.

Überprüfungen zur Compile- oder Laufzeit – Von CORBA werden zwei Arten an Schnittstellen zur Verfügung gestellt: IDL[19] und DII[20]. Bei ersterem handelt es sich um eine getypte

[18] Object Request Broker
[19] Interface Description Language
[20] Dynamic Invocation Interface

Schnittstelle, welche gewisse statische Überprüfungen erlaubt, das dynamische DII erlaubt diese hingegen nicht. Bei der Laufzeitunterstützung kann CORBA auf Fähigkeiten der Zielsprache zurückgreifen. Zur Zeit wird bei Web Services keine standardisierte Infrastruktur zur statischen Überprüfung zur Verfügung gestellt. Zur Laufzeit wird nur die Wohlgeformtheit der XML-Nachricht sichergestellt. Für die Zukunft wäre allerdings eine WSDL Anbindung zu verschiedenen Programmiersprachen denkbar, wodurch Überprüfungen zur Compilerzeit ermöglicht würden. Die Überprüfung der Validität beim Verarbeiten der SOAP-Nachricht hingegen ist feiner im Vergleich zu IDL, da XML-Schemata ausdrucksstärker sind (z.B. reguläre Ausdrücke und Wertebereiche).

6.1.2 Eigenschaften

Firewall – Die Überwindung von *Firewalls* stellt für CORBA noch eine Herausforderung dar. Die *CORBA Firewall Traversal Specification* wurde im März 2004 verabschiedet, sie wird bisweilen aber nur von wenigen Anbieter unterstützt. Das für Web Services am meisten angewandte Protokoll ist `HTTP`. Firewalls sind für gewöhnlich so konfiguriert, dass sie sowohl eingehende als auch ausgehende `HTTP`-Nachrichten problemlos passieren lassen. Somit erübrigen sich jegliche Änderungen in der Konfiguration der Firewall.

Sicherheit – Sicherheit in verteilten Umgebungen umfasst Aspekte wie Authentifizierung, Autorisierung, Verschlüsselung oder Datenintegrität. All diese Anforderungen werden von dem *CORBA Security Service* unterstützt. Bei Web Services fehlt eine solche Standardisierung. Allerdings lassen sich aufbauend auf Internettechnologien wie SSL oder XML-Signaturen ähnliche Sicherheitsmechanismen realisieren.

Persistenz – Wiederum bietet CORBA durch die CORBA Persistent State Services einen Mechanismus zur Persistenz an. Web Services überlassen Persitenzangelegenheiten den jeweiligen Anwendungen.

Plattforunabhängigkeit – Sowohl CORBA als auch Web Services sind Plattform- und Programmiersprachenunabhängig.

Minimalanforderungen – Mobile Endgeräte oder Thin Clients stellen Anforderungen an eine schlanke Technologie. Für eingebettete Systeme bietet CORBA eine Minimalspezifikation. An der Spezifikation für drahtlose Geräte wird momentan noch gearbeitet. Sie wird den Namen *Mobile Agent Facility* tragen und liegt zur Zeit in der Version 1.0 vor. Dennoch benötigen solche Ansätze eine minimale Infrastruktur in Form eines *ORB-lite*. Die Minimalanforderungen für Web Services sind abhängig von der Komplexität der Clients und Server. Im Extremfall könnte ein Client sogar ohne vollständigen XML-Parser auskommen.

6.2 Einsatzgebiete

Die beiden gegenübergestellten Technologien CORBA und Web Services haben unterschiedliche Stärken und Schwächen. Die Frage, welche Technologie verwendet werden soll hängt vom Einsatzgebiet ab. Im Folgenden wird eine Übersicht über die verschiedenen Anwendungsgebiete und die dafür zu bevorzugende Technologie gegeben:

Web Interfaces – Für Geschäftsanwendungen, die auf Web Schnittstellen basieren ist SOAP aufgrund der Verknüpfung mit XML und `HTTP` die erste Wahl. XML kann via XSLT einfach in eine (x)HTML-Darstellung transferiert werden. Web Schnittstellen mit CORBA zu realisieren ist erheblich aufwendiger.

