Aus dem Institut für Telematik
der Universität zu Lübeck

Direktor:
Prof. Dr. rer. nat. Stefan Fischer

Optimierte Protokolle für Web Services mit begrenzten Datenraten

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
– Aus der Technisch-Naturwissenschaftlichen Fakultät –

Vorgelegt von
Herrn Dipl.-Inf. Christian Werner
aus Salzgitter

Lübeck, im Juni 2006

Erster Berichterstatter:	Prof. Dr. rer. nat. Stefan Fischer
Zweiter Berichterstatter:	Prof. Dr. rer. nat. Volker Linnemann

Tag der mündlichen Prüfung: 10. August 2006

Zum Druck genehmigt.

Lübeck, den 10. August 2006

gez. Prof. Dr. rer. nat. Enno Hartmann
– Dekan der Technisch-Naturwissenschaftlichen Fakultät –

Vorwort

Kritiker behaupten, das Qualitätsniveau deutscher Universitäten sei im internationalen Vergleich auf eine Position im Mittelfeld zurückgefallen und sinke immer weiter – dieses Problem beträfe dabei Forschung und Lehre gleichermaßen. Ich möchte vorwegnehmen, dass ich nicht beabsichtige, diese provokante These zu widerlegen oder gar zu bestätigen, denn dies ist sicherlich nicht die Aufgabe eines Informatikers, sondern vielmehr die eines Sozialwissenschaftlers.

Vermutlich fragen sich aber viele frisch gebackene Universitätsabsolventen: Soll ich in Deutschland promovieren und mich für viele weitere Jahre an einer Bildungseinrichtung betätigen, die inzwischen gemeinhin als zweitklassig gilt?

Ich muss zugeben, dass ich mir diese Frage nie gestellt habe; hätte ich es getan, wäre ich aus meinem damaligen Blickwinkel möglicherweise zu der Entscheidung gekommen, dieses Vorhaben nicht zu wagen. Heute – nach vierjähriger Lehr- und Forschungstätigkeit – bin ich jedoch froh darüber, dass mir derartige Zweifel nie in den Sinn gekommen sind. Das Unterfangen hat sich auf jeden Fall gelohnt: In diesen vier Jahren hatte ich nicht nur die Möglichkeit, viele neue Ideen zu entwickeln, sondern konnte diese auch direkt in die Tat umsetzen. Zudem – und dies ist sicherlich ebenfalls ein ganz wesentlicher Punkt – hat mir die Arbeit im wissenschaftlichen Bereich sehr viel Spass gemacht.

Ich hatte also in meinem Umfeld keineswegs das Gefühl, dass man an deutschen Universitäten zur wissenschaftlichen Arbeit im Mittelfeld verdammt ist – im Gegenteil. Sicherlich lässt sich mein persönlicher Eindruck nicht verallgemeinern, aber dennoch bin ich der festen Überzeugung, dass der Erfolg eines (Nachwuchs-)Wissenschaftlers nicht primär vom Standort des Forschungsinstitutes abhängt. Man kann also auch heute noch an einer deutschen Universität durchaus international erfolgreich forschen. Viel entscheidender ist die Zusammensetzung der Forschergruppe, in der man tätig ist. Denn – und auch dies hat sich mir in den vergangenen Jahren klar gezeigt – in der modernen Wissenschaft kann man als „Einzelkämpfer“ nicht bestehen.

Daher danke ich allen, die mich bei meiner Arbeit immer wieder inspiriert, motiviert und begleitet haben, vor allem meinen Eltern.

Ich danke auch den zahlreichen Studierenden, die mich in meiner wissenschaftlichen Arbeit unterstützt haben. Hervorheben möchte ich hier Frau Ylva Brandt und Herrn Tobias Jäcker – beide haben an der TU Braunschweig ihre Diplomarbeiten angefer-

tigt und wurden von mir mit betreut. Beide Arbeiten sind von ganz hervorragender Qualität und haben wichtige Ergebnisse für meine eigene Forschung geliefert.

Herzlich danken möchte ich außerdem meinen Kollegen. Ihre kritischen Anregungen haben ganz wesentlich zum Gelingen meines Promotionsvorhabens beigetragen. Lars Wolf, Jens Brandt, Verena Kahmann, Xiaoyuan Gu, Muhammad Khan, Fadi Tirkawi, Jörg Diederich, Matthias Dick, Dieter Brökelmann, Ulrich Timm, Ulrich Abelmann, Oliver Wellnitz, Frank Strauß, Marc Bechler, Zefir Kurtisi, Andreas Kleinschmidt, Martin Gutbrod, Beda Hammerschmidt, Angela König, Horst Hellbrück, Birgit Schneider, Axel Wegener, Stefan Schmidt, Carsten Buschmann, Dennis Pfisterer, Martin Lipphardt, Daniela Krüger, Dirk Schmidt – Ihr alle habt dafür gesorgt, dass ich in den vergangenen vier Jahren nicht nur sehr viel gelernt habe, sondern mir die Arbeit in dieser Zeit auch großen Spass gemacht hat. Frau Birgit Schneider danke ich besonders für ihre Ausdauer, meine gesamte Dissertation auf Schreibfehler durchzusehen.

Ebenfalls zu großem Dank verpflichtet bin ich Herrn Professor Erik Maehle vom Institut für Technische Informatik für die Übernahme des Vorsitzes in der Prüfungskommission sowie Herrn Professor Volker Linnemann vom Institut für Informationssysteme für die gründliche Durchsicht meiner Dissertation und die Übernahme des Koreferats.

Mein ganz besonderer Dank gilt meinem Doktorvater Herrn Professor Stefan Fischer: Er hat meine Arbeit nicht nur in ausgezeichneter Weise fachlich betreut, sondern hat mir auch immer die Freiheit gegeben, die ich brauchte, um eigene Ideen zu entwickeln, zu verfolgen und zu verwirklichen. Außerdem hat er auf eindrucksvolle Weise demonstriert, wie man eine wissenschaftliche Arbeitsgruppe zielorientiert und erfolgreich leitet. Sein organisatorisches Geschick und seine pragmatische Art, Probleme anzugehen und zu lösen, werden mir stets als Vorbild in Erinnerung bleiben. Eine bessere Betreuung kann man sich als Doktorand nicht wünschen!

Lübeck, im August 2006 Christian Werner

Kurzfassung

Web Services konnten sich in den letzten Jahren als universelle Middleware-Technologie in vielen Bereichen der Informatik etablieren. Durch den konsequenten Einsatz von Web-Standards – hier vor allem XML – ermöglichen Web Services die Realisierung von verteilten Anwendungen auch in stark heterogenen Umgebungen. Allerdings bringt der durchgängige Einsatz von XML auch einen erheblichen Nachteil mit sich: Die Darstellung aller Daten als Text verursacht im Vergleich zu älteren Middleware-Technologien mit binärer Datenrepräsentation (z. B. CORBA oder Java RMI) ein deutlich höheres Datenaufkommen. Zwar spielt dieser Nachteil bei modernen, drahtgebundenen Netzwerken kaum noch eine Rolle, denn hier reicht die zur Verfügung stehende Datenrate in aller Regel aus. Im Gegensatz dazu konnte sich die Web-Service-Technologie in Anwendungsfeldern, in denen die Kommunikation über eine Funkschnittstelle erfolgt, bislang nicht durchsetzen. Im Rahmen dieser Arbeit untersucht der Autor daher Konzepte zur Reduzierung des Datenaufkommens.

Kern dieser Arbeit bilden zwei neuartige Ansätze zur Datenkompression, die auf die besonderen Erfordernisse von Web Services zugeschnitten sind. Das erste Verfahren basiert auf dem Ansatz der Differenzcodierung. Hier wird die Differenz zwischen dem zu übertragenden SOAP-Dokument und einem so genannten Skelettdatensatz berechnet, der zuvor aus der WSDL-Beschreibung des Web Services generiert wurde. Da der Skelettdatensatz bereits einen Großteil der zu übertragenden Daten enthält, fallen die Differenzdokumente sehr klein aus, und somit kann das Datenaufkommen wirksam reduziert werden. Das zweite Verfahren beruht auf der Erzeugung eines Kellerautomaten aus einer XML-Schema-Beschreibung. Das zu komprimierende SOAP-Dokument wird in den konstruierten Automaten eingegeben und verarbeitet. Dabei wird der Pfad durch den Automaten binär codiert und zum Empfänger übertragen. Die erzeugte Codewortfolge ist äußerst kompakt und charakterisiert das XML-Dokument eindeutig. Mit ihrer Hilfe kann der Empfänger das Originaldokument rekonstruieren.

Weiterhin untersucht der Autor das Datenaufkommen bei der Web-Service-Kommunikation in den einzelnen Protokollschichten und stellt ein neuartiges, besonders leichtgewichtiges Anwendungsprotokoll vor. Dieses verursacht nur ein sehr geringes Datenaufkommen und arbeitet damit deutlich effizienter als das für diesen Anwendungsfall gängige HTTP.

Schwerpunkt dieser Arbeit ist also eine Optimierung bzw. Ergänzung der zur Verfügung stehenden Web-Service-Protokolle, so dass sich diese Technologie auch in den Bereichen durchsetzen kann, in denen die zur Verfügung stehende Datenrate knapp bemessen ist. Vor allem könnten die äußerst zukunftsträchtigen Einsatzgebiete Ubiquitous Computing und Sensornetze hiervon profitieren.

Abstract

During the past years the web service technology emerged into more and more fields of application. It has become a key technology for bridging heterogeneity in various areas of computer science. A major success factor of web service technology is the usage of standardized web technologies, particularly XML. However, using web services as a universal middleware approach has also a major drawback: By representing all data as text, web services cause significantly more overhead than competing older technologies, e.g. CORBA or Java RMI. Anyhow, in wired networks this is usually not so critical, because usually more than sufficient bandwidth is available. By contrast, in wireless networks the bandwidth is typically quite limited and therefore web services are not the best choice in communication scenarios with radio linked devices. In this work, the author proposes novel concepts for reducing the overhead of the web service protocols in order to make this technology also applicable to mobile applications.

The author presents two novel data compression techniques which are custom-tailored for the special demands of web service applications. The first one is based upon differential encoding. The basic idea is to calculate the difference between the SOAP document to be sent and a so called skeleton message, which is previously generated from the WSDL description of the web service. Since this skeleton message already contains large parts of the message to be sent, the difference documents are rather small resulting in significantly reduced network traffic. The second approach is based upon the construction of a pushdown automaton, which is generated from an XML Schema description. The SOAP document to be sent is processed using this automaton. The path through the automaton is encoded using compact binary identifiers. This way we can generate an extremely compact binary representation of the XML document, because the path through the automaton describes the XML document in an unambiguous way.

As a third contribution the author analyzes the protocol overhead which is induced by the protocols in different layers of the protocol stack and proposes a new application layer protocol which tends to be very lightweight. It uses network bandwidth very economically and is much more efficient than the commonly used transport over HTTP.

This work provides optimizations and endorsements to the web service protocol stack in order to make this technology applicable in environments with tightly limited network bandwidth. Ubiquitous computing and wireless sensor networks are two examples of highly relevant application domains which could benefit from web services as a powerful middleware solution.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Vernetzte Computer sind in vielen Bereichen fester Bestandteil des täglichen Lebens geworden. Ein Anwendungsprogramm, das in einem solchen Netzwerk auf mehreren Rechnern verteilt abläuft, wird als *verteilte Anwendung* bezeichnet.

Ein besonders bekanntes Beispiel hierfür ist das World-Wide-Web (WWW); der Umgang mit ihm ist für viele Menschen längst alltäglich geworden. Diese verteilte Anwendung läuft zum einen auf einer Vielzahl von Web-Servern ab, welche die WWW-Inhalte bereithalten und über das Internet miteinander verbunden sind. Zum anderen gibt es WWW-Client-Rechner. Diese sind mit einem Web-Browser ausgestattet und ermöglichen dem Benutzer so den Zugriff auf sämtliche WWW-Inhalte. Ein wesentliches Charakteristikum dieses Systems ist somit eine geeignete Benutzerschnittstelle, über die ein Benutzer direkt auf die verteilte Anwendung zugreifen kann. Weitere Beispiele für verteilte Anwendungen dieses Typs sind E-Mail, FTP und Instant-Messaging-Dienste.

Eine andere Gruppe verteilter Anwendungen arbeitet dagegen weitgehend im Verborgenen und tritt nur selten oder indirekt in Kontakt mit Benutzern. Ein typischer Vertreter dieser Gruppe ist beispielsweise ein System zur Steuerung einer industriellen Anlage. Hier geht es nicht darum, dass Informationen zwischen einem Rechnersystem und einem Menschen ausgetauscht werden, sondern um den Austausch von Betriebsparametern zwischen den einzelnen Anlagenteilen. Die Steuerungsrechner kommunizieren hier also untereinander.

Beide Gruppen stellen dabei spezifische Anforderungen an die technische Infrastruktur. Menschen können im Vergleich zu Rechnern nur wenige Informationen pro Zeiteinheit aufnehmen und verarbeiten. Daher ist die Geschwindigkeit bei der Kommunikation zwischen Rechner und Mensch meist nicht durch technische Randbedingungen begrenzt. Somit war bei der Entwicklung geeigneter Protokolle und Datenformate für diesen Anwendungsfall auch nicht die Übertragungseffizienz primär ausschlaggebend; viel wichtiger war eine einfache Handhabung, sowohl während der Softwareentwicklung als auch zur Laufzeit. Daher hat man hier von Anfang an textorientiert gearbeitet, was die Programmierung und Fehlerdiagnose besonders einfach macht. Protokolle wie FTP oder HTTP sind hier typische Vertreter. Auch die Auszeichnungssprache HTML ist ein bekanntes Beispiel für ein textorientiertes Datenformat.

Ganz andere Ansprüche stellt dagegen die Rechner-zu-Rechner-Kommunikation. Hier werden Daten mitunter sehr schnell erzeugt und verarbeitet; die Netzwerkinfrastruktur sollte hierbei nicht zu einem Engpass werden. Daher hat man Protokolle und Datenformate für solche Anwendungen vorwiegend mit dem Ziel effizienter Datenrepräsentation und hoher Verarbeitungsgeschwindigkeit entworfen und optimiert. Das Ergebnis sind hochspezialisierte Protokolle, die auf relativ weit unten liegenden Netzwerkschichten aufsetzen und möglichst optimal auf eine konkrete Anwendung zugeschnitten sind. Das Datenformat der Netzwerknachrichten orientiert sich dabei in der Regel an den rechnerinternen Datenstrukturen. So wird eine 32-Bit-Zahl auch direkt als binäre 32-Bit-Zahl gesendet und nicht, wie bei der Mensch-Rechner-Kommunikation, zunächst in Dezimalziffern umgewandelt und dann als Text übertragen.

Dieses Vorgehen hat jedoch einen entscheidenden Nachteil: In heterogenen Umgebungen, d. h. bei Vorhandensein vieler verschiedener Hardware-Plattformen, können sehr leicht Inkompatibilitäten auftreten. In Abhängigkeit von der jeweiligen Rechnerarchitektur variieren Bit-Breite und Byte-Order der einzelnen Datentypen. Dadurch ist die Entwicklung verteilter Anwendungen in heterogenen Umgebungen mitunter ein mühevolles und fehlerträchtiges Unterfangen.

1.1 Zeitliche Entwicklung von Middleware-Technologien

Als sich etwa im Jahr 1985 abzeichnete [88], dass das Internet Protocol zu einer Triebfeder für die Entwicklung verteilter Systeme werden könnte und heterogene Umgebungen dabei immer häufiger auftraten, wurde die Entwicklung einer speziellen Form von Systemsoftware, so genannter *Middleware*, stark vorangetrieben. Unter diesem Begriff verstehen wir Software, die eine Zwischenschicht zwischen dem Netzwerk und der Anwendung realisiert [53, 145]. Sie sorgt – unabhängig von der zu Grunde liegenden Rechnerarchitektur – für eine einheitliche Programmierschnittstelle, so dass sich der Anwendungsentwickler nicht mehr um diese Problematik zu kümmern braucht. Genau wie die klassische Rechner-zu-Rechner-Kommunikation über die Socket-Schnittstelle basiert auch eine Middleware im Allgemeinen auf binären Protokollen und Datenformaten. Die wesentliche Neuerung besteht darin, dass die Binärformate für unterschiedliche Programmiersprachen und Betriebssysteme vereinheitlicht werden und diese vereinheitlichte Darstellung automatisch erzeugt wird.

Erste Arbeiten in diesem Bereich wurden bereits um 1980 bei der Firma Xerox durchgeführt. 1981 wurde von Xerox *Courier* veröffentlicht, eine Software zur Realisierung von *Remote Procedure Calls* [8]. Die grundlegende Idee bei diesem Produkt bestand darin, die gesamte Netzwerkprogrammierung bei der Anwendungsentwicklung vor dem Entwickler zu verbergen. Entfernte Prozesse konnten über herkömmliche Prozeduraufrufe angesprochen werden, so dass es für den Programmierer vollständig transparent war, ob eine Prozedur lokal oder entfernt aufgerufen wurde. Courier sorgte automatisch für eine geeignete Netzwerkrepräsentation der Aufrufparameter,

die Erzeugung geeigneter Netzwerknachrichten für Anfrage und Antwort sowie für die Umsetzung der Antwortnachricht in einen Rückgabewert der Prozedur. Als technische Basis diente dabei das Client/Server-Modell: Der Prozeduraufruf wird von einem Client implementiert, dagegen wird der Code der entfernten Prozedur auf einem Server im Netzwerk ausgeführt.

Diese Idee wurde bald auch von anderen Softwareherstellern aufgegriffen und weiterentwickelt. Im Jahr 1987 veröffentlichte die Firma Sun Microsystems schließlich eine Spezifikation für eine plattformunabhängige Datenrepräsentation: *eXternal Data Representation (XDR)* [139]. Ein Jahr später stellte Sun dann ein eigenes Protokoll zur Realisierung von Remote Procedure Calls vor [140]. Ein Novum zu dieser Zeit war, dass Sun seine Spezifikationen öffentlich zugänglich machte. Wie sich herausstellen sollte, war diese Entscheidung von großem strategischem Wert. Schon bald implementierten auch andere Betriebssystemhersteller wie IBM, AT&T und Cray die Spezifikationen von Sun. Sun-RPC wurde bald zu einer sehr breit akzeptierten Lösung. Insbesondere das Network File System (NFS) [137], das auf Sun-RPC aufsetzt, war ein wichtiger Erfolgsfaktor dieser Technologie.

Eine weitere wegweisende Entwicklung, die parallel zu Sun-RPC entstand, ist das Konzept der *Interface Definition Language (IDL)*. Mit solchen Sprachen werden die Schnittstellen von RPC-Servern beschrieben. Clients, die die angebotenen Prozeduren nutzen möchten, verwenden diese Beschreibung zur automatisierten Erzeugung von Programmcode, der sämtliche Kommunikationsaufgaben mit dem Server übernimmt.

Mit der zunehmenden Verbreitung objektorientierter Programmiersprachen veränderten sich schrittweise auch die Anforderungen an Middleware-Lösungen. Im Jahre 1991 wurde von der *Object Management Group (OMG)*, einem Konsortium führender Softwarehersteller, erstmals eine Middleware vorgestellt, mit deren Hilfe nicht nur RPCs realisiert werden konnten. Mit der *Common Object Request Broker Architecture (CORBA)* ist es darüber hinaus auch möglich, in einer verteilten Umgebung objektorientiert zu arbeiten [87]. Obwohl bei CORBA die wesentlichen Konzepte, insbesondere auch das der IDL, im Kern von der RPC-Technologie übernommen wurden, hat die Komplexität von CORBA im Vergleich zur älteren RPC-Technologie stark zugenommen – und zwar sowohl in Bezug auf technische Konzepte, aber auch hinsichtlich der Benutzerschnittstellen, die CORBA einem Programmierer zur Verfügung stellt.

Im Jahr 1997 stellte Sun seine neu entwickelte Programmiersprache Java in der Version 1.1 vor. Diese bot zusätzlich zur Unterstützung für CORBA auch eine eigene Middleware-Komponente an: *Remote Method Invocation (RMI)*. Im Vergleich zu CORBA war RMI deutlich einfacher in der Handhabung. Allerdings wurde RMI ausschließlich für Java angeboten, so dass die Integration von Anwendungen, die in anderen Programmiersprachen implementiert wurden, nicht ohne weiteres möglich war. Dennoch setzte sich Java RMI in einigen Bereichen gegen CORBA durch. Zum Beispiel wird RMI auch heute noch als eine wichtige technische Basis für Java-basierte Unternehmensanwendungen (J2EE) eingesetzt.

Alle zuvor vorgestellten Middleware-Ansätze arbeiten aus Effizienzgründen mit binären Datenformaten. Allerdings waren Netzwerke und Rechner inzwischen so leistungsstark geworden, dass dieser Effizienzaspekt bei den Middleware-Lösungen zunehmend an Bedeutung verlor.

Etwa zur gleichen Zeit, ungefähr ab dem Jahr 1994, wurde in einem anderen Forschungsbereich der Informatik eine wegweisende Entwicklung vorangetrieben: die *eXtensible Markup Language (XML)*. In Anbetracht des immer größer werdenden World-Wide-Webs war abzusehen, dass die ständig steigende Informationsflut ohne allgemein verständliche Datenformate bald nicht mehr beherrschbar sein würde.

Daher griff man Ideen auf, die bereits 1969 von Charles Goldfarb, Edward Mosher und Raymond Lorie bei IBM entwickelt wurden: Ziel war damals die Schaffung einer Sprache, mit der man strukturierte Informationen jeglicher Art beschreiben kann. Goldfarb et al. entwickelten auf dieser Basis die *Generalized Markup Language (GML)* [51]. Die Hauptanwendungsgebiete waren vor allem Dokumentenmanagement sowie Textverarbeitung für die US-amerikanische Luftfahrtindustrie, das Militär und die Regierung. Goldfarb entwickelte diesen Ansatz dann weiter zur *Standard Generalized Markup Language (SGML)*, die schließlich 1986 als ISO-Standard 8879 veröffentlicht wurde. Leider ist SGML sehr komplex, so dass sich dieser Ansatz in weiten Anwendungsbereichen zunächst nicht durchsetzen konnte.

Der Durchbruch dieser Technologie begann erst rund zehn Jahre später, als man die Wichtigkeit einer generischen Datenbeschreibungssprache für das rasant wachsende WWW erkannte. Im Juli 1996 gründeten Vertreter aus Industrie und Forschung eine W3C-Arbeitsgruppe[1] mit dem Ziel, den SGML-Ansatz für das WWW nutzbar zu machen. Da man sich der Tatsache bewusst war, dass die große Komplexität von SGML ein starkes Hemmnis bei der Etablierung in der Praxis darstellt, beschränkte man den Umfang der neu zu entwerfenden Sprache auf eine Teilmenge von SGML. Diese Bemühungen resultierten in der XML-Spezifikation 1.0, die zwei Jahre später als *W3C Recommendation* [165] verabschiedet wurde.

Der Erfolg von XML war beachtlich. Durch die große Flexibilität und die vergleichsweise einfache Handhabung etablierte sich XML bald in vielen Bereichen der Informatik als universelles Datenformat.

Einer der Hauptvorteile gegenüber den bisher gängigen Dokumentformaten, wie etwa *Comma Separated Values (CSV)* oder einfachen Textdateien, besteht darin, dass Zeichensatz und -codierung im Prolog des XML-Dokuments klar festgelegt werden. Hierdurch kann der weitere Inhalt des Dokuments auf dem Zielrechner unmissverständlich interpretiert werden. Unterschiedliche Zeichensätze und -codierungen auf Quell- und Zielrechner waren vor der Etablierung von XML ein durchaus ernstzunehmendes Problem, denn in Abhängigkeit von der verwendeten Rechnerarchitektur,

[1] Das W3C ist ein Standardisierungsgremium für WWW-Technologien. Einen Überblick über die wichtigsten Standardisierungsgremien im Zusammenhang mit Web Services findet der Leser am Anfang von Kapitel 2.

dem Betriebssystem und den Ländereinstellungen auf einer Plattform waren viele unterschiedliche Einstellungskombinationen durchaus gängig. Die Überwindung dieser Plattformabhängigkeit war daher eine zentrale Zielsetzung bei der Spezifikation von XML.

Obwohl XML ursprünglich nicht als Datenformat für Middleware-Anwendungen entworfen wurde, begann fast zeitgleich mit der Standardisierung von XML eine Konvergenz beider Technologien: Im Jahr 1998 entwickelte Dave Winer in Zusammenarbeit mit Microsoft ein Protokoll namens XML-RPC [152]. Die Idee zur Zusammenführung der RPC-Technik mit XML war viel versprechend. Schließlich war XML mächtig genug, um damit strukturierte Informationen jeglicher Art codieren zu können, insbesondere also auch Datentypen und Prozeduraufrufe. Wegen seiner Plattformunabhängigkeit war XML zudem eine ideale Basistechnologie für Middleware-Anwendungen.

Microsoft entwickelte diese Idee zum *Simple Object Access Protocol (SOAP)* weiter und reichte die SOAP Protokollspezifikation in der Version 1.0 im November 1999 bei der *Internet Engineering Task Force (IETF)* zur Standardisierung ein. Im Vergleich zu XML-RPC war SOAP deutlich komplexer. Insbesondere war SOAP unabhängig von einem speziellen Transportmechanismus, XML-RPC dagegen funktionierte ausschließlich mit HTTP-Post. Weiterhin stellte SOAP besonders umfangreiche Möglichkeiten für Protokollerweiterungen bereit.

Zunächst waren die Reaktionen verhalten, doch schon bald erkannten auch andere Softwarehersteller den Wert von XML-RPC und SOAP. Im Jahr 2000 wurde die *W3C XML Protocol Working Group* mit dem Ziel ins Leben gerufen, XML-Protokolle weiterzuentwickeln und zu standardisieren. Heute zählen neben Microsoft auch Vertreter von IBM, Sun Microsystems, Nokia, BEA Systems, Oracle und SAP zu den Mitgliedern dieser Arbeitsgruppe.

Seit dem 24. Juni 2003 liegt SOAP in der Version 1.2 als *W3C Recommendation* vor [176, 177] und hat in vielen Bereichen ältere Middleware-Ansätze wie CORBA oder Java-RMI bereits verdrängt. Neben SOAP wurden inzwischen noch weitere XML-Protokolle und Datenformate spezifiziert. Zu nennen sind hier vor allem die *Web Service Description Language (WSDL)* [170], eine auf XML basierende IDL, sowie die *Universal Description, Discovery and Integration (UDDI)* Technologie [113], welche das Auffinden von Diensten und Dienstanbietern jeglicher Art ermöglicht.

Anwendungen, die auf derartigen Technologien beruhen, bezeichnet man auch als *Web Services*. Ein Web Service ist dadurch gekennzeichnet, dass er über eine URL adressiert wird, auf XML-Technologie beruht und ein- und ausgehende Nachrichten über Standardprotokolle wie etwa HTTP transportiert werden.

Web Services stehen also am Ende einer Entwicklung, die durch drei entscheidende Einflussfaktoren geprägt wurde:

- Das World Wide Web entwickelt sich von einer Informationsplattform für Menschen zu einer universellen Infrastruktur, die auch weltweite Middleware-Anwendungen mit einschließt.
- XML etabliert sich als universelle Basistechnologie.
- Durch Standardisierung, insbesondere im Rahmen der verschiedenen W3C-Arbeitsgruppen, wird die Interoperabilität zwischen Protokollen und Datenformaten verschiedener Softwarehersteller sichergestellt.

Es sei angemerkt, dass sich bis heute keine einheitliche Definition für den Begriff Web Service durchgesetzt hat. Im Rahmen dieser Arbeit verwendet der Autor den Begriff so, wie er von der W3C Web Service Architecture Working Group festgelegt wurde [181]:

Definition 1.1 (Web Service): A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

1.2 Motivation

Die durchgängige Verwendung von XML und Web-Protokollen zur Implementierung verteilter Anwendungen bringt auch einen Nachteil mit sich, der je nach Anwendungsbereich mehr oder weniger gravierend ist: Bei der XML-basierten Kommunikation ist das Verhältnis zwischen *Payload* und *Overhead* deutlich ungünstiger als bei der binären Übertragung. Der Begriff Payload kennzeichnet dabei die Datenmenge, die die anwendungsrelevanten Informationen in einer Nachricht beschreibt, also etwa Messwerte oder Berechnungsergebnisse. Overhead dagegen charakterisiert die Differenz aus gesamter Datenmenge abzüglich Payload. Overhead wird zum Beispiel durch Daten hervorgerufen, die der Adressierung oder der Datenflusssteuerung dienen.

Der hohe Overhead von XML wurde bereits in mehreren Forschungsarbeiten umfassend untersucht und quantitativ bewertet. Eine umfassende Übersicht findet sich in den Arbeiten von Tian et al. [148] und Marahrens [91].

Besonders deutlich werden die Auswirkungen der XML-Datenrepräsentation in Vergleichsmessungen mit anderen Middleware-Ansätzen. Daher hat Marahrens in [91] typische Anwendungsbeispiele mit Hilfe von verschiedenen Technologien implementiert und dann die jeweiligen Datenaufkommen miteinander verglichen. Abbildung 1.1 stellt die Ergebnisse anhand eines Beispiels zusammenfassend dar.

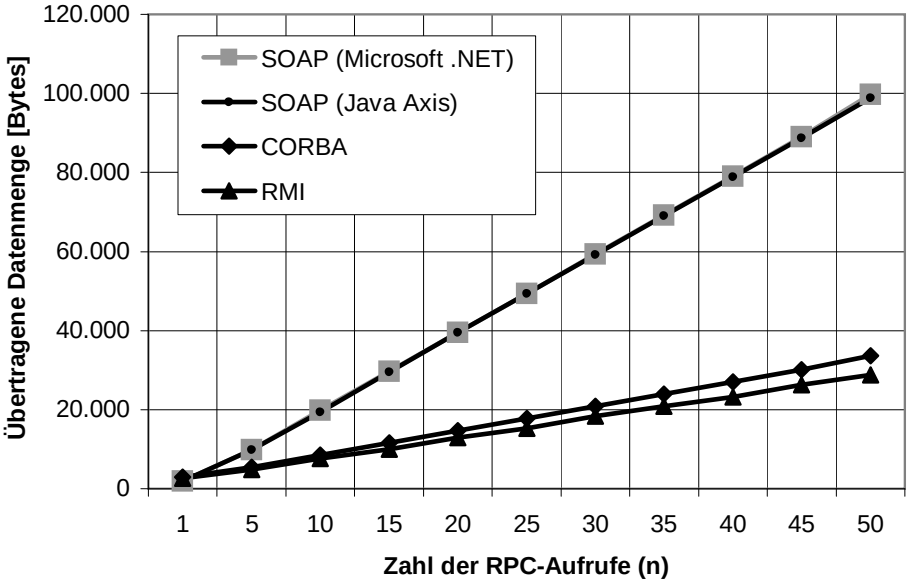


Abbildung 1.1: Datenaufkommen in Abhängigkeit von der verwendeten Middleware-Technologie nach [91]

Für die zu Grunde liegende Messreihe wurden für die verschiedenen Middleware-Technologien RPC-Server und -Clients implementiert. Die RPC-Server stellen dabei jeweils eine Operation `void sendString(String s)` bereit, die von den Client-Applikationen im Rahmen der Messung n-mal hintereinander aufgerufen wird. Der Aufrufparameter `s` ist dabei eine Zufallszeichenkette mit der konstanten Länge von 250 Zeichen. Die resultierenden Datenaufkommen sind jeweils kumuliert für n Messungen dargestellt. Bei allen Middleware-Technologien steigt das Datenaufkommen nahezu linear mit der Anzahl der RPC-Aufrufe. Weiterhin zeigen sich nur marginale Unterschiede zwischen den beiden SOAP-Implementierungen *Microsoft .NET* und *Java Axis*. Die jeweiligen Messwerte liegen unabhängig von n sehr eng zusammen.

Wie klar in der Abbildung zu erkennen ist, verursacht SOAP im Allgemeinen ein Mehrfaches an Overhead im Vergleich zu Java RMI und CORBA. Lediglich bei der Messung mit $n = 1$ arbeitet SOAP am effizientesten (die zugehörigen Werte sind in der Abbildung wegen des gewählten Maßstabs nicht mehr zu erkennen): SOAP (Java Axis) 1.972 Bytes, SOAP (Microsoft .NET) 1.976 Bytes, Java-RMI 2.626 Bytes und CORBA 2.887 Bytes. Dies liegt daran, dass die Java-RMI- und CORBA-Clients zunächst mit einem Registrierungsdienst kommunizieren müssen, um hier unter anderem die Portnummer der Server-Applikation zu erfragen. Dies verursacht bei diesen Technologien ein vergleichsweise hohes Datenaufkommen, noch bevor der erste RPC-Aufruf stattfindet. Jedoch ist der hier gegebene Fall, dass ein RPC-Client nur ein einziges Mal mit einem RPC-Server kommuniziert, für die allermeisten Anwendungen eher untypisch.

Zusammenfassend lässt sich feststellen, dass SOAP im Vergleich zu Java-RMI und CORBA bei dem dargestellten Anwendungsfall ein etwa dreimal höheres Datenaufkommen verursacht.

Wie bereits oben angedeutet, führt diese größere Datenmenge in vielen Anwendungsbereichen zu keinen nennenswerten Nachteilen. In Unternehmensnetzwerken – und diese sind heute zweifelsohne ein Hauptanwendungsbereich für XML-Protokolle wie SOAP – ist die Infrastruktur in aller Regel ausreichend dimensioniert, so dass der zusätzlich verursachte Overhead hier kaum ins Gewicht fällt.

Bei drahtlos vernetzten Geräten ist indessen eine effiziente Nutzung der zur Verfügung stehenden Datenrate[2] oftmals ein wichtiges Kriterium. Durch die physikalischen Eigenschaften der Luftschnittstelle arbeiten drahtlose Kommunikationstechniken in aller Regel langsamer als drahtgebundene. Kommunizieren viele Geräte im selben Frequenz- und Kommunikationsbereich, kommt hinzu, dass sich diese Geräte die ohnehin knappe Datenrate teilen müssen. Ein weiterer Aspekt sind die entstehenden Kosten: Kommuniziert ein Rechner beispielsweise über ein Mobilfunknetzwerk, erfolgt die Abrechnung zumeist anhand der übertragenen Datenmenge. Daher ist es bei drahtlos vernetzten Geräten in vielen Fällen sinnvoll, die eingesetzten Protokolle im Hinblick auf eine effiziente Nutzung der zur Verfügung stehenden Datenrate zu optimieren.

Ebenso negativ wirkt sich das hohe Datenaufkommen von XML-Protokollen im Bereich der Hochleistungsdatenverarbeitung aus. Etwa bei Grid-Anwendungen, die die verteilte Berechnung komplexer algorithmischer Aufgaben zum Ziel haben, kommt es in hohem Maße auf die effiziente Nutzung der zur Verfügung stehenden Netzwerkdatenrate an [26]. In aller Regel bildet die Netzwerkinfrastruktur hier einen Engpass bei der Koppelung einzelner Knoten im Grid, so dass eine Steigerung der Effizienz bei der Kommunikation auch direkt zu einer Erhöhung der Leistungsfähigkeit des Gesamtsystems führt. Obwohl XML-Protokolle erhebliche Vorteile bieten, setzt man bei Computing-Grids aus Effizienzgründen teilweise noch immer proprietäre Lösungen zur Koppelung der Grid-Knoten ein [138]. Durch eine effizientere Codierung der XML-Nachrichten könnten auch in diesem Anwendungsbereich die Vorteile der XML-Kommunikation voll zum Tragen kommen.

Jüngste Entwicklungen zeigen noch ein neues Anwendungsgebiet für XML-Protokolle auf: mobile Kleinstcomputer. Diese Geräte sind dadurch gekennzeichnet, dass sie nur über einen sehr begrenzten Hauptspeicher verfügen. Die Stromversorgung erfolgt über Batterien oder Akkus, und somit ist der Energievorrat der Geräte ebenfalls vergleichsweise gering.

[2] Der Begriff *Datenrate* bezeichnet das Datenvolumen pro Zeiteinheit, gemessen in Bytes pro Sekunde. Alternativ ist in der Informatik auch der Begriff *Bandbreite* gebräuchlich. Dieser wird vom Autor allerdings nicht verwendet, da er in der Physik und Nachrichtentechnik bereits mit einer anderen Bedeutung belegt ist (Breite eines Frequenzbandes).

Ein modernes Einsatzgebiet solcher Geräte sind Sensornetzwerke [2]: Eine große Anzahl mobiler Kleinstcomputer, welche mit verschiedenen Sensoren ausgestattet sind, werden so programmiert, dass sie bestimmte Kontroll- oder Überwachungsaufgaben wahrnehmen. Diese so genannten Sensorknoten werden dann in dem Gebiet ausgebracht, das überwacht werden soll. Mit Hilfe ihrer Sensoren messen sie bestimmte Umgebungsparameter wie Temperatur, Erschütterungen oder den Gehalt von Schadstoffen. Die Geräte kommunizieren dabei über Funk und tauschen so Informationen über die gemessenen Werte aus. Charakteristisch für Sensornetzwerke ist dabei, dass im Rahmen dieses Kommunikationsprozesses niederwertige Informationen zu höherwertigen aggregiert werden. Beispielsweise könnten einzelne hohe Temperaturwerte, die von Sensoren in einem begrenzten Bereich gemessen werden, zu der Information „ein Feuer ist ausgebrochen“ aggregiert werden.

Dieses Vorgehen bietet gegenüber der Messung mit einzelnen, nicht vernetzten Sensoren deutliche Vorteile: Ein Sensornetzwerk wird dadurch tolerant gegenüber einzelnen fehlerhaften Messungen sowie gegenüber Ausfall oder Beschädigung einzelner Messeinrichtungen.

Geeignete Verfahren zum Austausch von Informationen zwischen den Sensorknoten sowie zur Aggregation dieser Einzelinformationen sind seit langem ein aktives Forschungsgebiet. Herkömmliche Kommunikationsparadigmen wie „Client/Server“ sind für den Einsatz in Sensornetzwerken wegen einer fehlenden permanenten Konnektivität zwischen einzelnen Knoten nämlich nur bedingt geeignet. Einen ausführlichen Überblick über diese Problemstellung und mögliche Lösungsansätze findet der Leser in [188].

Ein neuerer Ansatz zur Organisation von Informationen in einem Sensornetzwerk wird von Koberstein et al. in [79] beschrieben: Sämtliche Informationen werden in einem *virtual distributed Shared Information Space (dvSIS)* organisiert und gespeichert. Der dvSIS ist eine hierarchische Datenstruktur, die in jedem einzelnen Sensorknoten als Kopie gespeichert wird. Allerdings ist keine dieser Kopien vollständig, d. h. jeder Knoten speichert nur die für seine aktuellen Aufgaben erforderliche Datenmenge. Neue Informationen werden dabei über Broadcast-Nachrichten an benachbarte Knoten weitergegeben und bei Bedarf dort gespeichert.

Der gesamte Informationsbestand wird also verteilt im Sensornetzwerk gehalten. Zur Implementierung des dvSIS schlagen Koberstein et al. die Verwendung von XML vor, da hiermit die hierarchische Struktur des Datenbestandes optimal abgebildet wird. Weiterhin kann durch den Einsatz von XML-Schema-Dokumenten eine automatisierte Typüberprüfung stattfinden. Zudem kommen auch in diesem Einsatzgebiet dieselben Vorteile zum Tragen wie bei XML-basierten Middleware-Lösungen. Sollen zum Beispiel die Daten aus dem Sensornetz zu anderen Systemen – etwa zu einer über das Internet angeschlossenen Überwachungsstation – übertragen werden, so kann durch den Einsatz von XML die Heterogenität der beteiligten Systeme wirkungsvoll überwunden werden.

Koberstein et al. weisen aber auch auf Nachteile hin, die durch den Einsatz von XML in diesem Anwendungsgebiet entstehen. Das Parsen von XML auf solchen Kleinstgeräten lässt sich wegen der hier systemimmanenten Knappheit der Hauptspeicherkapazität nicht mit herkömmlichen Ansätzen wie SAX oder DOM bewerkstelligen. Folglich müssen hier alternative Strategien angewendet werden. Die Autoren schlagen zur Lösung dieser Problemstellung das in [79] beschriebene Anwendungsentwicklungskonzept CCASTAX vor. Hiermit können äußerst kompakte Softwarekomponenten zur Verarbeitung von XML-Daten erzeugt werden. Der Grundgedanke bei diesem Ansatz besteht darin, keine universellen Funktionen zur Verarbeitung von XML bereitzustellen, sondern spezialisierte, die auf einen bestimmten Anwendungsfall hin optimiert sind.

Neben den Einschränkungen, die sich durch den sehr begrenzten Hauptspeicher solcher Sensorknoten ergeben, ist die Übertragung von XML-Nachrichten mit Sensorknoten auch durch einen zweiten entscheidenden Faktor limitiert. Der Energiebedarf dieser Geräte hängt in hohem Maße von der Einschaltdauer und der Nutzung der Funkschnittstelle ab [124, 97]. Folglich führt die Übertragung von XML-Nachrichten, die ja – wie bereits erläutert – durch ein vergleichsweise hohes Datenaufkommen gekennzeichnet ist, zu einem erhöhten Energiebedarf und damit auch zu einer verkürzten Betriebsdauer der batteriebetriebenen Geräte. Dieses Problem wird durch CCASTAX nicht gelöst.

1.3 Zielsetzung und Aufbau dieser Arbeit

XML hat sich in den letzten Jahren in vielen Bereichen der Informatik als universelles Mittel zur Beschreibung strukturierter Daten durchgesetzt. Damit wird XML zu einer idealen Basis für die Implementierung von Middleware-Anwendungen in heterogenen Umgebungen. Im Bereich der Hochleistungsdatenverarbeitung und bei Anwendungen mit drahtlos vernetzten Systemen kommt jedoch ein entscheidender Nachteil von XML zum Tragen: das deutlich höhere Datenaufkommen im Vergleich zu alternativen Middleware-Technologien.

Ziel dieser Arbeit ist daher die Entwicklung wirksamer Strategien zur Reduzierung des Datenvolumens von XML-Nachrichten. Der Autor stellt zum einen zwei neuartige Codierungsverfahren für XML vor, die durch den Einsatz von Datenkompressionstechniken die Größe der XML-Daten deutlich verringern. Zum anderen wird ein alternatives Übertragungsprotokoll entwickelt, das im Vergleich zu den heute üblichen Transportmechanismen einen Großteil des Overheads vermeidet.

Da SOAP in der Praxis eine besonders wichtige Rolle spielt, wird diese moderne Ausprägung eines XML-Protokolls als Ausgangspunkt für alle weiteren Überlegungen gewählt. Die Kernideen der im Folgenden diskutierten Ansätze sind jedoch grundsätzlich auch auf andere XML-Protokolle anwendbar.

In Kapitel 2 stellt der Autor zunächst die grundlegenden Technologien vor, welche die Basis für heutige Web-Service-Anwendungen bilden. Diese Technologien werden in ein Schichtenmodell, den so genannten Web Service Technology Stack, eingeordnet und anhand einiger Beispiele erläutert. Eine Analyse, an welchen Stellen dieses Schichtenmodells Ansätze zur Effizienzsteigerung implementiert werden können, schließt dieses Kapitel ab.

In Kapitel 3 werden dann die grundlegenden Konzepte der Datenkompression beschrieben. In den einzelnen Unterabschnitten geht es um elementare Begriffe der Informationstheorie und um eine Klassifizierung der in der Fachliteratur vorgestellten Algorithmen.

In Kapitel 4 wird ein neuartiges Verfahren zur Kompression von SOAP-Nachrichten präsentiert, das auf dem Konzept der Differenzcodierung beruht. Dieser Ansatz nutzt die Tatsache aus, dass die meisten Ein- und Ausgabenachrichten eines Web-Services eine sehr ähnliche Struktur aufweisen und nur ein vergleichsweise kleiner Anteil der transportierten Daten anwendungsrelevante Informationen beschreibt. Diese Beobachtung lässt sich bei der Codierung von SOAP Nachrichten vorteilhaft ausnutzen und führt zu einer deutlichen Reduktion des Datenvolumens.

In Kapitel 5 geht es um ein weiteres, alternatives Datenkompressionsverfahren für XML-Nachrichten, das jedoch nicht auf dem Ansatz einer Differenzcodierung basiert. Stattdessen werden die Strukturinformationen eines Datensatzes aus einer Grammatikbeschreibung, zum Beispiel in Form einer DTD oder eines XML-Schema-Dokuments, extrahiert. Eine solche Grammatikbeschreibung ist bei typischen Web-Service-Anwendungen sowohl dem Sender einer Nachricht als auch dem Empfänger bekannt, und folglich müssen Informationen, die aus dieser Grammatikbeschreibung algorithmisch zurückgewonnen werden können, nicht in den XML-Nachrichten mit codiert werden. Auf dieser Grundlage kann das Datenvolumen der Netzwerknachrichten ebenfalls deutlich reduziert werden.

Zu Beginn von Kapitel 6 analysiert der Autor das Gesamtdatenaufkommen bei der SOAP-Kommunikation. Hierbei wird insbesondere der Anteil des Overheads untersucht, der durch die Protokolle zum Nachrichtentransport verursacht wird. Die vorgestellten Ergebnisse zeigen deutlich, dass auch geeignete Anwendungsprotokolle wirksam dazu beitragen können, das Datenaufkommen zu senken. Im weiteren Verlauf von Kapitel 6 stellt der Autor daher ein von ihm neuentwickeltes, besonders leichtgewichtiges Anwendungsprotokoll zum Transport von SOAP-Nachrichten vor. Abschließende Messungen zeigen, dass damit – in Verbindung mit geeigneten Datenkompressionstechniken – die Web-Service-Technologie auch für Anwendungen mit begrenzten Datenraten nutzbar wird.

Die Kapitel 4, 5 und 6 behandeln somit jeweils komplementäre Strategien zur Effizienzsteigerung und bilden damit den Kern dieser Arbeit. Um den inhaltlichen Bezug zu verwandten Arbeiten optimal darzustellen, werden diese nicht an zentraler Stelle, sondern in separaten Unterabschnitten der jeweiligen Kapitel behandelt.

In Kapitel 7 fasst der Autor die Ergebnisse dieser Arbeit schließlich zusammen und zeigt mögliche Ansatzpunkte für weitere wissenschaftliche Untersuchungen auf diesem Themengebiet.

Da XML im Rahmen dieser gesamten Arbeit eine zentrale Rolle einnimmt, ist es unabdingbar, dass der Leser mit den gängigsten XML-Begriffen vertraut ist. Die Begriffe werden durchgängig so verwendet, wie sie in den entsprechenden W3C-Dokumenten [165, 167, 178, 179] definiert sind. Die wichtigsten XML-Grundlagen und -Konzepte hat der Autor in den Anhängen A und B für den Leser zusammengefasst.

Kapitel 2

Grundlagen von Web Services

Wie bereits in Kapitel 1 erläutert, hat sich die Web-Service-Technologie im Wesentlichen aus zwei Basistechnologien heraus entwickelt: dem WWW und XML (bzw. der Vorgängersprache SGML). Daher ist es auch nicht weiter verwunderlich, dass die Standardisierung von Web-Service-Protokollen und -Datenformaten primär in den Gremien stattfindet, die sich mit diesen Basistechnologien beschäftigen – nämlich dem *World Wide Web Consortium (W3C)* und der *Organization for the Advancement of Structured Information Standards (OASIS Open)*.

Das W3C wurde im Jahre 1994 von Tim Berners-Lee, dem Erfinder des WWW, gegründet und beschäftigt sich seitdem mit der Standardisierung von Web-Technologien. Neben der *Hypertext Markup Language (HTML)* und verwandten Technologien wie *Cascading Stylesheets (CSS)* hat das W3C inzwischen mehr als 90 Web-Standards verabschiedet, so genannte *W3C Recommendations*. Dazu gehören auch solche, die für Web Services von zentraler Bedeutung sind – zu nennen sind hier vor allem XML, SOAP und WSDL. Ein Großteil der Arbeit des W3C wird in Arbeitsgruppen (*Working Groups*) durchgeführt, die sich jeweils auf einen inhaltlichen Teilbereich konzentrieren.

Die OASIS Open wurde bereits 1993 gegründet und beschäftigte sich damals vor allem mit der Weiterentwicklung von SGML für Unternehmensanwendungen; bis 1998 hieß diese Organisation auch noch *SGML Open*. Mit der Entstehung von XML hat SGML jedoch immer mehr an Einfluss verloren, und XML wurde auch für betriebswirtschaftliche Anwendungen immer wichtiger. Daher beschäftigt sich die OASIS seit etwa 1998 nicht mehr ausschließlich mit SGML, sondern zusätzlich auch mit XML im Unternehmensumfeld. Dabei geht es insbesondere um die Nutzung dieser Sprachen für die Beschreibung und die technische Abwicklung von Geschäftsprozessen. Im Bereich Web Services ist vor allem der *Universal Description, Discovery and Integration (UDDI)* Standard der OASIS zu nennen, der neben SOAP und WSDL zum Kern der Web-Service-Technologie gehört. UDDI ist ein universeller Ansatz für einen Verzeichnisdienst, der unter anderem das Auffinden von Web Services ermöglicht.

Neben dem W3C und der OASIS gibt es noch eine weitere wichtige Organisation, die sich zwar nicht auf Web-Service-Standards im engeren Sinne konzentriert, sondern hauptsächlich die Technologien standardisiert, auf denen Web Services auf-

setzen: Die *Internet Engineering Task Force (IETF)* entwickelt allgemeine Internet-Standards, hierzu gehören grundlegende Protokolle wie IP, TCP oder HTTP. Jedes IETF-Dokument wird dabei als ein so genannter *Request For Comments (RFC)* veröffentlicht. Es gibt zwei verschiedene Typen von RFCs – solche, die einen verbindlichen Standard beschreiben (Standard Track), und solche, die keinen verbindlichen Charakter haben (Non-Standard Track) [11].

Alle drei Gremien streben *offene* Standards an. Die durchgängige Verwendung solcher offenen Standards ist ein ganz wesentlicher Grundgedanke der Web-Service-Technologie. „Offen“ bedeutet in diesem Zusammenhang, dass sämtliche Spezifikationen öffentlich zugänglich sind und die dort beschriebenen Protokolle und Datenformate keinen besonderen Nutzungseinschränkungen unterliegen. Die Standardisierung von Techniken, deren Nutzung durch Patente oder andere Schutzrechte beschränkt ist, soll daher nur unter besonderen Randbedingungen erfolgen [12, 180, 107]. Ein weiterer wesentlicher Aspekt eines offenen Standards ist die Möglichkeit zur Mitwirkung am Standardisierungsprozess. Es steht prinzipiell jedem Unternehmen und – mit einigen Einschränkungen – auch jeder Privatperson offen, sich an den Standardisierungsprozessen in den oben genannten Gremien zu beteiligen [11, 184, 106].

Die Web-Service-Technologie fußt somit auf solch offenen Standards. In diesem Kapitel stellt der Autor zunächst ein Schichtenmodell vor, den *Web Service Technology Stack*. Dieses Modell beschreibt die grundlegende Architektur eines Web Services und stellt die Beziehungen zwischen den beteiligten Standards zueinander dar. In den dann folgenden Abschnitten werden die wichtigsten dieser Technologien weiter erläutert: SOAP, WSDL und UDDI. Diese Textteile dienen in erster Linie dazu, die grundlegenden Funktionsweisen im Überblick darzustellen, so dass sich dem Leser der Inhalt der danach folgenden Kapitel leichter erschließt.

Im letzten Teilabschnitt stellt der Autor das *Web-Service-Rollenmodell* vor. Es beschreibt das zeitliche Zusammenspiel zwischen den einzelnen Web-Service-Komponenten. Anhand dieses Rollenmodells diskutiert der Autor auch die Frage, welche Ansatzpunkte zur Reduzierung des Datenaufkommens bei der Web-Service-Kommunikation viel versprechend sind.

Die in diesem Kapitel dargelegten Ausführungen hat der Autor bereits in [162] in wesentlichen Teilen vorab veröffentlicht.

2.1 Web Service Technology Stack

Die *W3C Web Service Architecture Working Group* beschäftigt sich mit grundlegenden Konzepten zur Architektur von Web Services. Zentraler Bestandteil ist dabei der in Abbildung 2.1 dargestellte *Web Service Technology Stack*. Er beschreibt auf einem abstrakten Niveau, welche Komponenten bei der Implementierung und Nutzung eines Web Services relevant sind. Abstrakt deshalb, weil diese Spezifikation noch keine

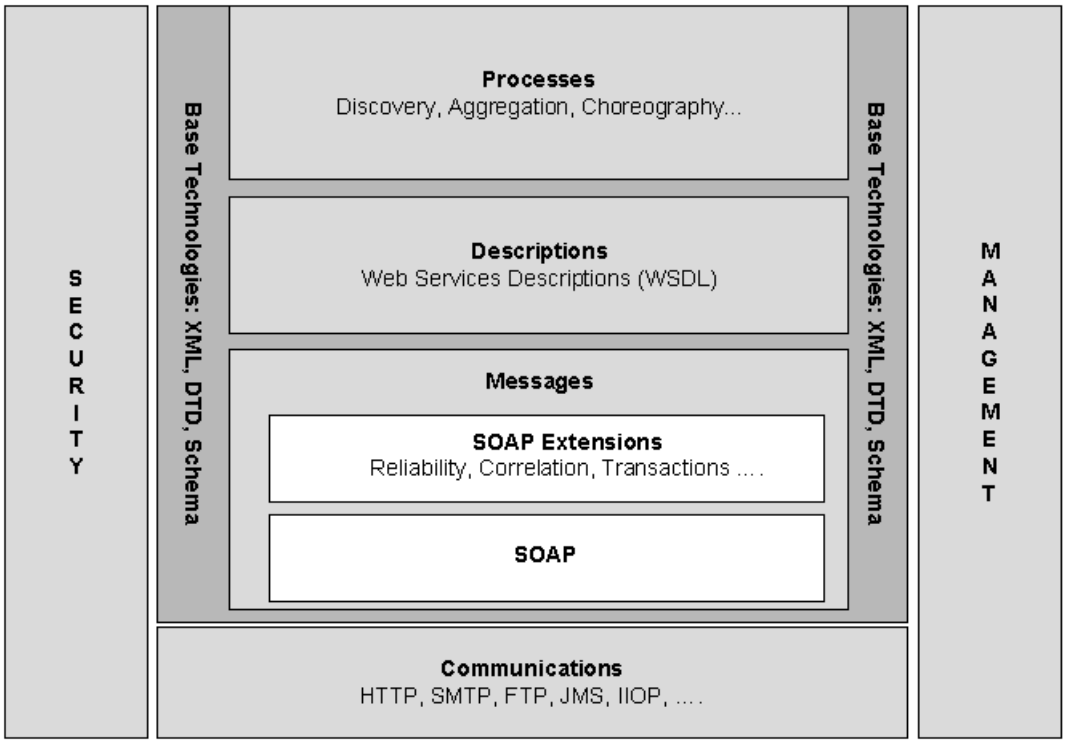


Abbildung 2.1: Der Web Service Technology Stack (entnommen aus [181])

technischen Details für die Umsetzung – etwa in Form konkreter Protokolle oder Datenformate – vorschreibt.

Die genaue technische Ausgestaltung wird in anderen W3C-Arbeitsgruppen festgelegt, zu nennen sind hier vor allem die *XML Protocol Working Group* und die *Web Services Description Working Group*. Sie haben technische Standards wie SOAP oder WSDL erarbeitet, die im Web Service Technology Stack jeweils eine Schicht ausfüllen.

Durch die strikte Trennung zwischen abstrakter Architektur und konkreter technischer Umsetzung bietet die Web-Service-Technologie eine stark ausgeprägte Modularität. Denn zu einer Schicht im Web Service Technology Stack kann es durchaus mehrere Möglichkeiten für eine konkrete technische Umsetzung geben. Der Web-Service-Entwickler hat damit die Freiheit, die in den einzelnen Schichten eingesetzten Protokolle und Datenformate den speziellen Bedürfnissen eines konkreten Anwendungsfalls anzupassen.

Es sei angemerkt, dass diese Modularität auch zu Inkompatibilitäten zwischen Web-Service-Implementierungen führen kann: Verwenden zwei Web Services inkompatible Technologien in einzelnen Schichten, so ist eine Kommunikation zwischen beiden zunächst nicht möglich. Ein modularer Aufbau bietet gleichwohl den klaren Vorteil, dass im Falle einer solchen Inkompatibilität eine der beiden Web-Service-

Implementierungen auf einfache Weise um die fehlende Technologie erweitert werden kann. Ein Web Service kann also durchaus mehrere Technologien auf den einzelnen Schichten anbieten, um so mit möglichst vielen Web-Service-Typen kompatibel zu sein.

Betrachten wir nun den Aufbau des Web Service Technology Stacks im Einzelnen: Die Schicht *Communications* bildet die Basis dieses Protokollstapels, sie repräsentiert den Nachrichtentransport. Eine grundlegende Idee besteht hier darin, ausgereifte und bereits standardisierte Internetprotokolle, wie etwa HTTP [44], SMTP [77] oder FTP [123], einzusetzen. Allerdings ist die Verwendung solcher Standardprotokolle nicht verbindlich festgeschrieben. Es werden keine besonderen Eigenschaften oder Fähigkeiten vorausgesetzt – prinzipiell eignet sich also jeder Mechanismus, mit dessen Hilfe sich Nachrichten zu einem spezifizierten Empfänger übertragen lassen [181]. Wie in Abbildung 2.1 dargestellt, eignen sich sogar hochspezialisierte Lösungen zur Nachrichtenübertragung, wie etwa das *Internet Inter-Orb Protocol (IIOP)* [114], das ursprünglich für den Transport von CORBA-Nachrichten entwickelt wurde, oder auch *Java Message Service (JMS)* [141], ein System zum asynchronen Austausch von Nachrichten für komplexe Geschäftsanwendungen. Der Einsatz solcher speziellen Transportmechanismen ist immer dann sinnvoll, wenn die benötigte Infrastruktur, beispielsweise in Form von JMS-Servern, bereits in einem Unternehmen vorhanden ist, oder wenn der Anwendungsfall ganz besondere Anforderungen an den verwendeten Transportmechanismus stellt. Der Autor hat ein solches Spezialprotokoll entwickelt. Es ist auf die besonderen Bedürfnisse von mobilen Web Services mit begrenzten Datenraten zugeschnitten und wird in Kapitel 6 vorgestellt.

Die Schicht oberhalb von *Communications* ist mit *Messages* bezeichnet und bildet der Kern der Web-Service-Technologie. Die hier angesiedelten Funktionalitäten sorgen für eine betriebssystem- und programmiersprachenunabhängige Darstellung der zu übertragenden Netzwerknachrichten. Obwohl auch die in dieser Schicht eingesetzten Technologien prinzipiell modular austauschbar sind, nimmt das Protokoll *SOAP* eine Sonderstellung ein. Die W3C Web Service Architecture Working Group stellt in [181] klar dar, dass es zwar auch möglich sei, Web Services ausschließlich auf Basis von benutzerdefinierten XML-Sprachen – also ohne SOAP – zu implementieren, allerdings ließen sich dann die in [181] beschriebenen Architekturüberlegungen nicht ohne Einschränkungen umsetzen. SOAP bildet somit einen festen Rahmen für die Umsetzung von Web Services im Sinne dieser W3C-Arbeitsgruppe.

Wie wir im folgenden Abschnitt noch genauer betrachten werden, war Erweiterbarkeit ein zentraler Aspekt bei der Spezifikation von SOAP. Im Web Service Technology Stack ist SOAP daher auch in zwei Teilschichten unterteilt, diese sind in Abbildung 2.1 weiß dargestellt. Die untere von beiden – bezeichnet mit *SOAP* – beinhaltet die in der SOAP-Spezifikation [176, 177] festgeschriebenen Grundfunktionalitäten. Hierzu gehören etwa Regeln für die Repräsentation von Datentypen oder auch Mechanismen zur Fehlerbehandlung. Die obere Teilschicht *SOAP Extensions* repräsentiert dagegen optionale Komponenten, welche über Erweiterungsmechanismen in das SOAP-

Protokoll integriert werden können. Typische Beispiele sind hier *WS-Reliability* [109], ein Mechanismus zur Realisierung von zuverlässiger Nachrichtenübertragung, oder auch *WS-AtomicTransactions* [19], eine Erweiterung zur Umsetzung von Transaktionen.

Oberhalb von *Messages* liegt die Schicht *Descriptions*. Sie beinhaltet Technologien zur Beschreibung von Web Services. Das Vorhandensein einer möglichst exakten Dienstbeschreibung ist von zentraler Bedeutung, denn ohne sie wüsste ein Dienstaufrufer nicht, wie gültige Anfragen an einen Web Service und dessen zurückgelieferte Antworten syntaktisch aufgebaut sind. Neben Angaben über die vom Dienst zur Verfügung gestellten Operationen, den unterstützten Datentypen und Nachrichtenformaten enthält diese Dienstbeschreibung auch Angaben über die Adresse, unter der ein Web Service erreichbar ist. Sie enthält somit sämtliche Angaben, die ein Dienstnutzer benötigt, um einen Web Service erfolgreich aufrufen zu können. Die vom W3C standardmäßig vorgesehene Sprache zur Dienstbeschreibung ist die *Web Service Description Language (WSDL)* [170].

Die oberste Schicht im Web Service Technology Stack heißt *Processes*. Sie repräsentiert sämtliche Prozesse, die beim Umgang mit Web Services eine Rolle spielen. Besonders relevant ist hier der Prozess des Auffindens (Discovery) eines Web Services – schließlich muss ein potentieller Dienstnutzer zunächst einen geeigneten Web Service finden, der zur vorliegenden Problemstellung passt. Der technische Lösungsansatz für diesen Vorgang heißt *Universal Description, Discovery and Integration (UDDI)* [113]. Mit Hilfe dieser Technologie lassen sich Dienstregister (Registries) implementieren, bei denen sich ein Dienstanbieter zusammen mit allen von ihm angebotenen Diensten registrieren kann. Ein potentieller Dienstnutzer fragt diese Register über eine API ab und kann so geeignete Dienste ausfindig machen.

Neben dem Auffinden eines Dienstes ist auch das Integrieren und Zusammenfassen von mehreren Web Services zu einer funktionalen Einheit ein wichtiger Prozess. Diese Vorgänge bezeichnet man als *Service Aggregation* oder auch *Service Choreography*. Eine genaue Abgrenzung dieser Begriffe sowie weitere Erläuterungen hierzu findet der Leser in [73].

Wie bereits in Kapitel 1 erläutert, bildet die Sprache XML den technologischen Rahmen für alle zentralen Web-Service-Komponenten. In Abbildung 2.1 ist dies durch einen dunkel abgesetzten Bereich dargestellt, der die Schichten *Messages*, *Descriptions* und *Processes* umschließt. Neben XML selbst sind auch XML-Grammatikbeschreibungssprachen – hier insbesondere *XML Schema* [178, 179] – von zentraler Bedeutung für Web Services. Da XML Schema ein umfangreiches System zur Beschreibung von Datentypen bereitstellt, bot es sich an, dieses auch als Grundlage für sämtliche Datentypen im Web Service Technology Stack zu verwenden. Die ältere XML-Grammatikbeschreibungssprache *Document Type Definition (DTD)* [165] ist zwar in Abbildung 2.1 noch mit aufgeführt, in aktuellen Web-Service-Spezifikationen spielt sie aber so gut wie keine Rolle mehr.

Ergänzend zu den Funktionalitäten, die in den horizontalen Schichten des Web Service Technology Stack eingegliedert sind, sieht dieses Modell auch zwei Bereiche vor, die für alle Schichten relevant sind. Diese sind in Form von zwei Säulen im äußersten linken und rechten Bereich von Abbildung 2.1 dargestellt: *Security* und *Management*.

Der Bereich Sicherheit war von Anfang an ein erklärtes Entwicklungsziel bei Web Services. Technologisch wird er vor allem durch den Standard *WS-Security* [103] abgedeckt. Dieser basiert seinerseits auf den beiden W3C-Standards *XML Encryption* [172] und *XML Signature* [173]. Diese legen fest, wie Verschlüsselungstechniken auf XML-Dokumente (oder Teile davon) anzuwenden sind, so dass sich der resultierende Datensatz wiederum in XML darstellen lässt. Auf diese Weise lassen sich verschlüsselte und signierte XML-Dokumente mit Hilfe normaler XML-Parser verarbeiten, was die Implementierung von sicheren Web Services enorm erleichtert.

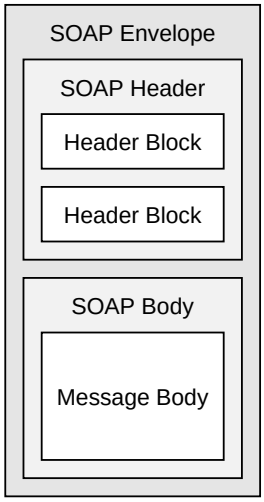
Web-Service-Management ist im Sinne von Netzwerkmanagement zu verstehen. Hierbei geht es also darum, die Funktionen einzelner Web Services permanent zu überwachen, um so schnell auf den Ausfall eines Dienstes oder etwaige Leistungsengpässe reagieren zu können. Zwei umfassende Spezifikationen zu diesem Themenbereich wurden kürzlich als OASIS-Standards verabschiedet. Die Spezifikation *Management of Web Services (MOWS)* [110] beschreibt Abläufe und Nachrichtenformate zum Überwachen des Betriebszustandes eines Web Services. Dabei ist zu beachten, dass diese Überwachungsfunktionalität wiederum über Web-Service-Schnittstellen bereitgestellt wird. Die Spezifikation *Management Using Web Services (MUWS)* [111, 112] beschreibt eine solche Web-Service-Schnittstelle für Managementaufgaben. MUWS ist aber nicht auf die Überwachung von Web Services beschränkt, sondern eignet sich auch zur Überwachung von beliebigen Geräten im Netzwerk – neben der Überwachung von anderen Web Services im Sinne von MOWS wären also typische Anwendungsbereiche Router oder Switches.

Nachdem der Autor in diesem Abschnitt die einzelnen Komponenten des Web Service Technology Stacks vorgestellt hat, wird er in den nächsten drei Abschnitten die wichtigsten Technologien zur Umsetzung der Kernkomponenten näher erläutern.

2.2 SOAP

Der Name SOAP war ursprünglich ein Akronym für *Simple Object Access Protocol*. Allerdings hat man diese Bedeutung in der aktuellen Spezifikation [176, 177] nicht mehr festgeschrieben – SOAP ist damit nur noch ein Eigenname für ein XML Protokoll. Allerdings wird die Langform *Simple Object Access Protocol* nach wie vor in der Web-Service-Literatur verwendet. Mitunter interpretiert man den Namen heute auch als *Service Oriented Access Protocol*, beides ist im aktuellen Standard jedoch eigentlich nicht vorgesehen.

Der Grund für diese neue Ausrichtung der Namensgebung liegt vermutlich darin, dass das ursprüngliche Akronym schlichtweg unpassend war. Zum einen ist SOAP, vor al-




```
<?xml version="1.0" encoding="UTF-8"?>
 <env:Envelope xmlns:env="http://
   www.w3.org/2003/05/soap-envelope">
  <env:Header>
  ...
  </end:Header>
  <env:Body>
  ...
  </env:Body>
</env:Envelope>
```

Abbildung 2.2: Struktur einer SOAP-Nachricht, schematisch (links) und in XML-Darstellung (rechts)

lem im Vergleich zur älteren XML-RPC-Technologie [152], recht komplex, so dass das Attribut *simple* alles andere als zutreffend ist. Weiterhin ist der Funktionsumfang auch nicht speziell auf die Erfordernisse zum Zugriff auf Objekte im Sinne der objektorientierten Programmierung ausgerichtet. Es ist damit eigentlich kein (*Object Access*) Protokoll, sondern dient, wie bereits in Kapitel 1 erläutert, zum plattformunabhängigen Austausch hierarchisch strukturierter Daten jeglicher Art.

Zurzeit liegt SOAP in der Version 1.2 vor, welche im Juni 2003 in Form einer *W3C Recommendation* standardisiert wurde. Die Spezifikation ist in zwei Teile gegliedert: *SOAP Version 1.2 Part 1: Messaging Framework* und *SOAP Version 1.2 Part 2: Adjuncts*. Der erste Teil beschreibt auf einem abstrakten Niveau, wie eine SOAP Nachricht aufgebaut ist und wie sich das Protokoll an einen Mechanismus zum Nachrichtentransport binden lässt. Eine solche Bindung bezeichnet man auch als *Transport Binding* oder kurz *Binding*. Der zweite Teil geht dann auf konkrete technische Realisierungsvarianten ein. Hier wird vor allem festgelegt, wie einfache und zusammengesetzte Datentypen in SOAP repräsentiert werden können und wie sich mit Hilfe von SOAP RPC-Aufrufe realisieren lassen. Weiterhin beschreibt dieser zweite Teil auch ein *SOAP-über-HTTP-Binding*; dieses erläutert, wie man SOAP-Nachrichten mit Hilfe der Kommandos HTTP-GET und HTTP-POST transportiert (für weitere Erläuterungen zu den HTTP-Kommandos siehe [44]).

2.2.1 Nachrichtenformat

Das grundlegende Format einer SOAP-Nachricht ist in Abbildung 2.2 dargestellt. Die Funktionen der hier dargestellten Nachrichtenteile lassen sich zur Veranschauli-

chung mit denen eines herkömmlichen Post-Briefs vergleichen: Der *SOAP Envelope* entspricht dem Umschlag des Briefs, er umfasst die einzelnen Nachrichtenteile und schließt sie zu einer Einheit zusammen. Der *SOAP Header* entspricht der Umschlagbeschriftung und enthält Informationen, die für die am Nachrichtentransport beteiligten Systeme bestimmt sind – beispielsweise Adressierungs- oder Priorisierungsinformationen (in SOAP sind solche Angaben optional). Der *SOAP Body* entspricht dem eigentlichen Inhalt des Briefs.

Wie Abbildung 2.2 zeigt, wird diese Struktur in SOAP mit Hilfe von XML dargestellt: Das Wurzelelement ist stets *Envelope*, es enthält entweder ein oder zwei Kindelemente – einen optionalen *Header* und einen obligatorischen *Body*.

Sämtliche Stationen, die eine SOAP-Nachricht auf ihrem Weg vom Sender zum Empfänger zurücklegt, werden als *SOAP Nodes* bezeichnet. Der Absender heißt in der Spezifikation *Initial Sender* und der Empfänger *Ultimate Receiver* – beides sind damit spezielle SOAP Nodes. SOAP Nodes, welche die Nachricht weiterleiten, heißen *Intermediaries* – in unserer Analogie wären dies die Post-Verteilzentren.

Obwohl der Vergleich mit einem Post-Brief die Funktionen der einzelnen Elemente schon recht gut charakterisiert, wollen wir im Folgenden die hier zu Grunde liegenden Konzepte zusätzlich unter technischen Gesichtspunkten betrachten:

Über den optionalen SOAP-Header lassen sich Protokollerweiterungen realisieren. Dies können beispielsweise erweiterte Mechanismen zur Adressierung oder Priorisierung einzelner Nachrichten, Unterstützung für Transaktionen oder aber Verfahren zur Authentifizierung sein. Wenn ein Header in einer SOAP-Nachricht vorhanden ist, dann trägt er also Informationen, die nicht unmittelbar anwendungsbezogen sind, sondern der Protokollsteuerung dienen. Die Header-Informationen sind dabei in einzelne Abschnitte unterteilt, diese bezeichnet man jeweils als *Header Block*. In einem ersten Abschnitt könnten beispielsweise Benutzername und Passwort zur Benutzerauthentifizierung übertragen werden, ein zweiter könnte eine Transaktions-ID enthalten, die diese Nachricht einer laufenden Transaktion zuordnet.

Das Body-Element ist obligatorisch. Es enthält die Anwendungsdaten, die in dieser SOAP-Nachricht übertragen werden. Die Spezifikation stellt zunächst keine besonderen Anforderungen an den Inhalt des Body-Elements, es kann also beliebige anwendungsspezifische XML-Daten enthalten.

Allerdings birgt dieser hohe Freiheitsgrad auch die Gefahr von Namenskonflikten. Beispielsweise könnte im SOAP-Body eine XHTML-Seite übertragen werden, welche wiederum ein Body-Element enthält. Dass eine SOAP-Nachricht zwei Body-Elemente enthält, ist in der SOAP-Spezifikation nicht vorgesehen und führt folglich zu Fehlern bei der Verarbeitung.

Zur Vermeidung solcher Namenskonflikte sieht die SOAP-Spezifikation die Verwendung von XML-Namensräumen vor, diese bezeichnet man als *Namespaces*. Es wird empfohlen, sämtliche anwendungsspezifischen Daten über separate Namespaces zu

kennzeichnen, so dass Konflikte mit dem SOAP-Protokoll oder einer der zahlreichen SOAP-Erweiterungen vermieden werden. Wie in Abbildung 2.2 zu erkennen ist, stammen sämtliche Elemente, die zum SOAP-Envelope gehören, aus einem einheitlichen Namespace. Dieser wird in der SOAP-Spezifikation exklusiv für SOAP-Protokollfunktionalitäten reserviert.

Die Verwendung von Namespaces dient zusätzlich auch der Versionierung. Alle SOAP-Versionen verwenden charakteristische Namespaces, und ein Empfänger kann daran die vorliegende Protokollversion eindeutig identifizieren. In der Abbildung 2.2 deutet der Namespace `http://www.w3.org/2003/05/soap-envelope` auf die SOAP-Version 1.2 hin. Die ältere, aber noch immer weit verbreitete SOAP-Version 1.1 verwendet stattdessen den Namespace `http://schemas.xmlsoap.org/soap/envelope`.

2.2.2 Nachrichtentransport und Kommunikationsmuster

Ausgehend vom modularen Aufbau des Web Service Technology Stacks schreibt die SOAP-Spezifikation nicht die Verwendung eines speziellen Protokolls zum Transport von SOAP-Nachrichten vor. In Abhängigkeit von den besonderen Anforderungen des jeweiligen Anwendungsfalls und der bereits bestehenden Infrastruktur im Netzwerk kann der Anwender ein beliebiges Protokoll zum Nachrichtentransport wählen und dieses an SOAP binden. Wie bereits oben erwähnt, bezeichnet man eine solche Bindung auch als *Binding*. Dieser Binding-Mechanismus ist ein wesentlicher Unterschied zu älteren Technologien, wie Java RMI oder CORBA, denn bei diesen ist der Nachrichtentransport jeweils fester Bestandteil der Kernspezifikation.

In Abhängigkeit vom verwendeten Binding ergeben sich auch unterschiedliche Kommunikationsmuster, so genannte *Message Exchange Patterns (MEPs)*. Jede Binding-Spezifikation muss mindestens ein solches MEP bereitstellen. Ein MEP spezifiziert dabei den zeitlichen Ablauf des Kommunikationsprozesses zwischen den beteiligten Parteien und legt auch Maßnahmen zur Signalisierung von Fehlerzuständen fest.

Teil 1 der SOAP-Spezifikation definiert, was ein MEP ist und welche Bereiche bei der Spezifikation neuer MEPs zu beachten sind. In Teil 2 werden dann exemplarisch zwei MEPs für HTTP angegeben: ein *Response-MEP* und ein *Request-Response-MEP*. Das erste spezifiziert die Übertragung von SOAP-Nachrichten mit Hilfe des HTTP-GET-Kommandos, das zweite bezieht sich auf HTTP-POST.

Ein typisches Beispiel für die Verwendung des Response-MEPs ist in Abbildung 2.3 dargestellt: In der Filiale einer Autovermietung wird der Status einer Wagenreservierung über den dafür zuständigen Web Service des Unternehmens abgefragt. Als Aufrufparameter übermittelt der Dienstnutzer die zugehörige `ReservationID`.

Da es sich bei HTTP um ein Client/Server-Protokoll handelt, ist der Kommunikationsprozess zeitlich in zwei Abschnitte gegliedert: Request und Response. Im Re-

```
Request:
GET /info?reservationID=384DA3F  HTTP/1.1
Host: sunshinecars.example.org
Accept: text/html;q=0.5, application/soap+xml

Response:
HTTP/1.1 200 OK
Content-Type: application/soap+xml
Content-Length: nnnn
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <res:reservationInfo
    xmlns:res="http://sunshinecars.example.org/reservation">
    <res:vehicle>Standard SUV</res:vehicle>
    <res:customer>
        <res:name>John Smith</res:name>
        <res:ID>1674927356</res:ID>
    </res:customer>
    <res:pickup>2005-09-12T12:00:00.000Z</res:pickup>
    <res:dropoff>2005-09-19T12:00:00.000Z</res:dropoff>
    <res:rate>69,00 USD per day</res:rate>
  </res:reservationInfo>
 </env:Body>
</env:Envelope>
```

Abbildung 2.3: SOAP-Transport über HTTP-GET

quest wird der Anfrageparameter `ReservationID` an den Server übertragen, dies geschieht als Parameter des GET-Kommandos. In der Abbildung ist dieser Wert durch Fettdruck hervorgehoben. Der Web Service sendet in der Response-Nachricht das Ergebnis der Anfrage im XML-Format zurück. Die anwendungsspezifischen Daten werden dabei im SOAP-Body eingefügt, diese sind ebenfalls im Fettdruck dargestellt. Im Beispiel erkennt man weiterhin die bereits beschriebene Nutzung von separaten Namespaces zur eindeutigen Kennzeichnung von Protokoll- und Anwendungsdaten.

Bei Verwendung des Response-MEPs wird ein Aufrufparameter also direkt als Name-Wert-Paar in die Request-URI [44] codiert. Es lassen sich auch mehrere Aufrufparameter als Sequenz von Name-Wert-Paaren angeben; eine hierarchische Struktur, beispielsweise um den Wert eines zusammengesetzten Datentyps (Struct) darzustellen, ist aber nicht vorgesehen.

Um auch hierarchisch strukturierte Daten beim Aufruf eines Web Services übertragen zu können, bietet sich das Request-Response-MEP an, welches auf HTTP-POST aufsetzt. Abbildung 2.4 zeigt das gleiche Anwendungsbeispiel unter Verwendung dieses MEPs. Wie hier klar zu erkennen ist, enthält bereits der Request eine SOAP-Nachricht, und in ihrem Body-Element ist der Aufrufparameter in XML-Schreibweise

```
Request:
POST /info HTTP/1.1
Host: sunshinecars.example.org
Content-Type: application/soap+xml
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <res:reservationRequest
    xmlns:res="http://sunshinecars.example.org/reservation">
    <res:reservationID>384DA3F</res:reservationID>
  </res:reservationRequest>
 </env:Body>
</env:Envelope>

Response:
HTTP/1.1 200 OK
Content-Type: application/soap+xml
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <res:reservationInfo
    xmlns:res="http://sunshinecars.example.org/reservation">
    <res:vehicle>Standard SUV</res:vehicle>
    <res:customer>
        <res:name>John Smith</res:name>
        <res:ID>1674927356</res:ID>
    </res:customer>
    <res:pickup>2005-09-12T12:00:00:00.000Z</res:pickup>
    <res:dropoff>2005-09-19T12:00:00:00.000Z</res:dropoff>
    <res:rate>69,00 USD per day</res:rate>
  </res:reservationInfo>
 </env:Body>
</env:Envelope>
```

Abbildung 2.4: SOAP-Transport über HTTP-POST

codiert. Die Response-Nachricht ist identisch mit der aus dem Response-MEP-Beispiel (vergleiche Abbildung 2.3).

Durch die Verwendung der XML-Darstellung auch im Request ist das Request-Response-MEP besonders flexibel einsetzbar. Allerdings ergibt sich der Nachteil eines deutlich größeren Protokoll-Overheads; auch dies wird beim Längenvergleich der Request-Abschnitte beider Abbildungen sofort sichtbar.

Obwohl diese beiden HTTP-MEPs bislang die einzigen sind, die vom W3C standardisiert wurden, eignen sich prinzipiell auch andere Protokolle für den Transport von SOAP-Nachrichten – beispielsweise ist ein Transport über E-Mail- oder Message-Queuing-Dienste möglich. Solche alternativen SOAP-Bindings werden wir in Kapitel 6 ausführlich betrachten.

2.2.3 RPC und SOAP Data Encoding

Wie wir in den obigen Beispielen gesehen haben, stellt SOAP zunächst Funktionalitäten zur Kapselung beliebiger anwendungsspezifischer XML-Daten bereit. Allerdings reicht dies allein in der Praxis mitunter nicht aus, denn bei der Entwicklung verteilter Anwendungen liegen die zu versendenden Daten möglicherweise gar nicht in XML-Form vor. Vielmehr möchte ein Entwickler die Datentypen übertragen, die seine Programmiersprache vorsieht. Sicherlich könnte er hierzu, typischerweise unter Zuhilfenahme von APIs wie DOM oder SAX, selbst für die XML-Serialisierung und -Deserialisierung der ein- bzw. ausgehenden Daten sorgen – dies widerspricht jedoch dem Kerngedanken von Middleware-Lösungen. Schließlich sollen diese die Komplexität sämtlicher Kommunikationsaufgaben möglichst vollständig vor dem Entwickler verbergen. In diesem Teilabschnitt geht es daher um Techniken, mit deren Hilfe sich die SOAP-Kommunikation für den Entwickler besonders einfach gestaltet.

Wie bereits in Kapitel 1 erläutert, ist das RPC-Programmierparadigma ein besonders geeigneter und gängiger Ansatz zur Realisierung von Middleware-Lösungen. Er kapselt sämtliche Kommunikationsaufgaben in Form von Prozedur- bzw. Operationsaufrufen[1], was die Anwendungsentwicklung deutlich vereinfacht. Daher sieht die SOAP-Spezifikation Mechanismen vor, um das RPC-Paradigma auf Web Services zu übertragen. Obwohl diese unabhängig vom gewählten Transport Binding funktionieren, eignet sich eine Bindung an HTTP ganz besonders gut zur Umsetzung von RPC-Funktionalitäten; denn HTTP unterstützt als typisches Client/Server-Protokoll die ebenfalls Client/Server-orientierte RPC-Kommunikation besonders gut.

Zur Umsetzung bedarf es der Lösung zweier Teilprobleme; beide werden in der SOAP-Spezifikation (Teil 2) behandelt: Zum einen muss ein Datenformat festgelegt werden, um RPC-Aufruf und -Antwort samt aller übergebenen Prozedurparameter in XML

[1] Da der Begriff *Prozedur* stark durch das prozedurale Programmierparadigma geprägt ist, gängige SOAP-Implementierungen aber auch den objektorientierten Programmierstil unterstützen, verwendet der Autor im Folgenden den neutralen Begriff *Operation*.

```
Request:
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
  xmlns:soapenc="http://www.w3.org/2003/05/soap-encoding"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:res="http://sunshinecars.example.org/reservation">
   <soapenv:Body>
     <res:getReservationInfo soapenv:encodingStyle=
        "http://www.w3.org/2003/05/soap-encoding">
       <reservationID xsi:type="soapenc:string">
           dab37a3e4f</reservationID>
     </res:getReservationInfo>
  </soapenv:Body>
</soapenv:Envelope>

Response:
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
 xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
 xmlns:soapenc="http://www.w3.org/2003/05/soap-encoding"
 xmlns:rpc="http://www.w3.org/2003/05/soap-rpc"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:res="http://sunshinecars.example.org/reservation">
  <soapenv:Body>
    <res:getReservationInfoResponse soapenv:encodingStyle=
      "http://www.w3.org/2003/05/soap-encoding">
      <rpc:result>getReservationInfoReturn</rpc:result>
      <getReservationInfoReturn
        xsi:type="res:ReservationInfo">
        <customerInfo xsi:type="res:CustomerInfo">
          <ID   xsi:type="soapenc:int">27463782</ID>
          <name xsi:type="soapenc:string">John Smith</name>
        </customerInfo>
        <dropoff xsi:type="soapenc:dateTime">
            2005-09-12T12:00:00.000Z</dropoff>
        <pickup  xsi:type="soapenc:dateTime">
            2005-09-19T12:00:00.000Z</pickup>
        <rate    xsi:type="soapenc:string">
            69,00 USD per day</rate>
        <vehicle xsi:type="soapenc:string">
            Standard SUV</vehicle>
      </getReservationInfoReturn>
    </res:getReservationInfoResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Abbildung 2.5: SOAP-Nachrichten zur Darstellung eines RPC-Aufrufs

darzustellen. Dieser Teilaspekt eines Datenformates für RPC-Nachrichten heißt in der SOAP-Spezifikation *RPC Representation*. Zum anderen müssen Regeln für die einheitliche XML-Repräsentation von einfachen und zusammengesetzten Datentypen festgelegt werden, so dass die Operationsparameter automatisch serialisiert und deserialisiert werden können. Dieser Teilaspekt heißt *SOAP Encoding*.

Abbildung 2.5 zeigt exemplarisch die Realisierung des Anwendungsbeispiels aus dem vorigen Teilabschnitt – allerdings nun unter Verwendung von RPC Representation und SOAP Encoding. Die dargestellten SOAP-Nachrichten realisieren den RPC-Aufruf `ReservationInfo getReservationInfo(String reservationID)`. Ein Vergleich mit Abbildung 2.4 auf Seite 23 zeigt, dass das Nachrichtenformat an einigen Stellen deutlich abweicht. Die wesentlichen Änderungen sind im Fettdruck dargestellt und werden im Folgenden erläutert.

Betrachten wir zunächst die fünf grundlegenden Regeln für RPC Representation. Die Anwendung dieser Regeln kann der Leser anhand von Abbildung 2.5 nachvollziehen.

- RPC-Aufruf:
  1. Der SOAP-Body enthält genau ein Kindelement k_1. Sein Name entspricht dem der gerufenen Operation.
  2. Die Aufrufparameter der Operation werden jeweils als Kindelement von k_1 übertragen. Ihr Name entspricht jeweils dem Namen des Parameters.
- Antwort auf einen RPC-Aufruf:
  3. Der SOAP-Body enthält genau ein Kindelement k_2. Sein Name ist beliebig.
  4. Die Rückgabeparameter werden jeweils als Kindelement von k_2 übermittelt. Ihr Name entspricht jeweils dem Namen des Rückgabeparameters. (Ein RPC-Aufruf im Sinne von SOAP kann also durchaus mehrere Rückgabeparameter haben.)
  5. Einer der Rückgabeparameter hat einen besonderen Stellenwert. Er ist der Rückgabewert des RPC-Aufrufs. Ist dieser Rückgabewert nicht `void`, so muss es ein zusätzliches Kindelement von k_2 geben, das den Namespace `http://www.w3.org/2003/05/soap-rpc` und den Namen *result* trägt. Es enthält den Namen des Kindes von k_2, der als Rückgabewert interpretiert werden soll. Ist der Rückgabewert dagegen `void`, darf es ein solches Element *result* nicht geben.

Es sei angemerkt, dass die Unterscheidung zwischen „normalen“ Rückgabeparametern und dem Rückgabewert einer Operation lediglich der Tatsache Rechnung trägt, dass heutige Programmiersprachen unterschiedliche Formen von Rückgabeparametern vorsehen. Zum einen gibt es die Ergebnisrückgabe in Form von (ggf. mehreren) Call-by-Reference-Parametern; zum anderen gibt es den Rückgabewert einer Operation (*return value*). Die SOAP-Spezifikation versucht mit Hilfe des Elements *result*,

beide Techniken möglichst flexibel in XML nachzubilden. Es ist also keineswegs so, dass nur der ausgezeichnete Rückgabewert von Bedeutung ist und die anderen Parameter weniger wichtig sind bzw. ignoriert werden sollen. Das Element *result* dient ausschließlich dazu, der empfangenden SOAP-Implementierung anzuzeigen, welcher der Rückgabeparameter als Rückgabewert des RPC-Aufrufs zu interpretieren ist. Ob eine solche Unterscheidung überhaupt sinnvoll ist, hängt von der Mächtigkeit der eingesetzten Programmiersprache ab.

Neben diesen fünf Regeln zur Formulierung von SOAP-RPC-Nachrichten gibt es noch einige weitere, die insbesondere der Signalisierung und Behandlung von Fehlerzuständen dienen. Da diese Abläufe jedoch nicht zum weiteren Verständnis dieser Arbeit beitragen, wird der Autor auch nicht darauf eingehen.

Betrachten wir nun den zweiten Teilaspekt *SOAP Encoding*: Um die Aufruf- und Rückgabeparameter einer RPC-Nachricht unmissverständlich in SOAP codieren zu können, muss eine eindeutige Zuordnung zwischen den Datentypen der verwendeten Programmiersprache und der korrespondierenden XML-Darstellung möglich sein. Die SOAP-Spezifikation definiert einen Mechanismus zur Lösung dieser Problemstellung unter dem Namen SOAP Encoding.

Wie man in den Abbildungen 2.3 und 2.4 erkennt, sind die übertragenen Werte im Allgemeinen nicht typbehaftet. In diesen Beispielen deutet also nichts darauf hin, dass es sich bei `ID` um einen Integer-Wert handelt oder bei `pickup` und `dropoff` um Datumsangaben mit Uhrzeit. Folglich kann eine SOAP-basierte Middleware bei einer eingehenden Nachricht auch nicht automatisch entscheiden, auf welche Datentypen der vorliegenden Programmiersprache die in der SOAP-Nachricht angegebenen Werte abgebildet werden sollen. Diese Funktionalität ist zur Realisierung von RPC-Aufrufen aber zwingend erforderlich.

Die SOAP-Spezifikation Teil 2 sieht daher die Möglichkeit vor, den Datentyp jeweils über das Attribut *type* anzugeben. Welches Typsystem dabei verwendet wird, ist nicht verbindlich festgelegt. Eine Variante besteht darin, das Typsystem von XML Schema zu verwenden, dieses bietet eine breite Auswahl gängiger Datentypen an – beispielsweise Boolean, Integer, Date oder String [179]. Diese Datentypen tragen den Namespace `http://www.w3.org/2001/XMLSchema`.

Allerdings war bei der Entwicklung von SOAP 1.1 im Jahr 2000 das Typsystem von XML Schema noch unvollständig und nicht ausgereift. Daher wurde für SOAP 1.1 ein eigenes Typsystem entwickelt, welches auch in SOAP 1.2 noch enthalten ist und von vielen SOAP-Implementierungen – darunter auch Apache Axis – noch immer verwendet wird. Dieses Typsystem hat man aber in der Version 1.2 soweit angepasst, dass sich die SOAP-Datentypen unmittelbar von den korrespondierenden XML-Schema-Typen ableiten. Somit gibt es nur noch marginale Unterschiede zwischen den beiden Typsystemen. Sämtliche SOAP-Datentypen tragen den Namespace `http://www.w3.org/2003/05/soap-encoding`. In Abbildung 2.5 erkennt man

deutlich die Kennzeichnung der einzelnen Datentypen über das Attribut *type* mit Datentypbezeichnungen aus dem SOAP-Typsystem.

Ein weiterer Aspekt des SOAP Encodings bezieht sich auf einheitliche Regeln zur Serialisierung bzw. Deserialisierung von einfachen und zusammengesetzten Typen, wie Structs oder Arrays. Finden diese Regeln bei der Serialisierung Anwendung, so wird dies über das Attribut `encodingStyle` mit dem Wert `http://www.w3.org/2003/05/soap-encoding` in der SOAP-Nachricht vermerkt. Der Empfänger der Nachricht erkennt anhand dieser Kennzeichnung, dass er die enthaltenen XML-Daten mit Hilfe der in der SOAP-Spezifikation festgelegten Regel automatisch deserialisieren kann. In Abbildung 2.5 erkennt man diese Kennzeichnung sowohl in der Request- als auch in der Response-Nachricht – und zwar jeweils im Kindelement des SOAP-Bodys.

Es sei angemerkt, dass RPC Representation und SOAP Encoding zwei Mechanismen sind, die grundsätzlich auch unabhängig voneinander funktionieren. Allerdings ist es durchaus typisch, beide Funktionalitäten zu kombinieren, denn dies ermöglicht die Entwicklung von SOAP-Anwendungen, die einerseits die Serialisierung und Deserialisierung von Datentypen vollständig automatisch vornehmen, und andererseits auch das Senden und Empfangen von Nachrichten in einem RPC-Aufruf kapseln. Dies macht die Anwendungsentwicklung mit SOAP besonders komfortabel.

Web Services, die RPC Representation und SOAP Encoding verwenden, bezeichnet man gemeinhin auch als *RPC-orientiert*. Im Gegensatz dazu heißen Web Services, die hiervon keinen Gebrauch machen, *dokumentorientiert*. Obwohl beide Begriffe nicht in der SOAP-Spezifikation verankert sind, haben sie sich bei der Web-Service-Entwicklung etabliert und werden daher auch im Rahmen dieser Arbeit verwendet.

Abschließend sei noch auf den Begriff „SOAP-Engine“ hingewiesen: Dieser bezeichnet eine Software-Komponente, die ausgehende SOAP-Nachrichten erzeugt und eingehende verarbeitet. Sie stellt dem Anwendungsentwickler sämtliche SOAP-Funktionalitäten zur Verfügung, typischerweise in Form einer API. Auf diese Weise wird die Komplexität des SOAP-Protokolls vor dem Softwareentwickler verborgen.

Zusammenfassend lässt sich feststellen, dass das Protokoll SOAP im Web Service Technologie Stack eine zentrale Rolle einnimmt. Wie bereits zu Beginn dieses Abschnitts angedeutet, wäre es prinzipiell zwar auch möglich, auf SOAP zu verzichten und die anwendungsspezifischen XML-Nachrichten direkt über ein Protokoll wie HTTP zu übertragen. Auch so ließen sich Daten plattformunabhängig austauschen. Erst SOAP aber ermöglicht eine Kapselung der Daten im Sinne einer vollwertigen Middleware-Lösung: Über die beiden standardisierten Mechanismen SOAP Encoding und RPC Representation kann eine SOAP-Engine die technische Repräsentation der Daten vollständig vor dem Entwickler verbergen. Zudem können durch Aufteilung einer Nachricht in SOAP-Header und -Body auf einfache Weise Protokollerweiterungen realisiert werden. Der Funktionsumfang von SOAP lässt sich also je nach Bedarf erweitern und an einen konkreten Anwendungsfall anpassen; entscheidend ist hierbei,

dass die Anwendungsdaten im Body klar von den jeweils benötigten Protokollinformationen im Header separiert bleiben.

2.3 WSDL

Ein weiterer wesentlicher Aspekt für Web Services als eine universelle Middleware-Technologie ist das Vorhandensein einer flexiblen und mächtigen Schnittstellenbeschreibungssprache *(engl.: Interface Definition Language, IDL)*. Die grundlegende Aufgabe einer solchen IDL besteht in einer exakten und maschinenlesbaren Beschreibung der Dienstschnittstellen. Ein Anwendungsentwickler kann mit Hilfe eines IDL-Compilers aus einem IDL-Dokument Quellcode erzeugen, der sämtliche Kommunikationsroutinen für die Interaktion mit diesem Dienst bereitstellt.

Wie bereits in Kapitel 1 erwähnt, gibt es für ältere Middleware-Ansätze wie Java RMI oder CORBA bereits Schnittstellenbeschreibungssprachen und passende IDL-Compiler, doch diese eignen sich nicht für den Einsatz mit Web Services. Grundlegende Kriterien, wie Erweiterbarkeit oder die Verwendung von XML als universelle Datenbeschreibungssprache, werden von diesen IDLs nicht erfüllt.

Aus diesem Grund wurde die *Web Service Description Language (WSDL)* entwickelt. Die aktuelle WSDL-Version 2.0 hat das W3C im März 2006 veröffentlicht, sie liegt zurzeit als *Candidate Recommendation* vor. Die Verabschiedung der endgültigen *Recommendation* ist für Mitte 2006 geplant. Analog zu SOAP ist auch die WSDL-Spezifikation in zwei Teile aufgeteilt: *WSDL Version 2.0 Part 1: Core Language* und *WSDL Version 2.0 Part 2: Adjuncts*. Der erste Teil legt die Struktur von WSDL-Dokumenten auf einem abstrakten Niveau fest, d. h. unabhängig von konkreten technischen Ausprägungen der zu beschreibenden Web Services. Der zweite Teil spezifiziert die Verwendung von WSDL zur Beschreibung von Web Services, die auf SOAP in Verbindung mit HTTP zum Nachrichtentransport basieren.

2.3.1 Dokumentstruktur

Wie in Abbildung 2.6 dargestellt, ist ein WSDL-Dokument typischerweise in fünf Hauptabschnitte unterteilt: *documentation*, *types*, *interface*, *binding* und *service*. Im Folgenden wird der Autor die Funktionen dieser Abschnitte jeweils erläutern und auch die Zusammenhänge zwischen ihnen darstellen.

Der Abschnitt *documentation* enthält eine ergänzende Dienstbeschreibung in einer für Menschen verständlichen Klartextform. Die übrigen Angaben einer WSDL-Beschreibung sind, wie bereits erläutert, primär für eine Interpretation durch Maschinen bestimmt. Allerdings beschreibt der maschinenlesbare Teil lediglich die technische Ausgestaltung der Dienstschnittstelle und ist damit notwendigerweise nicht vollständig. Er beschränkt sich auf die Beschreibung von technischen Parametern, insbesondere

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/2006/01/wsdl"...>

  <documentation>
  ...
  </documentation>

  <types>
    <xs:schema>
        <xs:element name="someElementName" .../>
    ...
    </xs:schema>
  </types>

  <interface>
    <operation name="someOpName" ...>
        <input element="someElementName" ...>
        ...
        <output ...>
        ...
    </operation>
    ...
  </interface>

  <binding name="someBindingName"...>
      <operation ref="someOpName" .../>
  ...
  </binding>

  <service>
      <endpoint binding="someBindingName" ...
                address="someURI"/>
  ...
  </service>
</description>
```

Abbildung 2.6: Typische Struktur eines WSDL-Dokuments

den Aufbau gültiger Ein- und Ausgabenachrichten, sowie auf die Bereitstellung von Adressierungs- und Protokollinformationen. Er enthält aber keinerlei Angaben zur Dienstsemantik, d. h. in welcher Weise der Dienst zu benutzen ist und welche Bedeutung die einzelnen Operationen und Operationsparameter haben. Ein Anwendungsentwickler kann daher auf die ergänzenden Informationen aus dem *documentation* Abschnitt zurückgreifen, um die Funktionsweise des Dienstes besser zu verstehen.

Im Abschnitt *types* werden Datentypen definiert, die in ein- und ausgehenden Nachrichten des beschriebenen Web Services verwendet werden. Das hierzu verwendete Typsystem ist in der WSDL-Spezifikation nicht festgelegt. Es könnten also beispielsweise die im vorherigen Abschnitt behandelten SOAP-Datentypen verwendet werden oder aber auch XML-Schema-Datentypen.

Der Abschnitt *interface* ist die Kernkomponente einer jeden WSDL-Beschreibung. Hier werden alle vom Web Service bereitgestellten Operationen aufgeführt und die korrespondierenden Ein- und Ausgabenachrichtentypen spezifiziert. Dabei referenziert man die Datentypen aus dem Abschnitt *types*.

Bis zu diesem Punkt ist die Dienstbeschreibung noch abstrakt, d. h. sie enthält noch keine Angaben über Nachrichtenformate zur Kapselung der Anwendungsdaten oder über Protokolle zum Nachrichtentransport.

Der Abschnitt *binding* ergänzt diese noch fehlenden Angaben. Hierzu werden die *operation* Elemente aus dem Abschnitt interfaces referenziert. Es folgen Angaben zum verwendeten Nachrichtenformat (z. B. SOAP 1.2) und zum Transportmechanismus für den Nachrichtenaustausch (z. B. HTTP GET). Service-Operationen können dabei durchaus mehrfach referenziert werden. Auf diese Weise ist es möglich, Services zu beschreiben, bei denen die gleiche Operation über verschiedene technische Schnittstellen abrufbar ist. Also beispielsweise sowohl mit Hilfe von SOAP 1.2 über HTTP GET als auch mit SOAP 1.1 über HTTP POST.

Im Abschnitt *service* werden schließlich so genannte Service-Endpunkte (service endpoints) definiert. Diese legen fest, wo ein Dienstnutzer den beschriebenen Dienst aufrufen kann. Wie in Abbildung 2.6 dargestellt, referenziert jedes *endpoint* Element zunächst ein Binding. Über ein weiteres Attribut *address* wird dann typischerweise die Adresse dieses Endpunkts in Form einer URI angegeben. Da ein Dienst auch durchaus über mehrere Endpunkte erreichbar sein kann, beispielsweise weil er auf mehreren Servern repliziert wird, ist es auch erlaubt, ein Binding in mehreren Endpunktdefinitionen zu referenzieren.

Es sei darauf hingewiesen, dass die Angabe eines *address* Attributs in einer Endpunktdefinition nach der WSDL-Spezifikation Teil 1 optional ist. Dies mag auf den ersten Blick verwundern, da die Kenntnis einer Zugriffsadresse für die erfolgreiche Nutzung eines Dienstes offenbar von entscheidender Bedeutung ist. In der Erläuterung zum *address* Attribut heißt es jedoch, dass die Endpunktadresse auch über andere Mechanismen bekannt gegeben werden kann; als Beispiel wird die Verwendung von WS-Addressing [185] aufgeführt. Außerdem seien auch Web-Service-Anwendungen möglich, die völlig ohne Endpunkt-Adressierung auskommen. Hierzu zählen beispielsweise Broadcast-Anwendungen, bei denen eine Nachricht an alle erreichbaren Systeme geschickt wird.

2.3.2 Beispiel

Abbildung 2.7 stellt exemplarisch ein vollständiges WSDL-Dokument dar, welches den Web Service für Mietwagenreservierungen aus dem vorangegangenen Abschnitt beschreibt (Nicht-RPC-Variante in Verbindung mit dem Request-Response-MEP, vergleiche auch Abbildung 2.4, Seite 23). Man erkennt deutlich die Einteilung in die Ab-

schnitte *types*, *interface*, *binding* und *service*. Der optionale *documentation* Abschnitt ist in diesem Beispiel nicht enthalten.

Im Abschnitt *types* werden die zur Kommunikation verwendeten Datentypen mit Hilfe von XML Schema definiert. Betrachtet man die Elementdeklarationen für *reservationRequest* und *reservationInfo*, so zeigt ein Vergleich mit Abbildung 2.4 auf Seite 23, dass die im Body der Request- bzw. Response-Nachricht enthaltenen XML-Strukturen mit diesen Datentypen konform sind.

Der Abschnitt *interface* spezifiziert alle Operationen, die der Web Service anbietet. Im Beispiel gibt es nur eine Operation namens *reservationInfoOperation*. In WSDL 2.0 wird jede Operation einem in der WSDL-Spezifikation vordefinierten Kommunikationsmuster, genannt *Message Exchange Pattern (MEP)*, zugeordnet. Hierbei ist aber zu beachten, dass diese WSDL-Interface-MEPs nicht den in Abschnitt 2.2.2 vorgestellten SOAP-MEPs entsprechen. Der Begriff MEP hat also in der SOAP- und WSDL-Spezifikation jeweils eine andere Bedeutung. Während er sich in SOAP unmittelbar auf den verwendeten Transportmechanismus bezieht, steht er im Interface-Abschnitt einer WSDL-Beschreibung lediglich für die Richtung der Kommunikation; d. h. dafür, ob eine Service-Operation eingehende, ausgehende oder sowohl ein- als auch ausgehende Nachrichten unterstützt. Im Beispiel werden sowohl ein- als auch ausgehende Nachrichten unterstützt, was durch die Verwendung der vordefinierten URI `http://www.w3.org/2006/01/wsdl/in-out` als Wert für das Attribut *pattern* angezeigt wird.

Im dann folgenden Abschnitt *binding* wird die abstrakte Interface-Definition an ein konkretes Nachrichtenformat sowie an einen Mechanismus zum Nachrichtentransport gebunden. Auch diese Schritte funktionieren über die Referenzierung von vordefinierten URIs. Wie man im dargestellten Beispiel erkennen kann, erfolgt die Bindung hier an SOAP und HTTP, ausgedrückt über die beiden URIs `http://www.w3.org/2006/01/wsdl/soap` und `http://www.w3.org/2003/05/soap/bindings/HTTP`. Schließlich wird noch das SOAP-MEP für die referenzierte Web-Service-Operation festgelegt, dies geschieht durch Verwendung der URI `http://www.w3.org/2003/05/soap/mep/request-response`.

Im letzten Abschnitt der dargestellten WSDL-Beschreibung *service* wird ein Service-Endpunkt spezifiziert. Da im Abschnitt *binding* eine Bindung an das HTTP-Protokoll aufgeführt ist, muss der Endpunkt auch die Form einer HTTP-URL aufweisen: `http://sunshinecars.example.org:8080/reservation.cgi`. Würde man hingegen E-Mail-Protokolle für den SOAP-Transport einsetzen, müsste hier eine E-Mail-Adresse angegeben werden.

```
<?xml version="1.0" encoding="utf-8" ?> <description
  xmlns="http://www.w3.org/2006/01/wsdl"
  targetNamespace= "http://sunshinecars.example.org/reservation"
  xmlns:tns= "http://sunshinecars.example.org/wsdl/reservation"
  xmlns:res= "http://sunshinecars.example.org/reservation"
  xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap">
  <types><xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://sunshinecars.example.org/reservation"
    xmlns="http://sunshinecars.example.org/reservation">
    <xs:element name="reservationRequest">
      <xs:compleyType><xs:sequence>
        <xs:element name="reservationID" type="xs:string"/>
      </xs:sequence></xs:complexType>
    </xs:element>
    <xs:element name="reservationInfo">
      <xs:complexType><xs:sequence>
        <xs:element name="vehicle" type="xs:string"/>
        <xs:element name="customer" type="customerType"/>
        <xs:element name="pickup" type="xs:dateTime"/>
        <xs:element name="dropoff" type="xs:dateTime"/>
        <xs:element name="rate" type="xs:string"/>
      </xs:sequence></xs:compleType>
    </xs:element>
    <xs:complexType name="customerType"><xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="ID" type="xs:int"/>
    </xs:sequence></xs:complexType>
  </xs:schema></types>
  <interface name="reservationInterface">
    <operation name="reservationInfoOperation"
      pattern="http://www.w3.org/2006/01/wsdl/in-out">
      <input messageLabel="In"   element="res:reservationRequest"/>
      <output messageLabel="Out" element="res:reservationInfo"/>
    </operation>
  </interface>
  <binding name="reservationSOAPBinding"
    interface="tns:reservationInterface"
    type="http://www.w3.org/2006/01/wsdl/soap"
    wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">
    <operation ref="tns:reservationInfoOperation" wsoap:mep=
      "http://www.w3.org/2003/05/soap/mep/request-response"/>
  </binding>
  <service name="reservationService"
    interface="tns:reservationInterface">
    <endpoint name="reservationEndpoint"
      binding="tns:reservationSOAPBinding"
      address="http://sunshinecars.example.org:8080/reservation.cgi"/>
  </service>
</description>
```

Abbildung 2.7: Vollständiges Beispiel eines WSDL-Dokuments

2.4 UDDI

Die UDDI-Spezifikation beschreibt eine Reihe von Datenformaten und APIs zur Implementierung von elektronischen Unternehmensregistern (business registries). UDDI ist dabei ein Akronym für *Universal Description, Discovery and Integration* und wurde erstmals im Jahr 2000 von Microsoft, IBM und der Firma Ariba vorgestellt. Ursprünglich zielte diese Technologie auf die Realisierung von allgemeinen Business-to-Business-Anwendungen (B2B) ab. Die Vision dabei war, dass Unternehmen potentielle Handelspartner schnell und einfach über ein weltweites, elektronisches Verzeichnis ausfindig machen können. Dieses sollte nicht nur allgemeine Angaben, wie Name, Anschrift oder Branche, zu einem Unternehmen liefern können, sondern zusätzlich auch Informationen über die technischen Schnittstellen, über die das Unternehmen mit potentiellen B2B-Partnern kommuniziert.

Allerdings konnte sich das UDDI-Konzept im Unternehmensumfeld nie wirklich durchsetzen, und alle heute existierenden internetweiten UDDI-Verzeichnisse dienen vorwiegend Demonstrations- oder Testzwecken. Inzwischen hat sich der Fokus von UDDI jedoch etwas gewandelt. Obwohl Web Services von Anfang an ein wesentliches technologisches Mittel zur Umsetzung von UDDI bildeten, sind diese erst in der aktuellen UDDI-Version 3.02 auch klar als Zielsetzung in der UDDI-Spezifikation verankert: Die Ausrichtung von UDDI hat sich von einem allgemeinen elektronischen Unternehmensregister hin zu einem spezialisierten Register verlagert, das nunmehr auf die besonderen Bedürfnisse zum Auffinden von Web Services ausgerichtet ist. Ein wesentlicher Aspekt war dabei auch die vollständige Automatisierung. Mit Hilfe von UDDI sollen geeignete Dienste dynamisch und vollautomatisch zur Laufzeit einer Web-Service-Anwendung gefunden werden.

Der UDDI 3.02 Standard, welcher im Oktober 2004 spezifiziert und im Februar 2005 ratifiziert wurde, ist in vier verschiedene Gruppen von Dokumenten eingeteilt: eine *Feature List* beschreibt die Änderungen im Vergleich zur Version 2.0, die *Technical Specification* erläutert die grundlegende Architektur von UDDI, die eingesetzten Datenstrukturen *(data structures)* werden in Form mehrerer XML-Schema-Dokumente spezifiziert und schließlich gibt es noch eine Reihe von WSDL-Dokumenten, welche die verschiedenen *APIs* von UDDI beschreiben.

UDDI ist im Grundsatz zentralisiert aufgebaut. Wenn man einen Web Service in einem UDDI-Register eintragen möchte, so muss man die zugehörigen Daten entweder per Web-Browser oder über die im Standard vorgesehene SOAP-Schnittstelle bei einem UDDI-Anbieter ablegen. In der UDDI-Spezifikation heißt ein solcher UDDI-Anbieter *operator*. Die Daten werden zunächst ausschließlich bei diesem Anbieter vorgehalten, so dass ein potentieller Dienstnutzer auch explizit bei diesem UDDI-Anbieter nachfragen muss, um einen hier registrierten Web Service finden zu können.

Diese rein lokale Datenspeicherung führt jedoch dazu, dass jeder UDDI-Anbieter verschiedene Ergebnisse zu einer Suchanfrage liefert. Somit steht ein Dienstnutzer zu-

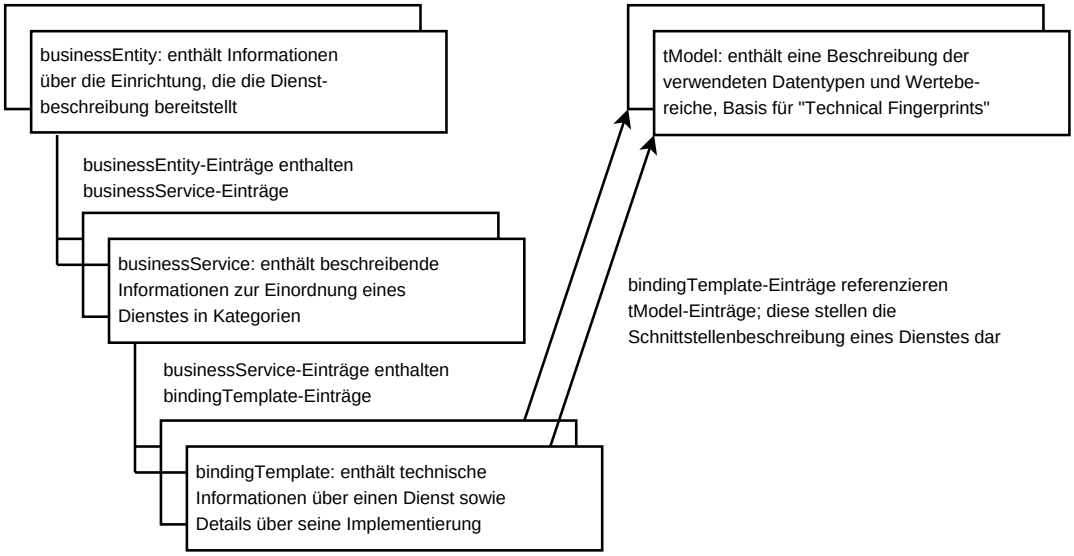


Abbildung 2.8: Beziehungen zwischen den grundlegenden UDDI-Datentypen (entnommen aus [108], vom Autor aus dem Englischen übersetzt)

nächst vor dem Problem, einen UDDI-Anbieter ausfindig zu machen, der die Suchanfrage in geeigneter Weise bearbeiten kann. Möglicherweise sind Anfragen bei verschiedenen UDDI-Anbietern erforderlich, bevor ein passender Web Service gefunden wird.

Um dieses Problem zu vermeiden, stellt UDDI Mechanismen bereit, um die Daten zwischen mehreren UDDI-Anbietern zu replizieren und somit zu vereinheitlichen. Anbieter, die an einem solchen Verbund teilnehmen, formen eine so genannte *operator cloud*.

2.4.1 Datentypen

Abbildung 2.8 veranschaulicht die Beziehungen zwischen den grundlegenden Datentypen, aus denen ein UDDI-Verzeichnis aufgebaut ist. Die Datentypen sind dabei hierarchisch organisiert, als Datenbeschreibungssprache kommt durchgängig XML zum Einsatz. Es gibt vier verschiedene Basisdatentypen:

Das *businessEntity* Element ist das Wurzel-Element eines jeden UDDI-Eintrags, es enthält sämtliche Angaben zu einem Dienstanbieter. Für jeden angebotenen Dienst gibt es ein *businessService* Kindelement, welches Informationen zu diesem Dienst bereitstellt. Die anderen Kindelemente von *businessEntity* enthalten allgemeine Angaben zum Dienstanbieter – beispielsweise Telefon- oder Faxnummer, Postanschrift usw.

Ein Element vom Typ *businessService* enthält eine nichttechnische Beschreibung eines Dienstes. Hierzu gehören typischerweise ein Name sowie ein beschreibender, natürlichsprachlicher Text. Weiterhin kann ein Element dieses Typs eine oder mehrere Kategorien enthalten, die diesen Service charakterisieren. Mögliche Kategorien könnten sich beispielsweise auf die geographische Lage eines Dienstes, auf einen industriellen Sektor oder ein konkretes Produkt beziehen. In der UDDI-Spezifikation heißt eine solche Sammlung von zutreffenden Kategorien *categoryBag*. Die Kategorisierung ist in der UDDI-Spezifikation nicht vorgegeben, prinzipiell sind also beliebige Einteilungen möglich. Schließlich gibt es mindestens ein *bindingTemplate* Kindelement.

Ein *bindingTemplate* Element enthält eine technische Dienstbeschreibung. Insbesondere wird hier der Dienstzugangspunkt angegeben, an dem der Dienst erreichbar ist. Die UDDI-Spezifikation sieht keinerlei Einschränkungen hinsichtlich der Beschreibungsform für einen solchen Zugangspunkt vor, es kann ein beliebiger String angegeben werden – beispielsweise eine URL, eine E-Mail-Adresse oder auch eine Telefonnummer. Ein *bindingTemplate* Element kann ebenfalls einen *categoryBag* enthalten, im Gegensatz zu dem *categoryBag* unterhalb eines *businessService* Elements enthält dieser aber ausschließlich technische Kategorien. Eine Einteilung könnte sich hier beispielsweise auf den Entwicklungsstatus eines Dienstes beziehen. Eine mögliche Klassifizierung wäre hier „production“, „stable“, „testing“ und „unstable“. Ein *bindingTemplate* enthält außerdem optionale Elemente zur natürlichsprachlichen Erläuterung einzelner Funktionalitäten, weiterhin wird hier mindestens ein *tModel* Element referenziert.

Ein *tModel* Element liefert eine genaue technische Spezifikation einer Dienstschnittstelle. Hierzu gehören insbesondere Angaben zum verwendeten Nachrichtentransportprotokoll sowie zu Datentypen für die ein- und ausgehenden Nachrichten. Auf dieser Basis kann ein potentieller Dienstnutzer entscheiden, ob dieser Dienst technisch kompatibel ist. In der UDDI-Spezifikation wird dies anschaulich als *technical fingerprint* bezeichnet: Mehrere Dienste können ein *tModel* Element referenzieren und auf diese Weise anzeigen, dass sie zu der hier spezifizierten Schnittstelle kompatibel sind. Es ist offensichtlich, dass eine einheitliche und allgemein verständliche Beschreibungssprache für die praktische Umsetzung dieser Idee essentiell ist. Allerdings lässt UDDI auch diesen Punkt offen und empfiehlt lediglich die Verwendung eines beliebigen, standardisierten Vokabulars. Eine Möglichkeit besteht in der Referenzierung eines WSDL-Dokuments, dieser Lösungsansatz wird in der UDDI-Spezifikation über das Attribut *wsdlInterface* auch explizit unterstützt.

Alle Instanzen von UDDI-Datentypen sind in einem UDDI-Register über eindeutige Bezeichner (UDDI-IDs) gekennzeichnet. Seit UDDI 2.0 sind diese ID-Werte 128 Bits lang und werden in Hexadezimalschreibweise angegeben. Ein Beispiel für eine UDDI-ID sieht wie folgt aus: `uddi:3DF43C1F-83A1-BB39-007E-F487BA25CC04`.

2.4.2 APIs

Um den Datenbestand eines UDDI-Registers abzufragen, zu replizieren, zu aktualisieren oder zu ergänzen, sieht die UDDI-Spezifikation verschiedene APIs vor, welche über WSDL-Dokumente beschrieben sind. Sämtliche APIs stellen eine Bindung an das SOAP-Protokoll bereit, wobei die meisten UDDI-Implementierungen grundlegende Funktionalitäten zusätzlich auch über eine Web-Browser-Schnittstelle anbieten.

Da die Verwendung der UDDI-APIs für das weitere Verständnis dieser Arbeit nicht relevant ist, wird der Autor diese APIs nicht im Einzelnen vorstellen. Nachfolgend erläutert er anhand der *UDDI Inquiry* API, wie eine UDDI-Abfrage technisch realisiert wird. Weiterführende Erläuterungen findet der Leser beispielsweise in [39] oder direkt im UDDI-Standard [108].