Interaktion mit Legacy-Systemen – Ein Vorteil von CORBA ist die einfache Kommunikation mit bestehenden Middlewaretechnologien wie zum Beispiel EJB oder COM/DCOM. Einige Legacy-Systeme sind bereits mit CORBA-Schnittstellen ausgestattet. Allerdings existieren CORBA-SOAP-Gateways, um diese Lücke zu überwinden.

Mobile Endgeräte – In einem solchen Anwendungsfall sind die Clients oder die Server mobil. Die Herausforderung besteht in sich ändernden Netzwerkadressen, dem Weiterleiten von Nachrichten sowie unzuverlässigen Verbindungen. Zur Zeit ist CORBA für solche Anforderungen schlecht ausgerüstet. Arbeiten der OMG[21] an *Wireless CORBA* sind noch nicht abgeschlossen, das Dokument liegt momentan in Version 1.1 (unabgeschlossen) vor. Web Services haben dank Proxys kaum Probleme mit diesen Anforderungen.

Thin Clients – Die im Abschnitt 'Minimalanforderungen' erwähnte minimale Infrastruktur für auf CORBA basierende Anwendungen führt zu höheren Kosten, verursacht durch größeren Speicher oder schnellere Prozessoren. Web Services hingegen sind hier sehr genügsam. Die Aufgabe besteht lediglich im Senden und Empfangen von SOAP-Nachrichten. Ein XML-Parser könnte auf die Anwendungsdomäne beschränkt sein.

Zustandsorientierte Anwendungen – Bei CORBA wird ein Zustand durch Objekte repräsentiert. Zustandsveränderungen können via Methodenaufrufen erreicht werden. SOAP ist hingegen ein zustandsloses Protokoll. Um einen Status über mehrere Nachrichten zu erhalten, ist das wiederholte Versenden aller Informationen notwendig. In zustandsorientierten Anwendungen ist CORBA durch seinen objektorientierten Ansatz Web Services überlegen.

Bei der Auswahl einer Technologie für eine Anwendungsdomäne spielen noch weitere zwei Faktoren eine entscheidende Rolle. Zum Einen betrifft dies die Entwickler. Hierbei spielen die Vertrautheit mit den Technologien, die Lernkurve, aber auch die Anzahl der benötigten Entwickler zur Implementierung eine wichtige Rolle. Hier haben Web Services aufgrund der bekannten und verbreiteten Protokolle und Formate einen Vorteil. Zum Anderen betrifft dies die Anwendbarkeit der einzelnen Technologien. Die Anwendbarkeit hängt sowohl vom Grad der Standardisierung als auch von der Verfügbarkeit der Werkzeugen ab. Hier hingegen kann CORBA gegenüber Web Services einen Vorteil aufweisen, da bereits viel Entwicklungsaufwand in CORBA investiert wurde.

7 Zusammenfassung

Der Begriff Web Services erfreut sich sowohl in der Industrie als auch im akademischen Umfeld großer Popularität. Wie bei vielen *Hype*-Begriffen ist auch bei Web Services mit dieser Popularität eine verschwommene Abgrenzung des Begriffs verbunden. Eines der Ziele dieses Seminars war es, die Begriffe Web Service und serviceorientierte Architektur genauer zu beschreiben.

Einige der Technologien und Standards als auch Werkzeuge für Web Services sind momentan noch in der Entwicklungsphase. Trotz positiver Erfahrungsberichte ist es deshalb unklar, ob Web Service die an sie gesetzten Erwartungen erfüllen werden. Die wichtigsten Vorteile von Web Services liegen in ihrer Struktur. Sie bauen auf vertraute und etablierte Protokolle und Standards wie `http` oder XML auf, sind plattform- und programmiersprachenunabhängig, haben einen modularen Aufbau und sind selbstenthaltend sowie selbstbeschreibend. Sie stellen einen kleinen gemeinsamen Nenner für den Nachrichtenaustausch zwischen Programmen über Netzwerke dar. Ein weiterer Punkt, warum sich Web Services durchsetzen werden, liegt in der starken Unterstützung durch die Industrie. Firmen wie IBM, Microsoft und BEA sind die treibenden Kräfte hinter Web Services. Die Basistechnologien stellen die in diesem Aufsatz näher beschriebenen Technologien SOAP, WSDL und UDDI dar. Alternative Ansätze wie XML-RPC oder REST werden sich nicht durchsetzen. Beide sind durch ihre Einfachheit was den Aufbau betrifft nicht mächtig genug, um mit SOAP konkurrieren zu können.

[21] Object Management Group

A Anhang

A.1 SOAP

A.1.1 Listing SOAP Message

Das Listing der SOAP Message ist an Vasudevan [15] angelehnt.

```
POST /perl/soaplite.cgi HTTP/1.0
Host: http://thirdparty.example.org
ontent-Type: text/xml; charset=utf-8
SOAPAction: ""

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
   <env:Header>
      <t:transaction
         xmlns:t="http://thirdparty.example.org/transaction"
         env:encodingStyle="http://example.com/encoding"
         env:mustUnderstand="true" >
      </t:transaction>
   </env:Header>
   <env:Body>
      <m:chargeReservation
         env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
         xmlns:m="http://travelcompany.example.org/">
         <m:reservation xmlns:m="http://travelcompany.example.org/reservation">
            <m:code>
               FT35ZBQ
            </m:code>
         </m:reservation>
         <o:creditCard xmlns:o="http://mycompany.example.com/financial">
            <n:name xmlns:n="http://mycompany.example.com/employees">
               Thomas Mustermann
            </n:name>
            <o:number>
               123456789099999
            <o:number>
            <o:expiration>
               2005-02
            </o:expiration>
         </o:creditCard>
      </m:chargeReservation>
   </env:Body>
</env:Envelope>
```

A.2 WSDL

A.2.1 Listing WSDL

Das Listing WSDL erfolgt in Anlehnung an Cerami [1] (Seite 122f).

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService"
   targetNamespace="http://www.wenzler.info/wsdl/HelloService.wsdl"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.wenzler.info/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string" />
   </message>
   <message name="SayHelloResponse">
      <part name="greetings" type="xsd:string" />
   </message>

   <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest" />
         <output message="tns:SayHelloResponse" />
      </operation>
   </portType>

   <binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
```

```
24            transport="http://schemas.xmlsoap.org/soap/http" />
25         <operation name="sayHello">
26            <soap:operation soapAction="sayHello" />
27            <input>
28               <soap:body
29                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
30                  namespace="urn:examples:helloservice"
31                  use="encoded" />
32            </input>
33            <output>
34               <soap:body
35                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
36                  namespace="urn:examples:helloservice"
37                  use="encoded" />
38            </output>
39         </operation>
40      </binding>
41
42      <service name="Hello_Service">
43         <documentation>
44            WSDL file for HelloService
45         </documentation>
46         <port binding="tns:Hello_Binding" name="Hello_Port">
47            <soap:address
48               location="http://localhost:8080/soap/servlet/rpcrouter" />
49         </port>
50      </service>
51   </definitions>
```

A.3 UDDI

A.3.1 Listing `businessEntity`

Das Listing `businessEntity` ist an Cerami [1] (Seite 162f) angelehnt.