Die *UDDI Inquiry* API dient zum Abfragen eines UDDI-Registers. Sie stellt zwei verschiedene Arten von SOAP-Operationen bereit: Find- und Get-Operationen. Für jeden der vier grundlegenden Datentypen gibt es jeweils eine Find- und eine Get-Operation. Find-Operationen dienen zum überblicksweisen Durchsuchen eines UDDI-Registers; sie geben Ergebnisse zurück, die nur wenige Details enthalten. Die Get-Operationen liefern hingegen eine vollständige Ausgabe der gefundenen Datensätze. Diese Einteilung in Get- und Find-Operationen hat den Zweck, den Suchprozess zweistufig zu gestalten: mit Hilfe einer Find-Operation kann sich ein Anfrager zunächst einen Überblick über relevante Einträge in diesem UDDI-Register verschaffen und ggf. die Ergebnismenge über weitere Find-Operationen schrittweise vergrößern. In einem zweiten Schritt kann er dann mit Hilfe von Get-Operationen die Details interessanter Treffer abrufen.

Abbildung 2.9 zeigt exemplarisch die Verwendung der Operation *find_tModel*. Wie der Name vermuten lässt, dient diese Operation dem Auffinden von tModel-Datensätzen. Die Nachricht des Aufrufers ist in der Abbildung mit „Request“, die Antwortnachricht des UDDI-Registers mit „Response“ gekennzeichnet. Zur besseren Übersicht hat der Autor in der Abbildung nur die UDDI-spezifischen Nutzdaten, also den Inhalt vom SOAP-Body, dargestellt – die umgebende SOAP-Hülle ist nicht mit abgebildet.

Im Request wird als Kindelement von *find_tModel* der Suchparameter spezifiziert, im Beispiel ist dies *name*. Zusätzlich können im Request *findQualifier* Elemente enthalten sein, welche genauer festlegen, wie die Suche ablaufen soll. Im Beispiel sorgt die Angabe *approximateMatch* dafür, dass in der Suchanfrage bestimmte Wildcard-Zeichen ausgewertet werden: Die beiden Prozentzeichen in `%Sunshine%` geben an, dass sämtliche tModel-Einträge gefunden werden sollen, deren Namen das Muster `Sunshine` enthalten, wobei vor und nach `Sunshine` beliebige Zeichen enthalten sein können.

Die Response-Nachricht liefert eine Übersicht über die gefundenen tModel-Einträge. Allerdings wird nicht der vollständige Eintrag geliefert, sondern nur der Wert von *name* sowie die zum Eintrag gehörige UDDI-ID. Ist der Anfrager mit einem dieser

```
Request:
<uddi:find_tModel generic="3.0"
  xmlns:uddi="urn.uddi-org:api_v3">
  <findQualifiers>
     <findQualifier>approximateMatch</findQualifier>
  </findQualifiers>
  <uddi:name>%Sunshine%</uddi:name>
</uddi:find_tModel>

Response:
<uddi:tModelList generic="3.0" operator="IBM"
    xmlns:uddi="urn.uddi-org:api_v3">
  <uddi:tModelInfos>
    <uddi:tModelInfo
        tModelKey="uddi:3DF43C1F-83A1-BB39-007E-F487BA25CC04">
      <uddi:name>Sunshine Cars Inc. reservation service</uddi:name>
    </uddi:tModelInfo>
    <uddi:tModelInfo
        tModelKey="uddi:2342D4B5-8AA0-FE04-83DB-FFBB023CABC5">
      <uddi:name>Sunshine Inn Motel electronic check-in</uddi:name>
    </uddi:tModelInfo>
    <uddi:tModelInfo
        tModelKey="uddi:BB54CF23-F52B-40A4-B8FA-0038B35A9FD3">
      <uddi:name>Sunshine Fruits Corp. price list service</uddi:name>
    </uddi:tModelInfo>
  </uddi:tModelInfos>
</uddi:tModelList>
```

Abbildung 2.9: Nutzung der *UDDI Inquiry* API am Beispiel der *find_ tModel* Operation

Treffer zufrieden, so würde er nun über die Operation *get_ tModelDetail* unter Angabe der zugehörigen UDDI-ID den vollständigen tModel-Eintrag abfragen.

2.4.3 Verbreitung von UDDI

Obwohl ein Dienstregister wie UDDI sicherlich ein wichtiger Baustein bei der Implementierung einer Web-Service-Infrastruktur ist, konnte sich diese Technik in der Praxis bislang kaum durchsetzen. Vermutlich liegt ein wesentlicher Grund hierfür in der Tatsache, dass UDDI zentralisiert aufgebaut ist, was dem Grundgedanken der Web-Service-Technologie im Kern widerspricht. Denn es erscheint unter technologischen Gesichtspunkten nicht zweckmäßig, dass in einer massiv verteilten Web-Service-Umgebung der Namensdienst zentralisiert aufgebaut ist.

Ein weiterer Grund für mangelnde Akzeptanz könnte in der zum Teil sehr offen und allgemein formulierten Spezifikation liegen. Wegen ungenügender Richtlinien zur konkreten Ausgestaltung der einzelnen Datentypen ist eine konsistente Beschreibung von

Diensten kaum möglich. Wie bereits erläutert, gibt es insbesondere kein festes Klassifikationsschema zur Verwendung in *categoryBag* Elementen. Auch die konkrete inhaltliche Ausgestaltung von *tModel* Elementen bleibt weitestgehend ungeregelt.

Obwohl über die UDDI-APIs eine Replikation von UDDI-Daten möglich ist, konnte sich bislang kein weltweit einheitliches Web-Service-Verzeichnis etablieren. Ein Pilotversuch der beiden führenden UDDI-Anbieter Microsoft und IBM wurde am 12. Januar 2006 ohne Angabe von Gründen abgebrochen. Beide Unternehmen haben ihren öffentlich zugänglichen UDDI-Dienst vollständig eingestellt. Lediglich die Firma SAP bietet zurzeit noch einen öffentlich zugänglichen (allerdings nicht-replizierten) UDDI-Dienst an (`http://uddi.sap.com`).

Angesichts dieser geringen Akzeptanz bleibt fraglich, ob UDDI für zukünftige Web-Service-Anwendungen überhaupt noch eine wesentliche Rolle spielen wird. Derzeit sind nach Kenntnis des Autors lediglich lokale UDDI-Dienste im praktischen Einsatz, d. h. Unternehmen, die diese Technologie einsetzen, unterhalten jeweils eigene Registries, die nicht mit denen anderer Unternehmen abgeglichen werden.

Es gibt bereits einige Forschungsarbeiten zu alternativen Ansätzen für Dienstregister. Zu nennen ist hier vor allem das Projekt „METEOR-S“ [153] der Universität Georgia. Hier werden semantisch reichhaltige Dienstbeschreibungen mit Hilfe einer Peer-to-Peer-Infrastruktur verteilt im Netzwerk gespeichert. Jedoch hat sich auch dieser Ansatz, vermutlich wegen seiner sehr hohen technologischen Komplexität, bislang nicht etablieren können.

2.5 Rollenmodell

Nachdem wir die wesentlichen Komponenten des Web-Service-Technology-Stacks auch unter technischen Gesichtspunkten kennen gelernt haben, stellt sich die Frage, wie diese Technologien zu einem sinnvollen Ganzen kombiniert werden können. In diesem Abschnitt erläutert der Autor die einzelnen Interaktionsvorgänge, die bei der Nutzung eines Web Services relevant sind, und setzt sie in einen zeitlichen Zusammenhang zueinander.

Die *W3C Web Service Architecture Working Group* hat in einem separaten Dokument [175] mehr als 30 typische Anwendungsbeispiele für Web Services zusammengestellt. Diese zeigen zunächst, dass sich mit Hilfe dieser Technologie sehr komplexe Interaktionsformen und Abhängigkeiten zwischen den beteiligten Systemen ergeben können. Allerdings ist auch ein regelmäßiges Muster in sämtlichen Kommunikationsvorgängen zu erkennen. Dieses Muster heißt *Web-Service-Rollenmodell* und ist in Abbildung 2.10 dargestellt.

Die in der Abbildung dargestellte Form des Web-Service-Rollenmodells wird in einer älteren Version des *W3C Web Service Architecture* Dokuments beschrieben [174]. Die aktuelle Version dieses Dokuments [181] stellt ebenfalls dieses Rollenmodell dar,

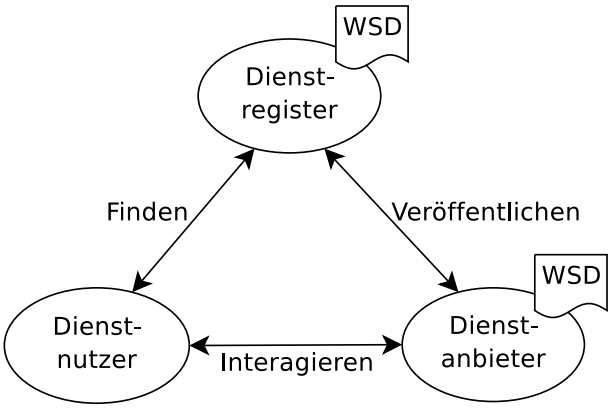


Abbildung 2.10: Das Web-Service-Rollenmodell

behandelt es aber in einer erweiterten Form, welche zusätzlich die Semantik eines Dienstes mit berücksichtigt. Überlegungen zur Semantik spielen im hier betrachteten Zusammenhang jedoch keine Rolle; daher wird hier die ältere und einfachere Darstellung erläutert.

Das Web-Service-Rollenmodell sieht vor, dass ein beteiligtes System drei verschiedene Rollen einnehmen kann: *Dienstnutzer (Service Requestor)*, *Dienstanbieter (Service Provider)* oder *Dienstverzeichnis (Discovery Agency)*. Diese Begriffe sind zunächst abstrakt zu verstehen, d. h. es kann sich hierbei um Einzelpersonen, Unternehmen oder auch Computer handeln. Zwischen diesen Systemen spielen sich die Prozesse *Veröffentlichen (Publish)*, *Finden (Find)* und *Interagieren (Interact)* ab.

Die Interaktionsprozesse zwischen diesen Systemen sind zeitlich wie folgt unterteilt: Im allgemeinen Fall kennen sich Dienstanbieter und Dienstnutzer vor Beginn des eigentlichen Kommunikationsvorgangs noch gar nicht, d. h. der Dienstnutzer muss zunächst einen geeigneten Dienstanbieter ausfindig machen. Dies geschieht über das Dienstverzeichnis, dessen Adresse als allgemein bekannt vorausgesetzt wird. Doch bevor ein Dienstnutzer einen Dienstanbieter finden kann, muss der Dienstanbieter seinen Dienst in einem ersten Interaktionsschritt beim Dienstregister veröffentlichen. Dies geschieht durch Bereitstellen eines *Web Service Description (WSD)* Dokuments, das den Dienst mit seinen charakteristischen Eigenschaften beschreibt. In einem zweiten Schritt kann der Dienstnutzer den Dienst finden und das zugehörige WSD-Dokument vom Dienstregister abrufen. In einem dritten Schritt folgt die eigentliche Interaktion zwischen Dienstnutzer und Dienstanbieter, hier wird der Dienst also aufgerufen und schließlich genutzt.

Dieses Rollenmodell ist, genau wie der eingangs behandelte Web Service Technology Stack, zunächst unabhängig von einer konkreten technologischen Umsetzung. Heutige Web-Service-Implementierungen verwenden hierfür in aller Regel die drei grundlegenden Technologien, die von den Web-Service-Standardisierungsgremien zur Umsetzung einer service-orientierten Architektur vorgesehen sind: WSDL, SOAP und UDDI.

UDDI dient dabei zum Auffinden und Veröffentlichen von Diensten, WSDL als Datenformat für WSD-Dokumente und SOAP als universelles Protokoll zur Interaktion zwischen Dienstnutzer und Dienstanbieter. Da UDDI auf SOAP aufsetzt, kommt SOAP zusätzlich auch als Basisprotokoll bei den Prozessen „Veröffentlichen“ und „Finden“ zum Einsatz.

In der Praxis wird dieses Rollenmodell jedoch in leicht abgewandelter Form umgesetzt: Wie bereits in Abschnitt 2.4.3 erläutert, konnte sich UDDI bisher kaum etablieren, und auch alternative Ansätze zur Implementierung von Dienstregistern haben sich bislang nicht durchgesetzt. Folglich können die Prozesse „Veröffentlichen“ und „Finden“ nicht mit Hilfe eines solchen Registers dynamisch zur Laufzeit einer Web-Service-Anwendung ablaufen, sondern müssen vom Entwickler manuell durchgeführt werden: Ein Dienstanbieter veröffentlicht das WSDL-Dokument seines Dienstes mit Hilfe herkömmlicher Datenaustauschmechanismen, beispielsweise über das WWW. Ein Dienstnutzer, also typischerweise der Programmierer einer Web-Service-Anwendung, lädt sich dann diese Beschreibung herunter und verarbeitet sie mit Hilfe eines geeigneten IDL-Compilers.

Ein gängiger WWW-Dienst zum Veröffentlichen von Web Services ist beispielsweise *xMethods* (`http://www.xmethods.com/`). Hier kann jeder Web-Service-Anbieter seine Dienste zusammen mit einer WSDL-Beschreibung registrieren und so öffentlich zugänglich machen. Alternativ ist auch der Austausch von WSDL-Dokumenten über E-Mail gebräuchlich.

Heutige Web-Service-Werkzeuge arbeiten also in aller Regel nicht mit UDDI, sondern verwenden nur WSDL und SOAP. Die Nutzung eines Web Services ähnelt dabei der Verwendung einer Funktionsbibliothek: Zur Entwicklungszeit wird ein geeigneter Dienst (zusammen mit seiner WSDL-Beschreibung) vom Programmierer gefunden. Dann erzeugt dieser mit Hilfe eines IDL-Compilers aus der WSDL-Beschreibung Quellcode, der die Kommunikationsroutinen zur Interaktion mit diesem Web Service bereitstellt. Der Programmierer integriert diese Quellcodefragmente in seine Anwendung und kann mit ihrer Hilfe komfortabel die Funktionen des Web Services nutzen – vergleichbar mit einer Funktionsbibliothek. Zur Laufzeit werden dann bei Nutzung dieser Funktionalitäten SOAP-Nachrichten mit dem eingebundenen Web Service ausgetauscht.

Somit repräsentiert SOAP klar die technologische Kernkomponente der Web-Service-Kommunikation im heutigen Sinne und bildet daher auch in dieser Arbeit den Ausgangspunkt für alle weiteren Überlegungen zur Verringerung des Datenaufkommens. SOAP hat sich inzwischen als universelles Protokoll zum XML-basierten Nachrichtenaustausch in weiten Bereichen etabliert. Sollten zukünftige Arbeiten Mechanismen hervorbringen, die auch das dynamische Auffinden von Diensten und deren Beschreibungen zur Laufzeit praktikabel machen, ist davon auszugehen, dass diese (ebenso wie UDDI) auf SOAP aufsetzen werden. Somit würden auch sie von Strategien zur effizienten Übertragung von SOAP-Nachrichten profitieren.

Bevor der Autor jedoch seine Ansätze zur Effizienzsteigerung vorstellt, behandelt er im folgenden Kapitel zunächst die theoretischen und konzeptionellen Grundlagen der Datenkompression. Denn diese stellen die Basis für die im Anschluss präsentierten Optimierungsansätze dar.

Kapitel 3

Grundlagen der Datenkompression

Die Idee, Kommunikationswege effizienter zu nutzen, indem man die Repräsentation der zu transportierenden Daten ändert, ist schon vergleichsweise alt. Bereits im Jahre 1837 erkannten Samuel Morse und seine Mitarbeiter, dass in englischsprachigen Texten die Buchstaben E und T besonders häufig vorkommen [57]. Bei der Entwicklung der später als Morse-Code bekannt gewordenen Zeichencodierung nutzten sie diese Entdeckung aus und berücksichtigten die Häufigkeiten von einzelnen Buchstaben bei der Konstruktion des Codes: Die Buchstaben E und T werden im Morse-Code jeweils durch ein einzelnes Codezeichen dargestellt, wohingegen alle anderen Buchstaben, Zahlen und Sonderzeichen aus einer Sequenz mehrerer Codezeichen bestehen. Die Codewortlänge steht dabei etwa im umgekehrten Verhältnis zur Häufigkeit des codierten Zeichens. Auf diese Weise werden häufig vorkommende Zeichen besonders kompakt dargestellt, was der Übertragungsgeschwindigkeit beim Morsen zugute kommt.

Zur weiteren Steigerung der Effizienz wurden bestimmte, häufig gebrauchte Sätze durch reservierte Wörter abgekürzt. Diese Technik hatte sich bereits Anfang des 19. Jahrhunderts kurz nach der Erfindung der Telegraphie etabliert, doch erst im Jahre 1912 wurde sie standardisiert[36]. Der so genannte *Q-Code* ist eine Sammlung reservierter Wörter, jeweils bestehend aus drei Buchstaben, die alle mit dem Buchstaben Q beginnen. Jedes dieser Wörter repräsentiert dabei einen ganzen Satz. So steht beispielsweise das Wort „QSL“ für „Ich bestätige den Empfang“.

Günstige Zeichencodierungen und auch solche reservierten Abkürzungswörter wurden dabei intuitiv konstruiert, ein mathematisches Modell gab es noch nicht. Bis zum Anfang des 20. Jahrhunderts war es somit mehr Kunst als Wissenschaft, effiziente Codes zu entwerfen.

Ab 1945 beschäftigte sich dann Claude Elwood Shannon mit einer mathematischen Beschreibung von Kommunikationsvorgängen. Er untersuchte unter anderem die Frage, welche Randbedingungen erfüllt sein müssen, um Nachrichten zuverlässig über gestörte Kanäle übertragen zu können. Shannon lieferte dabei eine mathematische Definition für den Informationsgehalt einer Nachricht und legte damit den Grundstein für alle weiteren Arbeiten auf dem Gebiet der Informationstheorie.

In diesem Kapitel erläutert der Autor zunächst die elementaren theoretischen Grundlagen der Datenkompression. Hierzu geht er zunächst auf ein Modell zur Darstellung von Kommunikationsvorgängen ein und erläutert dann verschiedene Techniken zur Quellencodierung. Der Quellencodierer ist im vorgestellten Modell die Komponente, die für eine kompakte Darstellung der zu übertragenden Daten sorgt.

3.1 Übertragung von Informationen

Der Shannon'schen Informationstheorie [135] liegt das in Abbildung 3.1 gezeigte Modell zu Grunde. Die Ausführungen hierzu basieren auf den Darstellungen in [57] und [96].

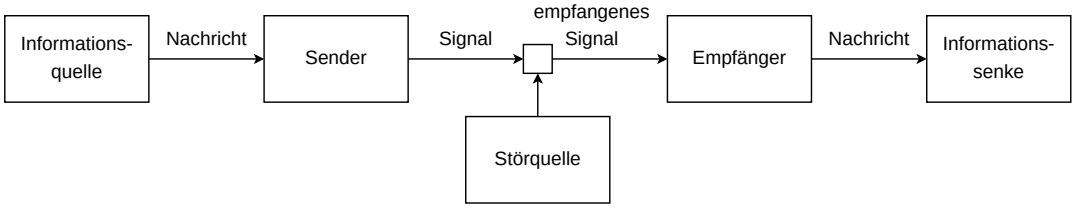


Abbildung 3.1: Schematisches Modell eines Kommunikationsprozesses

Die Informationsquelle ist dabei ein Mechanismus, der aus einer diskreten Wertemenge einzelne Elemente zufällig auswählt und diese sukzessive zum Bestimmungsort, der Informationssenke, schickt. Im Zustand der Übertragung bezeichnet man diese Sequenz von Werten als Nachricht.

Ein einfaches Beispiel für eine Informationsquelle ist ein Würfel. Durch Würfeln wird aus der Wertemenge $\{1, ..., 6\}$ ein Wert ausgewählt und zur Informationssenke, im Beispiel wären dies die Mitspieler des Würfelspiels, übertragen. Je nach Art des Würfelspiels kann eine Nachricht dabei aus einem oder auch mehreren Werten bestehen.

Der Begriff der Informationsquelle ist in diesem Modell absichtlich sehr allgemein gehalten – sie kann Elemente aus beliebigen diskreten Mengen ausgeben, also beispielsweise Buchstaben, physikalische Messwerte usw. Zur einfacheren Darstellung ist es in der Informationstheorie aber üblich, von diesem sehr allgemeinen Modell zu abstrahieren. Die Quelle gibt Zeichen aus, die durch eine diskrete Wertemenge repräsentiert werden. In unserem Beispiel verwenden wir die Werte aus der Menge $\{x_1, ..., x_6\}$.

Zwischen Informationsquelle und -senke liegt der Übertragungskanal. Hierunter versteht man ein Medium zur Übertragung physikalischer Signale. Typische Medien wären etwa elektrische Leiter oder auch, wie im oben genannten Beispiel eines Würfelspiels, Luft. Eine wesentliche Eigenschaft des Übertragungskanals ist, dass er die über ihn transportierten Signale durch Störungen verfälschen kann.

Um eine Information von der Quelle über den Übertragungskanal zur Senke transportieren zu können, muss die Repräsentation der von der Quelle gesendeten Zeichen an die physikalischen Eigenschaften des Übertragungskanals angepasst werden. Kann der verwendete Kanal beispielsweise nur elektromagnetische Wellen im Frequenzband von 350 bis 360 THz übertragen, so muss der Sender die von der Quelle ausgegebenen Zeichen auf Signalmuster in diesem Frequenzband abbilden. Bei dieser Umformungsoperation im Sender geht es darum, eine möglichst geeignete Repräsentation der zu übertragenden Information zu finden. Sie läuft in zwei Schritten ab.

Der erste Schritt ist die Quellencodierung. Hierunter versteht man Maßnahmen zur Reduzierung von Irrelevanz und Redundanz in der von der Quelle ausgesendeten Zeichenfolge: Repräsentieren die ausgesendeten Werte beispielsweise die Abtastwerte eines akustischen Signals, sind möglicherweise nur die von Menschen wahrnehmbaren Frequenzanteile für den Empfänger interessant, und folglich brauchen die übrigen, irrelevanten Informationsanteile nicht übertragen zu werden. Weiterhin kann die Repräsentation so geändert werden, dass sie besonders kurz wird und keine redundanten Informationen mehr enthält. Sendet die Informationsquelle beispielsweise sehr häufig das Zeichen a, aber nur selten die Zeichen b und c, so kann man diese Eigenschaft bei der Repräsentation der Zeichen berücksichtigen – diese Idee ist ja schon aus dem eingangs angeführten Beispiel des Morse-Codes bekannt. Eingabe des Quellencodierers ist also eine von der Quelle ausgesendete Zeichenfolge A, die Ausgabe ist eine neue Zeichenfolge B.

Der zweite Schritt heißt Kanalcodierung. Hierbei geht es vor allem darum, die Repräsentation so zu gestalten, dass die gesendeten Informationen auch dann vollständig zurückgewonnen werden können, wenn das gesendete Signal durch Störungen des Kanals verfälscht empfangen wird. Zusätzlich spielen in der Praxis oft noch technische Merkmale eine wichtige Rolle, wie etwa die Gleichspannungsfreiheit des erzeugten Signals oder die Möglichkeit zur Taktrückgewinnung. Der Kanalcodierer verarbeitet also die vom Quellencodierer ausgegebene Zeichenfolge B und liefert eine neue Zeichenfolge C, die schließlich durch Modulation an die physikalischen Eigenschaften des Kanals angepasst und dann übertragen wird.

Auf der anderen Seite des Kanals sorgt ein Empfänger für die Auswertung der übertragenen Signale und macht die Umsetzungen rückgängig.

3.2 Informationsgehalt und Redundanz

Jede Form der Datenkompression kann als Quellencodierungsproblem angesehen werden. Daher werden wir im Folgenden nur auf diese Komponente im Informationsübertragungssystem näher eingehen. Alle anderen Komponenten stehen mit Techniken zur Datenkompression nicht in direktem Zusammenhang und werden daher im Rahmen dieser Arbeit nicht weiter betrachtet.

Man unterscheidet zwischen verlustbehafteten und verlustfreien Kompressionsverfahren. Bei ersteren geht es um das Eliminieren von Informationen, die für die vorliegende Anwendung nicht relevant sind (Reduzierung der Irrelevanz). Bei letzteren werden lediglich redundante Informationen aus dem Datenstrom entfernt. Wir konzentrieren uns in dieser Arbeit auf die verlustfreie Kompression, da bei der Web-Service-Kommunikation typischerweise alle ausgesendeten Daten erhalten bleiben sollen.

Bei der Datenkompression liegt eine Zeichenfolge A aus dem Quellalphabet X vor, die vom Quellencodierer in eine neue, möglichst kompakte Zeichenfolge B aus dem Zielalphabet Y umgewandelt wird. Dabei soll die zweite Zeichenfolge alle Informationen aus der ersten enthalten. Zur Lösung dieses Problems muss zunächst der Begriff der Information mathematisch beschrieben werden.

Die von der Informationsquelle gesendeten Zeichen entstammen dem Zeichenvorrat des Quellalphabets X:

$$X = \{x_0, x_1, \ldots, x_{n-1}\} \tag{3.1}$$

In der Informationstheorie wird die Informationserzeugung als ein Zufallsprozess verstanden, d. h. die Informationsquelle wählt das zu sendende Zeichen zufällig aus dem Zeichenvorrat aus. $P(x_i)$ bezeichnet dabei die Wahrscheinlichkeit, mit der das Zeichen $x_i \in X$ ausgewählt wird. Da die Quelle immer irgendein Zeichen aus dem Alphabet auswählt, gilt:

$$\sum_{i=0}^{n-1} P(x_i) = 1 \tag{3.2}$$

Definition 3.1 (Informationsgehalt, Entropie): Sei $x_i \in X$ ein von einer Informationsquelle ausgegebenes Zeichen. Der Wert

$$H(x_i) = \log_b \frac{1}{P(x_i)} = -\log_b P(x_i) \tag{3.3}$$

heißt *Informationsgehalt* oder *Entropie* von x_i.

Die Einheit, die der Informationsgehalt trägt, ist dabei abhängig von der verwendeten Logarithmusbasis b. Bei der Verwendung der Basis e heißt die Einheit beispielsweise *natural digit (Nit)*, bei Verwendung der Basis 10 *digital digit (Dit)*. Im Folgenden gehen wir von der Basis $b = 2$ aus, da dies die gängigste Darstellungsform ist. Die Werte von $H(x_i)$ tragen dann die Einheit *binary digit (Bit)*.

Die Definition des Informationsgehalts lässt sich wie folgt interpretieren: Die Informationsquelle wählt mit der Wahrscheinlichkeit $P(x_i)$ das Zeichen x_i aus dem Zeichenvorrat X aus. Der Informationssenke ist der Ausgang dieses Zufallsexperiments noch nicht bekannt, so dass ein gewisser Grad an Unsicherheit besteht. Durch den Kommunikationsprozess wird bei der Senke diese Unsicherheit beseitigt. Der Informationsgehalt kann also als Maß angesehen werden, wie viel Unsicherheit bei der Senke

beseitigt wird. Nach Definition 3.1 steht dieses Maß im umgekehrten Verhältnis zu $P(x_i)$.

Besonders deutlich wird der Sinn dieser reziproken Abhängigkeit bei einer Grenzwertbetrachtung: Liegt $P(x_i)$ nahe 1, d. h. die Quelle wählt sehr häufig x_i aus, wird bei der Senke nur sehr wenig Unsicherheit beseitigt. Schließlich ist dieses Ergebnis zu erwarten. Liegt dagegen $P(x_i)$ nahe 0, so ist der Informationsgehalt sehr hoch. Dieses Ergebnis tritt nur sehr selten auf, dementsprechend wird durch die Kenntnis dieses Ergebnisses ein hohes Maß an Unsicherheit bei der Senke beseitigt.

Durch die Definition des Informationsgehalts als logarithmische Maßzahl kann der Informationsgehalt als Anzahl von Fragen mit b möglichen Antworten interpretiert werden, die die Senke brauchen würde, um das Ergebnis des Zufallsexperiments bei der Quelle zu erfragen. Ein Informationsgehalt von 3 Bits bedeutet also anschaulich, dass die Senke bei optimaler Fragestrategie dreimal eine Frage mit zwei möglichen Antworten stellen muss, um die bestehende Unsicherheit zu beseitigen.

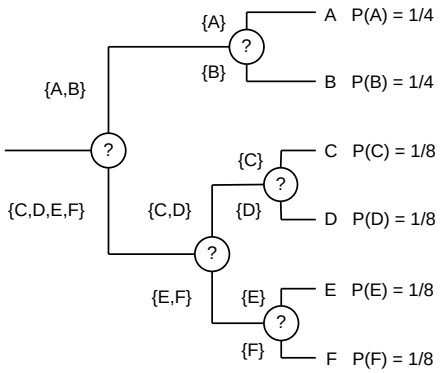


Abbildung 3.2: Veranschaulichung des Entropiebegriffs anhand eines Entscheidungsbaums

Abbildung 3.2 verdeutlicht diesen Zusammenhang anhand eines Beispiels: Die Quelle sendet ein Zeichen aus dem Zeichenvorrat $X = \{A, B, C, D, E, F\}$, die Wahrscheinlichkeiten für die einzelnen Zeichen sollen dabei $P(A) = P(B) = \frac{1}{4}$ und $P(C) = P(D) = P(E) = P(F) = \frac{1}{8}$ betragen. Die Senke erfragt nun dieses Zeichen bei der Quelle mit einer optimalen Fragestrategie. Um die Fragenanzahl möglichst gering zu halten, fragt die Senke nach dem Prinzip der Intervallschachtelung und beseitigt mit jeder Frage möglichst viel Unsicherheit. Die Senke teilt dazu die Menge der möglichen Antworten vor jeder Frage in zwei Hälften, und zwar jeweils so, dass die Summe der Wahrscheinlichkeiten für die Werte in den beiden Hälften gleich ist. Wegen $P(A)+P(B) = 0,5$ und $P(C)+P(D)+P(E)+P(F) = 0,5$ lautet eine mögliche Frage: „Liegt das gesendete Zeichen in der Menge $\{A, B\}$ oder in der Menge $\{C, D, E, F\}$?“. In Abhängigkeit von der Antwort fragt die Senke nach derselben Strategie weiter.

Wie in der Abbildung zu erkennen ist, benötigt die Senke zwei Fragen, um die Werte A oder B zu erfragen. Der Informationsgehalt beträgt also $H(A) = H(B) = \log_2 \frac{1}{\frac{1}{4}} = 2$ Bits. Für die übrigen Werte benötigt die Senke drei Fragen. Es ergibt sich ein Informationsgehalt von $H(C) = H(D) = H(E) = H(F) = \log_2 \frac{1}{\frac{1}{8}} = 3$ Bits.

Über den in Definition 3.1 angegebenen Zusammenhang wird demnach der Informationsgehalt eines einzelnen Zeichens bestimmt. Eine weitere wichtige Maßzahl beschreibt den Informationsgehalt der Informationsquelle als Ganzes.

Definition 3.2 (Entropie der Quelle): Der Wert

$$H(X) = \sum_{i=0}^{n-1} P(x_i) H(x_i) \text{ [Bits/Zeichen]} \tag{3.4}$$

heißt Entropie der Quelle.

Mit Formel 3.3 ergibt sich:

$$H(X) = \sum_{i=0}^{n-1} P(x_i) \log_2 \frac{1}{P(x_i)} = -\sum_{i=0}^{n-1} P(x_i) \log_2 P(x_i) \text{ [Bits/Zeichen]} \tag{3.5}$$

Die Entropie der Informationsquelle ist damit definiert als das gewichtete arithmetische Mittel aller Einzelentropien. Da dieser Wert den durchschnittlichen Informationsgehalt pro Zeichen angibt, trägt er die Einheit [Bits/Zeichen].

Wie in [96] gezeigt wird, ergibt sich ein maximaler Wert für die Entropie der Quelle genau dann, wenn alle Quellenzeichen gleich wahrscheinlich sind ($P(x_i) = \frac{1}{n}$ mit $i \in \{0, ..., n-1\}$):

$$H(X) = \sum_{i=0}^{n-1} \frac{1}{n} \log_2 \frac{1}{\frac{1}{n}} = H(X) = \sum_{i=0}^{n-1} \frac{1}{n} \log_2 n = \log_2 n \text{ [Bits/Zeichen]} \tag{3.6}$$

Definition 3.3 (maximale Entropie): Der Wert

$$H_{max}(X) = \log_2 n \text{ [Bits/Zeichen]} \tag{3.7}$$

heißt maximale Entropie.

Definition 3.4 (Redundanz): Die Differenz zwischen der maximalen Entropie und der Entropie der Quelle heißt Redundanz:

$$R(X) = H_{max}(X) - H(X) \text{ [Bits/Zeichen]} \tag{3.8}$$

Es sei darauf hingewiesen, dass der Autor die Begriffsdefinition für *Redundanz* lediglich der Vollständigkeit halber mit angegeben hat, denn dieser Begriff wird in der Literatur zum Thema Datenkompression sehr häufig verwendet. Für die weiteren Ausführungen im Rahmen dieser Arbeit ist er allerdings nicht zwingend erforderlich. Mit Blick auf eine möglichst kompakte Darstellung verwenden wir im Folgenden ausschließlich den Entropiebegriff.

3.3 Codes

Auf Basis der mathematischen Modellierung des Informationsbegriffs ist es nun möglich, den Informationsgehalt (Entropie) einer Informationsquelle zu bewerten. Bei der Datenkompression soll durch Umcodierung einer Zeichenfolge eine neue, kürzere Repräsentation entstehen. Wie wir sehen werden, gibt es einen direkten Zusammenhang zwischen der Entropie der Quelle und der minimal möglichen Länge dieser neuen Darstellung. Bevor ein Verfahren zur Konstruktion solcher optimalen Codierungen vorgestellt wird, benötigen wir jedoch noch einige formale Beschreibungen für grundlegende Begriffe (nach [92]):

Definition 3.5 (Code, Codierung): Seien $X = \{x_0, \ldots, x_{n-1}\}$ ein Quellalphabet, $Y = \{y_0, \ldots, y_{m-1}\}$ ein Zielalphabet und

$$c : X \to Y^+ : x_i \to (y_{i_1}, \ldots, y_{i_l}) \tag{3.9}$$

eine injektive Abbildung. Dabei bezeichnet $Y^+ = \bigcup_{l=1}^{\infty} Y^l$ die Menge aller Wörter über Y mit einer Länge l größer oder gleich 1. Die Abbildung c heißt Code oder Codierung. Das Bild dieser Abbildung heißt Codewortmenge. Ein Element $c(x_i) = (y_{i_1}, \ldots, y_{i_l})$ aus der Codewortmenge heißt Codewort vom Zeichen x_i; seine Codewortlänge ist l. Die Umkehrabbildung c^{-1} heißt Decodierung.

Leider ergibt sich aus der Injektivität der Codierung nur, dass die Umkehrabbildung eindeutig ist. Dies bedeutet aber nicht, dass auch jede zusammengesetzte Codewortfolge eindeutig decodierbar ist. Ein Beispiel verdeutlicht dies: Betrachten wir eine vereinfachte Variante des Morse-Codes mit

$X = \{a, d, e, i, s, u\}$,
$Y = \{\cdot, -\}$ und
$c = \{(a, \cdot-), (d, -\cdot\cdot), (e, \cdot), (i, \cdot\cdot), (s, \cdot\cdot\cdot), (u, \cdot\cdot-)\}$.

Die Wörter *idea* und *usa* werden beide auf die gleiche Codewortfolge abgebildet: $\cdot\cdot - \cdot\cdot\cdot - \cdot$

Folglich ist dieser Code nicht eindeutig decodierbar.

Zur Überwindung dieses Problems muss sichergestellt werden, dass die einzelnen Wörter aus $c(X)$ in einer zusammengesetzten Codewortfolge eindeutig identifiziert werden können. Beim Morse-Code erreicht man dies durch die Erweiterung der Menge Y um ein weiteres Alphabetszeichen: $Y = \{\cdot, -, \_\}$. Das Zeichen _ steht dabei für eine Sendepause, die beim Morsen hinter jedem Codewort eingefügt wird und somit zwei Morse-Codewörter klar voneinander trennt. Mit dieser Erweiterung wird der Code eindeutig decodierbar.

Die Einführung eines reservierten Trennzeichens zwischen einzelnen Codewörtern ist sicherlich dann sinnvoll, wenn ein Mensch ein zusammengesetztes Codewort decodie-

ren muss. Ein Trennzeichen macht es schließlich besonders leicht, die Anfänge der einzelnen Codewörter zu erkennen. Für Datenkompressionsaufgaben im Allgemeinen ist eine solche Erweiterung des Alphabets Y jedoch nicht sinnvoll, da ein zusätzliches Trennzeichen die Codewortlänge offensichtlich vergrößert.

In der Tat ist eine Erweiterung des Zielalphabets nicht immer notwendig. Für technische Aufgaben eingesetzte Codes kommen in aller Regel ohne ein solches reserviertes Trennzeichen aus. Robert M. Fano hat ein Kriterium formuliert, um die Decodierbarkeit eines Codes sicherzustellen.

Satz 3.1 (Fano-Bedingung): Ein Code c ist eindeutig decodierbar, wenn gilt:

$$\neg\exists\, c(x_i), c(x_j) \;:\; c(x_i) \text{ ist Anfang von } c(x_j) \wedge i \neq j$$

Den Beweis dieses Satzes findet der Leser in [92]. Ein Code, der der Fano-Bedingung genügt, heißt auch *präfixfrei* oder *Präfixcode*.

Es sei angemerkt, dass die Umkehrung dieses Satzes nicht gilt [54]. Es gibt also durchaus decodierbare Codes, die nicht der Fano-Bedingung genügen. In [90] wird jedoch gezeigt, dass jeder decodierbare Code durch einen Präfixcode mit gleichen Codewortlängen ersetzt werden kann.

In technischen Systemen wählt man zur Informationsübertragung in aller Regel Präfixcodes. Neben dem besonders einfachen Nachweis der eindeutigen Decodierbarkeit haben diese eine wichtige zusätzliche Eigenschaft. Bei ihnen ist es möglich, mit der Decodierung einer Codewortfolge schon zu beginnen, bevor die gesamte Nachricht beim Empfänger eingegangen ist [127].

3.4 Effizienz von Codes

Besonders interessant ist die Frage nach der Effizienz eines Codes, d. h. wie kompakt sich mit ihm die Informationen einer gegebenen Informationsquelle darstellen lassen. Als Gütekriterium definieren wir zunächst die mittlere Codewortlänge.

Definition 3.6 (mittlere Codewortlänge): Für einen Code c heißt der Wert

$$L_m(c) = \sum_{i=0}^{n-1} L(c(x_i))P(x_i) \text{ [Bits/Zeichen]} \tag{3.10}$$

mittlere Codewortlänge. Hierbei bezeichnet $L(c(x_i))$ die Länge des Codewortes für das Zeichen x_i.

Shannon konnte zeigen, dass es einen direkten Zusammenhang zwischen der Entropie der Quelle $H(X)$ und der kleinsten, erreichbaren mittleren Codewortlänge $L_m(c)$ gibt.

Satz 3.2 (Satz von Shannon): : Es lässt sich ein eindeutig decodierbarer Code c immer so konstruieren, dass die Beziehung gilt:

$$H(X) \leq L_m(c) < H(X) + 1 \tag{3.11}$$

Die mittlere Codewortlänge $L_m(c)$ kann demnach niemals kleiner werden, als die Entropie der Quelle. Ferner lässt sich immer ein Code konstruieren, dessen mittlere Codewortlänge kleiner ist als $H(X) + 1$ (Beweise siehe [96]).

Auf der Basis der mittleren Codewortlänge lässt sich weiterhin ein Optimalitätskriterium formulieren:

Definition 3.7 (optimaler Code): Sei c ein eindeutig decodierbarer Code, und sei C' die Menge aller eindeutig decodierbaren Codes, welche dasselbe Urbild und dasselbe Zielalphabet wie c haben. Der Code c heißt *optimal*, wenn gilt:

$$\neg\exists\, c' \in C' \;:\; L_m(c') < L_m(c)$$

Anschaulich bedeutet dies, dass es keinen „gleichwertigen“ Code gibt, dessen mittlere Codewortlänge noch kleiner ist.

3.5 Modellbildung und Codierung

Bereits kurz nachdem Shannon die mathematischen Grundlagen für die optimale Codierung von Nachrichten gelegt hatte, begannen viele Wissenschaftler mit der Entwicklung von Verfahren zur Konstruktion solcher Codes.

Heute, rund 60 Jahre später, existieren mehr als 20 [132, 83] grundlegende Datenkompressionsverfahren; die meisten von ihnen wurden im Rahmen von mehreren Forschungsarbeiten schrittweise weiterentwickelt und für spezielle Anwendungen optimiert, so dass zusätzliche Varianten entstanden sind.

Es ist zunächst erstaunlich, dass zu einer Problemstellung mit einem derartig präzisen mathematischen Fundament so viele Forschungsarbeiten durchgeführt wurden. Tatsächlich beschränkt sich das Themenfeld der Datenkompression jedoch nicht auf die Konstruktion von Codes mit minimalen mittleren Codewortlängen; diese Problemstellung wurde bereits kurz nach der Entstehung der Informationstheorie in Arbeiten von Shannon und Fano [43] sowie von Huffman [59] umfassend behandelt und algorithmisch gelöst. So ermöglicht etwa das Verfahren von Huffman, das im nächsten Abschnitt anhand einer ausführlichen Beispielrechnung exemplarisch vorgestellt wird, die Konstruktion eines optimalen Präfixcodes.

Der Grund, warum dieses Forschungsgebiet immer wieder interessante und neue Fragestellungen hervorbringt, liegt in den Eigenschaften der Informationsquelle. Die Informationstheorie geht zunächst davon aus, dass die Quelle Zeichen aussendet und die

Auswahl der Zeichen dabei nach einem perfekten Zufallsprozess abläuft. Bei realen Datenquellen trifft dieses Modell jedoch häufig nicht zu.

Denkt man beispielsweise an eine Informationsquelle, die deutschsprachige Texte als Zeichenfolge ausgibt, so ist die Auswahl eines zu sendenden Zeichens eben kein statistisch unabhängiger Prozess. Wurde als letztes Zeichen etwa ein *c* ausgegeben, so ist es sehr unwahrscheinlich, dass ein *z* folgt. Obwohl der Buchstabe *z* in deutschen Texten nicht selten vorkommt, ist er hinter einem c sehr ungewöhnlich. Die Wahrscheinlichkeit (und damit auch der Informationsgehalt) von *z* ändert sich also in Abhängigkeit von den zuvor gesendeten Zeichen. Somit ist das Senden des Buchstabens *z* kein statistisch unabhängiges Ereignis. Solche Abweichungen vom Modell eines perfekten Zufallsprozesses lassen sich bei der Datenkompression zusätzlich ausnutzen.

Folglich ist das Erzeugen eines optimalen Codes nur ein Teilprozess bei der Datenkompression. Man unterscheidet insgesamt zwei Schritte: In einem ersten müssen zunächst die charakteristischen Eigenschaften einer Datenquelle erfasst werden. Hierzu gehört insbesondere auch die Trennung von relevanten und nicht relevanten Informationsanteilen, denn in vielen Fällen sind nicht alle von der Informationsquelle ausgesandten Informationen für den Empfänger relevant. Diesen Prozess bezeichnet man als *Modellierung* der Informationsquelle (engl. *modeling*).

Das Finden einer möglichst kurzen Codierung im Zielalphabet nach den Gesetzmäßigkeiten der Informationstheorie ist der zweite Schritt. Ihn bezeichnet man als *Entropiecodierung* (engl. *entropy coding*).

Praktisch genutzte Verfahren zur Datenkompression decken in aller Regel beide Schritte ab. Jedoch lassen sich diese nicht immer klar voneinander trennen, denn mitunter hängt eine zweckmäßige Codierung auch von den spezifischen Eigenschaften der Quelle bzw. dem gewählten Modell ab.

Für die reine Entropiecodierung existieren im Wesentlichen zwei Basisverfahren, nämlich der Algorithmus von Huffman sowie der Ansatz der arithmetischen Codierung. Beide Verfahren liefern bereits das theoretisch bestmögliche Ergebnis, und folglich sind neue Ansätze zur Entropiecodierung nicht Gegenstand aktueller Forschungsarbeit.

Die Modellierung von Informationsquellen ist dagegen nach wie vor ein sehr aktives Forschungsgebiet, schließlich entstehen noch immer neue „Informationsarten“, wie Mehrkanal-Audio in der Unterhaltungselektronik oder auch XML als universelle Datenbeschreibungssprache. Für diese neuen Informationsquellen benötigt man folglich neue bzw. angepasste Modelle.

Im einfachsten Fall übernimmt der Programmierer einer Software zur Datenkompression die Modellbildung. Dies ist immer dann zweckmäßig, wenn sich die Eigenschaften der Quelle nicht mehr ändern. Bei vielen praktisch genutzten Verfahren zur Datenkompression geschieht die Anpassung an die Eigenschaften der Quelle jedoch permanent während des Kompressionsvorgangs. Hier kommen Algorithmen zum Einsatz,

die bei der Verarbeitung des zu komprimierenden Datensatzes Informationen über die Quelle sammeln und diese bei der Codierung berücksichtigen. Solche Ansätze fasst man unter dem Begriff *adaptive Verfahren* zusammen.

Wie bereits eingangs erwähnt, ist das Forschungsgebiet der Datenkompression äußerst umfangreich. Daher verzichtet der Autor auf eine ausführliche Darstellung einzelner Verfahren. Eine umfassende Übersicht und detaillierte Erläuterungen zu gängigen Algorithmen findet der Leser in [132].

In den folgenden beiden Abschnitten zeigt der Autor lediglich zwei ausgewählte Beispiele für Datenkompressionsstrategien.

In Abschnitt 3.6 geht es zunächst um das Verfahren von Huffman; es setzt das Modell einer perfekt zufälligen Informationsquelle voraus. Auf dieser Basis erzeugt das Verfahren optimale Codes und ist daher Bestandteil von vielen anderen Datenkompressionsverfahren, die dem Huffman-Algorithmus eine Vorverarbeitungsstufe zur Modellierung voranstellen. Ein bekanntes Beispiel dafür ist das Kompressionsprogramm bzip2. Bei diesem wird zunächst der Blocksortierungsalgorithmus von Burrows und Wheeler [16] angewendet, um die Eigenschaften von typischen Informationsquellen an die einer perfekt zufälligen Quelle anzupassen. Nach einigen weiteren Umformungsschritten kommt das Verfahren von Huffman zum Einsatz, um das Ergebnis der Blocksortierung effizient zu codieren.

In Abschnitt 3.7 wird dann die Differenzcodierung als ein möglicher Ansatz der Modellbildung für eine Informationsquelle beschrieben. In der Praxis wird dieses Modell einer Informationsquelle immer dann eingesetzt, wenn diese eine Sequenz von ähnlichen Zeichen oder von ähnlichen Zeichenketten sendet.

3.6 Huffman-Codierung

Das Verfahren von Huffman erzeugt für die Zeichen aus einem beliebigen, endlichen Quellalphabet einen präfixfreien Binärcode. Huffman zeigt in [59], dass sein Algorithmus immer einen optimalen Code erzeugt, d. h. es gibt keinen eindeutig decodierbaren Code mit einer kleineren mittleren Codewortlänge.

Bei der Konstruktion des Codes berücksichtigt der Algorithmus die Auftrittswahrscheinlichkeiten $P(x_i)$ der einzelnen Zeichen x_i. Die grundlegende Idee besteht darin, x_i mit großen Auftrittswahrscheinlichkeiten mit kurzen Codewörtern zu codieren, wohingegen solche mit kleinen Auftrittswahrscheinlichkeiten mit längeren codiert werden.

Wie wir später sehen werden, erzeugt der Algorithmus schrittweise einen vollständigen Binärbaum. Die besonderen Eigenschaften dieser Datenstruktur führen dazu, dass die Codewortlänge für jedes x_i in etwa dem zugehörigen Entropiewert $H(x_i)$ entspricht. Im Idealfall bedeutet dies, dass die mittlere Codewortlänge $L_m(c)$ mit der Entropie

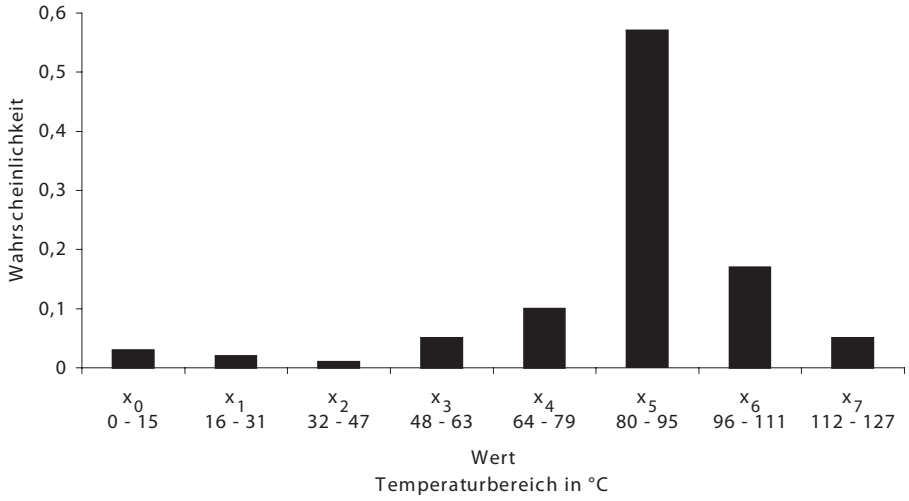


Abbildung 3.3: Wahrscheinlichkeitsverteilung der quantisierten Messwerte

der Quelle $H(X)$ übereinstimmt, was nach Satz 3.2 auch der unteren Schranke für einen optimalen Code entspricht.

Leider funktioniert diese Idee in der Praxis nur in den seltensten Fällen perfekt, weil die Entropie $H(x_i)$ in aller Regel kein ganzzahliger Wert ist, Codewortlängen aber immer ganzzahlig sein müssen – in technischen Systemen lassen sich schließlich keine Bitfragmente darstellen. Der Huffman-Algorithmus muss daher Codewörter so zuweisen, dass deren Länge gerundeten Entropiewerten entsprechen. Insgesamt liegt der entstehende Rundungsfehler gemittelt über alle Zeichen allerdings unter 1 Bit/Zeichen, so dass auch die obere Schranke aus dem Satz von Shannon immer eingehalten wird.

Im Folgenden wird der Algorithmus anhand eines praktischen Beispiels erläutert: Im Motor eines Fahrzeugs wird die Kühlwassertemperatur in äquidistanten Zeitintervallen gemessen. Die zur Signalerfassung eingesetzte Baugruppe erfasst den Temperaturbereich von 0 bis 127 °C und arbeitet mit acht Quantisierungsstufen. Sie liefert folglich acht verschiedene Messwerte, die mit x_0 bis x_7 bezeichnet werden. Abbildung 3.3 zeigt die Verteilung der zugehörigen Auftrittswahrscheinlichkeiten.

Nach den Zusammenhängen aus Abschnitt 3.2 lässt sich anhand dieser Werte nun der Informationsgehalt jedes einzelnen Messwertes ermitteln, Tabelle 3.1 zeigt die Ergebnisse. Sie zeigt weiterhin, dass sich der Wert für die Entropie der Quelle als $1,99$ Bits/Zeichen ergibt (vergleiche Formel 3.4 auf Seite 48). Mit dem Satz von Shannon lässt sich nun folgern, dass ein binärer Präfixcode existiert, dessen mittlere Codewortlänge im Bereich zwischen $1,99$ und $2,99$ Bits/Zeichen liegt.

Das Verfahren von Huffman konstruiert einen solchen Code. Der Autor führt das Verfahren in einer besonders anschaulichen Variante nach [31] vor, die auf einer Baumstruktur operiert. Der Originalalgorithmus [59] arbeitet dagegen mit einer tabellari-

Messwert x_i	Wahrscheinlichkeit $P(x_i)$	Informationsgehalt $H(x_i)$	$P(x_i) \cdot H(x_i)$
x_0	$0,03$	$5,06$	$0,15$
x_1	$0,02$	$5,64$	$0,11$
x_2	$0,01$	$6,64$	$0,07$
x_3	$0,05$	$4,32$	$0,22$
x_4	$0,10$	$3,32$	$0,33$
x_5	$0,57$	$0,81$	$0,46$
x_6	$0,17$	$2,56$	$0,43$
x_7	$0,05$	$4,32$	$0,22$
			$H(X) = 1,99$

Tabelle 3.1: Informationsgehalt in Abhängigkeit von der Auftrittswahrscheinlichkeit für alle x_i

schen Darstellung der Berechnungsschritte. Dieses Vorgehen ist jedoch weniger anschaulich, vor allem im Hinblick auf die Erzeugung der eigentlichen Bitcodierung am Schluss der Berechnung. Beide Varianten sind in der Fachliteratur aber durchaus gängig und arbeiten völlig äquivalent.

Die hier vorgestellte Variante des Algorithmus konstruiert nach einem Bottom-Up-Verfahren einen Baum, der die gesuchte Codierungsvorschrift repräsentiert. Wie in Algorithmus 1 zu erkennen ist, lässt sich der Huffman-Algorithmus mit wenigen Zeilen Pseudocode darstellen. Der Leser kann die Berechnungsschritte auch zusätzlich anhand der Abbildungen 3.4 und 3.5 nachvollziehen, die das Vorgehen mit den Werten unserer Beispielanwendung veranschaulichen.

Algorithmus 1 Erzeugung eines Huffman-Codes

```
1: for all x ∈ X do
2:     sortedQueue.add(new Node(P(x))
3: end for
4: while (|sortedQueue| > 1) do
5:     node1 ← sortedQueue.getFirst()
6:     sortedQueue.remove(node1)
7:     node2 ← sortedQueue.getFirst()
8:     sortedQueue.remove(node2)
9:     new_node ← new Node(node1.getMark() + node2.getMark())
10:    new_node.setLeftChild(node1)
11:    new_node.setRightChild(node2)
12:    new_node.markLeftEdge(0)
13:    new_node.markRightEdge(1)
14:    sortedQueue.add(new_node)
15: end while
```

Informationsquelle				Code		
x_i	$P(x_i)$	$H(x_i)$	$P(x_i) \cdot H(x_i)$	$c_1(x_i)$	$L(c_1(x_i))$	$P(x_i) \cdot L(c_1(x_i))$
x_0	0,03	5,06	0,15	01101	5	0,15
x_1	0,02	5,64	0,11	011001	6	0,12
x_2	0,01	6,64	0,07	011000	6	0,06
x_3	0,05	4,32	0,22	01110	5	0,25
x_4	0,10	3,32	0,33	010	3	0,30
x_5	0,57	0,81	0,46	1	1	0,57
x_6	0,17	2,56	0,43	00	2	0,34
x_7	0,05	4,32	0,22	01111	5	0,25
			$H(X) = 1,99$			$L_m(c_1) = 2,04$

Tabelle 3.2: Codewörter und Codewortlängen des Huffman-Codes c_1

Zunächst wird für alle zu codierenden Zeichen x_i aus dem Eingabealphabet X jeweils ein Blattknoten erzeugt und mit der zugehörigen Auftrittswahrscheinlichkeit $P(x_i)$ markiert. Sämtliche Knoten werden dabei in eine geeignete Datenstruktur eingefügt, wir arbeiten hier mit einer sortierten Warteschlange (engl.: sorted queue). Sie muss sicherstellen, dass zu jedem Zeitpunkt alle Elemente sortiert sind – und zwar aufsteigend nach ihren Markierungen. Diese Warteschlange ist in den Abbildungen 3.4 und 3.5 grau dargestellt.

Nach der Erzeugung der Blattknoten werden diese nun sukzessive miteinander verbunden, bis ein zusammenhängender Baum entstanden ist: In jedem Iterationsschritt betrachtet man zunächst die beiden vordersten Knoten in der Warteschlange, also die mit den kleinsten Markierungen. In zwei temporären Variablen (in Algorithmus 1 mit node1 und node2 bezeichnet) speichert man Referenzen auf diese Knotenobjekte zwischen und löscht sie aus der Warteschlange. Dann erzeugt man einen neuen Knoten (new_node) und markiert ihn mit der Summe der Markierungen von node1 und node2. Der neu erzeugte Knoten wird nun über zwei neue Kanten mit node1 und node2 verbunden. Die linke Kante wird dabei mit Null und die rechte mit Eins markiert. Schließlich wird der neu erzeugte Knoten in die Warteschlange eingefügt. Diesen Vorgang wiederholt man solange, bis nur noch ein Knoten in der Warteschlange vorhanden ist.

Das verbleibende Element ist der Wurzelknoten von einem vollständigen Binärbaum, dessen Kanten mit Nullen und Einsen markiert sind und dessen Blätter die Knoten sind, die anfangs aus den Zeichen des Quellalphabets erzeugt wurden (vergleiche Abbildung 3.5). Diesen Binärbaum bezeichnet man auch als Huffman-Baum. Die Codierungsvorschrift für jedes einzelne Zeichen $x_i \in X$ erhält man, indem man ihn von der Wurzel zum entsprechenden Blatt durchläuft. Die Markierungen der traversierten Kanten bilden die Bitsequenz für x_i im erzeugten Huffman-Code, im Folgenden mit c_1 bezeichnet. Tabelle 3.2 stellt die Ergebnisse für unser Beispiel in der fünften Spalte dar.

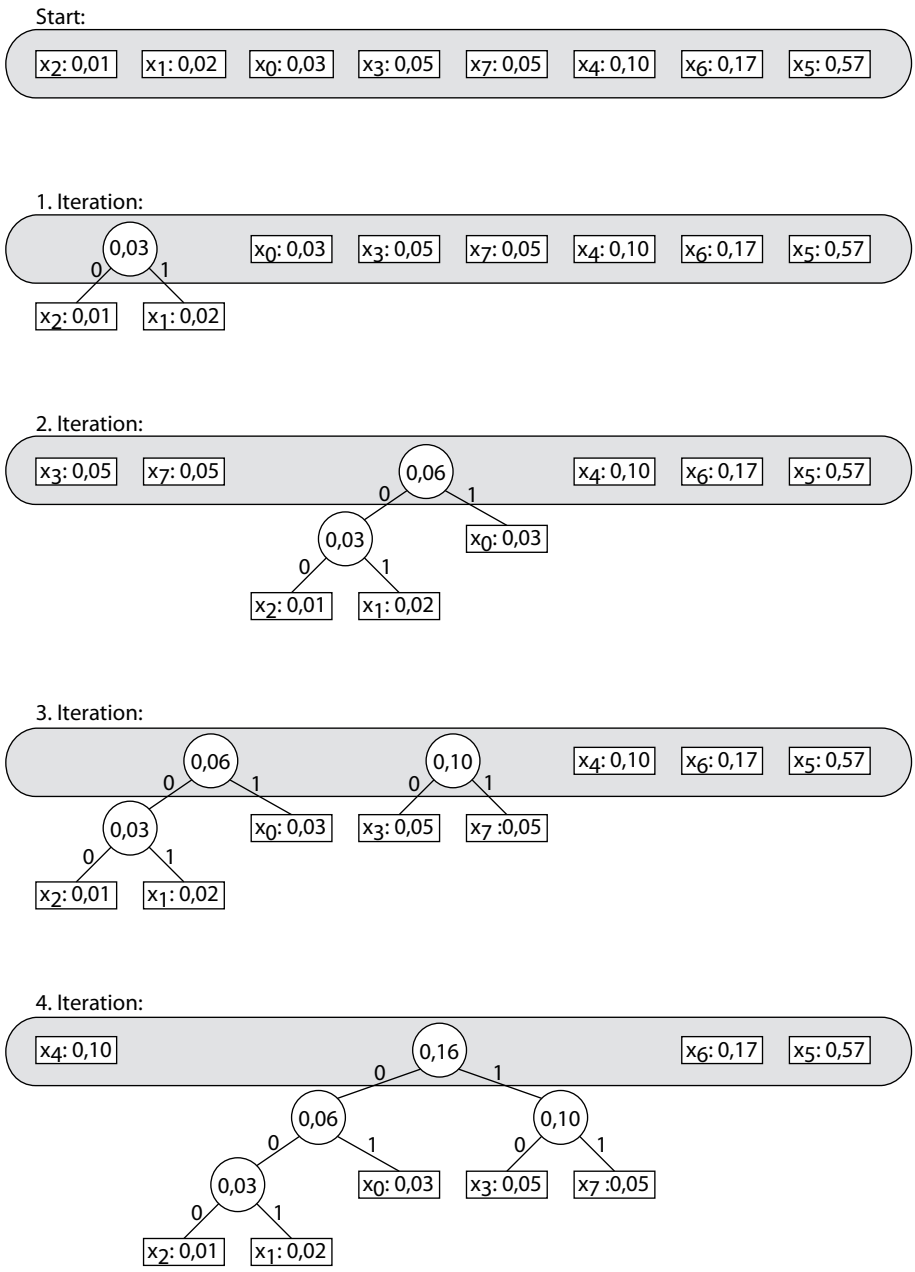


Abbildung 3.4: Ablauf des Huffman-Algorithmus (Start bis 4. Iteration)

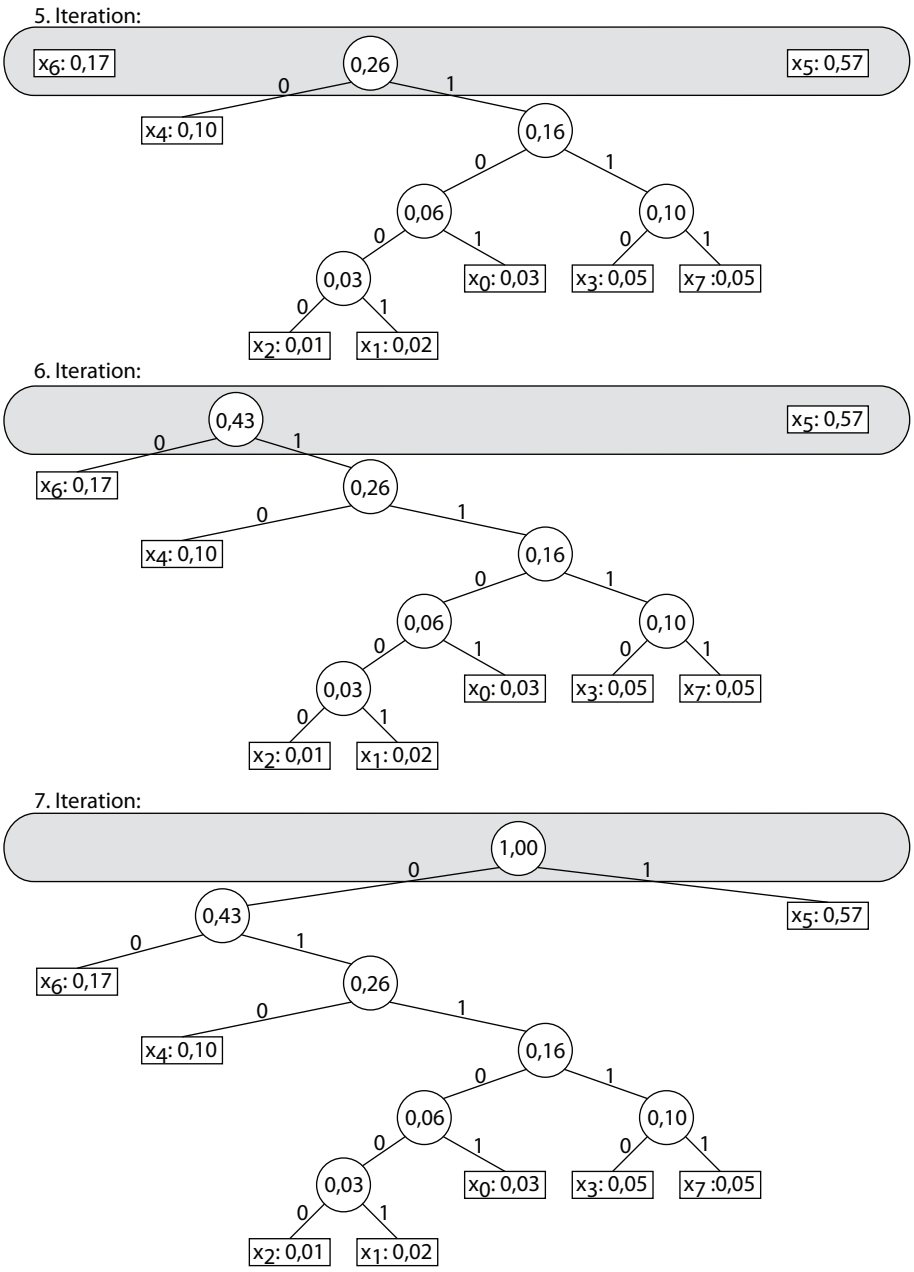


Abbildung 3.5: Ablauf des Huffman-Algorithmus (5. bis 7. Iteration)

Zeichenfolge Z_1:	x_1	x_2	x_3	x_4	x_5	x_5	x_5	x_5	x_6	x_5
Codewortfolge für Z_1:	011001	011000	01110	010	1	1	1	1	00	1

Abbildung 3.6: Zeichenfolge Z_1 und deren Binärcodierung mit dem Huffman-Code c_1. Die Länge der Codewortfolge beträgt 27 Bits.

Wie hier deutlich zu erkennen ist, ist kein Codewort Anfang eines anderen, der Algorithmus hat also einen Präfixcode erzeugt. Weiterhin liegen für alle Zeichen x_i die zugehörigen Entropie- und Codewortlängenwerte recht eng zusammen. Wie eingangs bereits erläutert, wäre es wünschenswert, dass beide Werte jeweils gleich sind. Wegen der Ganzzahligkeit der Codewortlängen ist ein Auf- oder Abrunden jedoch in aller Regel unvermeidlich. Es ergibt sich eine mittlere Codewortlänge von $2,04$ Bits/Zeichen und damit eine Abweichung von lediglich $0,05$ Bits/Zeichen zur unteren Schranke aus dem Satz von Shannon ($H(X) = 1,99$ Bits/Zeichen). Somit liegt die Effizienz des konstruierten Codes sehr nah an der theoretisch möglichen Schranke. Wie bereits erwähnt, arbeitet das Verfahren von Huffman optimal, d. h. es gibt keinen eindeutig decodierbaren Code mit einer Codewortlänge, die echt kleiner ist als der erreichte Wert.

In unserem Anwendungsbeispiel können wir nun alle auftretenden Messwerte codieren. Abbildung 3.6 zeigt exemplarisch eine Folge von zehn Werten mit der dazugehörigen Codierung. Diese bezeichnen wir im Folgenden mit Z_1. Die Leerzeichen dienen in der tabellarischen Darstellung nur der besseren Lesbarkeit. Da der konstruierte Code präfixfrei ist, ist die Codewortfolge auch ohne Leerzeichen eindeutig decodierbar. Die Länge der Codewortfolge beträgt 27 Bits.

In der dargestellten Folge von zehn Zeichen beträgt die durchschnittliche Codewortlänge demnach $2,7$ Bits/Zeichen. Die mittlere Codewortlänge des konstruierten Huffman-Codes $L_m(c)$ beträgt dagegen, wie weiter oben dargestellt, $2,04$ Bits/Zeichen. Die Differenz zwischen diesen beiden Kenngrößen kommt dadurch zu Stande, dass sich die Häufigkeiten der Zeichen in der zu codierenden Zeichenfolge Z_1 nicht mit den Auftrittswahrscheinlichkeiten decken, die bei der Konstruktion des Huffman-Codes zu Grunde gelegt wurden. Eine durchschnittliche Codewortlänge von $2,04$ Bits/Zeichen stellt sich also erst dann ein, wenn die Quelle eine sehr lange Zeichenfolge sendet, so dass sich die relativen Häufigkeiten der einzelnen Zeichen den zu Grunde gelegten Wahrscheinlichkeitswerten immer weiter annähern.

Gängige Praxis bei der Konstruktion von Huffman-Codes ist es daher, zunächst die zu codierende Zeichenfolge einzulesen und die Häufigkeiten der einzelnen Zeichen darin zu analysieren. Diese Häufigkeiten dienen dann bei der Konstruktion des Codes als Eingangsgröße. Es wird also aufgrund der vorliegenden Häufigkeitsverteilung eine Informationsquelle angenommen, die mit „passenden“ Wahrscheinlichkeitswerten die Zeichen aus dem Quellalphabet auswählt. Auf diese Weise wird immer ein optimaler Code für die vorliegende Nachricht konstruiert.

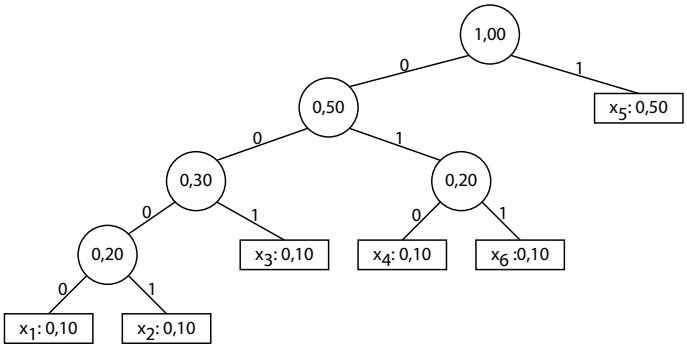


Abbildung 3.7: Optimierter Huffman-Baum für die Zeichenfolge Z_1

Tabelle 3.3 zeigt in der linken Spalte die resultierenden Wahrscheinlichkeitswerte bei diesem Vorgehen. Das Zeichen x_5 kommt in Z_1 fünfmal vor. Die Gesamtlänge von Z_1 beträgt 10. Folglich bekommt x_5 einen Wahrscheinlichkeitswert $P(x_5) = \frac{5}{10}$ zugewiesen. Die Wahrscheinlichkeitswerte aller anderen Zeichen ergeben sich auf die gleiche Weise als $\frac{1}{10}$. Mit Hilfe dieser Wahrscheinlichkeitswerte berechnet sich auch die Entropie dieser neuen Informationsquelle als $H(X) = 2,15$ Bits/Zeichen.

Ausgehend von dieser angepassten Wahrscheinlichkeitsverteilung ergibt sich auch ein neuer Huffman-Code c_2. Der zugehörige Huffman-Baum ist in Abbildung 3.7 dargestellt, die Codewörter und die zugehörigen Kenngrößen in Tabelle 3.3. Die mittlere Codewortlänge ergibt sich dabei als $L_m(c_2) = 2,20$ [Bits/Zeichen]. Somit hat der Huffman-Algorithmus auch für die Werte in diesem Beispiel einen Code konstruiert, dessen mittlere Codewortlänge nur wenig über der Entropie der Quelle liegt.

Abbildung 3.8 zeigt die resultierende Binärcodierung für die Zeichenfolge Z_1. Die Länge der Codewortfolge beträgt hier 22 Bits, was bei einer mittleren Codewortlänge von $2,20$ Bits/Zeichen und 10 zu codierenden Zeichen bei einem speziell auf diese Nachricht angepassten Huffman-Code auch zu erwarten war.

Informationsquelle				Code		
x_i	$P(x_i)$	$H(x_i)$	$P(x_i) \cdot H(x_i)$	$c_2(x_i)$	$L(c_2(x_i))$	$P(x_i) \cdot L(c_2(x_i))$
x_1	$0,10$	$3,32$	$0,33$	0000	4	$0,40$
x_2	$0,10$	$3,32$	$0,33$	0001	4	$0,40$
x_3	$0,10$	$3,32$	$0,33$	001	3	$0,30$
x_4	$0,10$	$3,32$	$0,33$	010	3	$0,30$
x_5	$0,50$	$1,00$	$0,50$	1	1	$0,50$
x_6	$0,10$	$3,32$	$\underline{0,33}$	011	3	$\underline{0,30}$
			$H(X) = 2,15$			$L_m(c_2) = 2,20$

Tabelle 3.3: Codewörter und Codewortlängen des Huffman-Codes c_2

Zeichenfolge Z_1:	x_1	x_2	x_3	x_4	x_5	x_5	x_5	x_5	x_6	x_5
Codewortfolge für Z_1:	0000	0001	001	010	1	1	1	1	011	1

Abbildung 3.8: Zeichenfolge Z_1 und deren Binärcodierung mit dem Huffman-Code c_2. Die Länge der Codewortfolge beträgt 22 Bits.

Es sei angemerkt, dass der Entwurf eines speziell auf eine Nachricht zugeschnittenen Huffman-Codes nicht nur Vorteile mit sich bringt. Da sich der Code mit jeder Nachricht ändert, muss der Sender die jeweils verwendete Codierungstabelle mit zum Empfänger übertragen. Folglich ist dieses Vorgehen bei sehr kurzen Nachrichten nicht immer vorteilhaft, denn die Übertragung der Codierungstabelle verursacht hier möglicherweise ein höheres Datenaufkommen als durch die Codeoptimierung eingespart wird. Ein weiterer Nachteil besteht darin, dass die gesamte Nachricht dem Sender zunächst vorliegen muss, damit er die Häufigkeiten der auftretenden Zeichen bestimmen kann.

Dieser zweite Aspekt lässt sich jedoch umgehen, indem eine Variante des Huffman-Algorithmus angewendet wird, die 1973 von Faller entwickelt wurde. Man konstruiert einen *adaptiven Huffman-Code* [42]. Die grundlegende Idee besteht dabei darin, die Codekonstruktion nicht vorab, sondern sukzessive mit der Übertragung einer Zeichenfolge vorzunehmen. Mit jedem gesendeten Zeichen wird entweder ein neues Blatt im Huffman-Baum angelegt oder – wenn das Zeichen bereits zuvor übertragen wurde – die zugehörige Blattmarkierung wegen der geänderten Häufigkeit dieses Zeichens aktualisiert. Genau wie beim herkömmlichen Huffman-Algorithmus ergibt sich aus den markierten Blattknoten ein Codebaum und schließlich ein Code – allerdings mit dem Unterschied, dass sich dieser mit jedem gesendeten Zeichen ändern kann. Da aber sowohl Sender als auch Empfänger die Häufigkeiten der bisher gesendeten Zeichen kennen, können beide Parteien ihren Huffman-Baum und damit auch den resultierenden Code synchron halten.

Da der genaue Aufbau eines adaptiven Huffman-Codes für das weitere Verständnis dieser Arbeit nicht von Bedeutung ist, geht der Autor nicht näher darauf ein. Eine ausführliche Darstellung dieser Codierungsvariante liefert beispielsweise [132].

Weiterführende Betrachtungen zu Huffman-Codes im Allgemeinen findet der Leser in [47]. Dort wird auch der Satz von Shannon weiter präzisiert, indem die obere Schranke für die mittlere Codewortlänge am Beispiel von Huffman-Codes noch genauer abgeschätzt wird.

3.7 Differenzcodierung

Die Optimalität von Huffman-Codes bedeutet nicht, dass es keinen Spielraum für weitere Verbesserungen gibt. In diesem Abschnitt zeigt der Autor, dass sich die Zeichen-

Zeichenfolge Z_1:	x_1	x_2	x_3	x_4	x_5	x_5	x_5	x_5	x_6	x_5
Zeichenfolge Z_2:	x_1	$+1$	$+1$	$+1$	$+1$	0	0	0	$+1$	-1

Abbildung 3.9: Überführung der Zeichenfolge Z_1 in die Zeichenfolge Z_2 durch Anwendung eines Differenzcodes

folge Z_1 noch kompakter darstellen lässt, als es mit dem speziell darauf angepassten Huffman-Code c_2 möglich ist.

Wie bereits in Abschnitt 3.5 erläutert, trifft das Modell einer Informationsquelle, die Zeichen im Rahmen eines perfekten Zufallsprozesses auswählt, bei praktischen Anwendungen nur selten zu. Die in Abschnitt 3.4 aufgezeigten theoretischen Grenzen gelten aber nur für diesen Idealfall. Folglich ist es eine Hauptaufgabe bei der Datenkompression, die Charakteristik der vorliegenden Informationsquelle genau zu erfassen und möglichst effektiv auszunutzen. Wie oben dargestellt, bezeichnet man diesen Schritt als Modellierung.

In dem Anwendungsbeispiel aus dem vorhergehenden Abschnitt ist davon auszugehen, dass der Erwärmungsprozess des Motors zunächst zu einem monotonen Ansteigen der Messwerte führt. Ist die Betriebstemperatur des Motors dann erreicht, bleiben die Werte konstant und ändern sich nur noch in besonderen Situationen. Gerät das Auto beispielsweise in einen Stau oder wird im Winter die Heizung eingeschaltet, ist mit einem spontanen Ansteigen oder Absinken der Kühlwassertemperatur zu rechnen. Somit sind langsam ansteigende und gleich bleibende Temperaturen bei diesem Anwendungsfall die Regel und fallende Werte eher die Ausnahme.

Durch diese zu erwartende Kontinuität ist das Modell einer perfekt zufälligen Informationsquelle für diesen Anwendungsfall offenbar nicht optimal geeignet. Es gibt eine stark ausgeprägte statistische Abhängigkeit zwischen einem Messwert und seinem Vorgänger, und diese Abweichung vom Modell eines perfekten Zufallsprozesses kann bei der Datenkompression ausgenutzt werden. Ein gängiges Konzept, um solche kontinuierlichen Prozesse zu modellieren, ist die Differenzcodierung. Hierbei codiert man nicht den Signalwert direkt, sondern nur dessen Abweichung zum vorherigen.

Abbildung 3.9 zeigt die geänderte Darstellung für die Wertefolge Z_1 aus unserem Anwendungsbeispiel, es entsteht die neue Zeichenfolge Z_2. Dabei wird zunächst der Startwert der Differenzwertfolge codiert: x_1. Alle folgenden Werte werden über ihre Abweichung zum vorherigen dargestellt. Dies erfolgt mit Hilfe der Zeichen „$+1$“, „0“ und „-1“.

Im nächsten Schritt konstruieren wir mit Hilfe des Huffman-Algorithmus einen optimalen Binärcode für Z_2. Tabelle 3.4 zeigt die Eigenschaften der zu Z_2 passenden Informationsquelle sowie den resultierenden Huffman-Code c_3. Der Huffman-Baum zu c_3 ist in Abbildung 3.10 dargestellt.

Informationsquelle				Code		
x_i	$P(x_i)$	$H(x_i)$	$P(x_i) \cdot H(x_i)$	$c_3(x_i)$	$L(c_3(x_i))$	$P(x_i) \cdot L(c_3(x_i))$
x_1	$0,10$	$3,32$	$0,33$	000	3	$0,30$
-1	$0,10$	$3,32$	$0,33$	001	3	$0,30$
0	$0,30$	$1,74$	$0,52$	01	2	$0,60$
$+1$	$0,50$	$1,00$	$0,50$	1	1	$0,50$
			$H(X) = 1,68$			$L_m(c_3) = 1,70$

Tabelle 3.4: Codewörter und Codewortlängen des Huffman-Codes c_3

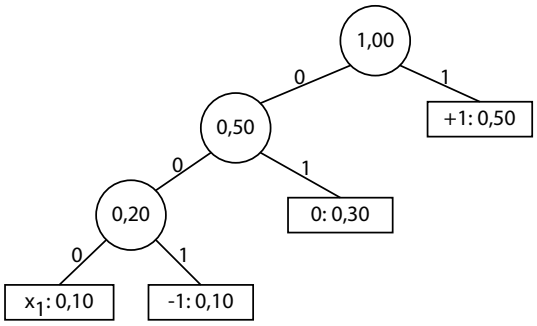


Abbildung 3.10: Optimierter Huffman-Baum für die Zeichenfolge Z_2

Wie aus der ersten Spalte in Tabelle 3.4 ersichtlich ist, ergibt sich durch die Differenzcodierung ein ganz anderes Quellalphabet X und damit auch ganz andere Wahrscheinlichkeitswerte für die einzelnen Alphabetszeichen x_i. Der Leser kann diese Werte mit denen aus Tabelle 3.3 auf Seite 60 vergleichen. Dabei fällt vor allem auf, dass die Entropie der Quelle mit $H(X) = 1,68$ Bits/Zeichen deutlich geringer ist als bei der auf Z_1 angepassten Informationsquelle. Die Entropie beträgt dort $2,15$ Bits/Zeichen. Durch den Zwischenschritt der Differenzcodierung konnte die Entropie der Quelle also erheblich gesenkt werden, was auch eine deutlich geringere mittlere Codewortlänge von nun $1,7$ Bits/Zeichen ermöglicht.

Abbildung 3.11 zeigt schließlich die resultierende Binärdarstellung für die Zeichenfolge Z_1 nach Überführung in die Differenzendarstellung Z_2 und nach Anwendung eines für Z_2 optimierten Huffman-Codes. Es ergibt sich eine sehr kompakte, 17 Bits lange Darstellung, also 5 Bits weniger als bei der direkten Codierung von Z_1.

Zeichenfolge Z_1:	x_1	x_2	x_3	x_4	x_5	x_5	x_5	x_5	x_6	x_5
Zeichenfolge Z_2:	x_1	$+1$	$+1$	$+1$	$+1$	0	0	0	$+1$	-1
Codewortfolge für Z_2:	000	1	1	1	1	01	01	01	1	001

Abbildung 3.11: Binärcodierung des erzeugten Differenzcodes mit Hilfe eines angepassten Huffman-Codes

Obwohl die Zeichenfolge Z_2 offensichtlich den gleichen technischen Vorgang wie Z_1 beschreibt (nämlich den Temperaturverlauf im Kühlwasser), ergibt sich eine geringere Entropie der Quelle und damit auch eine geringere mittlere Codewortlänge. Durch das Modell der Differenzcodierung wird zwar der Informationsgehalt laut mathematischer Definition gesenkt, es tritt jedoch kein Informationsverlust im technischen Sinne ein – schließlich lässt sich die Zeichenfolge Z_1 aus der Zeichenfolge Z_2 verlustfrei zurückgewinnen.

Der Entropiebegriff von SHANNON deckt sich damit nicht mit der allgemein üblichen Vorstellung, dass der Informationsgehalt einer Nachricht unabhängig von der gewählten Repräsentation ist. Er ergibt sich allein aus den statistischen Eigenschaften der vorliegenden Zeichenfolge; ändert man ihre Repräsentation – z. B. durch Differenzcodierung –, so ändert sich auch ihr Informationsgehalt.

Folglich erlaubt ein geeignetes Modell, eine Zeichenfolge noch kompakter darzustellen, als dies mit Entropiecodierungstechniken wie dem Huffman-Algorithmus allein möglich ist. Die Optimalitätseigenschaft eines Huffman-Codes ist also keineswegs ein hinreichendes Kriterium, um sicherzustellen, dass sich eine vorliegende Zeichenfolge nicht noch kompakter darstellen lässt.

Somit ist es auch heute, fast 60 Jahre nach Entstehung der Informationstheorie, mitunter mehr eine Kunst als eine Wissenschaft, geeignete Codierungen zu konstruieren. Die Kunst besteht darin, jeweils ein geeignetes Modell für die vorliegende Informationsquelle zu entwickeln, und zwar so, dass damit genau der Grad an Unsicherheit beschrieben wird, den die Informationsquelle durch Aussenden eines Zeichens beim Empfänger beseitigt.

Kapitel 4

SOAP-Differenzcodierung

Auf Basis der im vorhergehenden Kapitel dargestellten Informationstheorie hat der Autor untersucht, wie sich das Datenaufkommen bei der SOAP-Kommunikation mit Hilfe von Datenkompressionstechniken effektiv reduzieren lässt.

Im Folgenden stellt der Autor einen neuartigen Kompressionsansatz vor, bei dem er die Idee der Differenzcodierung aus Abschnitt 3.7 aufgreift und in einem entscheidenden Punkt weiterentwickelt: Bisher war es bei Verfahren zur Protokollkompression üblich, die Differenz zwischen zwei aufeinander folgenden Netzwerknachrichten zu codieren (vergleiche Abschnitt 4.4). Da diese Vorgehensweise beim SOAP-Protokoll wegen dessen Zustandslosigkeit nicht möglich ist, bedurfte es einer grundlegenden Neuausrichtung des Codierungsverfahrens. Die zentrale Idee des Autors besteht darin, nicht mehr die Differenz zwischen zwei aufeinander folgenden Nachrichten zu codieren, sondern die Differenz zwischen der zu übertragenden Nachricht und einem Skelettdatensatz, welcher zuvor aus der WSDL-Beschreibung des Web Services generiert wurde.

Dabei hat er die Tatsache ausgenutzt, dass die WSDL-Beschreibung eines Web Services bereits genaue Informationen über den Aufbau der ausgetauschten SOAP-Nachrichten enthält. Die WSDL-Beschreibung ist zum Zeitpunkt der Dienstnutzung sowohl dem Sender als auch dem Empfänger bekannt. Gehen Sender und Empfänger nun davon aus, dass nur solche Nachrichten ausgetauscht werden, die den Vorgaben aus der WSDL-Beschreibung entsprechen, so beseitigt die WSDL-Beschreibung bereits im Vorfeld einen erheblichen Teil an Unsicherheit im Sinne der Informationstheorie. Somit muss durch den eigentlichen Nachrichtenaustausch nur noch wenig Unsicherheit beseitigt werden, was sich positiv auf die Nachrichtengröße auswirkt.

In diesem Kapitel geht es zunächst um die technologischen Grundlagen der SOAP-Kompression. Der Autor erläutert, welchen Rahmen die SOAP-Spezifikation für Datenkompressionsansätze vorsieht und wie sich solche Ansätze mit dem allgemein gängigen SOAP-Transport über HTTP kombinieren lassen.

Dann gibt der Autor einen Überblick über verwandte Arbeiten. Er vergleicht diese nicht nur konzeptionell, sondern zeigt auch die Ergebnisse ausführlicher Vergleichsmessungen, die die Leistungsfähigkeit dieser Ansätze quantitativ gegenüberstellen.

Im Anschluss daran stellt der Autor sein neuartiges Differenzcodierungsverfahren vor. Zunächst erläutert er anhand eines schematischen Ablaufmodells die grundlegende Architektur, dann präsentiert er seine prototypische Implementierung und schließlich zeigt er anhand weiterer Messreihen die Effektivität seines Ansatzes.

Im letzten Abschnitt werden die Ergebnisse zusammengefasst und bewertet.

Die in diesem Kapitel dargelegten Ideen und Konzepte hat der Autor bereits in [158] und [159] in wesentlichen Teilen vorab veröffentlicht.

4.1 Überblick und technologische Grundlagen

Wegen der durchgängigen Repräsentation aller Daten als Text verursacht SOAP im Vergleich mit klassischen Middleware-Ansätzen wie CORBA oder Java RMI ein deutlich höheres Datenaufkommen. Vergleichende Messungen hierzu wurden bereits in Kapitel 1 vorgestellt (vergleiche Abbildung 1.1 auf Seite 7).

Obwohl heutige drahtgebundene Netzwerke durchaus in der Lage sind, auch sehr große Datenmengen zu bewältigen, stellt der Overhead, der durch das SOAP-Protokoll verursacht wird, in einigen Bereichen der Informationstechnologie einen gravierenden Nachteil dar: Beispielsweise ist es bei der Kommunikation über Mobilfunknetze wie GPRS oder UMTS noch immer überwiegend üblich, anhand des übertragenen Datenvolumens abzurechnen [144]. Bei einer Implementierung von Web Services auf Handys – erste Ansätze zu diesem Thema wurden unter anderem in [18] vorgestellt – könnten Datenkompressionsstrategien somit direkt zur Kostensenkung (und auch Leistungssteigerung) beitragen. Ähnliche Überlegungen treffen auch auf Wählverbindungen über ISDN oder Analog-Modem zu. Solche älteren Verbindungstechniken sind auch heute noch in vielen Bereichen durchaus üblich [78].

Ganz besonders wichtig sind Strategien zur Effizienzsteigerung aber im Umfeld von mobilen Kleinstcomputern, wie sie beispielsweise in Sensornetzwerken [2] zum Einsatz kommen. Auch dieser Aspekt wurde im Einleitungskapitel bereits ausführlich betrachtet.

4.1.1 Vorgaben der SOAP-Spezifikation

Tatsächlich sieht die SOAP-Spezifikation – wenn auch indirekt – Mechanismen zur Datenkompression vor. Im Standard-Dokument [176] heißt es wörtlich:

„The binding framework does not require that every binding use the XML 1.0 serialization as the 'on the wire' representation of the XML infoset; compressed, encrypted, fragmented representations and so on can be used if appropriate.“

Die SOAP-Spezifikation ordnet also die Problemstellung der Datenkompression dem *Binding Framework* zu, dieser Begriff bezieht sich auf den in Abschnitt 2.2.2 erläuterten Mechanismus zur dynamischen Bindung von Nachrichtentransportprotokollen an SOAP. Anders ausgedrückt bedeutet dies, dass für Datenkompressionskomponenten keine zusätzliche Schicht im Web Service Technology Stack vorgesehen ist; sämtliche Datenkompressionskomponenten sind Teil der Schicht *Communication* (vergleiche Abbildung 2.1 auf Seite 15).

Der im Zitat verwendete Begriff *XML Infoset* ist wie folgt zu verstehen: Die XML-Spezifikation [165] definiert ein XML-Dokument als Folge von Unicode-Zeichen, d. h. als Text. Diese Darstellungsform ist aber nicht die einzig mögliche – für technische Anwendungen sind andere Darstellungsweisen mitunter vorteilhafter (vergleiche auch Anhang A.4). Beispielsweise baut ein DOM-Parser aus der Textdarstellung zunächst eine baumartige Datenstruktur im Arbeitsspeicher eines Rechners auf, um so Schreib- und Leseoperationen auf diesem XML-Dokument in geeigneter Weise zu ermöglichen. Somit kann der Inhalt eines XML-Dokuments sowohl als Text, aber beispielsweise auch als DOM-Baum aufgefasst werden. Um von solchen unterschiedlichen technischen Darstellungsweisen zu abstrahieren, hat das W3C eine Spezifikation namens *XML Information Set (XML Infoset)* [171] erarbeitet, die eine vollständig abstrakte Datenstruktur für XML-Dokumente definiert. Sie repräsentiert also den Inhalt eines XML-Dokuments, und zwar unabhängig von einer konkreten technischen Darstellung.

Einem SOAP-Binding ist es damit gestattet, zum Transport der Nachricht eine von der Textform abweichende Darstellung („serialization“) zu verwenden. Das einer Nachricht zu Grunde liegende XML Information Set kann beliebig codiert werden. Dies ermöglicht neben Datenkompressionsanwendungen auch das Verschlüsseln oder Fragmentieren von SOAP-Nachrichten.

4.1.2 HTTP-Content-Encoding

Tatsächlich gibt es bereits SOAP-Implementierungen, die diesen Freiheitsgrad zur Datenkompression ausnutzen und von der Textdarstellung abweichen. So ermöglicht etwa Apache Axis [146] eine gzip-Kompression beim HTTP-Transport. Das Programm gzip ist ein universeller Datenkompressor, der auf dem Lempel-Ziv-Algorithmus (LZ77) [190] basiert.

Im Folgenden wird der hierfür verwendete *Content Encoding* Mechanismus von HTTP exemplarisch vorgestellt. Anhand dieses Beispiels wird deutlich, wie sich eine Komponente zur Datenkompression in die *Communications* Schicht des Web Service Technology Stacks integrieren lässt. Da HTTP das mit Abstand am weitesten verbreitete Protokoll zum SOAP-Nachrichtentransport ist, ist das HTTP-Content-Encoding für praktische Anwendungen besonders relevant.

Abbildung 4.1 zeigt die Anwendung eines gzip-Content-Encodings im Vergleich zu einem herkömmlichen HTTP-POST-Request. Der zusätzliche Header-Parameter `Con-`

```
POST /info HTTP/1.1
Host: sunshinecars.example.org
Content-Type: application/soap+xml
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <res:reservationRequest
   xmlns:res="http://sunshinecars.example.org/reservation">
   <res:reservationID>384DA3F</res:reservationID>
  </res:reservationRequest>
 </env:Body>
</env:Envelope>
```

(a) Transport einer SOAP-Nachricht in einem HTTP-Post-Request

```
POST /info HTTP/1.1
Host: sunshinecars.example.org
Content-Type: application/soap+xml;
Content-Encoding: gzip
Content-Length: nnnn

[Binärdaten]
```

(b) Wie (a), jedoch zusätzlich unter Verwendung eines gzip-Content-Encodings

Abbildung 4.1: Auswirkung eines HTTP-Content-Encodings

`tent-Encoding` signalisiert der empfangenden HTTP-Implementierung, dass der Absender die im HTTP-Body enthaltenen Daten mit einer zusätzlichen Codierung versehen hat, die beim Empfang rückgängig gemacht werden soll. Im Beispiel zeigt der Wert `gzip`, dass eine Datenkompression mit Hilfe des gzip-Kompressors vorgenommen wurde. Die empfangende HTTP-Implementierung soll also zunächst einen gzip-Dekompressor zur Decodierung der Daten aufrufen, bevor sie die Daten an die empfangene Applikation weiterleitet.

Durch die Verwendung dieses Mechanismus – er ist bei den HTTP-Implementierungen gängiger SOAP-Engines bereits weitestgehend vorkonfiguriert – kann man mit vergleichsweise wenig Aufwand die Textdarstellung einer SOAP-Nachricht für die Dauer des Nachrichtentransports in einen komprimierten Binärdatenstrom umwandeln.

Allerdings zeigen vergleichende Messungen [158], dass die Anwendung des gzip-Kompressors auf SOAP-Nachrichten zu keinen befriedigenden Kompressionsraten führt. Bei den Testdaten, die Abbildung 1.1 auf Seite 7 zu Grunde liegen, konnte das Datenaufkommen lediglich um etwa 25% verringert werden. Damit liegt es noch immer deutlich über dem Niveau von Java RMI und CORBA.

Der gzip-Kompressor ist damit offenbar nur bedingt zur Kompression von SOAP-Nachrichten geeignet. Im Folgenden geht es um die Frage, warum die gzip-Kompression – obwohl diese gemeinhin als recht leistungsfähig gilt – hier nur unbefriedigende Kompressionsergebnisse liefert. Weiterhin werden alternative Algorithmen vorgestellt und verglichen, die auf die Verarbeitung von XML-Daten spezialisiert sind.

4.2 Verwandte Arbeiten

Mit der zunehmenden Verbreitung von XML wurde auch die Problemstellung einer kompakten Repräsentation von XML-Dokumenten immer wichtiger. Es gibt bereits eine Vielzahl von Arbeiten zu diesem Thema. Allerdings beschäftigen sich diese zumeist mit Verfahren, die für den Einsatz in XML-Datenbank-Systemen optimiert und somit nicht auf die speziellen Anforderungen der SOAP-Kommunikation zugeschnitten sind.

SOAP-Nachrichten weisen nämlich eine ganz typische Eigenschaft auf, die bei der Kompression besonderer Beachtung bedarf: ihre vergleichsweise geringe Größe. Kleine Nachrichten sind, wie wir noch genauer betrachten werden, besonders schwierig zu komprimieren.

Im Folgenden stellt der Autor zunächst die wichtigsten verwandten Arbeiten auf dem Gebiet der XML-Datenkompression vor. Primäres Bewertungskriterium bei allen folgenden Betrachtungen ist die Effektivität der einzelnen Techniken bei der Kompression typischer SOAP-Daten, d. h. in welchem Maß sich das Datenvolumen, gemessen in Bytes, durch den Kompressionsvorgang verringern lässt.

4.2.1 Generische Kompressoren

Eine erste Möglichkeit, um die Größe von SOAP-Nachrichten zu reduzieren, ist der Einsatz generischer Datenkompressionstechniken. Hierunter sind solche Ansätze zu verstehen, die nicht nur für XML-Daten funktionieren, sondern für beliebige Eingabedaten. Hier wird der Eingabedatensatz als eine Folge von Bytes aufgefasst, die mit Hilfe eines Kompressionsalgorithmus umcodiert wird. Ein besonders gängiges Verfahren ist dabei die bereits erwähnte gzip-Kompression. Diese beruht auf dem Algorithmus von Lempel und Ziv (LZ77) [190]. Das für die Übertragung eingesetzte Datenformat wird in [37] beschrieben. Andere bekannte Beispiele für generische Kompressoren sind bzip2, ZIP, RAR, ARJ usw., einen sehr guten und umfassenden Überblick bietet [132].

Leider führt die Verwendung solcher generischen Kompressoren – neben der nur mäßigen Effektivität der Kompression – auch zu ungünstigen Nebeneffekten bei der XML-Verarbeitung: Das XML-Dokument wird als Ganzes (bzw. in Blöcken fester Größe) codiert, und zwar ohne Berücksichtigung der durch das XML-Markup gege-

benen Strukturinformation. Dies hat zur Folge, dass ein komprimiertes Dokument nicht geparst oder geändert werden kann, ohne das Dokument als Ganzes zu dekomprimieren. Im Anschluss an die durchgeführten Verarbeitungsschritte muss es dann erneut komprimiert werden.

4.2.2 Codierung von SAX-Events

Ein erster Schritt, um die Effektivität der XML-Datenkompression zu steigern, besteht in der Methode, bei der Codierung von XML-Nachrichten nicht die Textrepräsentation zu codieren, sondern lediglich die darin enthaltenen Informationen – formal repräsentiert durch das bereits erwähnte XML Information Set [171].

Bei vielen Anwendungen ist es beispielsweise nicht notwendig, die Leerzeichen und Zeilenumbrüche zu erhalten, die lediglich der besseren Lesbarkeit eines XML-Dokumentes dienen – folglich müssen diese Zeichen auch nicht mit codiert werden. Solche irrelevanten Informationsanteile sind also zu eliminieren.

Zur Separierung der technisch signifikanten Informationsanteile von den übrigen hat sich in der XML-Datenkompression die Verwendung von *SAX-Parsern (Simple API for XML)* durchgesetzt (vergleiche Anhang A.8). Erstmals wurde diese Vorgehensweise von Liefke und Suciu in [85] vorgeschlagen: Ein SAX-Parser verarbeitet dabei die Textdarstellung eines XML-Dokuments und übersetzt diese in eine Folge von Ereignissen, so genannte SAX-Events, die dann vom Kompressor interpretiert und schließlich codiert werden.

Anders als bei der Codierung mittels generischer Kompressoren, die die Informationsquelle als einen Mechanismus auffassen, der eine Folge von Bytes ausgibt, ist es bei der XML-Kompression somit üblich, das Modell der Informationsquelle so zu wählen, dass diese eine Folge von SAX-Events ausgibt. Diese grundlegende Technik wird bei allen im Folgenden behandelten Kompressoren – außer WBXML und Millau – eingesetzt.

4.2.3 XMill

Den ersten XML-spezifischen Kompressor, der sich auf beliebige XML-Dokumente anwenden lässt, haben Liefke und Suciu im Jahr 2000 vorgestellt [85]: Bei *XMill* werden die SAX-Events zunächst von einem *Path Processor* verarbeitet. Diese Komponente wertet die vom SAX-Parser gelieferten Events aus und speichert die darin enthaltene Information in verschiedenen Containern. Eine Open-Source-Implementierung von *XMill* ist unter [5] verfügbar.

Einer der XMill-Container speichert ausschließlich Strukturinformationen (*Structure Container*) – er enthält selbst keine Werte, sondern lediglich Referenzen auf Werte. Diese sind in den übrigen Containern abgelegt (*Data Container*).

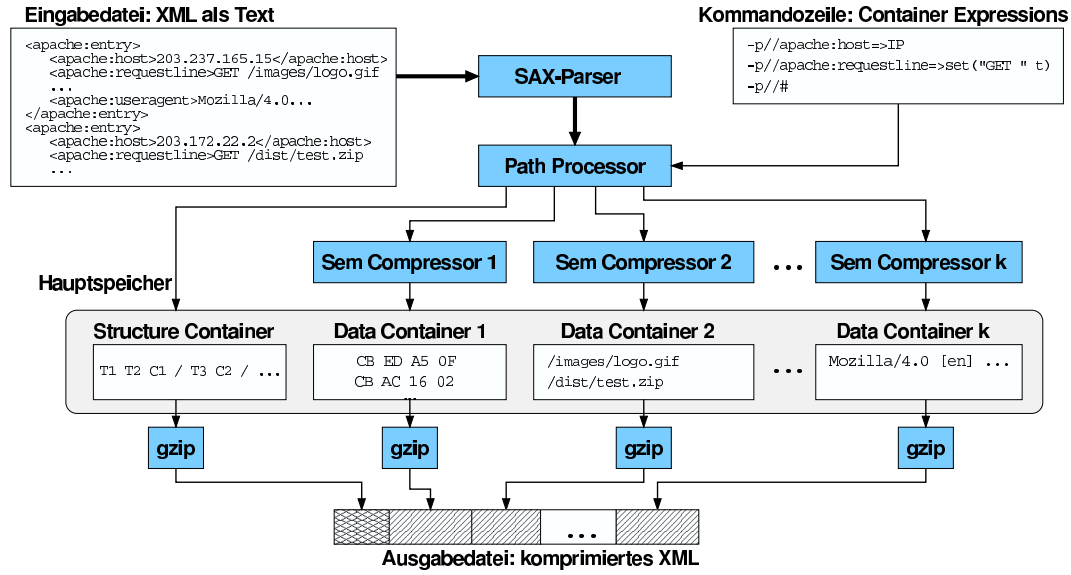


Abbildung 4.2: Architektur des XMill Kompressors (entnommen aus [85])

Ein großer Vorteil dieser Aufteilung ist die Möglichkeit, die Werte in den Datencontainern mit Rücksicht auf ihre besonderen Eigenschaften zu codieren, dies geschieht mit Hilfe so genannter *semantischer Kompressoren*. Beispielsweise müssen IP-Adressen nicht als Zeichenkette codiert werden, sondern lassen sich sehr kompakt durch vier Bytes darstellen. Solche besonderen Codierungsregeln werden bei XMill über Kommandozeilenparameter angegeben.

Zur Erzeugung einer kompakten Binärdarstellung werden schließlich sämtliche Container mit Hilfe eines herkömmlichen generischen Kompressors wie gzip codiert und die resultierenden Bytefolgen hintereinander in die Ausgabedatei geschrieben.

Abbildung 4.2 zeigt die Architektur von XMill und die Funktionsweise der einzelnen Komponenten anhand eines Beispiels. Das zu verarbeitende XML-Dokument beginnt mit zwei öffnenden Tags, die vom SAX-Parser in zwei *startElement* Ereignisse umgewandelt werden. Der *Path Processor* vermerkt das Auftreten dieser beiden Ereignisse im *Structure Container* über die Symbole `T1` und `T2`. Dabei steht `T` für ein öffnendes Tag, und die Nummer ist ein Zahlenwert, der den Namen dieses Tags referenziert. Der Tag-Name (zusammen mit den dazugehörigen Namespace-Informationen) wird in einem separaten Tag-Container abgespeichert (dieser ist in der Abbildung jedoch nicht mit dargestellt). Kommt ein Tag-Name im zu komprimierenden Dokument mehrfach vor, so muss er nicht noch einmal codiert werden. Er wird über seine Position im Tag-Container lediglich ein zweites Mal referenziert.

Dann folgt im Eingabedokument der Wert `203.237.165.15`, er repräsentiert eine IP-Adresse. Über den Kommandozeilenparameter `-p//apache:host=>IP` wird der

Path Prozessor angewiesen, für sämtliche Elemente mit dem QName[1] `apache:host` den semantischen Kompressor für IP-Adressen zu verwenden. Die Funktionsweise dieses Kompressors muss über weitere Kommandozeilenparameter definiert werden, dies ist im Beispiel ebenfalls nicht dargestellt. Der *Path Processor* erzeugt nun das Symbol `C1`, das eine Referenz auf den ersten Datencontainer darstellt.

Anders als bei den Tag-Namen wird also als Zahlenwert nicht die Nummer des Eintrags innerhalb eines Containers angegeben, sondern nur die Containernummer. Beim Kompressions- und Dekompressionsvorgang werden daher Zähler, die auf die Einträge innerhalb der Container zeigen, automatisch inkrementiert, so dass immer der nächste Eintrag ausgewählt wird.

In der Abbildung erkennt man im ersten Datencontainer die vier Bytes lange Codierung der IP-Adresse in Hexadezimalschreibweise.

Als Nächstes liest der SAX-Parser ein schließendes Tag. Der Path Processor codiert dieses mit dem Symbol / im Strukturcontainer. Da bei wohlgeformten XML-Dokumenten der Name eines schließenden Tags unmittelbar aus dem Namen des korrespondierenden öffnenden Tags hervorgeht, bedarf es keiner separaten Codierung der Namensinformationen von schließenden Tags.

Die Kommandozeilenoption `-p//apache:requestline=>set("GET " t)` signalisiert dem Path Processor, dass Elemente mit QName `apache:requestline` immer Text enthalten, der mit der Zeichenkette `"GET "` beginnt. Folglich muss dieses Präfix nicht mit codiert werden. Die Angabe `-p//#` führt dazu, dass sämtliche Werte, deren Verarbeitung nicht den zuvor spezifizierten Regeln unterliegt, ohne semantische Kompression zu verarbeiten sind und somit direkt in einen Datencontainer kopiert werden können. Der Leser kann das weitere Vorgehen bei der Codierung des Eingabedatensatzes anhand von Abbildung 4.2 selbst nachvollziehen.

Wie dieses Beispiel zeigt, ist XMill besonders flexibel zu konfigurieren. Mit Hilfe der Kommandozeilenoptionen für semantische Kompression lässt sich dieses Programm sehr genau an die besonderen Eigenschaften verschiedener XML-Dokumente anpassen. Allerdings wird diese Flexibilität damit erkauft, dass der Anwender die Angaben zur semantischen Kompression auf der Kommandozeile manuell spezifizieren muss. Folglich müsste man für jeden Web Service die Parameter separat anpassen, was bei realen Anwendungen sicherlich nicht praktikabel ist.

4.2.4 ESAX und Multiplexed Hierarchical Modeling

In [22] stellt Cheney das *ESAX* Konzept vor; ESAX steht dabei für *Encoded SAX*. Wie dieser Name bereits vermuten lässt, beruht auch ESAX auf der Verwendung eines SAX-Parsers. Allerdings werden hier die erzeugten Ereignisse nicht verschiedenen

[1] Der Begriff QName bezeichnet einen Elementnamen zusammen mit seinem Namespace-Präfix in der Form `Präfix:Name`. Für weitere Ausführungen hierzu siehe Anhang A.7.

XML als Text:	`<elt`	`att=`	`"asdf"`	`>`	`XYZ`	`</elt>`
ESAX-Code:	`10`	`0D att 00`	`asdf 00`	`FF`	`FE XYZ 00`	`FF`

Abbildung 4.3: Beispiel für die ESAX-Codierung

Containern zugeordnet, wie dies bei XMill der Fall ist. Vielmehr werden die SAX-Ereignisse mit Hilfe eines vergleichsweise einfachen, zustandsbasierten Algorithmus direkt codiert.

Abbildung 4.3 zeigt die ESAX-Codierung exemplarisch anhand des XML-Fragments `<elt att="asdf">XYZ</elt>`. Alle ESAX-Codes sind – mit Ausnahme von Zeichenketten – in Hexadezimalschreibweise angegeben.

In diesem Beispiel gehen wir davon aus, dass ein Element mit dem Namen `elt` bereits zuvor verarbeitet wurde. Zur Codierung des Namens genügt es daher, den entsprechenden Eintrag in der Element-Historie zu adressieren. Dies geschieht über das Byte `10` (Hex). Ein Attribut mit dem Namen `att` ist hingegen in der Attribut-Historie nicht gespeichert, und folglich muss der Name im Ausgabedatenstrom codiert werden. Das Präfix `0D` signalisiert, dass nun ein bisher unbekannter Attributname folgt. Im Anschluss wird die Zeichenkette `att` codiert. Die Codierung erfolgt dabei in der Zeichencodierung des Eingabedokuments, also beispielsweise in UTF-8. Sämtliche Zeichenketten werden mit dem Nullbyte `00` abgeschlossen. Es folgt der Attributwert `asdf` ebenfalls als Zeichenkette. Der Wert `FF` zeigt das Ende der Attributliste an. Das dann folgende Byte `FE` signalisiert, dass nun Zeichendaten folgen. Diese werden ebenfalls als Zeichenkette im Ausgabedatenstrom codiert. Der Datensatz endet mit dem Wert `FF`, welcher ein schließendes Tag repräsentiert.

An dieser Stelle wird deutlich, dass das Byte `FF`, je nachdem in welchem Verarbeitungszustand es auftaucht, entweder ein schließendes Tag oder das Ende einer Attributliste anzeigt. Eine genaue Auflistung über mögliche Verarbeitungszustände und deren Übergänge bei ESAX findet der Leser in [22].

Um die verbleibende Redundanz im erzeugten Binärstrom zu eliminieren, wird er mit Hilfe eines generischen Kompressors wie gzip weiterverarbeitet. Cheney hat mit einer Reihe verschiedener Kompressortypen für diesen Verarbeitungsschritt experimentiert. In Abhängigkeit vom gewählten Kompressor ergeben sich bei seinen Testdaten komprimierte Dateien, die nur 1% bis 7% größer sind als solche, die mit dem deutlich komplexeren XMill-Algorithmus erzeugt wurden.

Cheney schlägt in [22] bereits eine Verbesserung dieses Ansatzes vor: Er kombiniert die ESAX-Codierung mit dem *Prediction by Partial Match (PPM)* Algorithmus. Dem PPM-Algorithmus liegt das Modell einer Informationsquelle zu Grunde, bei der die ausgesendeten Zeichen mit großer Wahrscheinlichkeit in einem gleichen Kontext auftreten. Unter einem Kontext n-ter Ordnung versteht man dabei die n vorhergehenden Zeichen. Betrachtet man beispielsweise das Wort *ANANAS* und den Kontext erster Ordnung, taucht der Buchstabe A zweimal im Kontext N auf, und auch der Buch-

stabe *N* zweimal im Kontext *A*. Bei der Codierung berücksichtigt man nun solche Kontextinformationen, um anhand der bisher codierten Zeichenfolge das nächste Zeichen vorherzusagen. Im Ausgabedatensatz wird nur noch die Abweichung zu dieser Vorhersage codiert. Eine detaillierte Beschreibung des PPM-Algorithmus und seiner Varianten findet der Leser zum Beispiel in [132].

Cheney vermutete, dass sich auch bei typischen XML-Informationsquellen solche Kontextinformationen ausnutzen lassen. Sein Grundgedanke war dabei, die hierarchische Struktur eines XML-Dokuments mit einzubeziehen, um so geeignete Kontextinformationen extrahieren zu können. Diese Technik nennt er *Multiplexed Hierarchical Modeling (MHM)*. In [22] beschreibt Cheney verschiedene Algorithmen zur Bestimmung von geeigneten Kontextzeichen. Eine wesentliche Erkenntnis besteht darin, separate Kontexthistorien für Elemente, Attribute und Zeichendaten vorzusehen. Eine Open-Source-Implementierung dieses Ansatzes namens *xmlppm* ist unter [25] verfügbar.

4.2.5 WBXML und Millau

Ein weiterer Ansatz zur Erzeugung kompakter Binärcodierungen für XML-Dokumente ist *WAP Binary XML (WBXML)*. Diese binäre Repräsentation wurde bereits 1999 vom W3C standardisiert [166] und dient der effizienten Codierung von *Wireless Markup Language (WML)* [164] Dokumenten in schmalbandigen Mobilfunknetzen.

Die Funktionsweise von WML ist prinzipiell mit der von HTML vergleichbar, allerdings mit speziellem Zuschnitt auf die besonderen Anforderungen von kleinen Mobilgeräten. Ein weiterer Unterschied zwischen diesen beiden Sprachen besteht darin, dass WML im Gegensatz zu HTML eine XML-Sprache ist und sich folglich mit Hilfe von XML-Parsern verarbeiten lässt.

Um das Datenvolumen bei der Übertragung von WML-Dokumenten zu reduzieren, werden diese direkt auf dem Web-Server in die Binärdarstellung WBXML umcodiert. Wegen der begrenzten Ressourcen auf mobilen Geräten basiert diese Umformung jedoch nicht auf komplexen Algorithmen wie denen in XMill oder xmlppm. Stattdessen teilt ein WBXML-Codierer das Dokument in vordefinierte Blöcke ein, so genannte *Token*. Diese Token werden dann mit Hilfe von Codetabellen, genannt *Code Spaces*, codiert. Beispielsweise wird das Token `charset="UTF-8"` auf den Hexadezimalwert `6A` abgebildet. Sowohl die Token-Einteilung als auch die zu verwendenden Codetabellen sind im Standard festgeschrieben. Anders als bei XMill und xmlppm ist die Textdarstellung der XML-Daten Ausgangspunkt für die Kompression und nicht die Repräsentation in Form von SAX-Ereignissen. Das Vorgehen ist damit prinzipiell mit dem in Kapitel 3 vorgestellten Q-Code vergleichbar.

Da die WBXML-Codetabellen somit unveränderlich sind und nicht für jede Nachricht neu berechnet werden müssen, ist es nicht notwendig, diese zusammen mit der Nachricht zum Empfänger zu übertragen. Alle WBXML-fähigen Parser kennen diese

Tabellen und können mit ihrer Hilfe WBXML-Nachrichten codieren oder decodieren. Somit lassen sich auch bei kleinen Nachrichten gute Kompressionsraten erzielen.

Zeichenketten, die nicht mit Hilfe der vordefinierten Codetabellen in eine Binärrepräsentation übersetzt werden können, werden *in line*, d. h. als Folge von Einzelzeichen direkt im Ausgabedatenstrom, codiert. Das vordefinierte Token STR_I dient dabei als Anfangsmarkierung. Das Ende eines solchen Inline-Strings wird über ein reserviertes, zeichensatzspezifisches Abschlusssymbol codiert.

Ein besonderer Vorteil des WBXML-Ansatzes besteht darin, dass ein WBXML-fähiger SAX-Parser die binären Token mit Hilfe der Codetabellen sehr schnell in SAX-Events umwandeln kann, um so eine Anwendung zu steuern. Ein aufwändiger, mehrstufiger Decodierungsprozess, wie bei XMill oder xmlppm, ist damit nicht mehr erforderlich.

Das Prinzip, auf dem die WBXML-Codierung beruht, ist grundsätzlich nicht auf Anwendungen im Zusammenhang mit WML beschränkt. Es lassen sich prinzipiell auch andere XML-Sprachen auf diese Weise codieren, allerdings nur unter der Voraussetzung, dass geeignete Codetabellen zur Übersetzung der sprachspezifischen Tag- und Attributnamen im Parser vorhanden sind. Ein Parser für Windows und Unix Systeme, der besonders viele Sprachen unterstützt, ist Teil der *WBXML Library* [70]. Er unterstützt verschiedene Sprachen aus dem Bereich der Mobiltelefonie, darunter beispielsweise die *Synchronization Markup Language (SyncML)* [116] und auch die *Digital Rights Management Rights Expression Language (DRMREL)* [117]. Die meisten WBXML-Implementierungen sind jedoch nicht für den Betrieb auf PCs ausgerichtet. Sie sind direkt in die Firmware mobiler Geräte integriert.

Giradot und Sundaresan haben das WBXML-Konzept weiterentwickelt und daraus den Kompressor *Millau* konstruiert [50]. Millau bietet zwei zusätzliche Verbesserungen:

Zum einen speichert Millau Zeichenketten, die nicht mit Hilfe der Tabellen codiert werden können, nicht *in line*, sondern – ähnlich dem Vorgehen bei XMill – in einem separaten Container. Diese Trennung ermöglicht die Anwendung eines generischen Kompressors auf diese Daten, so dass sich auch Dateien mit einem großen Anteil solcher Zeichenketten effizient codieren lassen.

Zum anderen verbessert Millau die Adressierungsmöglichkeiten innerhalb der Code Spaces. In WBXML gibt es für mögliche Attributnamen und Werte nur jeweils 128 Tabelleneinträge. Giradot und Sundaresan argumentieren, diese Anzahl sei für komplexere Sprachen möglicherweise zu klein. Daher haben sie das Adressierungsschema so modifiziert, dass nun jeweils 256 Tabelleneinträge zur Verfügung stehen.

Leider gibt es keine öffentlich zugängliche Implementierung von Millau. Daher sind hier keine vergleichenden Messungen möglich. Da Millau aber eine Verbesserung von WBXML darstellt, ist davon auszugehen, dass die Kompressionsergebnisse besser als jene von WBXML ausfallen.

Es gibt bisher keine Ansätze zur Kompression von SOAP-Nachrichten mit Hilfe von Millau oder WBXML. Dies liegt offenbar an den besonderen Eigenschaften von SOAP: Anders als bei Sprachen wie SyncML, die nur eine sehr begrenzte Anzahl möglicher Tag- und Attributnamen erlauben, sind bei SOAP beliebige anwendungsspezifische XML-Daten innerhalb des Bodys möglich. SOAP ist damit eine besonders „dynamische“ XML-Sprache, für die es folglich kaum sinnvoll ist, feste Codetabellen vorzugeben.

4.3 Kompressionsergebnisse verwandter Ansätze

Im Folgenden präsentiert der Autor Untersuchungen zur Effektivität der genannten Kompressionsansätze. Zwar gibt es bereits etliche Veröffentlichungen, in denen die Autoren entsprechende Messergebnisse vorstellen, diese beziehen sich jedoch so gut wie ausschließlich auf Experimente mit großen Dokumenten ($>$ 100 kB) [85, 22, 50].

Bei vielen praktischen Anwendungen des SOAP-Protokolls treten dagegen recht kleine Nachrichten auf – mitunter nur wenige hundert Bytes groß [91]. Diese Tatsache macht eine effektive Kompression besonders schwierig, denn die meisten gängigen Kompressionsverfahren arbeiten mit Codetabellen oder Wörterbüchern, in denen bestimmte Zeichenfolgen aus der unkomprimierten Nachricht auf kurze Bitfolgen abgebildet werden (Beispiel: nicht-adaptiver Huffman-Algorithmus). Diese müssen im komprimierten Datensatz mit übertragen werden. Bei großen Datensätzen ist der Overhead durch Codetabellen und Wörterbücher vernachlässigbar. Bei der Kompression sehr kurzer Datensätze kann es allerdings im Extremfall vorkommen, dass allein diese Codetabellen mehr Raum einnehmen als der unkomprimierte Datensatz. Solche Datensätze werden durch die Kompression also größer statt kleiner.