```
1   <businessEntity
2      businessKey="0076b468-eb27-42e5-ac09-9955cff462a3"
3      operator="Microsoft Corporation"
4      authorizedName="Martin Kohlleppel">
5      <name>
6         Microsoft Corporation
7      </name>
8      <description xml:lang="en">
9         Empowering people through great software
10        any time, any place and on any device is Microsoft s
11        vision. As the worldwide leader in software for personal
12        and business computing, we strive to produce innovative
13        products and services that meet our customer s
14     </description>
15     <contacts>
16        <contact useType="Corporate Addresses and Telephone">
17           <description xml:lang="en">
18              Corporate Mailing Addresses
19           </description>
20           <personName />
21           <phone useType="Corporate Headquarters">
22              (425) 882-8080
23           </phone>
24           <address sortCode="~" useType="Corporate Headquarters">
25              <addressLine>Microsoft Corporaton</addressLine>
26              <addressLine>One Microsoft Way</addressLine>
27              <addressLine>Redmond, WA 98052-6399</addressLine>
28              <addressLine>USA</addressline>
29           </address>
30        </contact>
31        <contact useType="Technical Contact - Corporate UD">
32           <description xml:lang-"en">
33              World Wide Operations
34           </description>
35           <personName>
36              Martin Kohlleppel
37           </personName>
38           <email>
39              martink@microsoft.com
40           </email>
41        </contact>
```

```
42      </contacts>
43      <identifierBag>
44          <keyedReference
45              tModelKey="uuid:8609c81e-ee1f-4d5a-b202-3eb13ad01823"
46              keyName="D-U-N-S" keyValue="08-146-6849" />
47      </identifierBag>
48      <categoryBag>
49          <keydReference
50              tModelKey="uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bbc"
51              keyName="NAICS: Software Publisher" KeyValue="51121" />
52      </categoryBag>
53  </businessEntity>
```

A.3.2 Listing `businessService`

Das Listing `businessService` ist an Cerami [1] (Seite 165f) angelehnt.

```
1   <businessService}
2       serviceKey="d5921160-3e19-11d5-98bf-002035229c64"
3       businessKey="ba744ed1-3aaf-11d5-80dc-002035229c64">
4       authorizedName="Martin Kohlleppel">
5       <name>
6           XMethods Delayed Stock Quotes
7       </name>
8       <description xml:lang="en">
9           20-minute delayed stock quotes
10      </description>
11      <bindingTemplages>
12          <bindingTemplate>
13              serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
14              bindingKey="d5594a90-3e16-11d5-98bf-002035229c64"/>
15              <description xml:lang="en">
16                  SOAP binding for delayed stock quotes service
17              </description>
18              <accessPoint URLType="http">
19                  http://services/xmethods.net:80/soap
20              </accessPoint>
21              <tModelInstanceDetails>
22                  <tModelInstanceInfo
23                      tModelKey="uuid:0e727db0-3e14-11d5-98bf-00203529c64"/>
24              </tModelDetails>
25          </bindingTemplate>
26      </bindingTemplates>
27  </businessService>
```

A.3.3 Listing `tModel`

Das Listing `tModel` ist an Cerami [1] (Seite 167) angelehnt.

```
1   <tModel
2       tModelKey="uuid:0e757db0-3e14-11d5-98bf-002035229c64"
3       operator="www.ibm.com/services/uddi">
4       authorizedName="010000001QS1">
5       <name>XMethods Simple Stock Quote</name>
6       <description xml:lang="en">
7           Simple stock quote interface
8       </description>
9       <overviewDoc>
10          <description xml:lang="en">
11              wsdl link
12          </description>
13          <overviewURL>
14              http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl
15          </overviewURL>
16      </overviewDoc>
17      <categoryBag>
18          <keyedReference
19              tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
20              keyName="uddi-org:types"
21              keyValue="wsdlSpec" />
22      </categoryBag>
23  </tModel>
```

Funktionsname	Beschreibung
find_XXX Functions	
find_binding	sucht `bindingTemplate` Objekt für best. Service
find_business	sucht Unternehmen nach Kriterum
find_service	sucht Services einer Unternehmung
find_tModel	sucht `tModels` nach Kriterien
get_xxx Functions	
get_bindingDetail	liefert ein `bindingTemplate`-Datensatz
get_businessDetail	liefert einen `businessEntity`-Datensatz
get_serviceDetail	liefert einen `businessService`-Datensatz
get_tModelDetail	liefert einen `tModel`-Datensatz
Authentication Functions	
get_authToken	liefert Authentifizierungstoken
discard_authToken	Invalidierung eines authTokens
save_xxx Functions	
save_binding	erzeugt oder aktuallisiert `bindingTemplate`
save_business	erzeugt oder aktuallisiert `businessEntity`
save_service	erzeugt oder aktuallisiert `businessService`
save_tMdel	erzeugt oder aktuallisiert `tModel`
delete_xxx Functions	
delete_binding	löscht `bindingTemplate`
delete_business	löscht `businessEntity`
delete_service	löscht `businessService`
delete_tModel	versteckt `tModel` (löschen nicht möglich)

Tabelle 1: Ausschnitt aus der UDDI API

A.3.4 Ausschnitt aus der UDDI-API

Tabelle Auschnitt aus der UDDI API 1 ist übernommen aus Coyle [2] (Seite 155).

A.4 XML-RPC

Das folgenden Listings zu XML-RPC sind in Anlehnung an Cerami [1] (Seite 36ff).

A.4.1 XML-RPC request