Kurze Datensätze stellen daher eine besondere Herausforderung bei der Datenkompression dar. Infolgedessen sind die Ergebnisse von Messungen mit großen Datensätzen nicht notwendigerweise auf Anwendungen mit kleinen übertragbar.

Um nun die Effektivität der zuvor vorgestellten Ansätze zu untersuchen, hat der Autor die jeweiligen Kompressionsleistungen auf Basis von 182 XML-Dateien mit Größen zwischen 129 und 6.645 Bytes miteinander verglichen. In diesen Testdatensätzen sind acht verschiedene XML-Sprachen vertreten. Die verwendeten Dateien entstammen der WBXML-Bibliothek [70] und werden hier als Demonstrationsmaterial mitgeliefert. Sie eignen sich für Vergleichsmessungen vor allem deshalb, weil so auch der WBXML-Kompressor in den Vergleich mit einbezogen werden kann – dieser unterstützt schließlich nur einige ausgewählte XML-Sprachen.

Die Messergebnisse sind in Tabelle 4.1 dargestellt. Sie zeigt in der ersten Datenzeile zunächst die Werte für die unkomprimierte Textdarstellung der XML-Dokumente. In den folgenden Zeilen hat der Autor die Kompressionsleistungen der beiden generischen Kompressoren bzip2 und gzip mit denen der drei XML-spezifischen Kompres-

Codierung	S_{total}	S_{min}	S_{max}	$S_{\varnothing}$	λ_{best}	λ_{worst}	$\lambda_{\varnothing}$	s_{λ}
XML (unkompr.)	191.856	129	6.645	1.054,15	1,00	1,00	1,00	0,00
XML (text, bzip2)	80.296	157	1.253	441,19	0,16	1,22	0,62	0,31
XML (text, gzip)	73.439	153	1.202	403,51	0,16	1,19	0,58	0,30
XMill (bzip2)	87.646	195	1.282	481,57	0,18	1,51	0,73	0,43
XMill (gzip)	72.720	153	1.085	399,56	0,16	1,19	0,58	0,32
XMill (ppm)	68.981	145	1.064	379,02	0,15	1,12	0,55	0,30
xmlppm	62.598	75	1.099	343,95	0,14	0,70	0,43	0,14
WBXML	28.664	7	1.164	157,49	0,05	0,25	0,14	0,03

Tabelle 4.1: Kompressionsergebnisse für kleine XML-Dokumente, alle Größenangaben in Bytes

soren XMill, xmlppm und WBXML verglichen. Eine Besonderheit ist bei XMill zu beachten: Wie bereits beschrieben, lassen sich bei XMill verschiedene generische Kompressoren einstellen, die bei der Codierung der einzelnen XMill-Container zum Einsatz kommen. Es war zu vermuten, dass diese Einstellung die Kompressionsleistung signifikant beeinflusst. Daher hat der Autor separate Messreihen für die drei in der XMill-Implementierung enthaltenen Container-Kompressoren durchgeführt: XMill (gzip), XMill (bzip2) und XMill (ppm).

Zu jeder Codierungsvariante wurde die resultierende Gesamtgröße aller 182 Dateien ermittelt (S_{total}). Die Werte S_{min} und S_{max} repräsentieren die kleinste bzw. größte Datei in der jeweiligen Codierung. $S_{\varnothing}$ ist die über alle Dokumente gemittelte Dateigröße $S_{\text{total}}/182$. Die Verbesserung gegenüber der unkomprimierten Textdarstellung wird in Form der Kompressionsrate λ dargestellt, sie ist der Quotient $S_{\text{komprimiert}}/S_{\text{Text}}$. Kleinere Werte zeigen also bessere Kompressionsleistungen an. Es sind jeweils die beste (λ_{best}), die schlechteste (λ_{worst}) und auch die durchschnittliche Kompressionsrate ($\lambda_{\varnothing}$) dargestellt. Es sei angemerkt, dass der Durchschnittswert $\lambda_{\varnothing}$ nicht mit der Dateigröße gewichtet ist. Folglich ist dieser Wert auch nicht identisch mit dem Quotienten aus den S_{total} Werten der komprimierten und unkomprimierten Darstellung. Weiterhin zeigt die Tabelle die Standardabweichung s_{λ}. Sie gibt an, ob die erreichten Kompressionsraten für alle 182 Dateien eher konstant sind (kleine Werte für s_{λ}) oder ob sie sich in einem weiten Wertebereich bewegen (große Werte für s_{λ}).

Aus dem Vergleich der verschiedenen Werte für $\lambda_{\varnothing}$ geht hervor, dass die beiden generischen Kompressoren bzip2 und gzip im Vergleich zu den verschiedenen XMill-Testreihen vergleichsweise gut abschneiden. Obwohl sie die XML-Struktur bei der

Kompression nicht berücksichtigen, entsprechen ihre Kompressionsraten etwa denen der XMill-Varianten. Folglich lohnt sich bei kleinen XML-Dokumenten die Aufteilung der Daten auf verschiedene Container offenbar nicht. Durch die mehrfache Anwendung der generischen Kompressoren auf die Inhalte der einzelnen XMill-Container nimmt der Overhead durch Codetabellen und Wörterbücher zu. Dies wiegt offenbar die Vorteile des XMill-Ansatzes in etwa wieder auf.

Ein Vergleich zwischen gzip und bzip2 zeigt, dass gzip geringfügig bessere Ergebnisse auf solch kleinen Dateien produziert. Dies gilt sowohl für die Containerkompression in Verbindung mit XMill als auch bei direkter Anwendung auf die XML-Textdateien.

Der Kompressor xmlppm liefert nach WBXML klar die zweitbesten Ergebnisse. Besonders auffällig ist, dass xmlppm selbst sehr kleine Dateien noch weiter komprimieren kann ($\lambda_{worst} < 1$). Auch die Standardabweichung s_λ ist hier deutlich kleiner als bei XMill und den generischen Kompressoren. Dies deutet darauf hin, dass xmlppm sämtliche Dateien vergleichsweise gut komprimiert hat.

Weiterhin zeigen die Messungen deutlich, dass WBXML der mit Abstand effektivste XML-Kompressor in dieser Messreihe ist. Dieses gute Ergebnis kommt vor allem dadurch zu Stande, dass die Codetabellen bei diesem Ansatz im Kompressionsalgorithmus für die unterstützten XML-Sprachen fest vorgeben sind. Folglich muss in der komprimierten Nachricht lediglich ein kurzer, eindeutiger Bezeichner angegeben sein, der dem Dekompressor die vorliegende XML-Sprache signalisiert, so dass dieser die hierfür passenden Codetabellen bei der Dekompression auswählt. Die Tabellen selbst müssen aber – und dies ist der wesentliche Unterschied zu den anderen Ansätzen – nicht mit übertragen werden. Damit ist WBXML auf kleinen Dateien etwa doppelt so effektiv wie xmlppm. Auch der Wert 0,03 für die Standardabweichung s_λ ist mit Abstand der beste. Es ist zu vermuten, dass Millau als verbesserte Variante von WBXML diese Ergebnisse sogar noch übertreffen kann.

Allerdings lassen sich WBXML und auch Millau wegen ihrer inhärenten Beschränkung auf XML-Sprachen mit einem begrenzten Vorrat an Tag- und Attributnamen nicht zur Kompression von SOAP-Nachrichten verwenden. Der beste aller universell einsetzbaren Kompressoren für die Daten aus unserer Vergleichsmessung ist damit xmlppm.

4.4 Architektur

Im Folgenden stellt der Autor eine neuartige Datenkompressionstechnik vor, welche speziell auf eine besondere Charakteristik der SOAP-Kommunikation zugeschnitten ist: Betrachtet man den ein- und ausgehenden Datenverkehr eines Web Services, so fallen erhebliche Ähnlichkeiten zwischen einzelnen SOAP-Nachrichten auf.

Der Autor verdeutlicht diese Beobachtung anhand eines Beispiels: Ein Web Service stellt eine RPC-Operation `int add(int in0, int in1)` bereit. Bei Aufruf die-

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:add
      soapenv:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://somename.test/calculator">
      <in0 xsi:type="xsd:int">4</in0>
      <in1 xsi:type="xsd:int">9</in1>
    </ns1:add>
  </soapenv:Body>
</soapenv:Envelope>
```

(a) SOAP-Darstellung der RPC-Anfrage `add(4,9)`

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:add
      soapenv:encodingStyle= "http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://somename.test/calculator">
      <in0 xsi:type="xsd:int">5</in0>
      <in1 xsi:type="xsd:int">7</in1>
    </ns1:add>
  </soapenv:Body>
</soapenv:Envelope>
```

(b) SOAP-Darstellung der RPC-Anfrage `int add(5,7)`

Abbildung 4.4: Zeitlich hintereinander aufgezeichnete SOAP-Nachrichten als Eingabe einer RPC-Web-Services-Operation `add(int in0, int in1)`

ser Operation addiert er die beiden übergebenen Werte `in0` und `in1` und schickt die Summe an den Aufrufer zurück. Abbildung 4.4 zeigt exemplarisch zwei Beispiele für eingehende Nachrichten dieser Web-Service-Operation. Wie man leicht sieht, stimmen die Nachrichten mit Ausnahme der beiden Werte für `in0` und `in1` (jeweils im Fettdruck dargestellt) vollständig überein.

Aus technischer Sicht ist diese Beobachtung nicht weiter verwunderlich: Da jeder Web Service bestimmte Datentypen für ein- und ausgehende Nachrichten verwendet, die der Programmierer bei der Entwicklung fest vorgegeben hat, sind Nachrichten, die zu einer Operation gehören, zwangsläufig auch immer ähnlich strukturiert.

Aus der Sicht der Informationstheorie ist diese große Ähnlichkeit zwischen einzelnen SOAP-Nachrichten zunächst ein Indiz dafür, dass die hierin enthaltene Informationsmenge in einzelnen, abgegrenzten Bereichen konzentriert ist. Da ein Großteil der

Nachricht unveränderliche Textinformationen trägt, wird durch die Kenntnis dieser Bereiche bei der Informationssenke (Empfänger) keinerlei Unsicherheit beseitigt – die Auftrittswahrscheinlichkeit dieser statischen Bereiche x_{statisch} ist somit $P(x_{\text{statisch}}) = 1$, und folglich ergibt sich ein Informationsgehalt von $H(x_{\text{statisch}}) = 0$ Bits. Lediglich die veränderlichen Bereiche x_{variabel} beseitigen Unsicherheit bei der Informationssenke und haben somit einen positiven Informationsgehalt $H(x_{\text{variabel}}) > 0$ Bits.

Ausgehend von dieser Überlegung hat der Autor einen neuartigen Quellencodierungsansatz für SOAP-Nachrichten entwickelt, bei dem beide Bereiche zunächst separiert werden. Nur der variable Anteil wird codiert und schließlich in einer Netzwerknachricht übertragen.

Als konzeptionelle Basis dient dabei der gängige und bereits in Abschnitt 3.7 ausführlich erläuterte Differenzcodierungsansatz. Dieser ermöglicht eine sehr effiziente Codierung von Nachrichtensequenzen, bei denen aufeinander folgende Nachrichten einen hohen Grad an Ähnlichkeit aufweisen, indem nur die Änderungen zu der vorhergehenden Nachricht übertragen werden. In Rechnernetzen wird eine Differenzcodierung bereits erfolgreich zur Kompression von Protokoll-Headern eingesetzt [35, 20, 9, 40].

Allerdings lässt sich der Ansatz der Differenzcodierung nicht direkt auf SOAP-Nachrichten übertragen: Da SOAP ein zustandsloses Protokoll ist, gibt es hier keine Vorgängernachricht, zu der man die Differenz bestimmen könnte.

Daher kann die SOAP-Differenzcodierung nicht auf der Differenzbildung zwischen zwei aufeinander folgenden Nachrichten basieren. Stattdessen wird die Differenz zwischen der zu übertragenden Nachricht und einer so genannten *Skelettnachricht* berechnet, welche aus der WSDL-Beschreibung eines Web-Services vorab generiert wird. Eine Skelettnachricht ist eine „leere“ SOAP-Nachricht, die zwar bereits sämtliche statischen Bereiche der SOAP-Nachricht enthält, nicht aber die variablen. Anders ausgedrückt: Auf Basis des WSDL-Dokuments lässt sich bereits recht genau vorhersagen, wie die ausgetauschten SOAP-Nachrichten aussehen werden – mittels dieser Vorhersage werden Skelettnachrichten für die einzelnen Web-Service-Operationen erzeugt.

Die WSDL-Beschreibung dient bekanntlich dazu, einem Dienstnutzer mitzuteilen, wie gültige ein- und ausgehende SOAP-Nachrichten aufgebaut sind. Diese in der WSDL-Beschreibung enthaltenen Strukturinformationen werden nunmehr zur Erstellung von Skelettnachrichten wie folgt ausgenutzt:

Für jede hier aufgeführte Web-Service-Operation lassen sich Skelettnachrichten berechnen. Je nachdem, ob es sich dabei um eine Operation mit eingehenden, ausgehenden oder sowohl eingehenden als auch ausgehenden Nachrichten handelt, gibt es pro Operation entweder ein oder zwei Skelettnachrichten.

Da die Skelettnachrichten sowohl zur Codierung als auch zur Decodierung benötigt werden, müssen sie sowohl vom Dienstnutzer als auch vom Dienstanbieter berechnet werden. Sie bleiben – genau wie die WSDL-Beschreibung – über die gesamte Lebensdauer eines Web Services konstant, und folglich ist eine dynamische Berechnung der

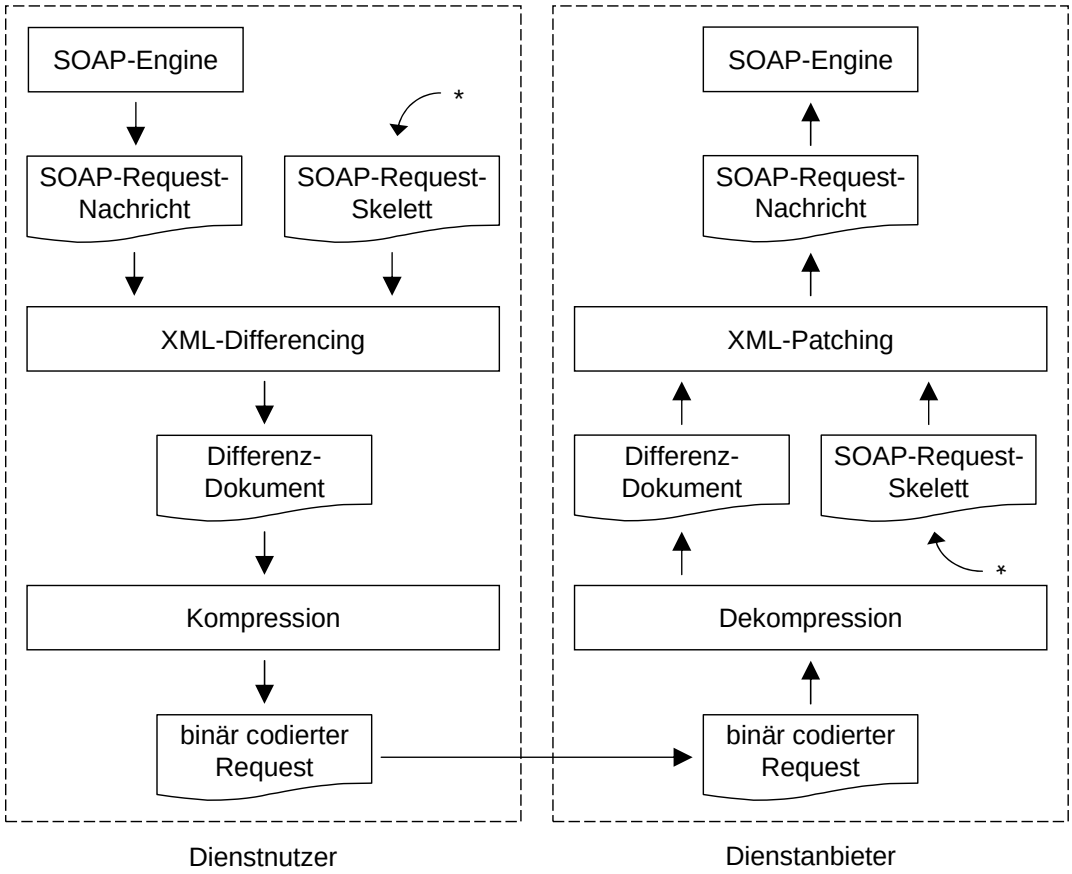


Abbildung 4.5: Schematische Darstellung der Differenzcodierung einer SOAP-Nachricht

Skelettnachrichten zur Laufzeit nicht erforderlich. Der Dienstanbieter generiert sie bei Inbetriebnahme des Dienstes, d. h. während des so genannten „Deployments“, und der Dienstnutzer generiert sie vor dem ersten Dienstaufruf. Für die Funktionsweise des Verfahrens ist es unabdingbar, dass Dienstanbieter und Dienstnutzer den selben Algorithmus zur Berechnung der Skelettnachrichten verwenden.

Abbildung 4.5 zeigt ein schematisches Schaubild des Verfahrens: Ein Dienstnutzer erzeugt mit Hilfe seiner SOAP-Engine eine SOAP-Nachricht. Diese soll an den Dienstanbieter gesendet werden. Im nächsten Schritt wird die Differenz zwischen dieser Nachricht und der zugehörigen, bereits vorab generierten Skelettnachricht berechnet; diesen Prozess bezeichnen wir als *XML-Differencing*. Das entstehende Differenzdokument wird dann in einer weiteren Verarbeitungsstufe mit Hilfe eines Kompressors in eine möglichst kompakte Binärdarstellung umgewandelt und zum Dienstanbieter übertragen. Dieser macht die Kompression zunächst rückgängig und führt danach

das Differenzdokument mit seiner Kopie der passenden SOAP-Skelettnachricht wieder zusammen. Dieser Prozess heißt *XML-Patching* und rekonstruiert die ursprüngliche Request-Nachricht, welche schließlich an die SOAP-Engine des Dienstanbieters weitergeleitet und dort verarbeitet wird. Würde der Dienstanbieter nun eine Antwortnachricht an den Dienstnutzer zurückschicken, so liefe der gesamte Prozess erneut, aber in umgekehrter Richtung ab.

Ausschlaggebend für die Effektivität dieses Ansatzes ist offenbar die Ähnlichkeit zwischen den generierten Skelettnachrichten und den eigentlichen SOAP-Nachrichten, welche von der sendenden SOAP-Engine erzeugt werden. Je kleiner hier die Abweichungen sind, desto kürzer sind die Differenzdokumente. Ein grundlegendes Problem besteht allerdings darin, dass die SOAP-Spezifikation gewisse Freiräume im Nachrichtenformat gestattet. Diese werden auch durch die WSDL-Beschreibung eines Dienstes nicht eingeschränkt.

Zum Beispiel sieht die SOAP-Spezifikation vor, dass das im SOAP-Body enthaltene Element einer RPC-Antwort einen beliebigen Namen tragen kann (vergleiche Aufzählung auf Seite 26). Existierende SOAP-Engines machen von diesem Freiheitsgrad zum Teil auch Gebrauch, so dass sich bisher keine einheitliche Konvention für die Namensgebung durchgesetzt hat. Dies führt zu Größenschwankungen der Differenzdokumente in Abhängigkeit von der verwendeten SOAP-Engine. Da diese in der SOAP-Spezifikation vorgesehenen Freiheitsgrade jedoch zu keinerlei technischen Vorteilen führen, sondern die Implementierung des SOAP-Protokolls eher erschweren, bleibt abzuwarten, ob auch künftige SOAP-Versionen solche Freiheitsgrade vorsehen werden.

Vorteilhaft dagegen ist die Tatsache, dass die durchschnittliche Effektivität der Differenzcodierung nicht davon abhängt, ob wirklich jede zu codierende SOAP-Nachricht ihrem korrespondierenden Skelett ähnlich ist: Weichen ausnahmsweise einzelne Nachrichten von der Struktur der Skelettnachricht ab, sind lediglich die zugehörigen Differenzdokumente größer. Werden aber viele Nachrichten ausgetauscht – und hiervon ist in der Praxis auszugehen – haben solche Ausnahmen kaum Einfluss auf die Effektivität im Mittel. Bei der Erzeugung der Skelettnachrichten kommt es also darauf an, dass die Struktur der Skelettdatensätze möglichst genau mit dem SOAP-Format übereinstimmt, das von den gängigsten SOAP-Engines erzeugt wird.

Bei dieser Form der Differenzcodierung ist es sogar möglich, Nachrichten zu übertragen, die vollständig vom vorhergesagten Skelettdatensatz abweichen. Dies könnte etwa passieren, wenn bei der Erzeugung der SOAP-Nachricht ein Fehler, ein so genannter *SOAP Fault*, aufgetreten ist. Zwar führt die Differenzcodierung in einem solchen Fall nicht zu kompakten Nachrichten, wichtig ist jedoch, dass sich in einer solchen Fehlersituation überhaupt Nachrichten zwischen den beiden Kommunikationspartnern übertragen lassen.

4.5 Implementierung

Zur Implementierung des vorgestellten Ansatzes werden drei Komponenten benötigt:

1. ein Werkzeug zur Erzeugung der Skelettdatensätze aus der WSDL-Beschreibung
2. ein Werkzeug für XML-Differencing und XML-Patching
3. ein geeignetes Datenformat zur Beschreibung der Differenzdokumente

Für die Erzeugung der Skelettdatensätze hat der Autor ein XSLT-Programm namens WSDL2SOAP prototypisch implementiert. XSLT [169] steht für *Extensible Stylesheet Language Transformations* und ist eine vom W3C standardisierte Programmiersprache zur Transformation von XML-Dokumenten. WSDL2SOAP liest also die WSDL-Beschreibung eines Web Services ein und erzeugt für jede Operation Skelette für ein- und ausgehende Nachrichten.

Zu den Komponenten 2 und 3 in der oben angegebenen Liste existieren bereits einige Forschungsarbeiten. Zu nennen sind hier vor allem die Arbeiten von Zhang et al. [189] sowie von Chawathe et al. [21]. Eine gut verständliche Zusammenfassung der gegenwärtig bekannten Lösungsansätze sowie weitere Erläuterungen zu diesem Themenbereich findet der Leser auch in [98].

Grundsätzlich lässt sich die Berechnung der Differenz zweier XML-Dokumente auf die Berechnung der „editing distance“ zweier ungerichteter azyklischer Graphen G und G' reduzieren; diese Problemstellung haben erstmalig Zhang et al. in [189] ausführlich beschrieben und untersucht. Hierbei geht es darum, den Graphen G mit Hilfe der drei Änderungsanweisungen „Knoten einfügen“, „Knoten löschen“ und „Knoten umbenennen“ so in den Graphen G' umzuwandeln, dass die Anzahl der dafür benötigten Operationen minimal ist. Zur Lösung dieser Problemstellung sind im Wesentlichen zwei Basisalgorithmen bekannt: der EZS-Algorithmus [189] und der FMES-Algorithmus [21].

Im Rahmen dieser Arbeit wurde die Thematik der Differenzbildung zwischen zwei XML-Dokumenten nicht näher untersucht. Der Autor hat stattdessen zwei frei verfügbare Implementierungen verwendet, die XML-Differencing und -Patching unterstützen.

Die erste heißt *diffxml* [99]. Sie basiert auf dem FMES-Algorithmus und verwendet als Ausgabeformat für Differenzdokumente die *Delta Update Language (DUL)*. DUL ist eine proprietäre Sprache, die zurzeit ausschließlich im Zusammenhang mit diffxml verwendet wird; in [98] ist der Sprachumfang genau beschrieben. Neben XML-Differencing unterstützt dieses Werkzeug auch XML-Patching.

Die zweite Implementierung heißt *xmldiff*. Sie basiert auf dem EZS-Algorithmus und unterstützt zwei verschiedene Ausgabeformate: zum einen ein proprietäres, nicht-XML-basiertes Textformat, für welches allerdings kein Patch-Werkzeug verfügbar ist,

und zum anderen die Ausgabe von XUpdate-Dokumenten. XUpdate ist ein Industriestandard, der von der *XML:db Initiative* (`http://xmldb-org.sourceforge.net/`) entwickelt wurde. Eine detaillierte Beschreibung von XUpdate findet der Leser in [187]. Für XUpdate-Dokumente gibt es beispielsweise das frei verfügbare Patch-Werkzeug *4xupdate*. Es basiert auf der Programmiersprache Python und ist Bestandteil der XML-Funktionsbibliothek *4Suite* [45].

Neben *diffxml* und *xmldiff* gibt es noch eine Reihe weiterer XML-Differencing-Implementierungen, welche sich allerdings nicht für die SOAP-Differenzcodierung eignen, da sie keine Differenzdokumente erzeugen, sondern lediglich die Unterschiede zwischen zwei XML-Dokumenten in Form einer farblich differenzierten Darstellung am Bildschirm ausgeben. Eine Auflistung findet der Leser in [98].

4.6 Beispiel

Im Folgenden demonstriert der Autor die SOAP-Differenzcodierung anhand eines praktischen Beispiels. Betrachten wir einen Web Service, der lediglich eine RPC-Operation `String concatString(String in0, String in1, String in2)` bereitstellt. Diese liefert als Ergebnis die Konkatenation der drei übergebenen Strings.

In einem ersten Schritt wird die WSDL-Beschreibung, welche automatisch von der SOAP-Engine des Dienstanbieters generiert wurde, mit Hilfe des oben genannten XSLT-Programms WSDL2SOAP verarbeitet. WSDL2SOAP wird hierzu einmal auf dem System des Dienstanbieters und einmal auf dem des Dienstnutzers ausgeführt. Auf beiden Systemen generiert WSDL2SOAP jeweils zwei Skelettnachrichten – eine für Request-Nachrichten von `concatString` und eine weitere für Response-Nachrichten. Die Skelettnachricht für Requests ist exemplarisch in Abbildung 4.6(a) dargestellt.

Zur Laufzeit erzeugt die SOAP-Engine des Dienstnutzers dann für jeden RPC-Aufruf eine SOAP-Nachricht, Abbildung 4.6(b) zeigt ein Beispiel. Diese Nachricht wird mit der zugehörigen, zuvor erzeugten Skelettnachricht verglichen und ein Differenzdatensatz berechnet. Dessen Format hängt dabei vom verwendeten Differencing-Werkzeug ab. Abbildung 4.7(a) zeigt ein Beispiel für das DUL-Format, zum Vergleich findet der Leser in Abbildung 4.7(b) den gleichen Differenzdatensatz als XUpdate-Dokument.

Wie in den Abbildungen ganz deutlich wird, ist das Differenzdokument zunächst nicht wesentlich kleiner als die zu codierende SOAP-Nachricht. Daher kommt ein XML-Kompressor (beispielsweise xmlppm) zum Einsatz, der das Differenzdokument in eine binäre Darstellung umwandelt. Die regelmäßigen, sich wiederholenden Strukturen von DUL- und XUpdate-Dokumenten ermöglichen deutlich bessere Kompressionsergebnisse als die direkte Kompression der SOAP-Nachricht ohne den zusätzlichen

```
<?xml version="1.0" encoding="UTF-8"?> <soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:concatString
      xmlns:ns1="http://somename.test/concat"
      soapenv:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/">
      <in0 xsi:type="xsd:string"/>
      <in1 xsi:type="xsd:string"/>
      <in2 xsi:type="xsd:string"/>
    </ns1:concatString>
  </soapenv:Body>
</soapenv:Envelope>
```

(a) Skelettnachricht

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:concatString
      soapenv:encodingStyle=
            "http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:ns1="http://somename.test/concat">
      <in0 xsi:type="xsd:string">foo</in0>
      <in1 xsi:type="xsd:string">bar</in1>
      <in2 xsi:type="xsd:string">foo</in2>
    </ns1:concatString>
  </soapenv:Body>
</soapenv:Envelope>
```

(b) SOAP-RPC-Request

Abbildung 4.6: Perfekte Identifikation statischer Nachrichtenanteile

Schritt der Differenzcodierung. (Genaue Ergebnisse hierzu präsentiert der Autor im folgenden Abschnitt 4.7).

Die entstehende Binärdarstellung wird dann zum Dienstanbieter übertragen. Damit der Dienstanbieter die richtige Skelettnachricht für den XML-Patching-Prozess auswählen kann, muss allerdings noch eine zusätzliche Information übermittelt werden: eine eindeutige Referenz auf die Web-Service-Operation. Eine Möglichkeit besteht hier darin, eine lexikographische Sortierung aller im WSDL-Dokument aufgeführten Operationen vorzunehmen (sämtliche Operationsnamen sind laut WSDL-Spezifikation eindeutig). Die Position der vorliegenden Operation in der sortierten Liste wird dann zusammen mit dem Differenzdokument als numerischer Wert übertragen.

```
<?xml version="1.0" encoding="UTF-8"?>
<delta>
  <insert charpos="1" childno="1" name="#text" nodetype="3"
   parent="/node()[1]/node()[2]/node()[2]/node()[2]">foo</insert>
  <insert charpos="1" childno="1" name="#text" nodetype="3"
   parent="/node()[1]/node()[2]/node()[2]/node()[4]">bar</insert>
  <insert charpos="1" childno="1" name="#text" nodetype="3"
   parent="/node()[1]/node()[2]/node()[2]/node()[6]">foo</insert>
</delta>
```

(a) Differenzinformationen als DUL-Dokument

```
<?xml version="1.0"?>
  <xupdate:modifications version="1.0"
    xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:append select="/soapenv:Envelope[1]/
        soapenv:Body[1]/ns1:concatString[1]/in0[1]" child="first()">
        <xupdate:text>foo</xupdate:text>
    </xupdate:append>
    <xupdate:append select="/soapenv:Envelope[1]/
        soapenv:Body[1]/ns1:concatString[1]/in1[1]" child="first()">
        <xupdate:text>bar</xupdate:text>
    </xupdate:append>
    <xupdate:append select="/soapenv:Envelope[1]/
        soapenv:Body[1]/ns1:concatString[1]/in2[1]" child="first()">
    <xupdate:text>foo</xupdate:text>
  </xupdate:append>
</xupdate:modifications>
```

(b) Differenzinformationen als XUpdate-Dokument

Abbildung 4.7: Unterschiedliche Datenformate zur Darstellung von XML-Differenzinformationen

In einem letzten Schritt dekomprimiert der Dienstanbieter das Differenzdokument und führt es mit dem zugehörigen SOAP-Skelett zusammen. Es entsteht eine Rekonstruktion der originalen SOAP-Nachricht, welche schließlich an die lokale SOAP-Engine weitergeleitet und dort verarbeitet wird. Die Antwort auf diesen RPC-Aufruf wird dann an den Dienstnutzer zurückgeschickt – wobei hier ebenfalls die Differenzcodierung zum Einsatz kommen kann.

4.7 Evaluation

Um die Effektivität der Differenzcodierung anhand realistischer Testdaten mit anderen Ansätzen vergleichen zu können, hat der Autor einen Web Service implementiert, der drei verschiedene RPC-Operationen bereitstellt:

- `void doNothing()`,
- `String echoString(String in0)` und
- `String concatString(String in0, String in1, String in2, String in3, String in4)`.

Jede dieser drei Operationen erzeugt bzw. verarbeitet Nachrichten, die einen unterschiedlich hohen Anteil an statischen und variablen Bestandteilen haben. Die Operation `doNothing` nimmt keinerlei Parameter entgegen und liefert auch keine Berechnungsergebnisse zurück. Folglich sind Request und Response vollständig statisch. Die Operation `echoString` liefert eine Kopie des übergebenen Strings zurück, und `concatString` berechnet die Konkatenation der fünf übergebenen Strings. Diese beiden Operationen arbeiten also mit einem kleinen bzw. größeren Anteil variabler Nachrichtenbestandteile.

Für jede dieser Operationen wurden die Request- und Response-Nachrichten aufgezeichnet und in Dateien abgespeichert. Als Aufrufparameter für `echoString` und `concatString` dienten dabei zufällig erzeugte Strings mit einer konstanten Länge von fünf Zeichen.

Sämtliche Request- und Response-Datensätze wurden zunächst mit xmlppm komprimiert, dem effektivsten Kompressor für SOAP-Nachrichten (siehe Abschnitt 4.3).

Zum Vergleich wurde dann die Differenzcodierung durchgeführt: Aus der WSDL-Beschreibung des Web Services wurden Skelettnachrichten für Requests und Responses generiert. Mit Hilfe der Werkzeuge *diffxml* und *xmldiff* hat der Autor dann die aufgezeichneten SOAP-Nachrichten mit ihren korrespondierenden Skelettnachrichten verglichen. Die Ergebnisse dieses Vergleichs wurden jeweils als DUL- (*diffxml*) bzw. XUpdate-Dokument (*xmldiff*) abgespeichert. Auch diese Differenzdokumente wurden anschließend mit Hilfe des Kompressors xmlppm komprimiert.

Die Ergebnisse dieses Versuchs sind in Abbildung 4.8 dargestellt:

Mit Hilfe von xmlppm allein kann die Dateigröße jeweils um etwas mehr als die Hälfte reduziert werden. Dieses Ergebnis stimmt auch annähernd mit der durchschnittlichen Kompressionsrate dieses Kompressors aus Abschnitt 4.3 überein (Tabelle 4.1 auf Seite 77, Spalte „xmlppm“, $\lambda_{\varnothing} = 0,43$).

Bei der Messung „Differenz (XUpdate, unkomprimiert)“ zeigt sich der erwartete Messwertverlauf: Bei `doNothing` stimmen die generierten Skelettdatensätze vollständig mit den aufgezeichneten SOAP-Nachrichten überein, denn Request und Response sind vollständig statisch und enthalten keinerlei variable Bestandteile. Folglich müssen in den entsprechenden XUpdate-Dokumenten keinerlei Änderungsanweisungen codiert werden, was zu einer sehr geringen Dateigröße führt. Bei der Request- und Response-Nachricht von *echoString* wird als variabler Nachrichtenbestandteil nur jeweils ein einzelner String übertragen. Gleiches gilt für die Response-Nachricht von

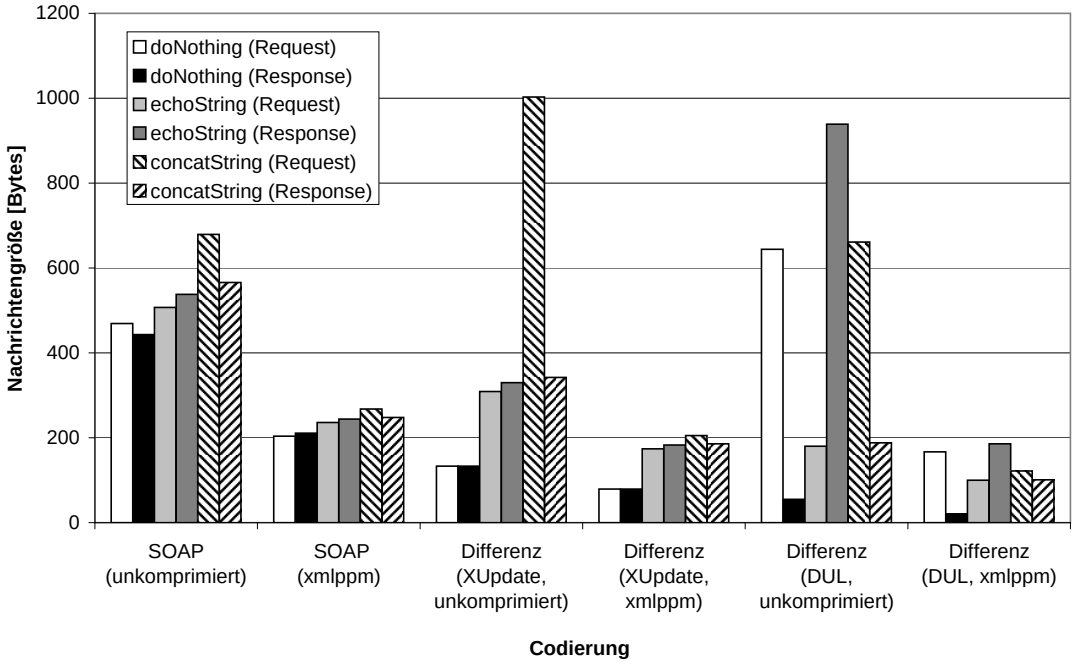


Abbildung 4.8: Effizienz verschiedener Codierungsvarianten für SOAP-Nachrichten

`concatString`. Das Einfügen dieses Strings in den Skelettdatensatz muss also in den erzeugten Differenzdokumenten codiert werden. Die resultierenden Dateigrößen schwanken zwischen 309 und 342 Bytes. Die Abweichungen kommen hier durch unterschiedliche Elementnamen zu Stande, die neben dem eigentlichen String auch mit codiert werden müssen. Bei der Request-Nachricht für `concatString` müssen fünf Strings eingefügt werden. Es ergibt sich daher ein deutlich größeres Differenzdokument mit 1.003 Bytes.

Die Werte für „Differenz (DUL, unkomprimiert)“ zeigen hingegen zum Teil unerwartete Ergebnisse: Das Werkzeug *diffxml* hat die Äquivalenz zwischen der `doNothing` Request-Nachricht und dem zugehörigen Skelettdatensatz offenbar nicht erkannt, obwohl die Skelettnachricht inhaltlich mit der SOAP-Nachricht völlig übereinstimmt. Hier werden daher unnötigerweise Änderungsanweisungen im DUL-Dokument codiert, und die resultierende Datei fällt sehr viel größer aus als erwartet. Dasselbe Problem tritt auch bei der Response-Nachricht von `echoString` auf – auch sie ist deutlich zu groß.

Der Grund für diese Abweichungen wird deutlich, wenn man die algorithmische Grundlage der Differenzbildung jeweils mit betrachtet: Zur Erzeugung des DUL-Dokuments kommt der FMES-Algorithmus zum Einsatz. Dieser zeigt zwar ein deutlich besseres Laufzeitverhalten als der EZS-Algorithmus, welcher zur Erzeugung der XUpdate-Dokumente eingesetzt wird, jedoch kann der FMES-Algorithmus die Optimalität der gefundenen Lösung nicht garantieren [21]. Optimal bedeutet in diesem

Zusammenhang, dass die Anzahl von Änderungsanweisungen im Differenzdokument minimal ist. Der FMES-Algorithmus ist somit besonders für die Verarbeitung großer XML-Dokumente geeignet. Bei derart kleinen Dateien steht hingegen nicht so sehr das Laufzeitverhalten im Vordergrund, sondern die Qualität der gefundenen Lösung. Hier ist demzufolge der EZS-Algorithmus besonders vorteilhaft.

Vergleicht man die Größen der unkomprimierten DUL-Dokumente mit denen der XUpdate-Dateien, fällt auf, dass die DUL-Darstellung (abgesehen von den oben genannten Abweichungen) deutlich kompakter ist. Wünschenswert wäre also die Kombination des EZS-Algorithmus mit dem DUL-Ausgabeformat.

Die Abbildung zeigt weiterhin die Dateigrößen bei zusätzlicher Anwendung des Kompressors xmlppm auf die DUL- und XUpdate-Dokumente. Bei allen Datensätzen kann die Dateigröße hierdurch noch einmal deutlich verringert werden. Damit sind die mit xmlppm komprimierten Differenzdokumente nicht nur erheblich kleiner als die unkomprimierten SOAP-Nachrichten, sie sind auch stets kleiner als die SOAP-Nachrichten, die direkt mittels xmlppm komprimiert wurden.

Der Vorteil, der durch die Differenzcodierung entsteht, ist dann am größten, wenn die SOAP-Nachricht einen sehr großen Anteil an statischen Bereichen aufweist (wie bei `doNothing`). Wenn der Anteil der variablen Bereiche zunimmt, schwindet auch der Vorteil gegenüber der direkten Kompression von SOAP-Nachrichten mittels xmlppm.

4.8 Ergebnis

Zunächst wurden in diesem Kapitel bereits existierende Ansätze für die Kompression von XML-Daten vorgestellt. Das WBXML-Format ist besonders kompakt, allerdings erlaubt der zu Grunde liegende Kompressionsansatz nur die Verarbeitung ausgewählter XML-Sprachen – für SOAP ist er ungeeignet. Daher bietet sich hier der xmlppm Kompressor an. Dieser kann beliebige XML-Dokumente verarbeiten und erzielt auch auf kleinen Datensätzen akzeptable Kompressionsraten.

Weiterhin wurde ein neuartiges Quellencodierungsverfahren erläutert, das speziell auf die Anforderungen von SOAP zugeschnitten ist. Dienstanbieter und -nutzer berechnen vor dem eigentlichen Kommunikationsvorgang die statischen Anteile möglicher SOAP-Nachrichten aus der WSDL-Beschreibung und legen diese als Skelettdatensätze ab. Zur Laufzeit wird nur die Differenz zwischen der SOAP-Nachricht und dem zugehörigen Skelettdatensatz übertragen.

Messungen zeigen, dass die Differenzcodierung in Verbindung mit einem XML-Kompressor wie xmlppm SOAP-Dokumente im Idealfall auf ein Zwanzigstel ihrer ursprünglichen Größe komprimieren kann. Auch im Vergleich zu SOAP-Dokumenten, welche direkt mit xmlppm komprimiert wurden, ergibt sich durch die Differenzcodierung noch immer eine deutliche Verringerung des Datenvolumens – und zwar bis auf

ein Zehntel. Die Effektivität hängt dabei maßgeblich davon ab, wie groß der Anteil an statischen Bereichen im SOAP-Dokument ist.

Bei der Interpretation dieser Ergebnisse ist allerdings zu beachten, dass die SOAP-Spezifikation in ihrer gegenwärtigen Fassung gewisse Freiräume im Nachrichtenformat vorsieht. So ist – wie schon weiter oben dargestellt – der Name des Kindelements des SOAP-Bodys in RPC-Response-Nachrichten beliebig. Folglich kann jede SOAP-Engine diesen Namen frei wählen. Bei praktischen Anwendungen mit SOAP-Engines verschiedener Hersteller kommt es also vor, dass hierbei unterschiedliche Namenskonventionen auftreten. Solche Unterschiede wirken sich bei der Differenzcodierung negativ aus. Bei den vorgestellten Untersuchungen wurden die Skelettnachrichten mit den Namenskonventionen erzeugt, die auch die verwendete SOAP-Engine „Apache Axis“ nutzt. Versuchsmessungen mit der Microsoft .NET SOAP-Engine unter Beibehaltung der Axis-Skelettdatensätze zeigten, dass durch die unterschiedliche Namensgebung die Differenzdokumente etwas größer ausfallen, da zusätzliche Umbenennungsanweisungen codiert werden müssen.

Zudem kamen bei den hier vorgestellten Messungen nur einfache Datentypen zum Einsatz. Verwendet man hingegen auch Listen- oder Array-Typen, lassen sich prinzipbedingt keine detaillierten Skelettdatensätze mehr generieren: Die Anzahl der Elemente in Instanzen dieser Datentypen ist zum Zeitpunkt der Erstellung des Skelettdatensatzes noch unbekannt, und folglich lassen sich die korrespondierenden Markup-Strukturen hier nur grob abschätzen.

Ein weiterer Aspekt, der die Nutzung dieses Ansatzes für praktische Anwendungen einschränkt, ist der Speicherplatz, der für die Umsetzung der Differenzcodierung benötigt wird. Zum einen müssen bei Dienstanbieter und -nutzer die Skelettdatensätze vorgehalten werden. Gerade bei Mobilanwendungen mit begrenztem Festspeicher könnte dies ein kritischer Aspekt sein. Weiterhin müssen zur Berechnung des Differenzdokuments sowohl die SOAP-Nachricht als auch der zugehörige Skelettdatensatz im RAM-Speicher des Geräts geladen sein, was besonders bei Geräten mit sehr begrenztem Hauptspeicher zu Problemen führen könnte.

Ein großer Vorteil dieses Codierungsansatzes ist hingegen die Tatsache, dass sich auch solche Dokumente übertragen lassen, die völlig vom Inhalt des zugehörigen Skelettdatensatzes abweichen. Ein solcher Fall tritt beispielsweise ein, wenn ein Verarbeitungsfehler aufgetreten ist und die sendende SOAP-Engine eine SOAP-Fault-Nachricht zurückgibt. In diesem Fall ist zwar keine Effizienzsteigerung mit der Differenzcodierung verbunden, der Kommunikationsvorgang kann aber dennoch ablaufen – für Spezialfälle wie Fehlerzustände müssen also keine gesonderten Skelettnachrichten vorgehalten werden.

Besonders interessante Möglichkeiten für weitere Arbeiten ergeben sich nach Ansicht des Autors im Bereich der Erzeugung und Darstellung von XML-Differenzdokumenten. Möglicherweise könnte es sich als zweckmäßig erweisen, die Differenzinformatio-

nen direkt binär zu codieren, anstatt sie zunächst als XML-Dokument auszugeben, um sie dann mit Hilfe eines Kompressors wie xmlppm weiter zu verarbeiten.

Technisch interessant erscheint auch der Ansatz, die WBXML-Technik zur Codierung von Differenzinformationen nutzbar zu machen. Da XML-Update-Sprachen wie XUpdate oder DUL über einen sehr begrenzten Vorrat an Attribut- und Elementnamen verfügen, müsste die WBXML-Codierung hier gut funktionieren.

Obwohl die SOAP-Differenzcodierung für praktische Anwendungen noch verfeinert werden müsste, zeigen die präsentierten Messungen bereits sehr deutlich, dass sich mit Hilfe der in der WSDL-Beschreibung enthaltenen Informationen wirkungsvolle Datenkompressionsstrategien realisieren lassen.

Kapitel 5

SOAP-Kompression mittels Kellerautomaten

Die Ergebnisse aus Kapitel 4 zeigen klar, dass die Informationen, die in einer WSDL-Beschreibung enthalten sind, effektive Datenkompressionsstrategien ermöglichen. Allerdings bringt der Ansatz der Differenzcodierung auch einige Nachteile mit sich, die sich vor allem im Bereich mobiler Anwendungen negativ auswirken:

Zum einen müssen Dienstnutzer und Dienstanbieter jeweils die Skelettdatensätze vorhalten. Gerade bei Web Services mit mehreren Operationen führt dies zu einem zusätzlichen Speicherbedarf, der bei Anwendungen auf mobilen Kleinstcomputern mitunter nicht mehr akzeptabel ist. Zum anderen ergibt sich ein mehrstufiger Verarbeitungsprozess (vergleiche Abbildung 4.5 auf Seite 81). Die Differenzcodierung und auch die anschließende Kompression des Differenzdokuments mit Hilfe eines Kompressors wie xmlppm sind zusätzliche Verarbeitungsschritte beim Versenden einer SOAP-Nachricht. Der Empfänger muss beide Schritte rückgängig machen, bevor er das Dokument mit Hilfe seiner SOAP-Engine weiterverarbeiten kann. Dieser zusätzliche Aufwand könnte auf Mobilgeräten mit begrenzten CPU- und RAM-Kapazitäten sicherlich in einigen Fällen problematisch werden.

Wünschenswert wäre also ein Kompressor, der auf die Bedürfnisse von Mobilgeräten zugeschnitten ist und möglichst wenig Zusatzaufwand bei der XML-Verarbeitung verursacht. Ein Beispiel für einen solchen Kompressor haben wir bereits kennen gelernt: WBXML erzeugt mit Hilfe statischer Codetabellen kompakte Binärrepräsentationen, die als Abkürzung für bestimmte Syntaxelemente eines XML-Dokuments dienen. Allerdings eignet sich dieser Ansatz nur für XML-Sprachen, bei denen die Element- und Attributnamen einer begrenzten Wertemenge entstammen.

Im Rahmen dieses Kapitels wird der Autor einen Kompressionsansatz vorstellen, der – ähnlich wie WBXML – hochspezialisiert ist. Allerdings hat die Spezialisierung hier keinen statischen Charakter. Anstatt einen Kompressor zu entwerfen, der nur für ausgewählte XML-Sprachen funktioniert, wird ein algorithmischer Ansatz vorgestellt, der einen spezialisierten Kompressor dynamisch erzeugt. Als Ausgangspunkt dient dabei eine XML-Schema-Beschreibung, die die zu komprimierende XML-Sprache definiert. Bei Web-Service-Anwendungen ist diese XML-Schema-Beschreibung im WSDL-

Dokument des Web Services enthalten. Hieraus wird schrittweise ein Spezialkompressor konstruiert, der nicht nur äußerst kompakte Binärrepräsentationen erzeugt, sondern zusätzlich auch Funktionalitäten zum validierenden Parsen von Dokumenten bereitstellt.

Im Folgenden wollen wir solche Kompressionsansätze, die die Eigenschaften der zu komprimierenden XML-Sprache bei der Kompression mit ausnutzen als *grammatikspezifisch* bezeichnen. Die Klasse der grammatikspezifischen Kompressoren lässt sich dabei in zwei Gruppen einteilen: *statische* und *dynamische*. Vertreter der ersten Gruppe sind dadurch gekennzeichnet, dass die grammatikspezifischen Codierungsregeln fest in den Kompressor integriert sind. Vertreter der zweiten Gruppe bieten dagegen die Möglichkeit, eine Grammatikbeschreibung – beispielsweise in Form einer DTD oder einer XML-Schema-Beschreibung – zur Laufzeit einzulesen und zu verarbeiten. Hier werden die Codierungsregeln also dynamisch erzeugt.

Als Vertreter der ersten Gruppe haben wir bereits WBXML kennen gelernt. Die vom Autor entwickelte Differenzcodierung nimmt zwar zunächst eine Sonderstellung ein, weil sie nicht auf Basis einer reinen XML-Grammatikbeschreibung arbeitet, sondern auf Basis der WSDL-Beschreibung eines Web Services. Da sie aber letztendlich die in einer WSDL-Beschreibung enthaltene XML-Schema-Grammatik verarbeitet, ordnen wir sie der zweiten Gruppe zu.

Da bei Web-Service-Anwendungen beliebige benutzerdefinierte XML-Sprachen zum Einsatz kommen können, sind statisch-grammatikspezifische Kompressoren hier prinzipiell nicht nutzbar. Folglich werden wir uns in diesem Kapitel ganz auf die dynamisch-grammatikspezifischen Kompressoren konzentrieren.

Im folgenden Abschnitt stellt der Autor zunächst verwandte Arbeiten vor. Etwa zeitgleich mit der Differenzcodierung wurden etliche weitere Arbeiten zum Thema dynamisch-grammatikspezifische XML-Kompressoren veröffentlicht. Diese neueren Arbeiten (und auch einige ältere, welche konzeptionell in diesem Zusammenhang auch relevant sind) wird der Autor im folgenden Abschnitt vorstellen. Im Anschluss präsentiert er die Ergebnisse vergleichender Messungen, welche die Vorteile gegenüber den Kompressoren aufzeigen, die wir bereits in Kapitel 4 betrachtet haben.

Im Anschluss daran beschreibt der Autor zunächst die Architektur seines neuartigen Kompressionsansatzes. Im Weiteren gibt er einen Überblick über eine prototypische Implementierung und vergleicht deren Leistungsfähigkeit mit anderen Ansätzen. Abschließend fasst er seine Ergebnisse zusammen und bewertet sie.

Die in diesem Kapitel vorgestellten Ideen und Konzepte wurden bereits in [156] und [157] in wesentlichen Teilen vorab veröffentlicht.

5.1 Verwandte Arbeiten

Die Idee, eine Kompression durchzuführen, indem man auf Basis einer XML-Grammatikbeschreibung Regeln für eine kompakte Binärdatenrepräsentation ableitet, wurde bereits in einigen Forschungsarbeiten diskutiert. Wie eingangs bereits erwähnt, bezeichnen wir solche Ansätze als *dynamisch-grammatikspezifisch*.

Im Folgenden gibt der Autor einen Überblick über die relevanten Forschungsarbeiten. Diese werden dabei in der Reihenfolge vorgestellt, die nach Meinung des Autors die zeitliche Entwicklung der verschiedenen XML-Kompressionstechniken am besten widerspiegelt. Wegen parallel und zum Teil auch unabhängig voneinander ablaufender Forschungsarbeiten ist diese Reihenfolge nicht exakt identisch mit der Reihenfolge der Veröffentlichungen.

5.1.1 ASN.1 Encoding und Fast Web Services

Einer der ersten in diesem Zusammenhang relevanten Ansätze basiert auf der formalen Beschreibungssprache für Datentypen *Abstract Syntax Notation One (ASN.1)*. Sie wurde bereits 1984 als Teil des Telekommunikationsstandards X.409 [29] standardisiert. In den Jahren 1992, 1997 und 2002 wurde sie grundlegend überarbeitet und ist heute im Standard X.680 [62] der *International Telecommunication Union (ITU)* festgeschrieben. Die ITU ist ein internationales Standardisierungsgremium für den Bereich Telekommunikation.

Die Sprache ASN.1 ist damit deutlich älter als XML oder die Web-Services-Technologie. Im Gegensatz zu XML stellt ASN.1 aber Mechanismen für kompakte Datenrepräsentationen bereit. Im Folgenden gibt der Autor zunächst eine knappe Einführung in die Grundlagen von ASN.1 und stellt darauf aufbauend *Fast Web Services* vor. Bei diesem Ansatz wird ein SOAP-Web-Service durch einen ASN.1-Web-Service ersetzt, was zu einem verringerten Datenaufkommen führt.

ASN.1 dient primär zur Spezifikation von Datentypen in Telekommunikationsprotokollen der ITU. Ein bekanntes Beispiel, dessen Spezifikation auf ASN.1 aufsetzt, ist das H.323-Protokoll [65]. Es kommt häufig bei Videokonferenz- oder Voice-over-IP-Anwendungen zum Einsatz. Bei Internetstandards ist die Verwendung von ASN.1 dagegen eher unüblich; IETF-Spezifikationen wie HTTP verwenden stattdessen zumeist die so genannte *Augmented Backus Naur Form (ABNF)* [33].

Obwohl ASN.1 primär zur Spezifikation von Datentypen gedacht ist, gibt es auch Compiler [115, 154], die abstrakte, programmiersprachenunabhängige ASN.1-Datentypdefinitionen in konkrete Datentypdefinitionen einer Programmiersprache umwandeln, beispielsweise in Form von C-Header-Dateien.

Zusätzlich gibt es noch die Möglichkeit, kanonische Codierungen für ASN.1-Datentyp-Instanzen automatisch vorzunehmen. Für diverse Programmiersprachen gibt es fertige

Funktionsbibliotheken [115, 154, 6], die für ASN.1-Datentyp-Instanzen verschiedene Serialisierungsvarianten anbieten. Besonders gängig sind die *Basic Encoding Rules (BER)* [63] und die *Packed Encoding Rules (PER)* [64].

Während BER die Zielstellung einer „selbstbeschreibenden" Datenrepräsentation verfolgt, d. h. die Strukturinformationen eines Datensatzes sollen explizit in der Codewortfolge enthalten sein, hat PER zum Ziel, den Datensatz möglichst kompakt darzustellen. Ohne Kenntnis der zugehörigen ASN.1-Datentypbeschreibung lassen sich aus einer PER-codierten Nachricht jedoch keine Strukturinformationen mehr zurückgewinnen.

ASN.1 ist prinzipiell mit XML-Grammatikbeschreibungssprachen wie DTD oder XML Schema vergleichbar, denn auch diese Sprachen lassen sich als Definitionssprachen für Datentypen auffassen. Geht es um die Erzeugung von kompakten Repräsentationen für die Instanzen der definierten Datentypen, ist ASN.1 aber den Definitionssprachen aus der XML-Welt augenscheinlich überlegen; mittels standardisierter Codierungsregeln (z. B. BER und PER) ist es hier vergleichsweise einfach möglich, eine kompakte Binärdarstellung von Datentypinstanzen zu erzeugen.

Mit der wachsenden Verbreitung von SOAP-Web-Services haben sich auch ITU-Arbeitsgruppen damit beschäftigt, wie man dem hohen Datenaufkommen bei der SOAP-Kommunikation begegnen kann. Da die ASN.1-Technologie bereits geeignete Verfahren zur Erzeugung solcher Repräsentationen vorsieht, hat man den X.694 Standard [66] entworfen. Mit dessen Hilfe lassen sich die Datentypdefinitionen aus einem XML-Schema-Dokument in eine ASN.1-Typdefinition überführen.

Betrachten wir als einfaches Beispiel einen Datentyp „Mitarbeiter": Ein Datensatz dieses Typs ist ein Struct, bestehend aus Name (String), Vorname (String) und Personalnummer (Integer). Abbildung 5.1(a) zeigt die Datentypdefinition in XML-Schema-Syntax, Abbildung 5.1(b) in ASN.1-Syntax. Die ASN.1-Darstellung wurde mit Hilfe des X.694-fähigen Konverters *xsd2asn1* aus dem XML-Schema-Dokument generiert. Eine Online-Version dieses Konverters ist unter `http://asn1.elibel.tm.fr/tools/xsd2asn1/` verfügbar.

Obwohl der von *xsd2asn1* generierte Code von einigen speziellen ASN.1-Funktionalitäten Gebrauch macht und daher etwas unübersichtlich wirkt, kann man dennoch recht gut die Parallelen zwischen beiden Beschreibungssprachen erkennen. Die relevanten Bereiche sind jeweils im Fettdruck dargestellt. In beiden Beschreibungssprachen wird der Datentyp „Mitarbeiter" als Sequenz von drei referenzierten Datentypen dargestellt: zwei String-Werte gefolgt von einem Integer-Wert.

Diese Konvertierung von XML Schema nach ASN.1 wird mitunter auch als *Fast Schema* bezeichnet.

Da die WSDL-Beschreibung eines SOAP-Web-Services typischerweise bereits eine XML-Schema-Beschreibung enthält, liegt es nahe, diesen Ansatz auch für SOAP nutzbar zu machen. Daher beschreibt der Standard X.892 [68] die Erzeugung von ASN.1-

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Mitarbeiter" type="ma-type"/>
  <xsd:complexType name="ma-type">
    <xsd:sequence>
      <xsd:element name="Nachname"       type="xsd:string"/>
      <xsd:element name="Vorname"        type="xsd:string"/>
      <xsd:element name="Personalnummer" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

(a) Datentypdefinition in XML Schema

```
Notarget-ns DEFINITIONS XER INSTRUCTIONS AUTOMATIC TAGS ::= BEGIN
IMPORTS
    String, Int
    FROM XSD;

Mitarbeiter ::= [ELEMENT] Ma-type

Ma-type ::= [NAME AS UNCAPITALIZED] SEQUENCE {
    nachname       [NAME AS CAPITALIZED] XSD.String,
    vorname        [NAME AS CAPITALIZED] XSD.String,
    personalnummer [NAME AS CAPITALIZED] XSD.Int
}

ENCODING-CONTROL XER
    GLOBAL-DEFAULTS MODIFIED-ENCODINGS
    GLOBAL-DEFAULTS CONTROL-NAMESPACE
        "http://www.w3.org/2001/XMLSchema-instance" PREFIX "xsi"
END
```

(b) Datentypdefinition in ASN.1

Abbildung 5.1: Verschiedene Sprachen zur Definition von Datentypen am Beispiel „Mitarbeiter“

Typdefinitionen für SOAP-Web-Services aus ihren WSDL-Beschreibungen. Solche ASN.1-basierten Web Services bezeichnet man als *Fast Web Services*.

Die Verwendung von Fast Web Services erhöht im Vergleich zu herkömmlichen SOAP-Implementierungen die Verarbeitungsgeschwindigkeit etwa um das Achtfache und führt auch zu einer Reduzierung des Datenaufkommens – etwa um den Faktor fünf [130]. Allerdings hat sich dieser Ansatz für praktische Anwendungen nicht durchsetzen können. Es gibt keine öffentlich zugänglichen Implementierungen.

Diese geringe Akzeptanz liegt vermutlich darin begründet, dass hier Dienstanbieter und auch Dienstnutzer gezwungen sind, statt der gewohnten XML-Werkzeuge die äußerst komplexe ASN.1-Technologie zu verwenden. Schließlich legt der Fast-Web-Service-Standard lediglich die Regeln zur Überführung eines WSDL-Dokuments in eine ASN.1-Beschreibung fest. Die Implementierung der Dienstfunktionalitäten wird damit nicht automatisiert.

Ein typisches Anwendungsszenario könnte so aussehen, dass ein Dienstanbieter – zusätzlich zu einem bereits vorhandenen SOAP-basierten Web Service – einen *Fast* Web Service bereitstellen möchte. Hierzu konvertiert er die bereits vorhandene WSDL-Beschreibung in eine ASN.1-Definition und verarbeitet diese dann mit Hilfe eines ASN.1-Compilers, welcher hieraus schließlich konkrete Datentypdefinitionen in einer Programmiersprache erzeugt. Auf dieser Basis kann der Dienstanbieter dann einen Fast Web Service mit äquivalenter Funktionalität implementieren. Mit Hilfe von ASN.1-Funktionsbibliotheken werden zur Laufzeit kompakte Binärrepräsentationen (z. B. mit BER oder PER) aller vom Fast Web Service ausgetauschten Netzwerknachrichten erzeugt.

Bei Fast Web Services und den verwandten ASN.1-Technologien handelt es sich also weder um einen Mechanismus zur Datenkompression noch um eine Technik, die auf XML aufsetzt. Der XML-Overhead wird bei diesem Ansatz dadurch eliminiert, dass auf XML zur Laufzeit völlig verzichtet wird. Dadurch gehen jedoch Merkmale wie Plattformunabhängigkeit und Erweiterbarkeit verloren – gerade diese machen aber den Wert von Web Services bei der Entwicklung verteilter Anwendungen aus.

5.1.2 Fast Infoset

Neben Fast Web Services hat das ITU einen weiteren Ansatz standardisiert, um dem hohen Datenaufkommen bei XML-basierten Kommunikationsprotokollen zu begegnen. Der Standard X.891 „Fast Infoset“ [67] legt eine kompakte, binäre Codierung für das vom W3C standardisierte XML Information Set [171] fest. Obwohl ASN.1 bei der Spezifikation von Fast Infoset zum Einsatz kommt, ist das Binärformat von Fast Infoset völlig unabhängig von ASN.1-Datenrepräsentationen wie BER oder PER – dies ist der wesentliche Unterschied zur Fast-Web-Services-Technologie.

```
<root>
  <tag>one</tag>
  <tag>two</tag>
  <anotherTag>one</anotherTag>
</root>
```

(a) Textdarstellung

```
{T0}<root>
  {T1}<tag>{C0}one</>
  [T1]{C1}two</>
  {T2}<anotherTag>[C0]</>
</>
```

(b) Fast-Infoset-Darstellung (schematisch)

Abbildung 5.2: String-Indizierung und Auslassen von End-Tag-Namen bei Fast Infoset

Bei Fast Infoset kommen primär zwei Techniken zum Einsatz: die Indizierung von Strings und das Auslassen von End-Tag-Namen. Beide Strategien haben wir bereits in Abschnitt 4.2 beim Kompressor XMill kennen gelernt.

Da sich bei wohlgeformten XML-Dokumenten die Namen aller schließenden Tags aus den Namen der korrespondierenden öffnenden Tags ergeben, brauchen diese nicht mit codiert zu werden.

Die zweite Strategie dient zur Reduzierung des Overheads, der durch wiederholt auftretende Zeichenketten entsteht. Sämtliche Zeichenketten werden daher bei ihrem ersten Auftreten indiziert. Bei erneutem Auftreten einer Zeichenkette wird nur noch die Adresse im Index referenziert. Es gibt separate Indextabellen für Tag-Namen, Attribut-Namen, Namespace-Informationen und Zeichendaten.

Abbildung 5.2 zeigt ein Beispiel zur Verdeutlichung beider Strategien: Die Namen aller End-Tags sind entfallen. Außerdem wird jede Zeichenkette bei ihrem ersten Auftreten in eine Indextabelle aufgenommen. In Abbildung 5.2(b) ist dies durch Werte in geschweiften Klammern angedeutet. Die Angabe `{T1}` deutet an, dass der Name des folgenden Tags in die Tag-Indextabelle an Adresse 1 mit aufgenommen werden soll. Taucht der indizierte Wert im weiteren Verlauf des Dokuments erneut auf, wird er nicht erneut im Ausgabestrom codiert, sondern lediglich referenziert. Eine solche Referenzierung wird in Abbildung 5.2(b) mit Hilfe eckiger Klammern dargestellt. Die Indizierung erfolgt bei Tag-Namen, Attribut-Namen, Namespace-Informationen und Zeichendaten völlig analog; in der Abbildung ist exemplarisch die Indizierung von Tag-Namen und Zeichendaten dargestellt.

Zu beachten ist allerdings, dass es sich bei Abbildung 5.2 um eine schematische Darstellung handelt. Bei realen Fast-Infoset-Dokumenten wird nicht mit geschweiften

oder eckigen Klammern bzw. mit „leeren“ End-Tags gearbeitet, sondern durchgängig mit binären Codierungen. Die hier gezeigte Darstellung dient lediglich der Anschaulichkeit und repräsentiert nicht die im Standard festgelegte technische Umsetzung.

Mit Hilfe von String-Indizierung und durch Auslassen von End-Tags ergibt sich bei kleinen SOAP-Nachrichten eine um etwa 25% reduzierte Dokumentgröße, bei größeren ergibt sich eine Einsparung um bis zu Faktor fünf. [131]

Die vergleichsweise schlechten Ergebnisse bei kleinen Dokumenten hängen damit zusammen, dass hier typischerweise kaum Wiederholungen einzelner Zeichenketten auftreten und somit die Einsparung durch Indexreferenzen kaum zum Tragen kommt.

Daher sieht der Fast Infoset-Standard optional die Verwendung von „Initial Vocabularies“ vor. Hierunter versteht man eine Vorbelegung der Indextabellen, so dass bereits zu Beginn der Verarbeitung Index-Einträge vorhanden sind, die referenziert werden können. Auf diese Weise kann der Referenzierungsmechanismus auch bei sehr kurzen Dokumenten wirksam werden. Der Standard X.891 sieht zwar die Verwendung solcher Initial Vocabularies grundsätzlich vor, lässt die genaue technische Umsetzung jedoch explizit offen. Es existieren bisher keine Implementierungen, die von Initial Vocabularies bei Fast Infoset Gebrauch machen.

Es gibt allerdings bereits Implementierungen von Fast Infoset, welche auf Initial Vocabularies verzichten. Hervorzuheben ist hier das *Fast Infoset Projekt*, welches eine X.891-konforme Implementierung erarbeitet hat, die dem Anwendungsentwickler die gängigen XML-APIs SAX und DOM bereitstellt. Auf diese Weise ist es möglich, Fast Infoset auf einfache Weise in bestehende XML-Projekte einzubetten.

Verwendet man Fast Infoset mit Initial Vocabularies, handelt es sich dabei um einen dynamisch-grammatikspezifischen Kompressor.

5.1.3 XGrind

Die Fast-Infoset-Codierung hat sehr große Ähnlichkeit mit einem Kompressionsansatz namens XGrind, den Tolani und Haritsa bereits 2002 in [149] vorgestellt haben. Insbesondere taucht in dieser Arbeit das Konzept von vorinitialisierten Symboltabellen erstmalig auf, welche prinzipiell mit den „Initial Vocabularies“ von Fast Infoset vergleichbar sind.

In einem ersten Verarbeitungsschritt werden bei XGrind initiale Werte für zwei Symboltabellen berechnet, hierbei wird eine DTD verarbeitet, die die Struktur der zu komprimierenden Datensätze beschreibt. Die erste Symboltabelle nimmt dabei die Namen von Elementen auf, die zweite die Namen möglicher Attribute. Hierzu wird die DTD sequentiell durchlaufen, und die gefundenen Namen werden in der jeweiligen Tabelle der Reihe nach indiziert. Weiterhin werden Informationen zu Aufzählungstypen aus der DTD extrahiert.

```
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<ELEMENT NAME (#PCDATA)>
<ELEMENT YEAR (#PCDATA)>
<ELEMENT PROG (#PCDATA)>
<ELEMENT DEPT EMPTY>
<ATTLIST DEPT name
(Computer_Science
    | Electrical_Engineering
    | ...
    | Physics | Chemistry)
>
```

(a) DTD

```
<STUDENT rollno = "604100481"/>
  <NAME>Pankaj Tolani</NAME>
  <YEAR>2000</YEAR>
  <PROG>Master of Engineering</PROG>
  <DEPT name = "Computer_Science"/>
</STUDENT>
```

(b) Textdatstellung

```
T0 A0 nahuff(604100481) /
   T1 nahuff(Pankaj Tolani) /
   T2 nahuff(2000) /
   T3 nahuff(Master of Engineering) /
   T4 A1 enum(Computer_Science)
/
```

(c) XGrind-Darstellung (schematisch)

Abbildung 5.3: Nutzung von vorinitialisierten Symboltabellen bei XGrind (entnommen aus [149])

In einem zweiten Verarbeitungsschritt wird schließlich der Eingabedatensatz mit Hilfe der zuvor erzeugten Tabellen komprimiert.

Abbildung 5.3(a) zeigt exemplarisch eine DTD. XGrind extrahiert hieraus fünf Elementnamen und legt sie in der entsprechenden Symboltabelle unter den Adressen `T0` bis `T4` ab. In der Tabelle für Attributnamen ergeben sich zwei Einträge `A0` und `A1`. Weiterhin analysiert der XGrind-Kompressor die DTD im Hinblick auf Aufzählungstypen. Diese werden ebenfalls in einer separaten Datenstruktur indiziert.

Abbildung 5.3(b) stellt ein XML-Dokument dar, das den Regeln der dargestellten DTD genügt. Abbildung 5.3(c) zeigt die von XGrind erzeugte Codierung dieses Dokuments: Alle Elemente und Attributnamen werden hier nicht mehr explizit codiert, stattdessen werden die Einträge in den Symboltabellen referenziert. Das Zeichen `/`

symbolisiert ein schließendes Tag – eine Codierung des Namens ist hier nicht erforderlich, da er mit dem des korrespondierenden öffnenden Tags identisch ist.

Zeichendaten verarbeitet XGrind mit Hilfe des nicht-adaptiven Huffman-Verfahrens (vergleiche Abschnitt 3.6). In Abbildung 5.3(c) wird dies über die Funktion `nahuff()` symbolisiert. Aufzählungstypen werden noch effizienter codiert, indem hier die bei der DTD-Verarbeitung indizierten Werte referenziert werden. In der Abbildung wird dies mit Hilfe der Funktion `enum()` verdeutlicht.

Obwohl die schematische Darstellung des komprimierten Datenstroms in diesem Beispiel leicht von der in Abbildung 5.2 (Seite 99) abweicht, ist die starke Ähnlichkeit zum Fast-Infoset-Ansatz unmittelbar ersichtlich. Beide Verfahren basieren im Wesentlichen auf der Indizierung von Strings und dem Auslassen von End-Tag-Namen.

Tolani und Haritsa haben selbst Leistungsmessungen vorgenommen, die zeigen, dass die von XGrind erreichten Kompressionsraten schlechter als die von XMill sind (welcher ohne die Verarbeitung einer DTD auskommt). Die Stärke von XGrind liegt damit nicht in einer besonders effektiven Kompression der Eingangsdaten. Vielmehr betonen Tolani und Haritsa, dass XGrind eine Navigation innerhalb des komprimierten Datensatzes mit Hilfe von Sprachen wie XPath [168] ermöglicht und so eine gezielte Dekompression einzelner Informationseinheiten sehr gut unterstützt. Dies sei ein entscheidender Vorteil gegenüber XMill und xmlppm. Da Möglichkeiten zur Navigation in komprimierten Datensätzen aber nicht Gegenstand dieser Arbeit sind, wird auf diesen Aspekt nicht näher eingegangen.

Unter `http://sourceforge.net/projects/xgrind/` ist eine freie Implementierung von XGrind verfügbar. Da XGrind eine DTD als Grammatikbeschreibung bei der Kompression verwendet, gehört dieser Ansatz zur Gruppe der dynamisch-grammatikspezifischen Kompressoren.

5.1.4 BiM

Fast Infoset und XGrind erzeugen Indextabellen, die beim Kompressionsvorgang quasi als Abkürzungswörterbuch dienen. Eine weitere Möglichkeit ergibt sich, wenn man die in einer XML-Sprache festgelegten Strukturinformationen in die Kompressionsstrategie mit einbezieht: Bei den meisten XML-Sprachen gibt es Regeln, die mögliche Sequenzen von Elementen und Attributen beschreiben. Beispielsweise beginnt jedes SOAP-Dokument mit dem Tag `<soap:Envelope>`, auf das Tag `</soap:Header>` folgt stets `<soap:Body>` usw. Da diese Regeln für jedes SOAP-Dokument zutreffen und sie sowohl dem Sender als auch dem Empfänger bekannt sind, beseitigt die Übertragung dieser Information keinerlei Unsicherheit. Der Informationsgehalt beträgt somit 0 Bit.

Folglich muss ein `<soap:Body>` Tag hinter einem `</soap:Header>` Tag auch nicht mit codiert werden – der Empfänger einer komprimierten SOAP-Nachricht kann die-

ses ausgelassene Tag allein mit Hilfe der Regeln zur Konstruktion gültiger SOAP-Nachrichten beim Dekompressionsvorgang selbständig ergänzen.

Beim *Binary Format for MPEG-7 Metadata (BiM)* wird diese Erkenntnis ausgenutzt. Wie der Name vermuten lässt, ist BiM allerdings ein Spezialkompressor für MPEG-7-Metadatendokumente [105] und kann nicht für die Kompression anderer XML-Sprachen eingesetzt werden (statisch-grammatikspezifischer Kompressor). BiM ist in diesem Zusammenhang aber dennoch relevant, denn hier werden erstmals die durch eine XML-Sprache vorgegebenen Strukturinformationen für Datenkompressionsanwendungen ausgenutzt.

Die Autoren Niedermeier et al. schlagen in [105] zwei mögliche Techniken für strukturbasierte Codierungen vor. Die eine heißt *Context Path Coding* und die zweite *Fragment Payload Coding*. Letztere ist an dieser Stelle besonders relevant, weil bei ihr erstmals der Einsatz von Automatenstrukturen zur Realisierung von XML-Kompressoren erwähnt wird. Niedermeier et al. schlagen vor, die Strukturinformationen, die eine XML-Sprache vorgibt, in Form von endlichen Automaten darzustellen. Allerdings wird von den Autoren lediglich angegeben, dass endliche Automaten hierfür verwendet werden; wie diese genau bei der Kompression eingesetzt werden und wie man sie konstruiert, bleibt weitestgehend unklar.

Niedermeier et al. haben ihre Kompressionsansätze prototypisch implementiert und mit dem Kompressor XMill verglichen [105]. Die Messungen zeigen, dass Context Path Coding tendenziell etwas schlechtere und Fragment Payload Coding etwas bessere Ergebnisse als XMill liefert. Die Abweichungen zu XMill bewegen sich bei den meisten Messungen zwischen 10 und 20%. Aufgrund der Tatsache, dass XMill ein Kompressor für beliebige XML-Sprachen ist und BiM lediglich MPEG-7 Metadaten verarbeiten kann, ist dieses Ergebnis auf den ersten Blick eher enttäuschend. Allerdings berücksichtigt BiM im erzeugten Binärformat auch diverse Anforderungen, die speziell bei der Verarbeitung von MPEG-7 Metadaten relevant sind; somit sehen die Autoren – trotz der nur mäßigen Kompressionsleistungen – klare Vorteile von BiM in diesem speziellen Anwendungsbereich.

5.1.5 Exalt

Exalt, vorgestellt von Toman in [150], ist ein *syntaktischer* Kompressor. Dieses Prinzip ähnelt sehr stark der strukturbasierten Kompression von BiM, allerdings sind die Strukturinformationen hier nicht vorab bekannt, sondern werden im Rahmen des Kompressionsprozesses schrittweise berechnet. Theoretische Grundlage zu diesem Vorgehen bildet ein vergleichsweise neuer Ansatz der Codierungstheorie: die grammatikbasierte Kompression (engl.: grammar-based Compression) [74].

Auch bei Exalt kommen endliche Automaten zur Beschreibung von Strukturinformationen zum Einsatz: Während des Kompressionsvorgangs wird jedem im Eingabedokument vorkommenden Element E ein azyklischer endlicher Automat zugeordnet.

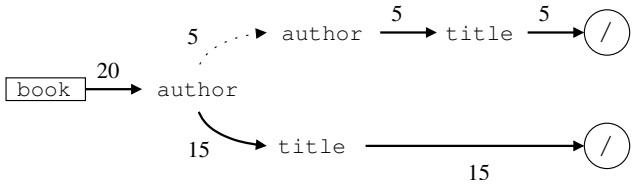


Abbildung 5.4: Automat für syntaktische Kompression für das Element `book` (entnommen aus [150])

Dieser Automat wird als *Modell von E* bezeichnet. Die Zustände in diesem Automaten modellieren mögliche (direkte) Kindelemente von E. Die Zustandsübergänge beschreiben mögliche Anordnungen der Kindelemente. Numerische Werte, welche die Auftrittshäufigkeiten möglicher Anordnungen beschreiben, dienen als Kantenmarkierungen.

Die Automatenkonstruktion erfolgt dabei nicht im Rahmen eines Vorverarbeitungsschritts, sondern sukzessive bei der Verarbeitung des Eingabedokuments. Taucht ein Element E also erstmalig auf, wird zunächst ein Automat angelegt, der noch keine Kindelemente für E vorsieht. Bei der weiteren Verarbeitung wird zu jedem erstmalig gelesenen Kindelement von E ein neuer Zustand im Automaten angelegt. Auch die Zustandsübergänge und Zustandsübergangsmarkierungen werden bei der Verarbeitung des Eingabedokuments laufend aktualisiert.

Abbildung 5.4 verdeutlicht die Funktionsweise dieser endlichen Automaten anhand eines Beispiels. Der dargestellte Automat repräsentiert die bisher aufgetretenen Kindelementsequenzen für Elemente mit dem Namen `book`. Jedes Element `book` besitzt als erstes Kindelement `author`. Die Kantenmarkierung `20` gibt an, dass dieser Zusammenhang bisher zwanzigmal im Eingabedokument festgestellt wurde. Auf `author` folgt entweder ein zweites `author` Element – dieser Fall trat bisher fünfmal auf – oder aber es folgt ein `title` Element. Letzteres ist bisher fünfzehnmal aufgetreten. Solche Alternativen werden im Automaten als Verzweigungen repräsentiert. Am Ende jeder Verzweigung gibt es ein Endsymbol, welches in der Abbildung als Kreis mit einem Schrägstrich dargestellt ist. Es dient bei Exalt zur Programmablaufsteuerung.

Bei der Verarbeitung eines XML-Dokuments werden nun für alle auftretenden Elementnamen solche Automaten angelegt, die mögliche Sequenzen für direkte Kindelemente abbilden. Zu jedem Zeitpunkt des Kompressionsvorgangs gibt es einen Vorrat solcher Automaten, die die bisher verarbeitete Dokumentstruktur widerspiegeln.

Der Exalt-Ansatz basiert darauf, dass auf Grundlage der bisherigen Dokumentstruktur die folgende vorhergesagt wird: Würde also ein weiteres Element `book` gelesen werden, so würde Exalt anhand des in Abbildung 5.4 dargestellten Automaten vorhersagen, dass zunächst `author` folgt und danach `title`. Schließlich ist `title` hinter `author` bisher häufiger aufgetreten als die Sequenz `author author title`. Solche

Zustandsübergänge, die Vorhersagen darstellen, sind in der Abbildung als durchgehende Pfeile dargestellt, andere als durchbrochene.

Im Ausgabedatenstrom wird bei der Kompression nur die Abweichung zur Vorhersage codiert. Diese Idee ist dem Leser bereits aus den Erläuterungen zum PPM-Algorithmus bekannt (vergleiche Abschnitt 4.2.4).

Toman hat Exalt mit anderen XML-Kompressoren verglichen und bei seinen Experimenten etwa 10 bis 30% schlechtere Ergebnisse im Vergleich zu xmlppm erzielt. Die Implementierung ist unter `http://exalt.sourceforge.net/` verfügbar. Er selbst resümiert, dass Exalt in der gegenwärtigen Form eher als Beweis für die Machbarkeit einer grammatikbasierten Codierung von XML-Daten anzusehen sei und nicht als produktiv einsetzbare Lösung.

Ein klarer Nachteil von Exalt ist die Notwendigkeit, die Grammatikregeln aus dem XML-Dokument schrittweise zu „erlernen“. Toman gibt an, seinen Kompressor so erweitern zu wollen, dass dieser eine DTD einliest und daraus bereits vor der Verarbeitung des Eingabedokuments die Automatenzustände und Zustandsübergänge anlegt. Allerdings wurde eine derartige Erweiterung bisher nicht veröffentlicht.

Somit ist Exalt in der gegenwärtigen Implementierung kein grammatikspezifischer Kompressor. Im Rahmen dieser Arbeit ist Exalt allerdings dennoch interessant, weil hier der Einsatz von Automatenstrukturen für die XML-Kompression erstmalig konkret demonstriert wird.

5.1.6 Xaust

Hariharan und Shankar haben in [55] erstmals die Konstruktion von Automatenstrukturen aus einer XML-Grammatikbeschreibung mit der Zielstellung der Datenkompression konkret demonstriert.

Die Autoren beschreiben hier den Kompressor *Xaust*, bei dem eine DTD eingelesen und zu jedem in der DTD definierten Elementtyp ein endlicher Automat erzeugt wird, der mögliche Kindelementsequenzen für diesen Elementtyp beschreibt. Bei diesem Verfahren findet sich also genau die bereits von Toman beschriebene Idee wieder.

In [55] werden auch Vergleichsmessungen mit xmlppm aufgeführt. Hierbei erreicht Xaust sehr ähnliche Kompressionsraten wie xmlppm, allerdings bei deutlich geringerer CPU- und RAM-Speicher-Auslastung.

Eine Implementierung von Xaust ist leider nicht verfügbar.

5.1.7 Xebu

Eine andere Arbeit in diesem Bereich haben Kangasharju et al. vorgelegt [71]. Dieser Kompressor vereinigt alle bisher vorgestellten Konzepte in sich und bietet

optionale Unterstützung für die Verarbeitung einer XML-Grammatik. Allerdings wird diese hier nicht als XML-Schema-Dokument angegeben, sondern in der Relax-NG-Syntax [27].

Tritt ein Element- oder Attributname das erste Mal auf, so wird er in einer Tabelle abgelegt, und bei einem wiederholten Auftreten wird nur noch eine Referenz auf diesen Tabelleneintrag codiert (String-Indizierung). Die Namen von End-Tags werden nicht mit codiert, weil sie sich aus dem korrespondierenden öffnenden Tag unmittelbar ergeben (Auslassen von End-Tags).

String-Indizierung und das Auslassen von End-Tags sind bei Xebu auch verfügbar, wenn keine Grammatikbeschreibung zu einem XML-Dokument vorliegt. Ist eine solche Beschreibung in Form einer Relax-NG-Grammatik verfügbar, kommen zwei weitere Mechanismen zum Tragen: Zum einen werden die Codetabellen der String-Indizierung bereits vorab auf Basis der Element- und Attributnamen initialisiert, die in der Grammatikbeschreibung vorgesehen sind. Dieses Merkmal wird hier als *Pre-Caching* bezeichnet und entspricht den *Initial Vocabularies* von Fast Infoset bzw. XGrind.

Weiterhin erzeugt *Xebu* aus der Relax-NG-Grammatik so genannte *Omission Automata*. Diese endlichen Automaten werden während des Kompressionsvorgangs eingesetzt, um mögliche Dokumentstrukturen vorherzusagen und nur solche Strukturinformationen zu codieren, die nicht schon durch die Relax-NG-Grammatik fest vorgegeben sind. Diese Idee der strukturbasierten Kompression ist bereits von BiM, Exalt und Xaust bekannt.

Allerdings sieht das Automatenkonzept bei Xebu noch ergänzende Funktionalitäten vor: Eine Relax-NG-Grammatik schreibt nicht nur die Dokumentstruktur vor, sondern enthält auch Angaben über die in einem XML-Dokument erlaubten Datentypen – diese Information lässt sich bei der Kompression ebenfalls ausnutzen. Ist beispielsweise bekannt, dass Elemente eines bestimmten Typs immer boolesche Werte enthalten, kann man diese im komprimierten Ausgabestrom stets mit Hilfe eines einzelnen Bits codieren. Auf gleiche Weise lassen sich auch Zahlen- oder Datumstypinstanzen besonders kompakt codieren.

Der Xebu-Kompressor kombiniert damit sämtliche bisher bekannten Kompressionsstrategien in einer einzigen Implementierung. Diese ist unter `http://hoslab.cs.helsinki.fi/downloads/xebu/` verfügbar.

Falls bei der Kompression eine Relax-NG-Grammatik verarbeitet wird, gehört Xebu zur Klasse der dynamisch-grammatikspezifischen Kompressoren.

5.1.8 Weitere Arbeiten und Ergebnis der Patentrecherche

Die bisherige Aufstellung verwandter Arbeiten beschränkt sich auf solche, in denen neuartige Konzepte oder Ideen vorgestellt werden. Es gibt noch eine ganze Reihe weiterer Veröffentlichungen, bei denen bereits bekannte Strategien modifiziert und

für andere Anwendungsbereiche nutzbar gemacht werden. Der Vollständigkeit halber werden diese aber ebenfalls genannt:

Vor allem für Kompressionsanwendungen im Bereich XML-Datenbanken gibt es eine ganze Reihe weiterer Arbeiten. Bei XML-Datenbanken steht nicht allein die erreichte Kompressionsrate im Zentrum der Betrachtungen, sondern zusätzlich auch die Möglichkeit, Anfragen an den komprimierten XML-Datensatz zu stellen. Solche „query-friendly“ Ansätze basieren im Wesentlichen auf den Konzepten von XGrind. [84, 7, 155, 75, 4, 17, 86]

In weiteren Arbeiten von Cheney sowie von Hariharan und Shankar geht es um Detailverbesserungen der Kompressoren xmlppm und Xaust. Außerdem werden hier weitere vergleichende Messungen angeführt. [23, 24, 56]

Arbeiten von Leighton et al. verfolgen ein Konzept, welches starke Parallelen zu Exalt aufweist. Auch hier wird eine Grammatikrepräsentation in Form von Automatenstrukturen während der Verarbeitung des Eingabedatensatzes erzeugt. [81, 82]

Des Weiteren existiert ein kommerzieller XML-Kompressor *XML Xpress* [61]. Allerdings bleibt hier unklar, wie dieser genau arbeitet. Auch eine Anfrage beim Hersteller brachte keine Klarheit. Offenbar werden bei XML Xpress in einem ersten Verarbeitungsschritt eine XML-Schema-Beschreibung sowie ein möglichst typischer Datensatz für die zu komprimierenden Daten eingelesen. Aus diesen beiden Dokumenten errechnet XML Xpress ein so genanntes *Schema Model File*. Im zweiten Verarbeitungsschritt wird die eigentliche Kompression durchgeführt; hierzu liest XML Xpress das Schema Model File und den zu komprimierenden Datensatz ein. Das Schema Model File ist auch bei der Dekompression erforderlich. Im Rahmen der Recherchen zu XML Xpress blieb insbesondere unklar, ob die Erzeugung vom Schema Model File vollautomatisch erfolgen kann oder hierzu ein manuelles Eingreifen erforderlich ist.

Im Rahmen einer umfangreichen Patentrecherche hat der Autor noch eine Reihe weiterer XML-Kompressionsverfahren gesichtet und analysiert. Allerdings ist es sehr schwierig, die in den Patentschriften beschriebenen Techniken mit den bisher vorgestellten zu vergleichen, weil die Darstellungen in den Patentschriften in aller Regel sehr allgemein formuliert sind, um den hiermit erzielten Patentschutz auf einen möglichst großen Wirkungsbereich auszudehnen. Der Autor beschränkt sich daher auf eine überblicksartige Darstellung:

Zu nennen sind zunächst fünf Patente, welche mit String-Indizierung arbeiten [48, 49, 104, 80, 46]. Die dort vorgestellten Techniken sind mit denen von Fast Infoset und XGrind vergleichbar. Weiterhin gibt es ein Patent, das sich speziell auf die Kompression von Namespace-Informationen konzentriert und ebenfalls mit solchen String-Indextabellen arbeitet [119]. Ein anderes Patent beschreibt ein Codierungsverfahren, welches wie Fast Web Services auf ASN.1-Codierungstechniken aufsetzt [60].

In der Patentschrift [58] geht es schließlich um ein Codierungsverfahren für XML-Daten, welches eine Sonderstellung einnimmt: Hier wird das XML-Dokument nicht

in eine binäre Repräsentation umgewandelt, sondern lediglich mit Hilfe so genannter *Entity Declarations* und *Entity References* umgeformt. Diese beiden Mechanismen sind in der XML-Spezifikation vorgesehen, um Abkürzungen für häufig auftretende XML-Fragmente zu realisieren. Wegen der Beibehaltung der Textdarstellung erreicht dieses Verfahren sicherlich nur bei ganz speziellen Datensätzen mit langen, sich häufig wiederholenden Strukturen nennenswerte Kompressionsraten. Allerdings bringt es auch den großen Vorteil mit sich, dass ein herkömmlicher XML-Parser in der Lage ist, Entity-Referenzen aufzulösen. Bei diesem Verfahren ist ein Dekompressionsschritt vor dem Parsen damit völlig entbehrlich. Eine Implementierung dieses Verfahrens ist nicht bekannt.

Abschließend sei noch auf den aktuellen Stand relevanter Standardisierungsbestrebungen hingewiesen:

Da kompakte XML-Repräsentationen vor allem im Umfeld mobiler Anwendungen immer wichtiger werden, hat das W3C bereits im März 2004 die „W3C XML Binary Characterization Working Group“ gegründet. Die Mitglieder dieser Arbeitsgruppe haben zunächst eine detaillierte Anforderungsanalyse durchgeführt und auch eine Aufstellung der existierenden Arbeiten erstellt. Als zentrales Arbeitsergebnis hat diese W3C-Arbeitsgruppe einen Bericht veröffentlicht [183], der 18 typische Anwendungsszenarien für den Einsatz von kompakten XML-Repräsentationen beschreibt. Im Dezember 2005 wurde eine weitere W3C-Arbeitsgruppe gegründet, die sich primär mit der Interoperabilität von binären XML-Repräsentationen beschäftigt: die *W3C Efficient XML Interchange Working Group* [182].

Beide Arbeitsgruppen haben noch keine Entwürfe oder gar fertige Standards erarbeitet. Allerdings gibt es unter [186] bereits einen Überblick, welche XML-Kompressoren gegenwärtig vom W3C evaluiert werden. Neben Xebu und den beiden ITU-Standards Fast Infoset und Fast Web Services sind hier vier kommerzielle Produkte aufgelistet. Diese wurden im Rahmen dieser Arbeit allerdings nicht näher betrachtet, weil die Hersteller keine Informationen über das jeweils zu Grunde liegende Kompressionsprinzip liefern.

5.2 Kompressionsergebnisse verwandter Ansätze

Nachdem wir nun einen Überblick über Arbeiten auf dem Gebiet der grammatikspezifischen XML-Kompression gewonnen haben, stellt sich die Frage, welche Kompressionsergebnisse sich mit Hilfe dieser Techniken auf typischen SOAP-Daten erzielen lassen. Hierzu hat der Autor vergleichende Messungen durchgeführt, deren Ergebnisse im Folgenden vorgestellt werden.

Besonders aufschlussreich wird dabei sein, ob sich mit Hilfe der grammatikspezifischen Ansätze tatsächlich Verbesserungen gegenüber den in Kapitel 4 behandelten

Techniken ergeben, welche – mit Ausnahme von WBXML, Millau und der XML-Differenzcodierung – nicht mit Grammatikinformationen arbeiten.

5.2.1 Erzeugung der Testdatensätze

Damit sich die Ergebnisse der Messungen möglichst gut verallgemeinern lassen, werden die Kompressoren anhand von realistischen SOAP-Daten miteinander verglichen. Der Autor hatte zunächst geplant, die bereits in Kapitel 4 vorgestellten Testdaten auch für diese Untersuchung heranzuziehen. Bei näherer Betrachtung zeigte sich indessen, dass das dort gewählte Web-Service-Beispiel die Vielfalt aktueller Web-Service-Anwendungen hinsichtlich der zu Grunde liegenden Grammatik nicht ausreichend widerspiegelt. Basis dieser Recherche war das Web-Service-Verzeichnis `http://www.xmethods.com`. Die dort aufgeführten Web Services lassen sich mit Blick auf ihre Grammatik in zwei Kategorien aufteilen:

Zum einen gibt es eine Vielzahl von Web Services, deren Nachrichtenstruktur durch eine sehr restriktive Grammatik beschrieben ist. Es gibt also nur sehr wenig mögliche Varianten, wie die SOAP-Nachrichten eines solchen Web Services aussehen können. Ein typischer Vertreter dieser Art ist beispielsweise ein Web Service zum Abrufen von Wertpapierkursen. Hier enthält die Request-Nachricht stets einen vergleichsweise kurzen Bezeichner, der das Wertpapier eindeutig kennzeichnet (typischerweise über die so genannte Wertpapierkennnummer); die Response-Nachricht enthält den aktuellen Börsenkurs als numerischen Wert. Bei dieser Anwendung sind die ausgetauschten Nachrichten also stets gleichartig aufgebaut. Eine weitere wichtige Eigenschaft von Web Services dieser Art ist, dass die ausgetauschten Nachrichten in der Regel recht kurz sind. Wie bereits in Abschnitt 4.3 erläutert, sind kurze Nachrichten ganz besonders schwierig zu komprimieren.

Als typischen Vertreter dieser ersten Kategorie hat der Autor einen Taschenrechner-Web-Service implementiert. Dieser stellt vier verschiedene Operationen bereit: `void doNothing()`, `int increment(int i1)`, `int add(int i1, int i2)` und `void add6ints(int i1, ..., int i6)`. Bei allen vier Operationen ist die Struktur der Nachrichten bereits durch die Grammatikbeschreibung fest vorgegeben; sie generieren jeweils Nachrichten unterschiedlicher Größe. Der Taschenrechner-Web-Service und eine entsprechende Client-Applikation wurden unter Verwendung der Microsoft-.NET-Plattform [95] implementiert. Der Client ruft die Dienstoperationen dabei mit zufälligen Integer-Werten auf. Die resultierenden Request- und Response-Nachrichten werden als Dateien abgespeichert und dann mit Hilfe der zu untersuchenden Kompressoren verarbeitet.

Kennzeichnend für die zweite Kategorie von Web Services ist, dass die Nachrichtenstruktur durch eine recht komplexe Grammatikbeschreibung vorgegeben ist und viele Variationen im Nachrichtenformat zulässig sind. Die Anwendungsdaten im Body einer

SOAP-Nachricht werden verstärkt über Attribute strukturiert, und zudem enthalten sie einen großen Anteil an Zeichendaten.

Ein typischer Vertreter der zweiten Web-Service-Kategorie ist der *Amazon E-Commerce Service* [3], mit dessen Hilfe man die Produktdatenbank des Online-Buchhändlers Amazon durchsuchen kann. Dieser Web Service wurde als zweiter für Vergleichsmessungen herangezogen: Der Autor hat – ebenfalls unter Verwendung der Microsoft-.NET-Plattform – einen Client für diesen Web Service implementiert, der die Operation `itemSearch` mit dem Parameter „web service“ aufruft. Der Amazon-Web-Service gestattet, über einen weiteren Parameter `ResponseGroup` die Ausführlichkeit der gelieferten Antworten zu beeinflussen. In einer ersten Request-Nachricht wird `ResponseGroup` auf den Wert `small` gesetzt. Dies bedeutet, dass der Amazon-Web-Service die Datensätze zum Thema „web service“ möglichst kompakt aufbereitet. Es werden in der Antwortnachricht also nur die wichtigsten Informationen zu den gefundenen Produkten übertragen, im Wesentlichen sind dies „Titel“, „Autor“ und „Produktgruppe“. In einer zweiten und dritten Request-Nachricht wird `ResponseGroup` auf den Wert `medium` bzw. `large` gesetzt. Dies führt dazu, dass der Amazon-Web-Service mit deutlich größeren SOAP-Nachrichten antwortet, die zusätzliche Informationen zu den gefunden Produkten enthalten. Dies sind beispielsweise Inhaltsangaben, Kundenrezensionen oder Bildmaterialien. Auch hier wurden die Request- und Response-Nachrichten jeweils in Dateien abgespeichert und mit Hilfe der einzelnen Kompressoren verarbeitet.

5.2.2 Erzeugung einer Grammatikbeschreibung für SOAP-Nachrichten

Um die Vorteile von grammatikspezifischen Kompressoren bei Web-Service-Anwendungen messen zu können, war es zudem erforderlich, geeignete Grammatikbeschreibungen für die beiden Test-Web-Services zu erzeugen. Zwar enthält die WSDL-Beschreibung eines Web Services bereits eine solche Grammatikbeschreibung, typischerweise in Form eines eingebetteten XML-Schema-Dokuments, doch bezieht sich diese nur auf die im SOAP-Body übertragenen Anwendungsdaten. Für Datenkompressionsanwendungen benötigen wir jedoch eine Grammatikbeschreibung der SOAP-Nachricht als Ganzes, d. h. inklusive SOAP-Envelope und -Header.

Das W3C hat eine XML-Schema-Beschreibung erarbeitet, die die generische Struktur dieser Nachrichtenbestandteile (also Header und Envelope) beschreibt – ebenfalls als XML-Schema-Dokument. Sie ist unter `http://schemas.xmlsoap.org/soap/envelope` abrufbar.

Über den Import-Mechanismus von XML Schema ist es nun möglich, das W3C-SOAP-Schema und das Schema für Anwendungsdaten aus der WSDL-Beschreibung so zusammenzuführen, dass eine geschlossene Grammatikbeschreibung für die vom Web Service ausgetauschten SOAP-Nachrichten entsteht. Dies geschieht in drei Schritten: Zunächst wird das XML-Dokument zur Beschreibung der Anwendungsdaten aus der

WSDL-Beschreibung des Web Services extrahiert und als separate Datei abgespeichert. Dann wird diese Datei über eine Import-Anweisung in die generische XML-Schema-Beschreibung für SOAP-Nachrichten importiert. Schließlich wird der Datentyp für den SOAP-Body auf einen Choice-Typ abgeändert, der die Datentypen aller Web-Service-Operationen referenziert.

Auf diese Weise entsteht aus der generischen W3C-Schema-Beschreibung für SOAP-Nachrichten eine Schema-Beschreibung, die genau die möglichen ein- und ausgehenden Nachrichten für den durch die WSDL-Datei beschriebenen Web Service umfasst.

Die Abbildung 5.5 veranschaulicht diese Modifikationen anhand des Taschenrechnerbeispiels.

Es sei an dieser Stelle angemerkt, dass die hier vorgestellte Lösung noch keine SOAP-Fault-Nachrichten abdeckt. Hierfür müsste im Choice-Datentyp noch eine weitere Verzweigung vorgesehen werden. Im Rahmen der hier vorgestellten Messungen hat der Autor auf eine Modellierung möglicher SOAP-Fault-Nachrichten aber verzichtet: Zum einen kommen im Test-Szenario keine SOAP-Fault-Nachrichten vor, und zum anderen ist nicht davon auszugehen, dass eine zusätzliche Verzweigung im Choice-Typ zu einer signifikanten Änderung der Kompressionsergebnisse führt.

Für praktische Anwendungen wäre es sicherlich zweckmäßig, die Erzeugung solcher XML-Schema-Beschreibungen aus WSDL-Beschreibungen zu automatisieren. Für die beiden Test-Web-Services hat der Autor die erforderlichen Schritte jedoch manuell vorgenommen.

Da der Xebu-Kompressor keine XML-Schema-Dateien verarbeiten kann, sondern auf eine Grammatik im Relax-NG-Format angewiesen ist, hat der Autor in einem zusätzlichen Schritt die XML-Schema-Beschreibungen beider Web Services mit Hilfe der Werkzeuge *Sun Relax NG Converter* [142] und *Trang* [143] in eine äquivalente Relax-NG-Grammatik übersetzt.

5.2.3 Durchführung der Messungen

Dem Autor war es leider nicht möglich, alle in Abschnitt 5.1 vorgestellten Kompressoren experimentell zu untersuchen. Die Autoren von Fast Web Service und Xaust haben keine Implementierungen ihrer Ansätze öffentlich zugänglich gemacht. Bei XGrind und Exalt ist eine Implementierung zwar veröffentlicht, jedoch waren vergleichende Messungen damit nicht möglich. Mit dem XGrind-Kompressor ließen sich XML-Dokumente zwar komprimieren, die Dekompression lieferte aber „defekte“ XML-Dokumente, welche unvollständig und nicht wohlgeformt waren. Bei Exalt wurde die Verarbeitung von Namespace-Informationen noch nicht unterstützt. Der Kompressor BiM ist wegen seiner Beschränkung auf MPEG-7-Metadaten für Messungen mit SOAP-Daten ebenfalls ungeeignet.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  <xsd:element name="Envelope">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="tns:Header" minOccurs="0"/>
        <xsd:element ref="tns:Body" minOccurs="1"/>
      [...]
    </xsd:complexType>
  </xsd:element>
  [...]
</xsd:schema>
```

(a) XML-Schema-Dokument zur generischen Beschreibung von SOAP-Nachrichten (W3C)

```
<?xml version="1.0" encoding="ISO-8859-1"?> <xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:calc="http://example.com/calculator"
  targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  <xsd:import namespace='http://example.com/calculator'
    schemaLocation="calculator_wsdl.xsd"/>
  <xsd:element name="Envelope">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="tns:Header" minOccurs="0"/>
         <xsd:element name="Body" minOccurs="1">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element ref="calc:doNothing"/>
              <xsd:element ref="calc:doNothingResponse"/>
              <xsd:element ref="calc:increment"/>
              <xsd:element ref="calc:incrementResponse"/>
              <xsd:element ref="calc:add"/>
              <xsd:element ref="calc:addResponse"/>
              <xsd:element ref="calc:add6ints"/>
              <xsd:element ref="calc:add6intsResponse"/>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      [...]
    </xsd:complexType>
  </xsd:element>
  [...]
</xsd:schema>
```

(b) Angepasstes XML-Schema-Dokument

Abbildung 5.5: Integration der XML-Schema-Dokumente für SOAP-Envelope und Anwendungsdaten am Beispiel des Taschenrechner-Web-Services

Experimentelle Messungen waren somit lediglich mit Fast Infoset und Xebu möglich (Implementierungen: [72, 129]). Wie in Abschnitt 5.1.2 erläutert, gibt es noch keine Fast-Infoset-Implementierungen mit Initial Vocabularies. Folglich handelt es sich bei der untersuchten Implementierung um einen nicht-grammatikspezifischen Kompressor.

Als Vergleichsgrößen hat der Autor die Kompressoren gzip (als generischen Kompressor), XMill und xmlppm (als weitere nicht-grammatikspezifische XML-Kompressoren) sowie den Differenzcodierungsansatz (dynamisch-grammatikspezifischer Kompressor), welcher in Abschnitt 4.4 vorgestellt wurde, in die Messungen mit einbezogen.

Alle Kompressoren wurden dabei in ihrer Grundeinstellung getestet, d. h. ohne Angabe von zusätzlichen Kommandozeilenparametern. Eine Ausnahme bildet hier Xebu, weil dieser Kompressor nicht als fertige Applikation zur Verfügung gestellt wird, sondern als Funktionsbibliothek für Java. Der Autor hat daher ein minimales Kompressionsprogramm auf Basis von Xebu implementiert.

Allerdings war die Xebu-Funktionsbibliothek nicht in der Lage, die Relax-NG-Grammatik für den Amazon-Web-Service korrekt zu verarbeiten. Dies ist offenbar auf einen Programmfehler in Xebu zurückzuführen. Daher hat der Autor Xebu von der Amazon-Testreihe ausgenommen.

Der Differenzcodierungsansatz wurde in der Variante „DUL + xmlppm“ untersucht, denn diese lieferte in Vergleichsmessungen (vergleiche Abschnitt 4.7) die besten Ergebnisse.

Der Autor hat die Effektivität der einzelnen Kompressoren für sämtliche Request- und Response-Nachrichten des Taschenrechner- sowie des Amazon-Web-Services gemessen. Dabei wurde jeweils die resultierende Dateigröße S und zusätzlich auch die Kompressionsrate λ ermittelt. Diese ist der Quotient aus $S_{\text{komprimiert}}/S_{\text{unkomprimiert}}$. Weiterhin wurde für jeden Kompressor die kumulierte Dateigröße Σ über alle Testdatensätze ermittelt.

5.2.4 Auswertung

Tabelle 5.1 zeigt die Ergebnisse der Messungen. Im Hinblick auf eine kompaktere Darstellung sind hier nicht alle Werte für λ aufgeführt, sondern nur der jeweils beste (λ_{best}), der schlechteste (λ_{worst}) und der Durchschnittswert ($\lambda_{\varnothing}$). Dabei ist zu beachten, dass der Durchschnittswert nicht mit den Dateigrößen gewichtet ist; folglich weicht $\lambda_{\varnothing}$ von $\Sigma_{\text{komprimiert}}/\Sigma_{\text{unkomprimiert}}$ ab.

Die Werte zeigen klar, dass die beiden grammatikspezifischen Verfahren, also Xebu und die Differenzcodierung, beim Taschenrechner-Web-Service die besten Ergebnisse liefern. Dies war auch zu erwarten, denn die Grammatik gibt hier bereits einen Großteil des Markups fest vor, und folglich braucht dieser Anteil im komprimierten Datensatz nicht codiert zu werden.

	unkom-primiert	gzip	XMill	xml-ppm	Fast Infoset	Xebu	Diff.-codiert
doNothing (Request)	336	224	338	167	210	103	21
doNothing (Response)	344	229	352	173	210	103	21
increment (Request)	381	236	334	177	246	127	52
increment (Response)	425	249	351	187	246	127	52
add (Request)	401	239	360	181	269	135	106
add (Response)	401	238	361	180	246	127	95
add6ints (Request)	559	307	436	242	397	217	155
add6ints (Response)	429	255	378	195	254	135	104
Amazon small (Req.)	680	371	519	319	455	–	255
Amazon small (Resp.)	9,144	1,625	2,072	1,576	7,446	–	2,366
Amazon med. (Req.)	681	373	518	321	456	–	257
Amazon med. (Resp.)	60,319	8,795	8,298	7,190	46,283	–	16,096
Amazon large (Req.)	680	371	520	319	455	–	255
Amazon large (Resp.)	299,619	45,841	31,977	32,171	236,349	–	88,103
Σ	374,399	59,353	46,814	43,398	293,522	–	107,938
λ_{best}	1.00	0.15	0.11	0.11	0.58	0.30	0.06
λ_{worst}	1.00	0.67	1.02	0.50	0.81	0.39	0.38
$\lambda_{\varnothing}$	1.00	0.50	0.71	0.39	0.67	0.32	0.24

Tabelle 5.1: Kompressionsergebnisse für typische SOAP-Nachrichten, alle Dateigrößenangaben in Bytes

Allerdings fällt auch auf, dass die Differenzcodierung bei den Response-Nachrichten des Amazon-Web-Services vergleichsweise schlechte Ergebnisse liefert. Der Grund dafür liegt in der Tatsache, dass hier die Erstellung detaillierter Skelettdatensätze prinzipbedingt nicht möglich ist: Der Amazon-Web-Service liefert in Response-Nachrichten eine Produktliste, in der sämtliche Produkte enthalten sind, die den in der Anfrage spezifizierten Suchparametern entsprechen (im Beispiel also alle Produkte zum Thema „web services“). Bei der Erzeugung des Skelettdatensatzes für Response-Nachrichten ist nicht vorhersehbar, wie viele Produkte in einer Antwortnachricht enthalten sein werden. In der gegenwärtigen Implementierung wird in diesem Fall ein Skelettdatensatz mit einem leeren Listentyp im SOAP-Body erzeugt (vergleiche Abschnitt 4.8). Als Folge ergibt sich ein vergleichsweise großes Differenzdokument.

In der Spalte für Fast Infoset ist deutlich zu erkennen, dass die Indizierung von Strings in Verbindung mit Auslassen von End-Tags auf den vorliegenden Testdaten nicht effektiv funktioniert – weder bei den SOAP-Dateien des Taschenrechner-Web-Services noch bei denen des Amazon-Web-Services. Das deutlich bessere Abschneiden von Xebu beim Taschenrechner-Web-Service ist folglich auf die drei hier zusätzlich implementierten Kompressionsstrategien zurückzuführen: Berücksichtigung von Datentypen bei der Codierung (Xebu codiert die Integer-Werte mit jeweils vier Bytes), Vorabinitialisierung von Indextabellen mit Hilfe der Grammatikbeschreibung und Einsatz von

Automatenstrukturen zum Auslassen des Markups, welches sich aufgrund der Grammatikbeschreibung vorhersagen lässt.

Zusammenfassend lässt sich feststellen, dass die Verwendung von grammatikspezifischen Kompressoren vor allem bei kleinen Dateien deutliche Vorteile bietet. Bei den größeren Dateien des Amazon-Web-Services waren wegen einer fehlerhaften Implementierung mit Xebu keine Messungen möglich. Der Differenzcodierungsansatz liefert hier prinzipbedingt nur mäßige Ergebnisse.

5.3 Architektur

Obwohl die Vergleichsmessungen zeigen, dass grammatikspezifische Kompressionstechniken bei kleinen Datensätzen klare Vorteile hinsichtlich der erzielten Kompressionsraten bieten, bleibt ihr Nutzen für praktische Anwendungen zunächst fraglich:

Die Differenzcodierung leidet vor allem darunter, dass dieser Ansatz bei Listentypen prinzipbedingt nicht effizient funktioniert. Außerdem müssen hier für alle Web-Service-Operationen Skelettdatensätze vorgehalten werden, was zu einem großen Bedarf an Festspeicher auf dem Gerät führt. Zudem ist die Berechnung von Differenzdokumenten algorithmisch recht komplex [159].

Auch der Xebu-Ansatz bringt einen großen Speicherplatzbedarf mit sich, denn er arbeitet mit String-Indextabellen, die mit der Länge des Datensatzes dynamisch wachsen können. Auch die Automatenstruktur, die zur Vorhersage von Markup verwendet wird, muss im Speicher vorgehalten werden.

Dynamisch wachsende RAM-Speicherstrukturen und auch ein großer Bedarf an Festspeicherplatz sind bei den meisten Mobilanwendungen nachteilig.

Auffällig ist außerdem, dass sämtliche bisher vorgestellten Ansätze auf einer Kombination mehrerer Techniken beruhen. Am deutlichsten wird dies bei Xebu, denn hier werden fünf Techniken miteinander kombiniert: String-Indizierung, Auslassen von End-Tags, Vorabinitialisierung der Indextabellen, datentypspezifische Zeichendatencodierung sowie Automatenstrukturen zum Vorhersagen von Markup. Für eine Implementierung auf ressourcenbeschränkten Geräten wäre es hingegen vorteilhaft, wenn die Kompression nur auf einem einzigen algorithmischen Ansatz beruhte, der sich mit geringem Ressourcenaufwand implementieren ließe.

Zielstellung dieser Arbeit ist daher die Konstruktion eines Kompressors, der einen sehr geringen RAM-Speicherbedarf und auch eine geringe algorithmische Komplexität aufweist. Ausgangspunkt bei der Entwicklung des Verfahrens war die Überlegung, wie viel Unsicherheit im Sinne der Informationstheorie durch die Kenntnis der Grammatikbeschreibung beim Empfänger der SOAP-Nachricht beseitigt wird.

Wie wir gesehen haben, schränkt die Grammatikbeschreibung zum einen mögliche Dokumentstrukturen ein, d. h. sie legt fest, welche Tags in welcher Reihenfolge auf-

einander folgen können. Zum anderen liefert sie Informationen über Datentypen. Mit Kenntnis der Datentypen ist es möglich, effizientere Codierungen für Zeichendaten vorzunehmen. Beispielsweise kann ein Zahlenwert vom Typ `xsd:int` immer mit vier Bytes codiert werden. Wie könnte nun ein geeignetes Modell der Informationsquelle aussehen, das beide Aspekte bei der Codierung berücksichtigt?

5.3.1 Automatenstrukturen als Modell der Informationsquelle

Existierende Arbeiten zum Thema XML-Kompression, insbesondere Exalt, Xaust und Xebu, haben bereits gezeigt, dass sich Automatenstrukturen zur Repräsentation möglicher Dokumentstrukturen einsetzen lassen.

Allerdings verwenden die Autoren hier ausschließlich deterministische endliche Automaten (DEA). Diese sind jedoch nicht mächtig genug, um die Grammatik, die durch ein XML-Schema-Dokument beschrieben wird, vollständig zu repräsentieren. Dies liegt daran, dass die Sprachen, die durch XML-Grammatiken wie DTDs oder XML Schema ausgedrückt werden können, nicht Teilmenge der regulären Sprachen sind. Ein DEA ist lediglich in der Lage, mögliche Sequenzen der *direkten* Kindelemente eines Elements zu beschreiben; dagegen ist eine Beschreibung, die auch die Kindelemente der Kindelemente (usw.) mit einschließt, mit einem einzigen DEA im Allgemeinen nicht zu realisieren. Segoufin und Vianu behandeln dieses Problem detailliert in [134].

Folglich müssen bisherige Kompressoren wie BiM oder Exalt pro Elementtyp einen separaten Automaten erzeugen, der mögliche Kindelementsequenzen beschreibt. Diese Kompressoren müssen daher auch zusätzliche Mechanismen bereitstellen, um während des Kompressionsvorgangs zwischen den einzelnen Automaten umzuschalten. Eine Sonderstellung nimmt hier Xebu ein. Bei diesem Kompressor wird nach Angabe der Autoren tatsächlich ein zusammenhängender DEA erzeugt, dieser repräsentiert jedoch nicht die gesamte Relax-NG-Grammatik, sondern dient lediglich als Zusatz, um bestimmte Elementsequenzen vorhersagen zu können.

Neben DEA werden in der Literatur noch eine Reihe weiterer Automatentypen zur Verarbeitung von XML-Dokumenten beschrieben – hier jedoch nicht im Kontext mit Datenkompression, sondern primär mit der Zielstellung des Parsens. Besonders bekannt sind die Baumautomaten [30]. Zu nennen sind hier außerdem endliche Automaten mit Levelangabe [41], kardinalitätsbeschränkte Automaten [126] sowie Raupenautomaten [14, 15]. Eine umfassende Übersicht über die einzelnen Funktionsweisen sowie einen Vergleich der Ansätze findet der Leser in [13]. Weiterhin wurde in [93] die Möglichkeit untersucht, spezialisierte XML-Parser mit Hilfe gängiger Parsergeneratoren aus einer DTD zu erzeugen.

Das vom Autor entwickelte Datenkompressionsverfahren basiert auf der Verarbeitung von XML mittels *deterministischer Kellerautomaten (DKA)*. Die theoretischen Grundlagen hierfür wurden von Segoufin und Vianu entwickelt [134]. Diese Auto-

matenklasse bietet – wie wir sehen werden – ideale Voraussetzungen für Datenkompressionsanwendungen.

5.3.2 Konstruktion eines deterministischen Kellerautomaten

Das vom Autor entwickelte Datenkompressionsverfahren für SOAP-Nachrichten setzt das Vorhandensein eines *deterministischen Kellerautomaten (DKA)* zum Parsen der von einem Web Service ausgetauschten XML-Dokumente voraus. Dieser DKA akzeptiert genau jene XML-Dokumente, die der XML-Schema-Beschreibung des Web Services genügen.

Dass sich aus XML-Grammatikbeschreibungen DKA konstruieren lassen, haben Segoufin und Vianu bereits 2002 in [134] gezeigt. Allerdings behandeln die Autoren diese Problemstellung ausschließlich theoretisch, eine algorithmische Lösung für die Konstruktion eines geeigneten DKA geben sie nicht an.

Daher haben Brandt [13] und Werner et al. [157] die Idee von Segoufin und Vianu aufgegriffen und einen Algorithmus für die Konstruktion eines DKA aus einem XML-Schema-Dokument entwickelt.

Die Erzeugung des DKA verläuft dabei in drei Schritten: Zunächst wird das XML-Schema-Dokument, welches die zu komprimierende XML-Sprache beschreibt, in eine *reguläre Baumgrammatik (RBG)* umgeformt. In einem zweiten Schritt wird aus dieser RBG eine Menge von DEA erzeugt. Im dritten Schritt wird aus der RBG und den DEA schließlich ein zusammenhängender DKA konstruiert.

Da im Rahmen dieser Arbeit nicht die Konstruktion des DKA im Mittelpunkt steht, sondern dessen Anwendung zur Datenkompression, gibt der Autor nur einen Überblick über die einzelnen Schritte. Eine ausführlichere Darstellung findet der Leser in [13] und [157].

Erzeugung einer regulären Baumgrammatik

In einem ersten Verarbeitungsschritt wird die XML-Schema-Grammatik des Web Services in eine RBG umgeformt. Dies ist notwendig, weil die XML-Schema-Syntax wegen ihrer hohen Komplexität algorithmisch nur sehr schwer zu verarbeiten ist. Dieser erste Umformungsschritt dient also lediglich dazu, von dieser sehr technischen Darstellungsweise zu abstrahieren.