```
1  POST /RPC2 HTTP/1.0
2  User-Agent: Frontier/5.1.2 (WinNT)
3  Host: betty.userland.com
4  Content-Type: text/xml
5  Content-length: 181
6
7  <?xml version="1.0"?>
8  <methodCall>
9     <methodName>
10       examples.getStateName
11    </methodName>
12    <params>
13       <param>
14          <value>
15             <int>
16                41
17             </int>
18          </value>
19       </param>
20    </params>
21 </methodCall>
```

A.4.2 XML-RPC response

```
1  HTTP/1.1 200 OK
2  Connection: close
3  Content-Length: 158
4  Content-Type: text/xml
5  Date: Fri, 17 Jul 1998 19:55:08 GMT
6  Server: UserLand Frontier/5.1.2-WinNT
7
8  <?xml version="1.0"?>
9  <methodResponse>
10    <params>
11       <param>
12          <value>
13             <string>
14                South Dakota
15             </string>
16          </value>
17       </param>
18    </params>
19 </methodResponse>
```

Literatur

[1] Ethan Cerami. *Web Services - Essentials*. O'Reilly, 2002. URL www.ecerami.com.

[2] Frank P. Coyle. *XML, Web Services, and the Data Revolution*. Addison Wesley, 2002.

[3] Stefan Deßloch. *Vorlesung: Web Services und Workflows*. Vorlesungsskript, 2003. URL http://wwwdvs.informatik.uni-kl.de/courses/WFWS/WS0304/.

[4] Tom Bellwood et. al. UDDI version 3.0. 2003. URL http://uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[5] Thomas Fielding. *Principled Design of the Modern Web Architecture*. 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[6] Aniruddha Gokhale, Bharat Kumar, and Arnaud Sahuguet. *Reinventing the Wheel? CORBA vs. Web Services*. 2002. URL http://www2002.org/CDROM/alternate/395.

[7] W3C Service Architecture Working Group. W3C WSDL Specification. *http://www.w3.org/TR/wsdl20/*, 2004.

[8] W3C Web Service Working Group. Web service architecture. 2004. URL http://www.w3.org/TR/ws-arch.

[9] Web Service Architecture Working Group. *W3C SOAP Specification*. W3C, 2003. URL http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[10] XML Schema Working Group. *W3C XML Specification*. W3C, 2002. URL http://www.w3.org/XML/.

[11] Hao He. What is service oriented architecture. 2003. URL http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html.

[12] Orientation in Objects Group. *REST Web Services – Eine Einführung*. 1999. URL http://www.oio.de/public/xml/rest-webservices.htm.

[13] Doug Tidwell. Web services - the web's next revolution. *IBM - Developer Works*, 2000. URL https://www6.software.ibm.com/developerworks/education/wsbasics/index.html.

[14] Stefan Tilkov. JavaMagazin - UDDI Revisited, 5 2004.

[15] Venu Vasudevan. A web services primer. 2001. URL http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/index.html?page=1.

[16] Dave Winer. *XML-RPC Specification*. 1999. URL http://www.xmlrpc.com/spec.