Eine RBG ist ein Spezialfall eines *Produktionssystems*. Hierunter verstehen wir eine formale Beschreibungsform für Grammatiken. Jedes Produktionssystem G ist ein 4-Tupel $G = (N, T, P, S)$. Dabei bezeichnet N die Menge der Nichtterminalsymbole, T die Menge der Terminalsymbole, P die Menge der Produktionsregeln und $S \subseteq N$ die Menge der Startsymbole.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a" type="A"/>
 <xsd:complexType name="A">
  <xsd:choice>
   <xsd:element name="b" minOccurs="0">
    <xsd:complexType>
     <xsd:choice>
      <xsd:element ref="a" minOccurs="2" maxOccurs="2"/>
      <xsd:element name="c" type="xsd:int"/>
     </xsd:choice>
    </xsd:complexType>
   </xsd:element>
  </xsd:choice>
 </xsd:complexType>
</xsd:schema>
```

Abbildung 5.6: Rekursive XML-Grammatik in XML-Schema-Darstellung

Die Besonderheit bei einer RBG im Vergleich zu anderen, gängigen Formen von Produktionssystemen (z. B. den Chomsky-Grammatiken [133]) besteht darin, dass bei Anwendung der Produktionsregeln nicht Zeichenketten, sondern Baumstrukturen abgeleitet werden. RBG beschreiben damit Sprachen, deren Wörter nicht aus einzelnen Zeichen, sondern aus hierarchischen Baumstrukturen gebildet werden. Solche Sprachen heißen Baumsprachen [30]. Da ein XML-Dokument ebenfalls als Baum aufgefasst werden kann (vergleiche Anhang A.4), sind Baumsprachen und die zugehörigen Grammatiken in diesem Anwendungsbereich besonders interessant.

Bei einer RBG haben die Produktionsregeln die folgende Form: Auf ihrer linken Seite steht ein Nichtterminalsymbol und auf der rechten Seite ein Terminalsymbol gefolgt von einem regulären Ausdruck über die Menge der Nichtterminalsymbole. Der reguläre Ausdruck $RegEx$ auf der rechten Seite einer Regel heißt auch Inhaltsmodell und ist für die Namensgebung der *regulären* Baumgrammatik verantwortlich.

$$Nichtterminal \rightarrow Terminal \;\; RegEx$$

Das Terminalsymbol repräsentiert dabei jeweils einen Knoten im Baum, der bei der Ausführung dieser Regel erzeugt wird – bei XML wäre dies ein Element. Der reguläre Ausdruck spezifiziert dabei mögliche Sequenzen von Nichtterminalen, die zu Kindern dieses Elements abgeleitet werden. Er ist also für die Struktur des abgeleiteten Baumes verantwortlich.

Eine XML-Schema-Grammatik wird nun in die formale Darstellungsweise einer RBG umgewandelt, indem man rekursiv (beginnend bei den globalen Elementdeklarationen) die Anweisungen im XML-Schema-Dokument durchläuft und jede Anweisung in eine entsprechende Produktionsregel bzw. in einen regulären Ausdruck umformt. Murata et al. geben in [101] Vorschriften für diese Umformung an. Wegen der

recht umfangreichen und komplexen Sprachstruktur von XML Schema werden wir diese Umformungen hier nicht im Detail ausführen, sondern lediglich anhand des in Abbildung 5.6 dargestellten Beispiels erläutern. Dieses dient als Ausgangspunkt für alle im Folgenden beschriebenen Umformungsschritte.

Für dieses Beispiel ergibt sich die folgende RBG G:

$G = (\{A, B, C, \text{xsd:int}\}, \{a, b, c\}, P, \{A\})$ mit $P = \{$

$$\begin{array}{lcl} A & \rightarrow & a\ (B + \varepsilon), \\ B & \rightarrow & b\ (AA + C), \\ C & \rightarrow & c\ (\text{xsd:int}) \end{array}$$

$\}$

Betrachten wir zunächst die Konstruktion der Mengen N, T und S: Alle im Schema über das Attribut `type` referenzierten Datentypen (sowohl einfache als auch komplexe) ergeben in der RBG Nichtterminale. Die „built-in“ Datentypen sind in der Beispiel-RBG mit dem Präfix `xsd:` gekennzeichnet, alle anderen werden durch Großbuchstaben dargestellt.

Alle im Schema deklarierten Elementnamen werden in der RBG zu Terminalsymbolen (dargestellt durch Kleinbuchstaben). Die Menge der Startsymbole ist gegeben durch die Nichtterminale, welche zu Elementnamen auf oberster Ebene gehören (globale Elementdeklarationen). Der Leser kann diese Beziehungen anhand von Abbildung 5.6 und der oben dargestellten RBG nachvollziehen.

Die Produktionsregeln im Beispiel (Menge P) haben die folgende Bedeutung: Ein Element a enthält entweder ein b oder[1] nichts (ϵ). Ein Element b enthält entweder zwei a Elemente oder ein c Element, welches einen `xsd:int` Wert enthält. Damit ist die dargestellte RBG G äquivalent zum in Abbildung 5.6 dargestellten Schema-Dokument.

Murata zeigt, dass sich mit Hilfe einer RBG auch komplexe XML-Schema-Funktionalitäten, wie Vererbung oder die Erweiterung bzw. Beschränkung von Datentypen, realisieren lassen. Auf diese Besonderheiten wird im Rahmen dieser Arbeit aber nicht näher eingegangen, Ausführungen hierzu findet der Leser in [101] und auch [13].

Erzeugung der endlichen Automaten

In einem nächsten Verarbeitungsschritt wird zu jedem Inhaltsmodell der RBG ein DEA erzeugt. Geeignete Algorithmen zur Erzeugung von DEA aus solchen regulären Ausdrücken sind aus der Theoretischen Informatik hinlänglich bekannt. Im Rahmen dieser Arbeit kam das in [1] beschriebene Verfahren zum Einsatz.

[1] Der Oder-Operator wird in den regulären Ausdrücken durch das Zeichen + dargestellt.

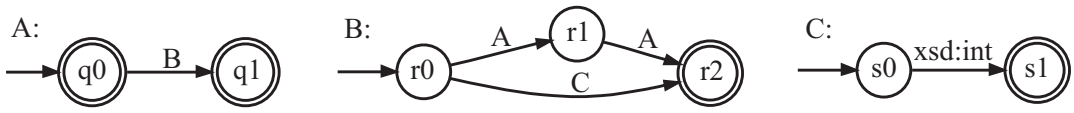


Abbildung 5.7: Endliche Automaten erzeugt aus den Inhaltsmodellen der Grammatik

Für das Beispiel ergeben sich die in Abbildung 5.7 dargestellten Automaten (Startzustände sind durch einen Pfeil gekennzeichnet, Endzustände durch einen Doppelkreis). Man erkennt leicht, dass diese Automaten genau die Inhaltsmodelle der Datentypen A, B und C unseres Beispiels repräsentieren. Auch in dieser Darstellungsweise ist unmittelbar ersichtlich, welche direkten Kindelemente in welcher Reihenfolge auftreten können. Betrachten wir exemplarisch den linken Automaten in Abbildung 5.7: Ein Element vom Typ A hat entweder keine Kinder (Startzustand ist auch Endzustand) oder aber ein Kind vom Typ B.

Erzeugung des Kellerautomaten

Die konstruierten DEA beschreiben also die Inhaltsmodelle unserer Grammatik G und damit auch die Struktur der von der Grammatik erzeugten Bäume.

Allerdings liegt bei vielen XML-Anwendungen das zu verarbeitende XML-Dokument gar nicht als Baum vor, sondern als Zeichenkette – dies gilt vor allem auch für die Übertragung von XML-Daten über ein Netzwerk. Folglich ist es vorteilhaft, das Dokument direkt in der Textdarstellung zu parsen (und zu komprimieren), anstatt es zunächst in die Baumdarstellung umzuwandeln.

Der im Folgenden konstruierte Parser-DKA verarbeitet daher das XML-Dokument als Zeichenkette, d. h. er liest öffnende und schließende Tags von der Eingabe. Dies ist ein ganz wesentlicher Unterschied zu den Automatenvarianten, die bisher bei der Datenkompression eingesetzt werden. Jene verarbeiten nicht die XML-Tags, sondern dienen lediglich zur Vorhersage möglicher Kindelemente [71, 55, 150]. Dies bringt allerdings den Nachteil mit sich, dass die Prozesse des Parsens und der Kompression getrennt voneinander ablaufen müssen. Die Besonderheit des vom Autor entwickelten Verfahrens besteht in der Kombination beider Prozesse zu einem einzigen Verarbeitungsschritt.

Die Konstruktion eines solchen Parser-DKA ist nicht trivial und bedarf einiger Formalismen. Der interessierte Leser kann eine detaillierte, formale Beschreibung des Konstruktionsverfahrens dem Anhang C entnehmen. Im Folgenden betrachten wir das Verfahren im Überblick:

Eingabe ist dabei eine RBG $G = (N, T, P, S)$ sowie eine Menge endlicher Automaten, die die einzelnen Inhaltsmodelle der RBG darstellen. Ausgabe ist ein DKA, welcher durch Leerung des Kellers terminiert: Zu Beginn der Verarbeitung befindet sich ein

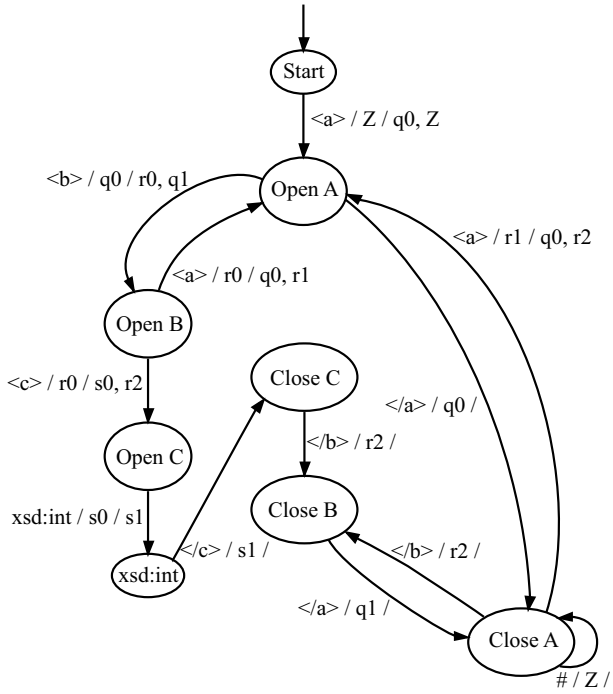


Abbildung 5.8: Aus der XML-Schema-Grammatik konstruierter Kellerautomat

spezielles Symbol im Keller (im Folgenden mit Z bezeichnet). Wird dieses Symbol aus dem Keller gelesen und nicht im selben Verarbeitungsschritt wieder zurückgeschrieben, terminiert der DKA. Diese Art eines DKA benötigt also keine Endzustände. Auf eine formale Definition für solche Kellerautomaten wird an dieser Stelle verzichtet, weil sie für das Verständnis des Verfahrens nicht notwendig ist. Der interessierte Leser findet eine solche Definition zum Beispiel in [133].

Die grundlegende Idee bei dem konstruierten Automaten besteht darin, über den Kellerspeicher die Ausführung der einzelnen DEA zu simulieren und dabei die öffnenden und schließenden Tags zu verarbeiten: Beim Absteigen des XML-Baumes (Verarbeitung öffnender Tags) werden die zu den verarbeiteten Datentypen gehörigen DEA-Zustände auf dem Kellerspeicher abgelegt. Bei der Verarbeitung von schließenden Tags werden diese DEA-Zustände wieder aus dem Keller gelöscht – der Speicherplatzbedarf des Kellers wird also durch die Verschachtelungstiefe des Eingabedokuments begrenzt. Auf dem Keller liegt zuoberst immer der DEA-Zustand, der als Nächstes eingenommen wird.

Der Leser kann die folgenden Ausführungen anhand von Abbildung 5.8 nachvollziehen:

Die Zustände des DKA ergeben sich wie folgt: Für jeden komplexen Typ werden im DKA zwei Zustände erzeugt, ein öffnender und ein schließender. Ein öffnender Zu-

stand wird nach der Verarbeitung eines öffnenden Tags eingenommen und ein schließender bei der Verarbeitung des schließenden Tags. Für jeden einfachen Datentyp aus G (z. B. `xsd:int`, `xsd:string`) wird hingegen nur ein Zustand erzeugt – einfache Datentypen beschreiben Zeichendaten und folglich werden keine öffnenden oder schließenden Tags verarbeitet. Weiterhin gibt es im DKA einen speziellen Startzustand.

Dann werden die Zustandsübergänge erzeugt: Jeder Zustandsübergang im DKA ist mit einem 3-Tupel (*read* / *pop* / *push*) als Zustandsübergangsmarkierung versehen. Hierbei bezeichnet *read* das Zeichen, das der Automat als Nächstes aus dem Eingabedokument liest, *pop* das Symbol, welches an oberster Position im Kellerspeicher steht (dieses wird bei Ausführung der Transition aus dem Keller entfernt), und *push* die Werte, die in den Keller geschrieben werden. Die Werte im Ausdruck *push* werden von rechts nach links abgearbeitet und in dieser Reihenfolge in den Keller geschrieben.

Den vollständigen DKA für unser Beispiel zeigt Abbildung 5.8. Wie man hier sieht, wird bei jeder Transition, die zu einem öffnenden Zustand führt, ein zusätzlicher DEA-Zustand auf den Stack geschrieben. Bei jeder Transition, die zu einem schließenden Zustand führt, wird dagegen ein DEA-Zustand vom Stack entfernt. Auf diese Weise wird über den Stack die Ausführung der einzelnen DEA simuliert. Beim Erreichen des Dokument-Endes (in der Abbildung dargestellt durch das Zeichen #) ist bei einem gültigen Eingabedokument der Stack leer, und der DKA terminiert.

Der Leser findet ein Beispiel für ein gültiges Dokument in Abbildung 5.10(a) auf Seite 125 – anhand dieses Dokuments lässt sich ein Automatendurchlauf nachvollziehen: Zu Beginn der Verarbeitung befindet sich der DKA im Zustand „Start“. Als öffnendes Wurzel-Tag wird `<a>` verarbeitet. Dabei wird die DKA-Transition von „Start“ nach „Open A“ ausgeführt und $q0$ (Startzustand vom DEA A) auf den Stack geschrieben. Der DKA befindet sich nun im Zustand „Open A“. Als Nächstes wird `<b>` gelesen. Bei einer RBG, die aus einem XML-Schema-Dokument erzeugt wurde, lässt sich aufgrund des Tag-Namens auf das zugehörige Nichtterminal schließen (sog. Single-Type-Eigenschaft, vergleiche [101]); im Beispiel ist das zu `<b>` gehörige Nichtterminal B. DEA A wechselt daher von $q0$ nach $q1$ und ruft dabei den DEA B auf. Folglich wird bei der korrespondierenden DKA-Transition $q0$ vom Stack genommen und $q1$ gefolgt von $r0$ (Startzustand vom DEA B) geschrieben. Der DKA befindet sich im Zustand „Open B“. Dann wird wieder `<a>` gelesen. DEA B wechselt von $r0$ nach $r1$ und ruft dabei wieder DEA A auf. Im DKA wird dies simuliert, indem $r0$ vom Stack genommen und $r1$ gefolgt von $q0$ (Startzustand von DEA A) geschrieben wird. Der DKA befindet sich im Zustand „Open A“. Als Nächstes wird `</a>` gelesen. DEA A wird beendet und $q0$ vom Stack abgeräumt ($q0$ ist gültiger Endzustand von DEA A). Der DKA befindet sich nun im Zustand „Close A“, und auf dem Stack liegt wieder $r1$ zuoberst. Es wird `<a>` gelesen, und DEA B wechselt von $r1$ nach $r2$. Folglich wird $r1$ vom Stack genommen und $r2$ gefolgt von $q0$ geschrieben. Alle weiteren Eingabezeichen werden nach gleichem Muster verarbeitet.

Den genauen Algorithmus zur Erzeugung der Transitionen kann der Leser gleichfalls dem Anhang C entnehmen. Weitere Erläuterungen zur Konstruktion derartiger Kellerautomaten findet der Leser auch in [13] und [157].

Konstruktion des Kompressors

Die zentrale Idee des vom Autor entwickelten Kompressionsverfahrens besteht nun darin, einen solchen Parserautomaten auch für die Datenkompression zu verwenden: Der DKA wird hierzu sowohl auf Sender- als auch auf Empfängerseite aus der WSDL-Beschreibung des Web-Services konstruiert, so dass beide Seiten über identische Kopien verfügen. Der Sender verarbeitet das zu übertragende Dokument mit seinem Automaten. Eine Datenkompression wird nun dadurch erreicht, dass lediglich der Pfad durch den DKA codiert wird. Anhand der Codewortfolge kann der Empfänger den Pfad durch den DKA nachvollziehen und so das XML-Dokument rekonstruieren. Wie wir im Folgenden sehen werden, ist die Codewortfolge, die den Pfad durch den Automaten repräsentiert, deutlich kompakter als die Textdarstellung des Dokuments.

Der Autor verwendet einen DKA-Parserautomaten somit als detailliertes Modell der Informationsquelle: Mögliche Zustandsübergänge im Automaten repräsentieren nämlich genau den Grad an Unsicherheit, den der Sender durch Aussenden eines Tags beim Empfänger beseitigt – dies ist die grundlegende Entdeckung, auf der dieser neuartige Kompressionsansatz beruht.

Zur Codierung der Zustandsübergänge im DKA wird jeder Zustandsübergang mittels eines binären Codeworts codiert. Um ein optimales Kompressionsergebnis zu erreichen, muss die Länge dieser Bitsequenz möglichst genau dem Entropiewert dieses Zustandsübergangs entsprechen (vergleiche Abschnitt 3.6). Alle möglichen Folgezustände werden dabei als gleich wahrscheinlich angenommen, denn in der Grammatikbeschreibung sind keine Angaben über die zu erwartenden Häufigkeiten enthalten.

Die Entropie eines Zustandsübergangs x_δ ergibt sich nach Definition 3.1 (Seite 46) als:

$$H(x_\delta) = -\log_2 P(x_\delta) = -\log_2 \frac{1}{n_\delta} = \log_2 n_\delta \ \text{[Bits]}$$

Dabei kennzeichnet n_δ die Anzahl möglicher Folgezustände.

Zur technischen Umsetzung dieser Idee kommt das Huffman-Verfahren zum Einsatz: Für jeden Zustand im DKA, der mehr als einen möglichen Folgezustand besitzt, wird das Verfahren von Huffman ausgeführt. Dabei wird jeweils ein Code erzeugt, der mögliche Folgezustände auf eindeutige Codewörter abbildet – und zwar so, dass die mittlere Codewortlänge minimiert wird (vergleiche auch hierzu Abschnitt 3.6). Zustände mit nur einem möglichen Folgezustand müssen offenbar nicht codiert werden – formal wird dies dadurch deutlich, dass sich nach der oben angegebenen Formel ein Entropiewert von 0 Bits ergibt. Da der Huffman-Algorithmus immer einen Präfix-

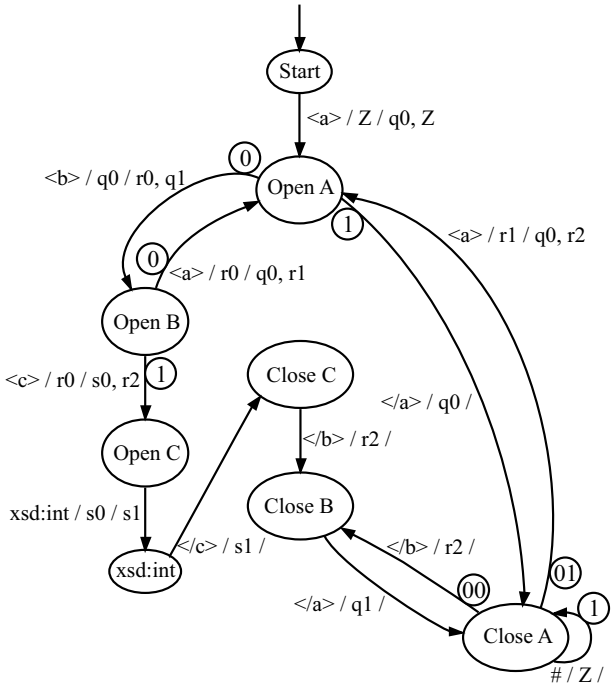


Abbildung 5.9: Kellerautomat aus Abbildung 5.8 ergänzt um binäre Codewörter zur Codierung von Zustandsübergängen

code konstruiert (d.h die Fano-Bedingung wird eingehalten), ist jede Codewortfolge eindeutig decodierbar.

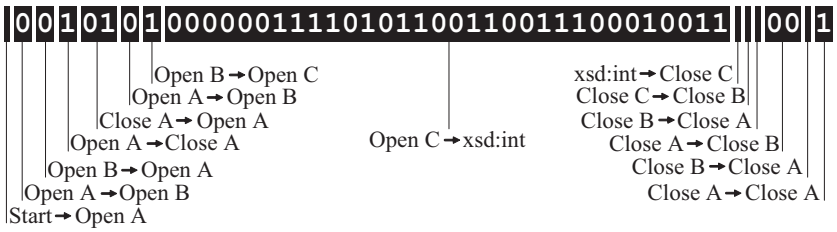
Abbildung 5.9 zeigt den Parser-DKA mit den so erzeugten Codewörtern (eingekreist dargestellt).

Neben dem Pfad durch den DKA – dieser repräsentiert schließlich nur das Markup des codierten Dokuments – müssen auch die Zeichendaten codiert werden: Für einige ausgewählte XML-Schema-Datentypen sind im Kompressions-DKA optimierte Codierungsregeln abgelegt. Diese erzeugen besonders kompakte Bitcodes fester Länge, z. B. 32 Bits für einen `xsd:int` Wert, ein einzelnes Bit für einen `xsd:boolean` Wert usw. Bei anderen Datentypen – also solchen, bei denen die Codierung als Bitfolge fester Länge nicht sinnvoll ist (z. B. bei `xsd:string`) – wird zeichenorientiert gearbeitet; sämtliche Zeichen werden in UTF-8-Codierung direkt in den komprimierten Bitstrom geschrieben und mit einer reservierten Stop-Byte-Sequenz terminiert.

In beiden Fällen wird der binär codierte Wert direkt an die aktuelle Position im Ausgabedatenstrom des Kompressors geschrieben. Dieses Vorgehen ist besonders vorteilhaft, denn auf diese Weise kann auf die Pufferung von Daten bei der Kompression oder Dekompression vollständig verzichtet werden (mitunter wird dies auch als Streaming-

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<a>
  <b>
    <a>
    </a>
    <a>
      <b>
        <c>64382739</c>
      </b>
    </a>
  </b>
</a>
```

(a) Codierung als Text (119 Bytes)



(b) Komprimierte Binärdarstellung (42 Bits ≈ 6 Bytes)

Abbildung 5.10: Beispieldokument vor und nach der Kompression

Fähigkeit bezeichnet). Dies ist ein klarer Vorteil gegenüber container-basierten Kompressionsverfahren wie etwa XMill.

Abbildung 5.10 zeigt ein Beispiel für ein Instanzdokument unserer Beispielgrammatik G und die zugehörige Binärdarstellung. Wie hier deutlich zu erkennen ist, ist der binäre Datenstrom deutlich kompakter als die Textdarstellung.

Anhand dieser Abbildung lässt sich auch der Dekompressionsprozess anschaulich nachvollziehen. Der Empfänger verwendet den Bitstrom zur Steuerung seiner Kopie des DKA. Wegen der Präfixfreiheit der verwendeten Huffman-Codierung kann ein Empfänger stets den Beginn und das Ende eines Codeworts erkennen. Bei jeder ausgeführten Transition schreibt er den korrespondierenden *read* Wert in die Ausgabe und rekonstruiert so die Tag-Sequenz aus dem unkomprimierten Dokument. Auch die Zeichendaten können stets eindeutig rekonstruiert werden: Bei Transitionen, die in einem Simple-Type-Zustand münden, wird in Abhängigkeit vom vorliegenden Datentyp entweder eine feste Anzahl von Bits gelesen (in unserem Beispiel wären dies 32 Bits für den `xsd:int` Wert) oder aber es werden solange Bytes von der Eingabe gelesen, bis die reservierte Stop-Byte-Sequenz im Bitstrom auftritt.

5.4 Implementierung

Der Autor hat das vorgestellte Verfahren prototypisch in Java implementiert. Diese Implementierung heißt *Xenia*, wobei dieser Name kein Akronym darstellt. Er trägt lediglich der Tatsache Rechnung, dass die Namen von Werkzeugen zur XML-Verarbeitung zumeist mit einem X beginnen. Im Folgenden weist der Autor auf einige Besonderheiten dieser Implementierung hin.

5.4.1 Verarbeitung von XML-Schema-Dokumenten

Als besonders vorteilhaft zur Verarbeitung von XML-Schema-Dokumenten hat sich die Verwendung der XML-Schema-API des Xerces-Parsers [147] erwiesen. Über diese API ist es vergleichsweise einfach möglich, auf die einzelnen Komponenten eines XML-Schema-Dokumentes zuzugreifen.

Ein Grundproblem bei der XML-Schema-Verarbeitung besteht in der hohen syntaktischen Komplexität und der Vielzahl möglicher Schreibweisen. Beispielsweise kann in einer Elementdeklaration der verwendete Datentyp über das Attribut `type` angegeben werden. Es ist aber auch möglich, die Typdefinition direkt als Kindelement der Elementdeklaration zu notieren. Solche syntaktischen Varianten machen die algorithmische Verarbeitung von XML-Schema-Dokumenten besonders aufwändig. Die XML-Schema-API von Xerces bietet dem Programmierer bereits eine stark abstrahierte Sicht auf die durch ein XML-Schema-Dokument ausgedrückte Grammatik. Insbesondere werden Anweisungen zur Erzeugung neuer Datentypen durch Vererbung bereits ausgewertet. Ein Programmierer kann also direkt auf die durch Vererbung erzeugten Datentypen zugreifen, und zwar ohne sich mit der XML-Schema-Syntax auseinander setzen zu müssen.

Mit Hilfe dieser API ist es somit möglich, die Komplexität der Xenia-Komponente zur Umwandlung eines XML-Schema-Dokuments in eine RBG vergleichsweise gering zu halten.

5.4.2 Behandlung von Attributen

In XML gibt es neben Elementen auch Attribute zur Strukturierung von Informationen. Allerdings gibt es für Attribute keine Entsprechung in einer RBG, auf der ja der hier vorgestellte Ansatz letztendlich aufsetzt.

Die hier realisierte Lösung dieser Problemstellung wurde erstmals von Bouchou et al. in [10] vorgeschlagen: Attributdeklarationen in einer XML-Schema-Beschreibung werden bei der Umformung in eine RBG genau wie Elementdeklarationen behandelt. Damit es nicht zu Namenskonflikten zwischen Elementen und Attributen kommt,

werden die zu Attributen gehörigen Terminalsymbole und Nichtterminalsymbole allerdings eindeutig gekennzeichnet.

Bei der Erzeugung der RBG und der anschließenden, schrittweisen Umwandlung dieser RBG in einen DKA werden Attribute also völlig analog zu Elementen behandelt. Auch Attribute haben also einen öffnenden und einen schließenden Zustand im DKA, wobei die zugehörigen Tags im Eingabedokument natürlich nicht explizit vorhanden sind. Wie aber leicht einzusehen ist, lässt sich jedes Attribut in Tag-Schreibweise umformen – dies geschieht bei Xenia automatisch bei der Verarbeitung des Eingabedatensatzes. Beispielsweise wird `<a b="c">d</a>` umgeformt in `<a><b`$_{\texttt{att}}$`>c</b`$_{\texttt{att}}$`>d</a>`.

5.4.3 Behandlung von Namespace-Informationen

Eine weitere Besonderheit betrifft die Verarbeitung von Namespace-Informationen. Da ein XML-Schema-Dokument bereits eine Angabe über den Namespace der deklarierten Elemente enthält (Target-Namespace-Angabe), braucht diese Information nicht im komprimierten Dokument codiert zu werden.

Bei der Erzeugung der RBG aus dem XML-Schema-Dokument wird die Target-Namespace-Angabe ausgewertet, und alle Terminalsymbole werden mit ihrem korrespondierenden Namespace markiert. Auch bei der schrittweisen Umformung in einen Kompressions-DKA bleiben diese Namespace-Informationen erhalten. Jedem Tag, das bei der Ausführung einer Transition von der Eingabe gelesen wird, ist also ein Namespace zugeordnet. Der DKA kann somit den zugehörigen Namespace stets mit berücksichtigen.

Über diesen Mechanismus werden auch bei der Dekompression die Namespace-Angaben rekonstruiert. Dabei ist jedoch zu beachten, dass neue, fortlaufende Namespace-Präfixe erzeugt werden (`ns0`, `ns1`, `ns2` usw.). Die Namespace-Informationen selbst bleiben bei der Kompression also erhalten, Namespace-Präfixe gehen dagegen verloren.

Dies stellt bei Web-Service-Anwendungen jedoch keinen Nachteil dar, weil hier die XML-Nachrichten ausschließlich durch Maschinen ausgewertet werden. Für Anwendungen, bei denen es auf aussagekräftige Namespace-Präfixe ankommt, müsste die Xenia-Implementierung um Funktionalitäten zur Codierung von Präfix-Informationen erweitert werden.

5.4.4 Varianten bei der Codierung von Zeichendaten

Wie bereits erläutert, werden Zeichendaten grundsätzlich (in Abhängigkeit vom vorliegenden Datentyp) entweder als Bitsequenz fester Länge oder als Sequenz von UTF-8-Zeichen codiert.

Bei Dokumenten, die einen sehr großen Anteil an Daten vom Typ `xsd:string` aufweisen, führt die zeichenweise Codierung allerdings zu keinen akzeptablen Kompressionsraten. Dies liegt daran, dass die zeichenweise Codierung keinerlei Kompression beinhaltet.

Zur Überwindung dieser Schwierigkeit hat der Autor die Xenia-Implementierung um zwei Varianten zur Codierung von `xsd:string` Daten erweitert:

Die erste besteht in der zusätzlichen Anwendung des adaptiven Huffman-Verfahrens auf die Byte-Sequenz, welche die UTF8-Zeichen darstellen, und die zweite basiert auf dem bereits in Abschnitt 4.2.4 vorgestellten PPM-Algorithmus – angewendet auf die zu codierende Zeichenkette.

Diese Zusätze erfordern zwar zusätzliche Berechnungen bei der Kompression und Dekompression, sie ermöglichen aber – wie wir im folgenden Abschnitt genauer betrachten werden – deutlich bessere Kompressionsergebnisse bei Datensätzen mit einem großen Anteil an Zeichendaten vom Typ `xsd:string`.

5.5 Evaluation

Der Autor hat die Kompressionsleistung seiner Implementierung Xenia anhand der beiden Test-Web-Services untersucht, die auch für die Messungen verwandter Ansätze in Abschnitt 5.2 zum Einsatz kamen.

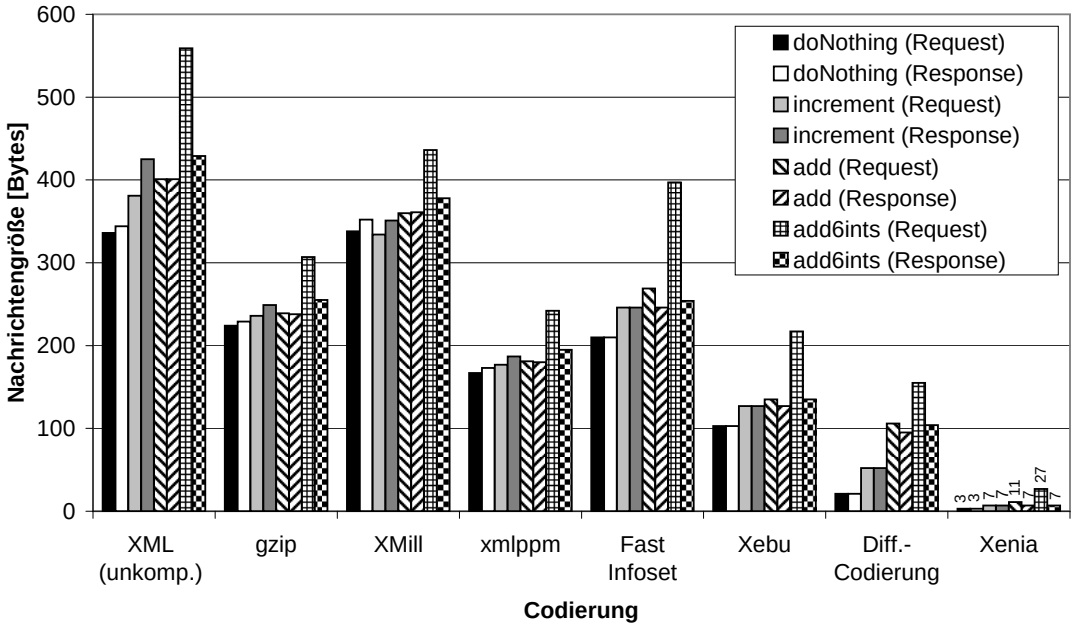
Abbildung 5.11(a) zeigt eine grafische Darstellung der Ergebnisse aus der Messreihe „Taschenrechner“ für die einzelnen Kompressoren.

In der Abbildung ist klar zu erkennen, dass Xenia mit Abstand die besten Kompressionsergebnisse liefert. Dies war auch zu erwarten, denn schließlich bestehen die hier ausgetauschten SOAP-Nachrichten nahezu vollständig aus Markup. Zudem schreibt das aus der WSDL-Datei erzeugte XML-Schema-Dokument sehr restriktiv vor, wie die einzelnen Tags geschachtelt sein können, und somit ergeben sich im Kompressions-DKA nur sehr wenige Zustände mit mehr als einem Folgezustand. Dies bewirkt, dass nur sehr wenige Bits für die Codierung des Markups erforderlich sind.

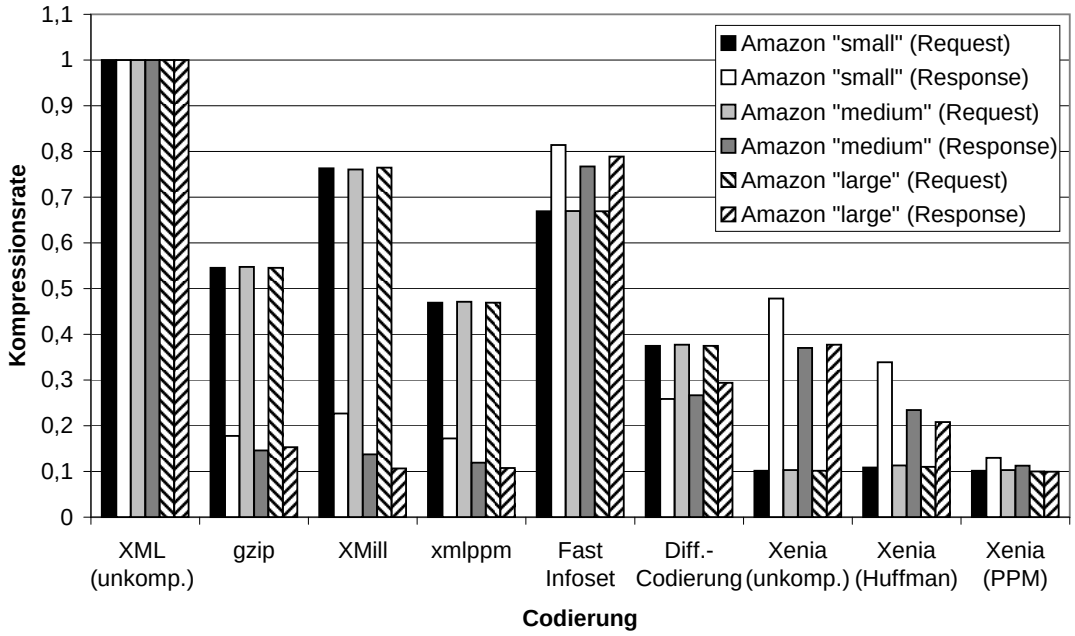
Abbildung 5.11(b) zeigt die Ergebnisse für den Amazon-Web-Service. Zur besseren Lesbarkeit ist in diesem Diagramm allerdings nicht die Nachrichtengröße, sondern die Kompressionsrate λ dargestellt. Bei Darstellung der Nachrichtengröße würde der sehr große Wertebereich[2] zu einer unleserlichen Darstellung bei kleinen Werten führen.

Da bei dem Amazon-Web-Service auch Zeichendaten vom Typ `xsd:string` übertragen werden, hat der Autor die Messungen mit allen drei verfügbaren Varianten

[2]Die Größe der unkomprimierten SOAP-Nachrichten variiert zwischen 680 und 229.616 Bytes, vergleiche Tabelle 5.1, Seite 114.



(a) Messreihe „Taschenrechner“



(b) Messreihe „Amazon“

Abbildung 5.11: Kompressionsleistung des Xenia-Kompressors im Vergleich zu verwandten Arbeiten

für die String-Codierung durchgeführt: unkomprimiert, adaptives Huffman-Verfahren und PPM (vergleiche Abschnitt 5.4.4).

Wie bei sämtlichen Amazon-Request-Nachrichten deutlich zu erkennen ist, liefern sowohl generische als auch nicht-grammatikspezifische Kompressoren (im Diagramm sind dies gzip, XMill, xmlppm und Fast Infoset) bei kleinen Dateien vergleichsweise schlechte Kompressionsraten. Bei größeren Nachrichten (d. h. den Amazon-Response-Nachrichten) wird die Kompressionsrate deutlich besser. Eine Ausnahme bildet hier Fast Infoset – die Kompressionsraten sind hier bei den Requests sogar etwas besser als bei den Responses. Der Autor hat diesen Effekt jedoch nicht näher untersucht, denn die Kompressionsleistungen von Fast Infoset blieben insgesamt deutlich hinter denen anderer Kompressoren zurück.

Der grammatikspezifische Kompressor „Differenzcodierung“ erreicht, wie bereits in Abschnitt 5.2 erläutert, beim Amazon-Web-Service nur mittelmäßige Kompressionsergebnisse: Bei den Requests ist die Kompressionsrate zwar erwartungsgemäß etwas besser als bei den generischen bzw. nicht-grammatikspezifischen Kompressoren, bei den Response-Nachrichten hingegen führt die Differenzcodierung zu nur mäßigen Ergebnissen. Wegen der in den Response-Nachrichten enthaltenen Listen-Typen lässt sich die Markup-Struktur bei Erzeugung der Skelettnachrichten nur sehr grob vorhersagen, und folglich sind die Differenzdokumente vergleichsweise groß.

Der grammatikspezifische Kompressor Xenia zeigt bei Amazon-Request-Nachrichten deutlich die besten Ergebnisse von allen Kompressoren. Die Nachrichten lassen sich hier auf rund 10% ihrer ursprünglichen Größe komprimieren ($\lambda = 0,1$). Da die Request-Nachrichten nur sehr kurze Strings enthalten, welche sich ohnehin kaum komprimieren lassen, macht es hier so gut wie keinen Unterschied, welche String-Codierungsvariante bei Xenia zum Einsatz kommt. Grundlegend anders verhält es sich bei den Amazon-Response-Nachrichten, denn diese enthalten einen großen Anteil (rund 40%) längerer Zeichenketten vom Typ `xsd:string`: Hier zeigt sich klar, dass die zusätzliche String-Kompression mit dem PPM-Algorithmus die besten Ergebnisse liefert. Xenia erreicht durchgehend eine Kompressionsrate von $\lambda \approx 0,1$. Allerdings wird dieses Ergebnis bei den Amazon-Response-Nachrichten auch annähernd von gzip, XMill und xmlppm erreicht, denn wegen des hohen String-Anteils an der Gesamtdatenmenge funktionieren auch diese Kompressoren hier sehr gut.

5.6 Ergebnis

Im Rahmen dieses Kapitels wurde ein völlig neuartiger, dynamisch-grammatikspezifischer XML-Kompressor vorgestellt. Anders als bisherige Ansätze kombiniert dieser nicht mehrere Kompressionsstrategien miteinander – wie beispielsweise die Indizierung von Strings oder das Auslassen von End-Tags; vielmehr basiert er auf dem geschlossenen und einfach strukturierten Modell eines Kellerautomaten.

Der für die Kompression verwendete DKA wird dabei schrittweise aus einem XML-Schema-Dokument konstruiert – bei Web Services ist dieses in aller Regel in der WSDL-Beschreibung enthalten. Der erzeugte Automat beschreibt über seine Transitionen genau die möglichen Sequenzen von Tags und Zeichendaten, welche nach den Datentypdefinitionen aus der XML-Schema-Beschreibung zulässig sind.

Konstruktionsverfahren für Kellerautomaten zum Parsen von XML-Dokumenten sind aus der Literatur zwar bereits bekannt, völlig neu aber ist die Idee, mit ihrer Hilfe eine XML-Datenkompression durchzuführen: Dabei wird das zu komprimierende Dokument mit Hilfe des Automaten verarbeitet, also geparsed. Der Pfad durch den Automaten charakterisiert dabei in eindeutiger Weise die Tag-Sequenz im Dokument. Folglich reicht es zur vollständigen Repräsentation der Tags aus, diesen Pfad im komprimierten Datensatz zu codieren. Neben den Tags sind auch die Zeichendaten eines XML-Dokuments zu codieren. Dies geschieht über spezielle Simple-Type-Transitionen. Wird eine solche Transition ausgeführt, erzeugt diese eine optimierte Binärcodierung in Abhängigkeit vom vorliegenden Datentyp – zum Beispiel eine 32-Bit-Codierung für einen `xsd:int` Wert.

Vergleichende Messungen anhand typischer SOAP-Nachrichten zeigten, dass der Ansatz des Autors hinsichtlich der erreichten Kompressionsrate den existierenden Ansätzen deutlich überlegen ist. Die Vorteile sind bei sehr kleinen Datensätzen am größten – bei der `void doNothing()` Operation konnten die Nachrichtengrößen auf weniger als ein Hundertstel (!) reduziert werden. Bei größeren Datensätzen nehmen die Vorteile der automatenbasierten Kompression zwar immer mehr ab, dennoch waren die Ergebnisse in sämtlichen Messreihen stets besser als die der anderen untersuchten Kompressoren. Bei der Amazon-Messreihe wurde die Nachrichtengröße durchgehend auf etwa ein Zehntel reduziert.

Der hier vorgestellte Ansatz hebt sich – neben den sehr guten Kompressionsleistungen – auch in einem zweiten wesentlichen Punkt von anderen Arbeiten ab: Die Vorgänge der Kompression (auf Senderseite) bzw. der Dekompression (auf Empfängerseite) beschränken sich im Wesentlichen auf den Durchlauf eines Kellerautomaten. Die Struktur dieses Kompressors ist damit so überschaubar, dass eine Implementierung selbst auf sehr kleinen, ressourcenbeschränkten Mobilgeräten praktikabel wird. Lediglich für die Realisierung des Kellerspeichers wird RAM-Speicher benötigt – dieser Speicherplatzbedarf ist durch die Schachtelungstiefe des Eingabedokuments begrenzt. Da der zur Kompression bzw. Dekompression verwendete Automat gleichzeitig auch als Parser fungiert, sind hierfür keine zusätzlichen Verarbeitungsschritte erforderlich. Dies ist ein erheblicher Vorteil gegenüber dem mehrstufigen Differenzcodierungsansatz aus Kapitel 4.

Da die Kompression mittels Kellerautomaten sich für etliche praktische Anwendungen sehr gut eignet, hat die Universität zu Lübeck den Kompressionsansatz des Autors im November 2005 zum Patent angemeldet [156]. Weiterhin ist geplant, diesen Ansatz auch der W3C Efficient XML Interchange Working Group vorzulegen und ihn

ggf. in den zurzeit stattfindenden Standardisierungsprozess mit einzubringen. Es ist allerdings grundsätzlich fraglich, ob das W3C ein durch ein Patent geschütztes Kompressionsverfahren im Rahmen des Standardisierungsprozesses berücksichtigt (siehe [180]).

Obwohl die prototypische Implementierung dieses Ansatzes – genannt Xenia – noch nicht vollständig standardkonform arbeitet, ist sie bereits in der Lage, auch vergleichsweise komplexe SOAP-Dokumente fehlerfrei zu verarbeiten. Um den XML-Standards zu genügen, bedarf es vor allem verbesserter Routinen zur Typüberprüfung. Dies betrifft zum Beispiel die Überprüfung der Eindeutigkeit von ID-/IDREF-Bezeichnern oder auch die Einhaltung der Vorgaben bezüglich des erlaubten Wertebereichs von Zeichendaten.

Neben der Vervollständigung der Implementierung hat der Autor auch zusätzliche Routinen zur Erzeugung von Quellcode geplant: In der aktuellen Version von Xenia wird bei jedem Kompressionsvorgang neben dem zu komprimierenden Dokument eine XML-Schema-Datei eingelesen. Aus dieser wird der Automat als Java-Objekt im RAM-Speicher des Rechners erzeugt; dieser Automat wird direkt nach dem Kompressionsvorgang, d. h. bei Programmende, wieder verworfen. Bei leistungsstarken Rechnern wie PCs ist dieses Vorgehen völlig unproblematisch, da der Automat vor jedem Kompressionsvorgang sehr schnell neu erzeugt werden kann. Jedoch bei Kleinstcomputern sollte dieser Schritt entfallen, zumal er prinzipiell unnötig ist – schließlich ist die Schema-Beschreibung für einen Web Service statisch, so dass auch der Automat für die gesamte Lebensdauer eines Web Services konstant ist. Daher plant der Autor, den Automaten, der gegenwärtig nur als Java-Objekt im RAM-Speicher existiert, in Form von C-Code zu exportieren. Auf diese Weise kann ein kompakter Spezialkompressor erzeugt werden, der gleichzeitig als Parser dient. Er funktioniert auf jedem System, für das ein C-Compiler verfügbar ist.

Die konsequente Weiterentwicklung dieses Gedankens ist die Erzeugung spezialisierter Hardwarestrukturen: Neben C-Code könnte Xenia auch Code in einer Hardwarebeschreibungssprache wie VHDL oder Verilog generieren. Mit Hilfe entsprechender Synthesewerkzeuge ließen sich daraus dann spezialisierte Hardwarestrukturen zum Parsen und Komprimieren von XML erzeugen.

Zusammenfassend ist festzustellen, dass sich Kellerautomaten zur Kompression von XML-Dokumenten als besonders vorteilhaft erwiesen haben – und zwar nicht nur mit Blick auf praktische Anwendungen, sondern auch konzeptionell: Über die Anzahl möglicher Folgezustände beim Automatendurchlauf lässt sich eine direkte Beziehung zum Shannon'schen Entropiebegriff herstellen. Damit ist dieses Modell ganz bestimmt auch für weitere theoretische Forschungsarbeiten interessant.

Kapitel 6

Reduzierung des Overheads beim SOAP-Transport

In den Kapiteln 4 und 5 hat der Autor zwei Verfahren zur SOAP-Datenkompression vorgestellt. Wie wir gesehen haben, lassen sich damit – in Abhängigkeit von den spezifischen Eigenschaften eines Datensatzes – deutlich bessere Kompressionsraten erreichen als mit generischen oder nicht-grammatikspezifischen Kompressoren.

Bisher unberücksichtigt blieb jedoch die Tatsache, dass der Overhead, der durch die textorientierte XML-Darstellung verursacht wird, nicht der einzige Grund für das vergleichsweise hohe Datenaufkommen bei der SOAP-Kommunikation ist. Das Datenaufkommen wird zusätzlich durch die für die Übertragung der SOAP-Nachrichten eingesetzten Protokolle maßgeblich beeinflusst, und dieser Anteil am gesamten Datenaufkommen wird durch die SOAP-Kompression schließlich nicht reduziert. Daher hat der Autor seine Betrachtungen auf den gesamten bei der SOAP-Kommunikation anfallenden Datenverkehr im Netzwerk ausgedehnt.

Im Folgenden geht der Autor davon aus, dass der Leser mit den wesentlichen Konzepten und Funktionsweisen von Internet-Protokollen vertraut ist. Hierzu gehören insbesondere Kenntnisse über das OSI-Referenzmodell sowie über die elementaren Funktionsweisen der beiden Transportprotokolle *Transmission Control Protocol (TCP)* [122] und *User Datagram Protocol (UDP)* [120]. Eine gute Zusammenfassung dieser Grundlagen findet der Leser zum Beispiel in [145].

Betrachteten wir zunächst eine Messreihe, welche das Problem des Overheads beim SOAP-Nachrichtentransport evident macht: Bei den hierzu durchgeführten Messungen hat der Autor zwei Rechner über Ethernet direkt miteinander verbunden und das Datenaufkommen bei der SOAP-Kommunikation mit Hilfe des Netzwerkanalysewerkzeugs *Ethereal* [28] untersucht. Bei den Messungen wurde sämtlicher Datenverkehr oberhalb von Schicht 2 im OSI-Referenzmodell ausgewertet. Die *Maximum Transfer Unit (MTU)* wurde bei den Messungen auf 1.300 Bytes eingestellt. Dieser Wert gibt die maximale Größe eines Schicht-2-Frames an; von ihm hängt damit vor allem ab, auf wie viele IP-Datagramme die zu transportierenden Daten verteilt werden – je kleiner die MTU, desto mehr IP-Pakete werden für den Transport benötigt und je größer ist der damit verbundene Overhead.

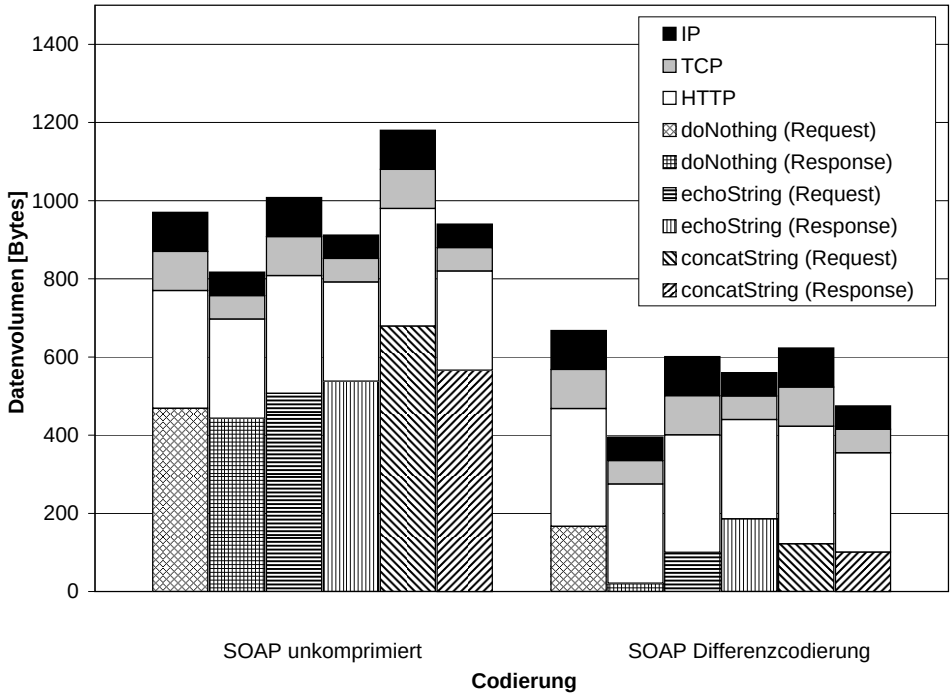


Abbildung 6.1: Datenaufkommen einschließlich Transport-Overhead in den einzelnen Protokollschichten

Abbildung 6.1 zeigt exemplarisch eine differenzierte Darstellung der beiden Messreihen „SOAP (unkomprimiert)“ und „Differenz (DUL, xmlppm)“ (vergleiche Abbildung 4.8 auf Seite 88). Die schraffierten Bereiche kennzeichnen die Größe der unkomprimierten bzw. komprimierten SOAP-Nachricht. Die weißen, grauen und schwarzen Bereiche kennzeichnen dagegen den Overhead, der durch die einzelnen zum Transport der SOAP-Nachricht eingesetzten Protokolle verursacht wird (HTTP, TCP und IP). Wie hier klar zu sehen ist, lässt sich durch die Kompression der SOAP-Nachrichten zwar die SOAP-Nachrichtengröße auf einen Bruchteil reduzieren, betrachtet man jedoch das Datenaufkommen insgesamt, d. h. einschließlich der anderen Protokolle, führt die Kompression der SOAP-Nachrichten nicht einmal zu einer Verringerung des Datenaufkommens um den Faktor zwei. Beim Einsatz von SOAP-Kompression verursacht der hier dargestellte Nachrichtentransport also ein Mehrfaches an Overhead im Vergleich zur eigentlichen Nachrichtengröße.

Diese Ergebnisse zeigen, dass der im Internet übliche Weg, SOAP-Nachrichten mit Hilfe von HTTP zu übertragen, für Anwendungen mit begrenzten Datenraten offensichtlich ungeeignet ist.

Wie bereits ausführlich in Kapitel 2 behandelt, ist der Transport über HTTP allerdings nicht der einzig mögliche Weg. Da der Web Service Technology Stack vollständig modular aufgebaut ist, kann SOAP prinzipiell an ein beliebiges Protokoll zum Nach-

richtentransport gebunden werden – wie bereits erläutert, bezeichnen wir ein solches Protokoll auch als *Binding*.

In nächsten Abschnitt gibt der Autor zunächst einen Überblick über die derzeit verfügbaren Bindings. Im Anschluss daran präsentiert er ausführliche Messungen zum Overhead dieser Bindings. Auf Basis dieser Ergebnisse entwickelt der Autor dann ein „minimales“ SOAP-Binding namens *PURE*, welches auf die besonderen Bedürfnisse für Web-Service-Anwendungen mit begrenzten Datenraten zugeschnitten ist. Nachdem er kurz einen Überblick über eine prototypische Implementierung dieses Ansatzes gegeben hat, vergleicht er diesen anhand weiterer Messungen mit verwandten Arbeiten. Abschließend fasst er seine Ergebnisse zusammen und gibt einen Ausblick auf mögliche Ansatzpunkte für weitere Forschungsarbeiten in diesem Bereich.

Die in diesem Kapitel dargelegten Ideen und Konzepte hat der Autor bereits in [160] und [161] in wesentlichen Teilen vorab veröffentlicht.

6.1 Überblick über existierende SOAP-Bindings

Ein bedeutsamer Vorteil der Web-Service-Technologie gegenüber älteren Ansätzen wie CORBA oder Java-RMI ist der modular austauschbare Mechanismus zum Nachrichtentransport, genannt *Transport Binding* oder kurz *Binding*.

Wie bereits in Abschnitt 2.2.2 erläutert, beeinflusst das verwendete Binding den Ablauf der SOAP-Kommunikation in entscheidender Weise. Das Binding muss (mindestens) ein so genanntes Kommunikationsmuster (*Message Exchange Pattern, MEP*) festlegen. Hierunter verstehen wir eine Spezifikation des zeitlichen Ablaufs des Kommunikationsprozesses, d. h. welche Netzwerknachrichten in welcher Reihenfolge und mit welchem Inhalt zwischen den Kommunikationspartnern ausgetauscht werden.

Allerdings wird bei heutigen Web-Service-Anwendungen so gut wie kein Gebrauch von der Modularität des Binding-Mechanismus gemacht. In nahezu allen Fällen kommt das HTTP-Binding mit dem Request-Response-MEP zum Einsatz, welches wir schon ausführlich in Abschnitt 2.2.2 betrachtet haben. Es gibt nur wenige, zumeist experimentelle Bindings neben HTTP. Im Folgenden gibt der Autor einen Überblick über solche Ansätze und erläutert jeweils die spezifischen Vor- und Nachteile.

6.1.1 E-Mail

Die *W3C XML Protocol Working Group* hat ein SOAP-über-E-Mail-Binding spezifiziert [100]. Dies liegt allerdings noch nicht als fertiger Standard vor, sondern hat den Status einer *Working Group Note*.

Anders als HTTP folgt die Kommunikation über E-Mail grundsätzlich einem asynchronen Muster: Eine E-Mail wird vom Absender zum Empfänger übermittelt; ob

der Empfänger dieser E-Mail antwortet, bleibt zunächst offen. Das Senden der Antwortnachricht ist hier also ein eigenständiger Kommunikationsvorgang – dies ist ein wesentlicher Unterschied zum Client/Server-Ansatz, bei dem Request- und Response-Nachricht stets eine Einheit bilden.

Bei vielen SOAP-Anwendungen – insbesondere solchen zur Realisierung von RPC-Aufrufen – ist es allerdings erforderlich, eine eindeutige Zuordnung zwischen Request- und Response-Nachrichten zu treffen. Diese Zuordnung kann entweder anhand des Feldes `Message-ID` im E-Mail-Header [125] erfolgen, oder aber es kommt die SOAP-Erweiterung *WS-Addressing (WSA)* [185] zum Einsatz. Diese Erweiterung implementiert einen leistungsfähigen Adressierungsmechanismus direkt im SOAP-Header, so dass hier die Adressierung weitestgehend unabhängig vom verwendeten Binding erfolgen kann.

Ein weiterer Unterschied zu HTTP besteht darin, dass bei einem SOAP-Transport über E-Mail ein Punkt-zu-Mehrpunkt-Kommunikationsmuster möglich ist; hierzu genügt es, mehrere Empfänger im Header einer E-Mail zu spezifizieren – typischerweise über die mehrfache Verwendung des Header-Feldes `To:`.

Eine Besonderheit des SOAP-über-E-Mail-Bindings ist, dass es völlig unabhängig von einem konkreten Protokoll zum Nachrichtentransport formuliert ist. Für den Transport von E-Mail sind mehrere Protokolle gängig. Im Folgenden beschränken wir uns auf das SMTP-Protokoll [77], da es für das in Abschnitt 6.2 betrachtete RPC-Szenario die sinnvollste Lösung darstellt; schließlich fordert das RPC-Paradigma eine enge zeitliche Koppelung zwischen Request und Response (synchroner Nachrichtenaustausch). Für andere Anwendungsfälle könnten allerdings auch Protokolle zum asynchronen E-Mail-Austausch wie POP [102] oder IMAP [32] durchaus interessant sein.

Da SMTP auf TCP aufsetzt, bietet dieses Binding grundsätzlich einen zuverlässigen Nachrichtentransport und auch Mechanismen zur Datenflusssteuerung und Staukontrolle im Sinne von TCP.

6.1.2 FTP

Obwohl das *File Transfer Protocol (FTP)* in der Literatur häufig als Binding-Alternative zu HTTP aufgeführt wird, gibt es bis heute weder eine Spezifikation noch eine Implementierung eines solchen Bindings.

Da die Möglichkeit eines FTP-Bindings aber mehrfach in der W3C-Spezifikation der Web-Service-Architektur [181] erwähnt wird, hat der Autor dieses Protokoll dennoch in den Vergleich des Overheads der einzelnen Protokolle mit einbezogen.

Auch bei FTP gibt es keine eindeutige Zuordnung zwischen Request- und Response-Nachrichten, denn die Vorgänge des Sendens und Empfangens von Nachrichten sind prinzipiell unabhängig voneinander (jeweils unidirektionale Kommunikation über das

`PUT` Kommando). Über die Verwendung von WS-Addressing lässt sich aber auch hier eine Zuordnung zwischen Request- und Response-Nachrichten erreichen.

Auch FTP basiert auf TCP, und folglich ergibt sich auch hier ein zuverlässiger Nachrichtentransport im Sinne von TCP.

6.1.3 Microsoft Message Queuing (MSMQ)

Microsoft Message Queuing (MSMQ) ist ein Mechanismus zum zuverlässigen und asynchronen Nachrichtenaustausch. Die Asynchronität wird dabei über einen Store-And-Forward-Mechanismus erreicht: Es gibt im Netzwerk einen oder mehrere Rechner, welche eingehende Nachrichten in einer Warteschlange zwischenspeichern, so genannte *Queuing-Server*. Sobald der Empfänger einer Nachricht erreichbar ist, wird die Nachricht ausgeliefert (vergleiche Abbildung 6.2). Ein besonderer Vorteil dieses Ansatzes ist also, dass der Empfänger zum Zeitpunkt des Sendens einer Nachricht nicht erreichbar sein muss.

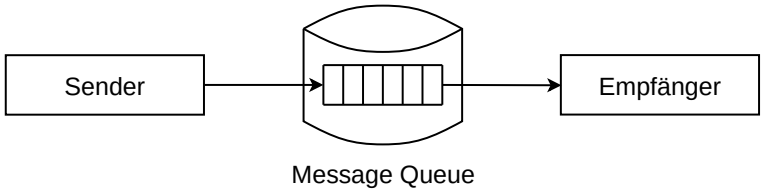


Abbildung 6.2: Sender und Empfänger kommunizieren asynchron über eine Message Queue

Das in Abbildung 6.2 dargestellte Beispiel zeigt eine herkömmliche Punkt-zu-Punkt-Kommunikation über eine Message Queue. Mit Hilfe von Message Queuing sind aber auch noch andere Kommunikationsformen möglich. Beispielsweise kann sich ein Empfänger für Nachrichten zu bestimmten Themengebieten registrieren. Der Queuing-Server leitet dann Nachrichten, die sich einem Themengebiet zuordnen lassen, automatisch an die Rechner weiter, die sich hierfür registriert haben (vergleiche Abbildung 6.3). Diese Kommunikationsform wird auch als Publisher/Subscriber-Modell bezeichnet.

Das MSMQ-System wurde als universeller Nachrichtendienst für Unternehmensnetzwerke entwickelt. Es eignet sich aber insbesondere auch zum Austausch von SOAP-Nachrichten. Ähnlich wie die bisher betrachteten Bindings ist die Kommunikation grundsätzlich unidirektional. Mit Hilfe von WS-Addressing lässt sich aber auch hier ein Request-Response-MEP realisieren. Weiterhin ist es möglich, eine Nachricht an mehrere Empfänger zu versenden.

Da auch MSMQ auf TCP aufsetzt, ergibt sich auch die damit einhergehende Zuverlässigkeit. Zusätzlich werden bei MSMQ eine Reihe weiterer Techniken eingesetzt, um

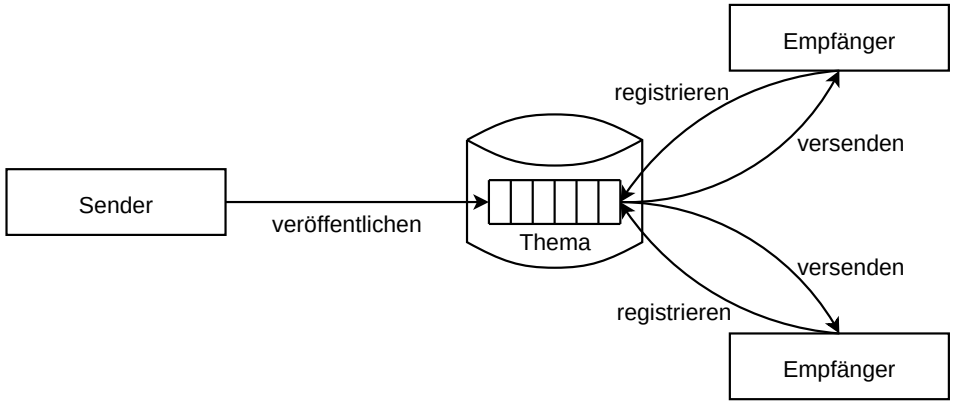


Abbildung 6.3: Kommunikation nach dem Publisher/Subscriber-Modell

die Zuverlässigkeit der Kommunikation – selbst beim Auftreten von Hard- oder Softwarefehlern – zu gewährleisten. Hierzu gehört insbesondere das Transaktionskonzept, welches auch häufig bei Datenbanksystemen eingesetzt wird. Dieses gesteigerte Maß an Zuverlässigkeit ist einer der Hauptvorteile gegenüber der Kommunikation über E-Mail.

6.1.4 TCP

Ein SOAP-über-TCP-Binding wurde von Microsoft spezifiziert. Eine entsprechende Implementierung ist in der Funktionsbibliothek *Web Service Enhancements (WSE) 2.0* [94] enthalten. Die grundlegende Idee bei diesem Binding besteht darin, das Schicht-7-Protokoll komplett aus dem Kommunikationsprozess zu entfernen und stattdessen die SOAP-Nachrichten direkt über eine TCP-Socket-Verbindung zu versenden.

TCP stellt allerdings nicht die für die SOAP-Kommunikation erforderlichen Adressierungsmechanismen bereit: TCP allein ist nämlich nur in der Lage, eine Applikation (bei SOAP-Anwendungen wäre dies die Web-Service-Engine) zu adressieren. Eine Web-Service-Engine kann aber durchaus mehrere Web Services bereitstellen, und folglich müssen diese beim Nachrichtentransport separat adressiert werden. Somit ist die Verwendung der SOAP-Erweiterung WS-Addressing hier zwingend erforderlich; über sie werden die fehlenden Adressierungsmöglichkeiten direkt im SOAP-Header realisiert.

WSE 2.0 implementiert zwei Varianten des TCP-Bindings: eine synchrone und eine asynchrone. Bei der synchronen Variante wird eine Request-Nachricht zusammen mit ihrer korrespondierenden Response-Nachricht über eine gemeinsame TCP-Verbindung übertragen. Bei der asynchronen Variante hingegen wird pro Richtung eine separate TCP-Verbindung aufgebaut, welche direkt nach der jeweiligen Nachrichtenübertragung wieder abgebaut wird. Die asynchrone Variante bietet den Vor-

teil, dass hier beide Kommunikationsvorgänge voneinander entkoppelt werden. Dauert beispielsweise eine Web-Service-Operation sehr lange, so könnte es bei der synchronen Variante vorkommen, dass die TCP-Verbindung abbricht, noch bevor die Response-Nachricht übertragen wurde. Das Berechnungsergebnis wäre in diesem Fall verloren. Bei der asynchronen Variante könnte der Web Service zu einem späteren Zeitpunkt die Verbindung aufbauen, um dann das Ergebnis zum Dienstnutzer zu übertragen. Allerdings wird diese gesteigerte Flexibilität auch durch einen etwas größeren Protokoll-Overhead erkauft, denn bei der asynchronen Variante müssen zwei TCP-Verbindungen auf- und wieder abgebaut werden.

6.1.5 UDP

Gudgin et al. haben in [52] ein SOAP-über-UDP-Binding vorgestellt. Es unterstützt den unidirektionalen Nachrichtenaustausch über IP-Unicast, -Multicast und auch -Broadcast. Bei Verwendung von IP-Multicast oder -Broadcast wird die Punkt-zu-Mehrpunkt-Nachrichtenübertragung direkt auf IP-Ebene realisiert. Dies führt im Vergleich zu den Punkt-zu-Mehrpunkt-Kommunikationsmustern von MSMQ oder E-Mail zu deutlich weniger Overhead.

Ein gravierender Nachteil dieses Bindings ist jedoch die damit einhergehende Beschränkung der Nachrichtengröße auf ca. 64 kBytes. Größere SOAP-Nachrichten passen nicht in ein UDP-Datagramm und können folglich nicht übertragen werden.

Anders als sämtliche TCP-basierten Ansätze stellt UDP keine Mechanismen für Datenfluss- oder Staukontrolle bereit. Weiterhin ist UDP ein unzuverlässiges Protokoll – das bedeutet, dass gesendete Nachrichten auf dem Weg zum Empfänger verloren gehen können, ohne dass der Absender darüber informiert wird.

6.1.6 Andere Ansätze

Neben den zuvor vorgestellten alternativen SOAP-Bindings existieren noch zwei weitere Binding-Spezifikationen: SOAP-über-JMS [38] und SOAP-über-BEEP [118].

Leider war es dem Autor nicht möglich, diese Bindings bei seinen Vergleichsmessungen mit zu berücksichtigen, da keine geeigneten Implementierungen zur Verfügung standen. Allerdings ist es im Blick auf den Protokoll-Overhead – allein bei Betrachtung des jeweils zu Grunde liegenden Konzepts – ohnehin nicht möglich, dass diese Protokolle effizienter als das oben betrachtete TCP-Binding funktionieren, denn beide Protokolle setzen auf TCP auf und fügen hier zusätzliche Header-Informationen hinzu. Folglich muss das Datenaufkommen hier größer sein als beim TCP-Binding.

Bei *Java Messaging Service (JMS)* [141] handelt es sich um einen Message-Queuing-Dienst für Java, der im Bezug auf Funktionsumfang und technische Realisierung MSMQ sehr ähnlich ist. Das *Block Extensible Exchange Protocol (BEEP)* [128] ist

ein Anwendungsprotokoll, welches zwar prinzipiell mit HTTP vergleichbar ist, aber einen deutlich größeren Funktionsumfang aufweist. Über benutzerdefinierte Profile können auch neue, benutzerspezifische Protokoll-Varianten von BEEP realisiert werden. Daher wird BEEP oft auch als ein Grundgerüst (*framework*) bezeichnet, auf dessen Basis ein Entwickler eigene Anwendungsprotokolle entwerfen kann.

6.2 Untersuchungen zum Overhead

Um die Effizienz bezüglich des verursachten Protokoll-Overheads zu messen, hat der Autor die zur Verfügung stehenden Bindings anhand von Messreihen miteinander verglichen. In diesem Abschnitt wird zunächst der Versuchsaufbau beschrieben. Im Anschluss daran interpretiert der Autor die Messergebnisse.

6.2.1 Versuchsaufbau und -durchführung

Der von einem Transport-Binding verursachte Overhead variiert mit der Nachrichtengröße, denn je größer eine Nachricht ist, desto mehr Datagramme (und damit auch mehr Header) werden für den Transport benötigt. Um nun den von den einzelnen Bindings verursachten Protokoll-Overhead zu messen, hat der Autor einen Test-Web-Service implementiert, der zwei RPC-Operationen zur Verfügung stellt: `void doNothing()` und `byte[] getImage(String in0)`. Die erste Operation erzeugt sehr kleine SOAP-Nachrichten mit 1.128 Bytes für die Request- und 1.021 Bytes für die Response-Nachricht. Die zweite liefert als Rückgabewert ein JPG-Bild in Form eines großen Byte-Arrays. Der Aufrufparameter `in0` bezeichnet den Dateinamen des Bildes. Die Größe für eine Request-Nachricht beträgt hier 1.161 Bytes, die der Response-Nachricht 143.501 Bytes. Damit ist die Response-Nachricht größer als die zulässige Größe eines IP-Datagramms (64 kBytes). Folglich müssen alle Bindings diese Response-Nachricht fragmentieren.

Bei allen Messungen kam die SOAP-Engine der Microsoft .NET-Plattform [95] zum Einsatz, denn für diese existieren bereits mehrere Implementierungen für alternative Bindings: Bei HTTP und TCP hat der Autor die Implementierungen aus der WSE 2.0 Erweiterung verwendet [94]. Von den zwei zur Verfügung stehenden Varianten beim TCP-Binding hat er die synchrone für die Messungen genutzt, da diese für die hier untersuchten RPC-Operationen am zweckmäßigsten ist (RPC-Paradigma erfordert synchrone Kommunikation). Bei HTTP kam die HTTP-POST-Variante zum Einsatz. Für SMTP, MSMQ und UDP hat der Autor die Implementierungen [89, 76, 163] verwendet. Wegen der Beschränkung der Nachrichtengröße auf etwa 64 kBytes konnte das UDP-Binding allerdings nicht mit der Test-Operation `byte[] getImage(String in0)` verwendet werden – eine Messung war hier also nicht möglich. Da es noch keine SOAP-über-FTP-Implementierung gibt, hat der Autor den Overhead hier auf andere Weise ermittelt. Dazu hat er die einzelnen Request- und Response-Nachrichten der

Test-Operationen zunächst in Dateien abgespeichert. Diese wurden dann mittels eines FTP-Clients zu einem FTP-Server übertragen (PUT-Kommando). Dabei wurde jeweils das Datenaufkommen gemessen.

Eine bei den Vergleichsmessungen wichtige Einflussgröße ist auch der gewählte Adressierungsmechanismus. Wie bereits in Abschnitt 6.1 erläutert, verfügen die Protokolle TCP und UDP für Web-Service-Anwendungen nicht über ausreichende Adressierungsmöglichkeiten und müssen deshalb auf zusätzliche Mechanismen zurückgreifen. Das W3C hat hierfür den Standard WS-Addressing erarbeitet, welcher die speziellen Anforderungen der SOAP-Kommunikation besonders gut unterstützt und sämtliche Adressinformationen direkt im SOAP-Header unterbringt. Zwar bieten Schicht-7-Protokolle wie HTTP bereits geeignete Adressierungsmöglichkeiten an, doch auch hier entschied sich der Autor dafür, die Adressierung über den WS-Addressing-Header zu realisieren. Auf diese Weise werden alle untersuchten Bindings mit denselben Adressierungsmöglichkeiten ausgestattet. Ein WS-Addressing-Eintrag im SOAP-Header hat dabei eine Größe von rund 400 Bytes.

Wie bereits zu Beginn dieses Kapitels erläutert, beeinflusst auch der eingestellte MTU-Wert das Datenaufkommen. Er wurde bei allen Messungen auf 1.300 Bytes gesetzt. Müssen IP-Datagramme transportiert werden, die nicht in ein Schicht-2-Frame dieser Größe passen, so werden sie von der IP-Implementierung auf Senderseite automatisch fragmentiert, d. h. auf mehrere Schicht-2-Frames aufgeteilt. Beim Empfänger sorgt die dortige IP-Implementierung dann dafür, dass diese Fragmentierung rückgängig gemacht wird. Detaillierte Ausführungen zur IP-Fragmentierung und deren Auswirkungen auf praktische Anwendungen findet der Leser in [136].

6.2.2 Auswertung

Tabelle 6.1 zeigt die Ergebnisse der Messungen für beide Test-Operationen, jeweils kumuliert für Request und Response.

Aus den Werten ergibt sich, dass der Transport von SOAP-Nachrichten über MSMQ, SMTP und auch FTP mit Blick auf den verursachten Overhead deutlich ineffizienter ist als der herkömmliche SOAP-Transport über HTTP-POST – und zwar sowohl bei `void doNothing()` als auch `byte[] getImage(String in0)`. Diese Ansätze sind somit für Web-Service-Anwendungen mit begrenzten Datenraten offenbar unzweckmäßig.

Es sei angemerkt, dass bei den hier gezeigten Messwerten für SMTP und FTP die in diesen Protokollen vorgesehenen Klartextmeldungen eingeschaltet waren. Allerdings sind diese Klartextmeldungen optional, und bei den meisten Server-Implementierungen lassen sie sich abschalten. Der Autor hat daher zusätzliche Messreihen durchgeführt, bei denen diese Klartextmeldungen abgeschaltet waren. Dies führte zwar jeweils zu rund 60% geringeren Werten für den Overhead auf Anwendungsschicht, auf Transport- und Vermittlungsschicht blieben die Werte allerdings nahezu unver-

Operation	`void doNothing()`					
Binding	HTTP	SMTP	FTP	MSMQ	TCP	UDP
Anwendungsschicht	560	2.535	576	2.959	-	-
Transportschicht	276	992	1.301	493	538	16
Vermittlungsschicht	260	960	1.300	480	320	40
Gesamt	1.096	4.487	3.177	3.932	858	56

Operation	`byte[] getImage(String in0)`					
Binding	HTTP	SMTP	FTP	MSMQ	TCP	UDP
Anwendungsschicht	558	2.581	577	2,996	-	-
Transportschicht	3.716	4.792	4.744	3.855	4.052	-
Vermittlungsschicht	3.700	4.760	4.680	3.880	3.820	-
Gesamt	7.974	12.133	10.001	10.731	7.872	-

Tabelle 6.1: Protokoll-Overhead verschiedener SOAP-Bindings, alle Größenangaben in Bytes

ändert. Somit schneiden FTP und SMTP auch bei abgeschalteten Klartextmeldungen noch immer deutlich schlechter ab als HTTP.

Das TCP-Binding schneidet erwartungsgemäß besser als HTTP ab, weil hier der Overhead auf Anwendungsschicht entfällt. Unerwartet hingegen ist die Tatsache, dass der Overhead auf Transport- und Vermittlungsschicht etwas größer ausfällt als bei HTTP. Dies dürfte eigentlich nicht sein, da bei TCP wegen des Wegfalls des HTTP-Headers insgesamt weniger Daten transportiert werden müssen.

Der Autor hat daher den Inhalt der ausgetauschten TCP-Segmente genauer untersucht und dabei festgestellt, dass die untersuchte TCP-Binding-Implementierung bei der Übertragung der Request- und Response-Nachrichten jeweils zwei zusätzliche TCP-Segmente übermittelt. Das erste enthält einen 100 Bytes großen Binärwert und das zweite einen einzelnen Bytewert. Da Microsoft die genaue Funktionsweise des TCP-Bindings nicht offen gelegt hat, lässt sich der Zweck dieser Werte nur vermuten: Anscheinend dient der 100 Bytes große Wert als zusätzliche Message-ID, und der einzelne Bytewert könnte genutzt werden, um die Übertragung einer Nachricht in mehreren Blöcken zu kennzeichnen – ähnlich der so genannten *chunked* Übertragung bei HTTP 1.1 [44].

Die Tabelleneinträge für `void doNothing()` zeigen deutlich, dass das UDP-Binding mit Abstand den geringsten Overhead verursacht. Allerdings ist es mit zwei wesentlichen konzeptionellen Nachteilen behaftet: Zum einen können hiermit nur solche SOAP-Nachrichten übertragen werden, die kleiner als 64 kBytes sind, und zum anderen ist der Transport unzuverlässig, so dass SOAP-Nachrichten verloren gehen können.

6.3 PURE: ein minimales SOAP-Binding

Der Vergleich der einzelnen SOAP-Bindings zeigte, dass der Transport von SOAP-Nachrichten über UDP in zweierlei Hinsicht viel versprechend ist. Zum einen ist der durch UDP verursachte Overhead erheblich geringer als bei allen anderen Bindings. Zum anderen ist UDP besonders flexibel einsetzbar, denn anders als HTTP, SMTP und auch FTP ist es kein Client/Server-Protokoll. Stattdessen sind hier Nachrichtensender und -empfänger gleichberechtigt. So können zwei Web Services untereinander kommunizieren, ohne zuvor aushandeln zu müssen, welcher Web Service als Client bzw. Server fungiert – in der Literatur wird diese Kommunikationsform mitunter auch als *Peer-To-Peer (P2P)* bezeichnet.

Wie bereits im vorhergehenden Abschnitt erläutert, bringt der Einsatz von UDP zum SOAP-Nachrichtentransport jedoch auch zwei wesentliche Nachteile mit sich – die Limitierung der Nachrichtengröße sowie die fehlende Zuverlässigkeit der Übertragung. Die zweite Einschränkung könnte über eine SOAP-Erweiterung wie WS-ReliableMessaging [109] kompensiert werden. Besonders kritisch ist daher die Begrenzung der Nachrichtengröße.

Wie bereits im Kapitel 4 dargestellt, ist es zwar durchaus ein typisches Merkmal der SOAP-Kommunikation, dass die ausgetauschten Nachrichten eher klein sind, allerdings ist es bei vielen Anwendungen nicht möglich, eine maximale Nachrichtengröße vorherzusagen: Zum Beispiel liefert der Amazon-Web-Service, den wir in Kapitel 5 kennen gelernt haben, bei einigen Anfragen Datensätze, die deutlich größer als 64 kBytes sind. Der Web-Service-Entwickler müsste also für die Einhaltung dieser Limitierung sorgen.

Vor allem stellt eine solche Begrenzung der Nachrichtengröße aber ein Problem bei Anwendungen mit so genannten SOAP-Zwischenknoten (*Intermediaries*) dar: Hier werden die SOAP-Nachrichten nicht direkt vom Absender zum Empfänger geschickt, sondern über einen oder mehrere SOAP-Zwischenknoten geleitet. Diese Zwischenknoten realisieren zusätzliche Funktionalitäten – typische Anwendungsbeispiele wären etwa die Nachrichtenverschlüsselung oder die Überprüfung von digitalen Signaturen. Bei der Weiterleitung einer Nachricht an den nächsten SOAP-Zwischenknoten kann prinzipiell ein beliebiges Transport-Binding zum Einsatz kommen. Ein Web-Service-Entwickler hat daher mitunter gar keine Kenntnis darüber, dass irgendwo auf dem Transportweg ein Binding zum Einsatz kommt, welches nur Nachrichten bis zu einer bestimmten Größe transportieren kann. Somit stellt eine Begrenzung der Nachrichtengröße bei praktischen SOAP-Anwendungen durchaus ein Problem dar.

Um diese Begrenzung der Nachrichtengröße beim SOAP-Transport über UDP zu überwinden, hat der Autor ein sehr leichtgewichtiges Anwendungsprotokoll namens PURE entworfen, welches auf UDP aufsetzt. Der Name PURE ist kein Akronym, sondern soll lediglich ausdrücken, dass sich der Funktionsumfang dieses Protokolls auf das Notwendigste beschränkt. Die Kernidee bei der Entwicklung von PURE war, nur

die grundlegendsten Funktionalitäten im Transport-Binding zu implementieren; bei Bedarf können Zusatzfunktionen über den SOAP-Header realisiert werden. Hierzu gehört insbesondere die Verwendung der bereits standardisierten SOAP-Erweiterungen WS-Addressing [185] und WS-ReliableMessaging [109].

Sicherlich könnte man argumentieren, dass wir damit den Overhead nur aus dem Transport-Binding in den SOAP-Header verlagern. Allerdings ist die Verwendung von SOAP-Erweiterungen wie WS-Addressing in vielen Anwendungsbereichen ohnehin zwingend erforderlich oder zumindest zweckmäßig [34]. Damit tritt der Overhead, welcher durch SOAP-Erweiterungen verursacht wird, ohnehin auf – und zwar unabhängig vom verwendeten Transport-Binding. Zudem haben wir bereits zwei leistungsstarke Kompressionsansätze für SOAP-Nachrichten kennen gelernt – auch unter diesem Aspekt erscheint die Auslagerung von Zusatzfunktionalitäten in den SOAP-Header vorteilhaft.

Dennoch ist der durchgängige Einsatz von SOAP-Erweiterungen sicherlich nicht bei allen Anwendungen zweckmäßig. Zielstellung dieser Untersuchung ist schließlich die Etablierung der SOAP-Kommunikation in Anwendungsbereichen, bei denen die zur Verfügung stehenden Ressourcen beschränkt sind – beispielsweise in Sensornetzwerken oder auch bei Ubiquitous-Computing-Anwendungen. Wegen der eingeschränkten Speicher- und Rechenleistung der hier eingesetzten Geräte ist es sicherlich nicht immer angezeigt, eine komplexe SOAP-Erweiterung wie WS-ReliableMessaging zu nutzen. Daher hat der Autor bei der Entwicklung von PURE auch grundlegende Mechanismen vorgesehen, die einen zuverlässigen Nachrichtentransport gewährleisten.

6.3.1 Aufbau des Headers

PURE ist ein Anwendungsprotokoll, welches auf UDP aufsetzt. Im Gegensatz zu der Implementierung von Gudgin et al. wird die SOAP-Nachricht hier nicht direkt in das Data-Feld eines UDP-Datagramms eingefügt; stattdessen wird bei PURE ein fünf Bytes großer Header vor der eigentlichen Nachricht eingefügt. Über diesen stellt PURE vier Basisdienste bereit:

- Nachrichtenfragmentierung
- Duplikaterkennung
- negative Bestätigungsnachrichten in Verbindung mit einer automatischen Wiederholung des Sendevorgangs
- optional: positive Bestätigungsnachrichten

Die Struktur des PURE-Headers ist in Abbildung 6.4 dargestellt. Das erste Oktett ist das Feld *Flags*. Wie die Abbildung zeigt, wird dieses Feld als Menge von Bit-Flags

interpretiert. In der gegenwärtigen Protokollversion werden nur die ersten vier Flags ausgewertet, alle übrigen sind für künftige Anwendungen reserviert.

Das Feld *SOAP ID* enthält einen 16 Bit langen Unsigned-Integer-Wert, welcher dazu benutzt wird, die einzelnen Fragmente einer SOAP-Nachricht einander zuzuordnen: Bevor eine PURE-Implementierung eine SOAP-Nachricht fragmentiert und verschickt, wird dieser Nachricht eine *SOAP ID* zugeordnet. Dieser Wert wird bei der Übertragung eines jeden Nachrichtenfragments im Feld *SOAP ID* mitgeschickt. Anhand dieses Wertes kann die empfangende PURE-Implementierung die einzelnen Fragmente einander zuordnen. Die sendende PURE-Implementierung erzeugt die *SOAP ID* Werte für aufeinander folgende SOAP-Nachrichten sequentiell, d. h. durch Inkrementieren eines Zählers.

Das Feld *Fragment* enthält ebenfalls einen 16 Bit langen Unsigned-Integer-Wert; er dient der sequentiellen Nummerierung zusammengehöriger Nachrichtenfragmente. Eine solche Kennzeichnung ist notwendig, denn schließlich garantiert UDP nicht, dass die einzelnen Datagramme in derselben Reihenfolge empfangen werden, wie sie versendet wurden. Die empfangende PURE-Implementierung kann anhand der *Fragment* Werte also feststellen, ob die einzelnen Fragmente in der richtigen Reihenfolge eingetroffen sind. Bei nicht korrekter Empfangsreihenfolge kann sie die Fragmente entsprechend umsortieren, bevor die Nachricht wieder zusammengesetzt und schließlich an die SOAP-Engine weitergeleitet wird. Weiterhin dient dieses Feld der Duplikaterkennung: Empfängt eine PURE-Implementierung innerhalb eines festgelegten Zeitintervalls mehrere Fragmente mit gleichen Werten für *SOAP ID* und *Fragment*, so werden diese als Duplikate erkannt und verworfen.

Schließlich wird im Feld *Data* die SOAP-Nachricht bzw. ein Fragment einer SOAP-Nachricht als Payload eingefügt.

Eine PURE-Nachricht kann inklusive Header maximal 65.507 Bytes groß sein. Dieser Wert ergibt sich wie folgt: 65.535 Bytes ist die maximale Größe eines IP-Pakets

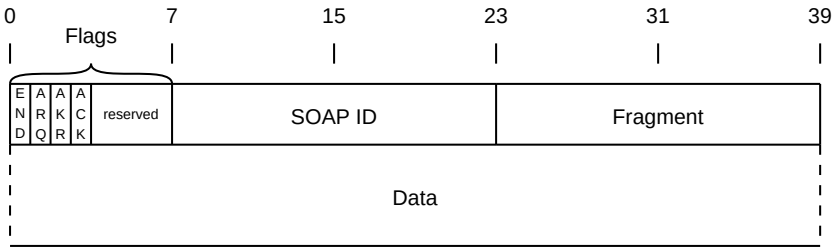


Abbildung 6.4: Schematische Darstellung des PURE-Nachrichtenformats, Bedeutung der Flag-Felder: „Letztes Fragment (*END*)“, „Automatic Repeat Request (*ARQ*)“, „Acknowledgement Requested (*AKR*)“ und „Reception Acknowledged (*ACK*)“

[121], mindestens 20 Bytes werden für den IP-Header und weitere acht Bytes für den UDP-Header benötigt. Da der MTU-Wert bei gängigen Netzwerktechnologien zwischen 1.200 und 1.500 Bytes liegt, würde eine solch große PURE-Nachricht allerdings zu erheblicher IP-Fragmentierung führen. Dies ist im fehlerfreien Fall zwar grundsätzlich unproblematisch, gehen jedoch einzelne IP-Fragmente verloren, kann die empfangende IP-Implementierung das Datagramm nicht wieder herstellen und verwirft es als Ganzes. Muss also aufgrund der Netzwerkinfrastruktur (z. B. in einem drahtlosen Netzwerk) damit gerechnet werden, dass einzelne IP-Fragmente verloren gehen, ist es ratsam, die Größe einer PURE-Nachricht auf den Wert MTU minus 28 Bytes (für IP- und UDP-Header) zu begrenzen. Auf diese Weise wird die IP-Fragmentierung vermieden.

Es sei angemerkt, dass der PURE-Header keine Prüfsumme vorsieht, um Übertragungsfehler zu erkennen. Eine solche Prüfsumme ist auch entbehrlich, denn dieser Mechanismus wird bereits im UDP-Header realisiert. Die darin enthaltene Prüfsumme wird über den gesamten Inhalt eines UDP-Datagramms berechnet und schließt folglich die PURE-Nachricht inklusive Header mit ein.

Im Folgenden stellt der Autor nun die Funktionsweise der einzelnen PURE-Header-Felder vor und zeigt jeweils ein Beispiel zur Verdeutlichung.

6.3.2 Nachrichtenfragmentierung

Um die UDP-bedingte Begrenzung der Nachrichtengröße auf 64 kBytes zu umgehen, unterstützt PURE die Fragmentierung von SOAP-Nachrichten. Eine SOAP-Nachricht wird hierbei vom Sender in mehrere Fragmente unterteilt, welche dann in mehreren UDP-Datagrammen verschickt werden.

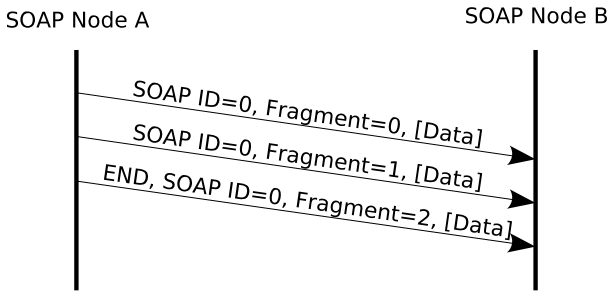


Abbildung 6.5: Nachrichtenfragmentierung

Abbildung 6.5 zeigt ein Beispiel für Nachrichtenfragmentierung. SOAP Node A möchte eine große SOAP-Nachricht an Node B senden. Hierzu unterteilt er zunächst die Nachricht in drei Fragmente. Jedes muss dabei – inklusive PURE-Header – kleiner als 65.507 Bytes sein.

Alle drei Fragmente tragen denselben *SOAP ID* Wert (im Beispiel 0). Hierüber wird ausgedrückt, dass die drei Fragmente zur selben SOAP-Nachricht gehören. Der Wert *Fragment* wird – bei 0 beginnend – mit jedem gesendeten Fragment inkrementiert. So kann der Empfänger die einzelnen Fragmente auch dann wieder in die richtige Reihenfolge bringen, wenn diese in einer anderen Sequenz empfangen werden. Schließlich wird über das *END* Flag im dritten Fragment angezeigt, dass dies das letzte Fragment der SOAP Nachricht mit der *SOAP ID* 0 ist. Die empfangende PURE-Implementierung kann bei Empfang der dritten Nachricht also sicher sein, dass sie alle Fragmente empfangen hat. Sie kann nun die zusammengesetzte SOAP-Nachricht an die SOAP-Engine weiterleiten.

In der gegenwärtigen Protokollversion ist *Fragment* ein 16-Bit-Wert und das Feld *Data* kann wegen der Größenbeschränkung von UDP höchstens 65.502 Bytes aufnehmen. Dies ergibt eine maximal mögliche SOAP-Nachrichtengröße von $2^{16} \cdot 65.502$ Bytes $\approx$ 4 Gigabytes.

Obwohl damit auch PURE eine inhärente Beschränkung der Nachrichtengröße mit sich bringt, dürfte diese bei praktischen Anwendungen zu keinen Komplikationen führen. Sollten in Zukunft tatsächlich Anwendungen existieren, die potentiell größere SOAP-Nachrichten austauschen, muss die Bitbreite des *Fragment* Feldes entsprechend vergrößert werden.

6.3.3 Negative Bestätigungsnachrichten und Duplikaterkennung

Da UDP nicht garantiert, dass ein gesendetes Datagramm auch beim Empfänger ankommt, kann es vorkommen, dass PURE-Nachrichten beim Transport verloren gehen – eine typische Ursache hierfür wäre beispielsweise ein überlasteter Router, der eingehende Datagramme verwirft.

Daher implementiert PURE eine Erkennung von verlorenen Nachrichten beim Empfang und fordert diese automatisch erneut an. Dieser Mechanismus heißt *Automatic Repeat Request (ARQ)*.

Für jede *SOAP ID* stellt PURE einen Empfangs-Timer bereit. Dieser wird immer dann zurückgesetzt, wenn ein Fragment mit der zugehörigen *SOAP ID* eintrifft. Läuft dieser Timer ab und wurden noch nicht alle Fragmente zu dieser *SOAP ID* empfangen, geht die PURE-Implementierung von einem verlorenen Fragment aus. Sie sendet eine PURE-Nachricht mit gesetztem *ARQ* Flag. Der Wert *SOAP ID* in dieser ARQ-Nachricht entspricht dem der noch ausstehenden Fragmente. Die noch fehlenden Fragmentnummern werden im Feld *Data* übertragen – auf diese Weise ist es möglich, gleich mehrere Nachrichten auf einmal erneut anzufordern. Das *Fragment* Feld einer ARQ-Nachricht trägt immer den Wert 0 und wird vom Empfänger nicht ausgewertet.

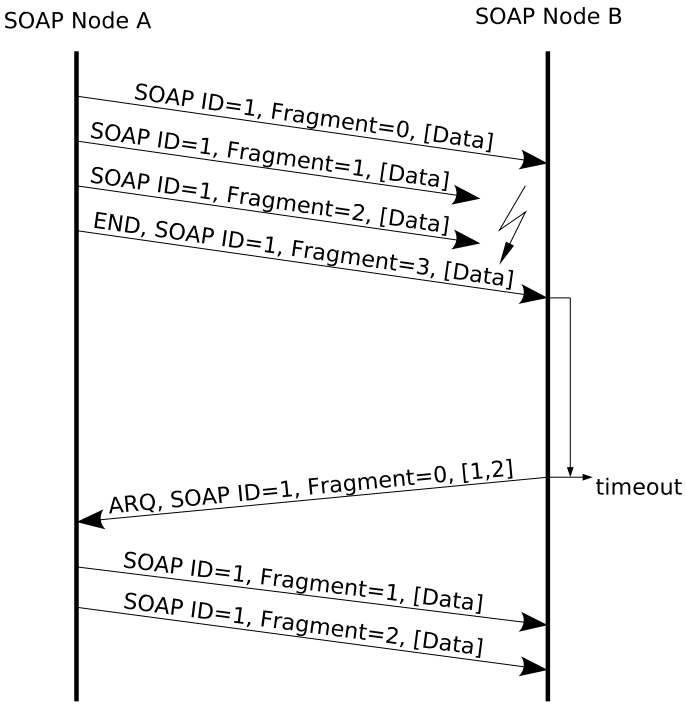


Abbildung 6.6: Erneute Übertragung von Nachrichten mit Hilfe des *ARQ* Flags

Abbildung 6.6 zeigt ein Beispiel: SOAP Node A teilt die zu sendende Nachricht in vier Fragmente ein, doch die Nachrichten mit den *Fragment* Werten 1 und 2 gehen bei der Übertragung verloren. Nach Empfang von *Fragment* 3 läuft der Empfangs-Timer ab, und Node B geht davon aus, dass die beiden noch ausstehenden Nachrichten verloren sind. Er schickt eine PURE-Nachricht mit gesetztem *ARQ* Flag zurück an den Absender und setzt die *Fragment* Werte der fehlenden Nachrichten ins *Data* Feld. SOAP Node A überträgt daraufhin die verlorenen Nachrichten erneut. Der ARQ-Mechanismus basiert also auf negativen Bestätigungsnachrichten (*negative acknowledgement, NACKs*).

Falls das letzte Fragment einer SOAP-Nachricht verloren geht (also das mit dem *END* Flag), können nicht mehrere Nachrichten auf einmal erneut angefordert werden – schließlich weiß die empfangende PURE-Implementierung hier nicht, aus wie vielen Fragmenten die SOAP-Nachricht insgesamt besteht. In diesem Fall würde die empfangende PURE-Implementierung nur das jeweils nächste Fragment anfordern. Dieser Vorgang wird solange wiederholt, bis ein Fragment mit dem *END* Flag empfangen wird.

Das Problem der Duplikaterkennung hängt eng mit dem ARQ-Mechanismus von PURE zusammen. Nachrichtenduplikate können entweder aufgrund von Routing-Fehlern entstehen (also auf der Vermittlungsschicht), oder aber die empfangende PURE-Implementierung fordert unnötigerweise eine erneute Übertragung an. Letzte-

res tritt immer dann auf, wenn eine Nachricht nicht verloren gegangen ist, sondern nur sehr lange unterwegs ist. Läuft der Empfänger-Timer ab, bevor die Nachricht eintrifft, so führt dies zu einer ARQ-Nachricht und schließlich zu einem Duplikat der verspäteten Nachricht.

PURE kann beide Typen von Duplikaten erkennen, indem es alle eingehenden Nachrichten in einem bestimmten Zeitintervall protokolliert. Werden zwei Nachrichten mit identischen Werten für *Fragment* und *SOAP ID* aufgezeichnet, so wird die später eingehende als Duplikat erkannt und verworfen.

6.3.4 Positive Bestätigungsnachrichten

Für einige Anwendungen ist es wichtig, dass der Absender vom Empfänger eine Bestätigung über den korrekten Empfang einer Nachricht erhält. Dies kann der ARQ-Mechanismus allein nicht leisten, denn wenn alle Fragmente verloren gehen, wird der Empfänger auch keine ARQ-Nachrichten generieren. Daher bietet PURE optional auch den Versand positiver Bestätigungsnachrichten (*positive acknowledgements, ACKs*) an.

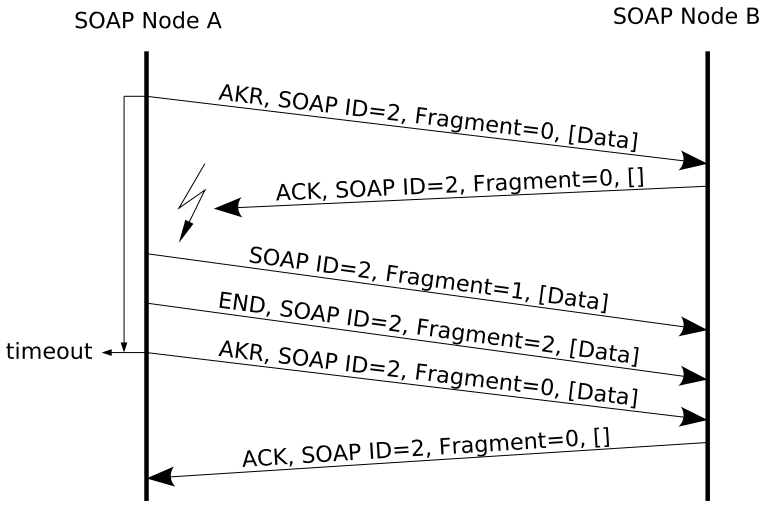


Abbildung 6.7: Positive Bestätigungsnachrichten mit Hilfe der Flags *AKR* und *ACK*

Wie Abbildung 6.7 zeigt, markiert der sendende SOAP Node A die PURE-Nachricht, deren Empfang bestätigt werden soll, mit dem Flag *AKR* (*Acknowledgement Requested*). SOAP Node B antwortet darauf mit einer PURE-Nachricht, die das *ACK* (*Acknowledged*) Flag gesetzt hat. Die Werte für *SOAP ID* und *Fragment* entsprechen denen aus der zu bestätigenden Nachricht. Das Feld *Data* bleibt leer.

Besonders interessant ist nun das Verhalten beim Verlust von Nachrichten: Im dargestellten Beispiel geht die ACK-Nachricht verloren. Nach Ablauf des AKR-Timers

geht die PURE-Implementierung auf SOAP Node A davon aus, dass entweder die mit *AKR* markierte Nachricht oder die korrespondierende ACK-Nachricht verloren gegangen ist. Somit wiederholt A seine Nachricht. Node B antwortet wiederum mit der ACK-Nachricht, verwirft jedoch die eingehende Nachricht, denn diese wird als Duplikat erkannt.

Node A kann nach Empfang der ACK-Nachricht sicher sein, dass B das bestätigte Fragment erhalten hat. Wie in diesem Beispiel deutlich wird, kann der AKR-Mechanismus dazu verwendet werden, nur für ausgewählte Nachrichtenfragmente eine Empfangsbestätigung einzuholen. Es genügt in aller Regel, nur ein Fragment pro SOAP-Nachricht über den AKR-Mechanismus zu schützen, denn wenn Node A sicher ist, dass ein Fragment seiner Nachricht den Empfänger erreicht hat, kann er sich auf den ARQ-Mechanismus verlassen, welcher automatisch die Wiederholung fehlender Fragmente veranlasst.

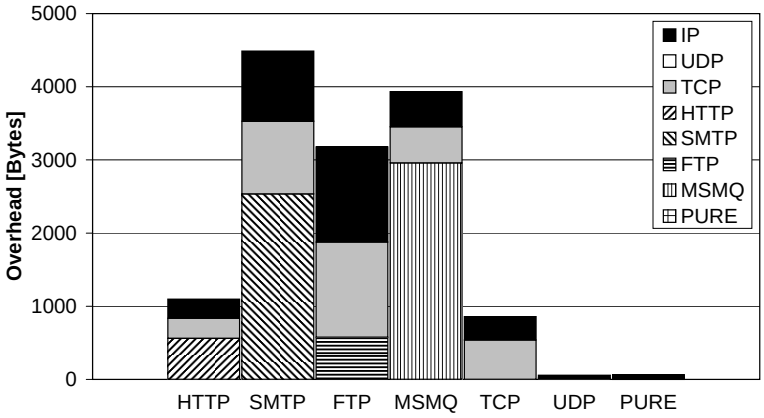
6.4 Implementierung und Evaluation

Im Rahmen einer Diplomarbeit [69] wurde PURE unter Verwendung der Microsoft-.NET-Plattform implementiert. Diese Implementierung beinhaltet alle oben beschriebenen Funktionen, allerdings gibt es zurzeit noch keine Begrenzung für die Anzahl der Nachrichtenwiederholungen, so dass bei permanenten Netzwerkfehlern sehr viele unnötige Nachrichten erzeugt werden. Auch Untersuchungen zur zweckmäßigen Wahl von Puffergrößen und Timer-Werten stehen noch aus. Die Implementierung ist jedoch bereits stabil genug, um SOAP-Nachrichten zwischen zwei SOAP Nodes zuverlässig auszutauschen.

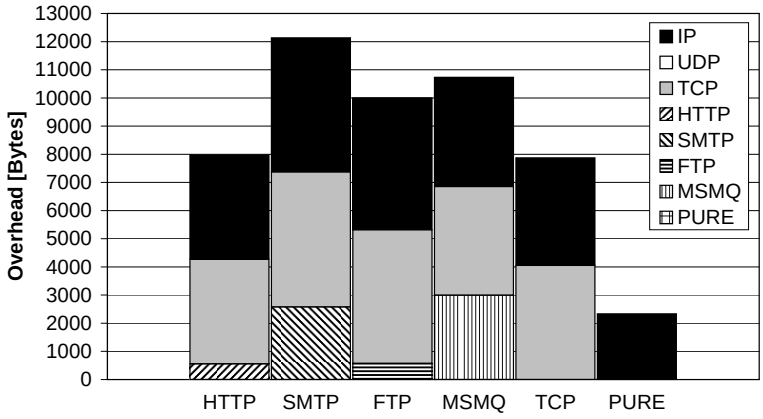
Es sei an dieser Stelle angemerkt, dass der Autor in [161] bereits Untersuchungen zur Latenz dieser PURE-Implementierung vorgestellt hat. Da dieser Aspekt im Rahmen dieser Arbeit allerdings nur am Rande relevant ist, wird auf eine detaillierte Darstellung der Ergebnisse verzichtet. Zusammenfassend lässt sich festhalten, dass PURE im Vergleich zu sämtlichen TCP-basierten Bindings eine etwa um den Faktor drei reduzierte Latenz aufweist. Der Grund hierfür liegt darin, dass bei PURE auf den TCP-typischen Verbindungsaufbau und -abbau verzichtet werden kann.

Im Folgenden präsentiert der Autor Messungen zum Protokoll-Overhead von PURE. Hierzu hat er das PURE-Protokoll anhand der beiden Test-Web-Service-Operationen `void doNothing()` und `byte getImage(String in0)` aus Abschnitt 6.1 untersucht.

Die Abbildung 6.8 zeigt die gemessenen Werte im Vergleich zu den anderen Bindings. In den Diagrammen ist jeweils nur der Overhead dargestellt, weil die Größe der SOAP-Payload für alle Bindings identisch ist (2.149 Bytes für `void doNothing()` und 144.662 Bytes für `byte[] getImage(String in0)`, jeweils Request und Response kumuliert).



(a) `void doNothing()`



(b) `byte[] getImage(String in0)`

Abbildung 6.8: Overhead verschiedener Transport-Bindings im Vergleich

Wie in Abbildung 6.8(a) deutlich zu erkennen ist, verursachen PURE und UDP hier erwartungsgemäß den weitaus geringsten Overhead. Das Datenaufkommen in den einzelnen Protokollschichten ist jeweils so gering, dass eine differenzierte Darstellung im Diagramm nicht mehr möglich ist; der Autor nennt daher im Text die absoluten Werte.

Bei `void doNothing()` können Request- und Response-Nachricht jeweils in einem einzigen PURE-Fragment übertragen werden. Es ergibt sich ein Overhead von zweimal 20 Bytes für den IP-Header, zweimal acht Bytes für den UDP-Header und zweimal fünf Bytes für den PURE-Header – insgesamt also 66 Bytes. Beim UDP-Binding ergeben sich entsprechend 56 Bytes, denn der fünf Bytes große PURE-Header entfällt hier bei Request und Response.

Bei der Operation `byte[] getImage(String in0)` konnte das UDP-Binding wegen seiner inhärenten Größenbeschränkung nicht mit untersucht werden. PURE

hingegen kann aufgrund seiner Fähigkeit zur Nachrichtenfragmentierung auch die 143.501 Bytes große Response-Nachricht dieser Testoperation transportieren. Dabei wurde mit der maximal möglichen Fragmentgröße von 65.507 Bytes gearbeitet (vergleiche Abschnitt 6.3.1). Es müssen also insgesamt vier PURE-Fragmente verschickt werden – eins für die Request- und drei für die Response-Nachricht. Folglich ergeben sich 20 Bytes für die vier PURE-Header und weitere 32 Bytes für vier UDP-Header. Weiterhin ergeben sich 2.300 Bytes IP-Overhead: Durch IP-Fragmentierung (MTU: 1.300 Bytes) ergeben sich 114 IP-Fragmente für die Response-Nachricht und ein weiteres Fragment für die Request-Nachricht. Insgesamt werden also 115 IP-Fragmente erzeugt, was bei einer Header-Größe von 20 Bytes insgesamt 2.300 Bytes ausmacht. Tatsächlich wurde genau dieser Wert für den IP-Overhead mit dem Ethereal-Werkzeug gemessen.

Bei beiden Testoperationen zeigt sich also deutlich, dass PURE jeweils nur einen Bruchteil des Overheads alternativer SOAP-Bindings verursacht. Ganz besonders kommt dieser Vorteil bei kleinen Nachrichten zum Tragen – somit erscheint eine Kombination von PURE mit Kompressionsstrategien, wie wir sie in den Kapiteln 4 und 5 kennen gelernt haben, äußerst viel versprechend.

6.5 Ergebnis

Obwohl die SOAP-Spezifikation einen modular austauschbaren Mechanismus für den Nachrichtentransport vorsieht, verwenden heutige SOAP-Implementierungen hierfür fast ausschließlich HTTP. HTTP bringt bei praktischen Anwendungen jedoch zwei wesentliche Nachteile mit sich: Zum einen verursacht es – wie Untersuchungen des Autors gezeigt haben – einen erheblichen Overhead. Zum anderen bietet es als typisches Client/Server-Protokoll nur unzureichende Unterstützung für einen asynchronen Nachrichtenaustausch.

Da der Einsatz von Web Services zunehmend auch in Anwendungsbereichen interessant wird, in denen ein asynchroner Nachrichtenaustausch und ein geringer Protokoll-Overhead von großer Relevanz sind, werden alternative Transport-Bindings dringend benötigt. Ganz besonders trifft dies auf Web Services zu, die auf mobilen oder ressourcenbeschränkten Geräten laufen.

Ein weiterer Grund für die Entwicklung von leichtgewichtigen Transport-Bindings besteht in der immer weiter fortschreitenden Etablierung von SOAP-Erweiterungen wie WS-Addressing und WS-ReliableMessaging. Diese ergänzen die Binding-Funktionalitäten – mitunter ersetzen sie sie sogar. Es gibt also einen Trend, Funktionalitäten nicht mehr im Transport-Binding, sondern direkt im SOAP-Header zu integrieren – hierzu gehören vor allem Mechanismen für die Adressierung oder den zuverlässigen Nachrichtenaustausch. Die Integration im SOAP-Header bietet dabei den großen Vorteil, dass hier die Informationen auch beim Nachrichtenaustausch über SOAP-Intermediaries erhalten bleiben.

Im Rahmen dieses Kapitels hat der Autor zwei Themenbereiche behandelt: Zum einen ging es um den Vergleich von existierenden SOAP-Binding-Implementierungen hinsichtlich des verursachten Protokoll-Overheads und zum anderen um die Entwicklung eines neuartigen und besonders leichtgewichtigen SOAP-Bindings namens PURE.

Beim Vergleich existierender Bindings zeigte sich im Wesentlichen, dass das von Microsoft entwickelte TCP-Binding nur wenig besser abschneidet als der Transport über HTTP. Das untersuchte UDP-Binding brachte bei der `void doNothing()` Testreihe hingegen eine sehr deutliche Reduzierung des Overheads auf etwa ein Zwanzigstel. Allerdings ist das UDP-Binding nur in der Lage, Nachrichten zu transportieren, die kleiner als 64 kBytes sind. Diese Einschränkung relativiert seinen Nutzen in der Praxis.

Der Autor hat die Idee eines UDP-basierten SOAP-Bindings aufgegriffen und in wesentlichen Punkten erweitert: Er hat ein Anwendungsprotokoll namens PURE entwickelt, welches auf UDP aufsetzt und elementare Zusatzdienste für die SOAP-Kommunikation bereitstellt. Hierzu gehören die Fragmentierung von Nachrichten, die Erkennung von Duplikaten sowie die automatische Neuübertragung verlorener Nachrichten. Optional ist auch der Versand positiver Bestätigungsnachrichten möglich. Trotz dieser Funktionalitäten ist der PURE-Header mit einer Länge von lediglich fünf Bytes äußerst kompakt.

Anhand von praktischen Messungen hat der Autor die Effizienz des PURE-Protokolls nachgewiesen. Erwartungsgemäß erreicht PURE nahezu die gleichen Werte wie das UDP-Binding. Vor allem wegen seiner Fähigkeit zur Nachrichtenfragmentierung lässt es sich aber wesentlich vielseitiger einsetzen.

Die vorgestellten Ergebnisse lassen klar erkennen, dass sich auch sehr leichtgewichtige Anwendungsprotokolle prinzipiell als SOAP-Binding eignen. Dennoch bleibt der vom Autor gewählte UDP-basierte Ansatz spezielleren Anwendungsfeldern vorbehalten: Da UDP keinerlei Mechanismen für die Datenfluss- oder Staukontrolle implementiert, ist PURE für Web-Service-Anwendungen im Internet sicherlich keine universell einsetzbare Lösung. Zwar ließe sich über den PURE-Header prinzipiell auch eine primitive Stop-and-Wait-Flusskontrolle implementieren, der praktische Nutzen einer solchen Erweiterung bleibt indes fraglich; bei Anwendungen, die einer Fluss- oder Staukontrolle bedürfen, ist der Einsatz von TCP sicherlich nach wie vor am zweckmäßigsten.

Interessante Anwendungsbereiche für PURE ergeben sich vor allem bei der Vernetzung mobiler Web Services untereinander; aber auch ein Einsatz in der Hochleistungsdatenverarbeitung, etwa für Grid-Computing-Anwendungen, ist denkbar. Je nach Einsatzbereich müsste überprüft werden, ob der vom Autor vorgeschlagene fünf Bytes große Header zweckmäßig ist. Mitunter ist es sicherlich vorteilhaft, etwas größere Bitbreiten für die einzelnen Felder zu wählen – beispielsweise um ein zu schnelles Überlaufen der *SOAP ID* Werte zu verhindern. Auch zusätzliche Header-Felder könnten sinnvoll sein – etwa für eine Versionierung des PURE-Protokolls. Ein fünf

Bytes großer Header ist im Protokolldesign ohnehin eher untypisch. Üblicherweise wählt man die Länge hier als Vielfaches von vier oder acht Bytes, so dass sich eine optimale Verarbeitungsgeschwindigkeit auf den heute üblichen 32-Bit- oder 64-Bit-Rechnerarchitekturen ergibt.

Die in diesem Kapitel gewählte Länge von fünf Bytes trägt also primär dem Anspruch des Autors Rechnung, die Machbarkeit eines möglichst leichtgewichtigen SOAP-Übertragungsprotokolls zu demonstrieren – PURE in der gegenwärtigen Fassung ist damit eher als Konzeptentwurf zu verstehen und nicht als universell einsetzbare Lösung.

Nach den vorgestellten Untersuchungen ist es grundsätzlich möglich, SOAP-Nachrichten mit sehr geringem Protokoll-Overhead zu übertragen. Neben dem vom Autor gewählten Ansatz, den Nachrichtentransport auf Basis von UDP zu realisieren, sind sicherlich auch andere Ansatzpunkte in diesem Bereich viel versprechend: Insbesondere der Einsatz von *TCP for Transactions (T/TCP)* sollte im Rahmen weiterer Arbeiten untersucht werden. Bei dieser TCP-Variante können bereits beim Verbindungsaufbau Daten ausgetauscht werden, so dass im Vergleich zu herkömmlichem TCP der Overhead reduziert werden kann. Auch der Einsatz der beiden modernen Schicht-4-Protokolle *Stream Control Transmission Protocol (SCTP)* und *Datagram Congestion Control Protocol (DCCP)* zum Transport von SOAP-Nachrichten wurde bislang ebenfalls noch nicht untersucht. Auch dies ist sicherlich eine interessante und höchst praxisrelevante Themenstellung.

Der Autor geht davon aus, dass in den nächsten Jahren ein Bedarf an alternativen Transport-Bindings entstehen wird, welche speziell auf die Erfordernisse eines konkreten Einsatzbereichs abgestimmt sind. Besonders relevant ist hier sicherlich der große Bereich der mobilen Web Services. Daher ist das Forschungsfeld der alternativen SOAP-Bindings ganz sicher sehr zukunftsträchtig, zumal – wie oben dargestellt – wesentliche Fragestellungen noch offen sind.

Kapitel 7

Zusammenfassung und Ausblick

Web Services haben sich als Schlüsseltechnologie zur Überwindung von Heterogenität in weiten Bereichen der Informationstechnik durchgesetzt. Allerdings führt der durchgängige Einsatz von XML nicht nur zu einer vollständig plattformunabhängigen Datendarstellung – vielmehr ist damit auch der Nachteil eines deutlich höheren Datenaufkommens im Vergleich zu älteren Middleware-Technologien wie CORBA oder Java RMI verbunden. Dieser Nachteil ist in vielen Anwendungsbereichen allerdings unerheblich, weil die Netzwerke hier so leistungsstark sind, dass der zusätzliche Overhead durch die textuelle Darstellung kaum ins Gewicht fällt. Hingegen bei kleinen, drahtlos vernetzten Geräten, wie sie beispielsweise bei modernen Ubiquitous-Computing-Anwendungen oder in Sensornetzwerken zum Einsatz kommen, ist die zur Verfügung stehende Datenrate in aller Regel knapp bemessen. Zudem ist der Anwender hier bestrebt, die gesendete Datenmenge so gering wie möglich zu halten, weil jeder Sendevorgang mit einem vergleichsweise hohen Energieaufwand verbunden ist und sich dadurch die Lebensdauer batteriebetriebener Geräte verringert.

Um die Web-Service-Technologie auch für Mobilanwendungen mit begrenzten Datenraten nutzbar zu machen, wurden in der vorliegenden Arbeit neuartige Konzepte und Verfahren zur Reduzierung des Datenaufkommens entwickelt.

Nachdem der Autor im ersten Kapitel seine thematische Motivation dargelegt und den Aufbau dieser Arbeit erläutert hatte, stellte er in Kapitel 2 die wesentlichen technologischen Grundkonzepte für Web Services vor. Er erläuterte den Aufbau des so genannten Web Service Technology Stacks; es handelt sich hierbei um ein Schichtenmodell, das die Protokolle und Datenformate der Web-Service-Kommunikation zueinander in Beziehung setzt. Danach behandelte er die Komponenten SOAP, WSDL und UDDI im Detail, denn diese bilden die Basis sämtlicher Web-Service-Anwendungen. Abschließend ging er auf das Web-Service-Rollenmodell ein, das die Kommunikationsvorgänge zwischen Dienstanbieter, Dienstnutzer und Dienstregister in einen zeitlichen Zusammenhang einordnet.

In Kapitel 3 gab der Autor einen Überblick über die Grundlagen der Informationstheorie und verschiedene Techniken zur Datenkompression. Er diskutierte dabei insbesondere den zentralen Begriff der Entropie; dieser bezeichnet den Grad an Unsicherheit, der beim Empfänger durch den Empfang einer Nachricht beseitigt wird. Auf

Basis dieses Begriffs demonstrierte der Autor anhand zahlreicher Beispielrechnungen, wie sich mit Hilfe von Datenkompressionstechniken das Datenaufkommen wirksam reduzieren lässt.

In den dann folgenden Kapiteln 4, 5 und 6 stellte der Autor die Ergebnisse seiner eigenen Forschungsarbeit vor:

In Kapitel 4 präsentierte er ein neuartiges Verfahren zur Datenkompression von SOAP-Nachrichten; es basiert auf dem Prinzip der Differenzcodierung. Es nutzt die Tatsache aus, dass bei der SOAP-Kommunikation die Struktur der ausgetauschten Nachrichten bereits in wesentlichen Teilen durch die öffentlich verfügbare WSDL-Beschreibung eines Web Services fest vorgegeben ist. Diese statischen Anteile beseitigen beim Empfänger keinerlei Unsicherheit – ihr Informationsgehalt im Sinne der Informationstheorie beträgt also 0 Bit, und ihre Übertragung ist somit entbehrlich. Die Grundidee bei dem vom Autor entwickelten Verfahren ist, die variablen Bestandteile der Nachricht zu extrahieren und nur sie zum Empfänger zu übertragen.

Sender und Empfänger generieren aus der WSDL-Beschreibung zunächst so genannte Skelettnachrichten. Diese enthalten sämtliche statischen Nachrichtenbestandteile, nicht aber die variablen. Der Sender berechnet nunmehr die Differenz aus der zu sendenden SOAP-Nachricht und der korrespondierenden Skelettnachricht (XML-Differencing). Dann überträgt er das XML-Differenzdokument zum Empfänger und dieser rekonstruiert daraus mit Hilfe seiner Kopie des Skelettdatensatzes die SOAP-Nachricht (XML-Patching). Messungen des Autors zeigten, dass die Kompressionsleistung dieses Verfahrens anderen Techniken deutlich überlegen ist. Nachteilig wirkt sich hingegen aus, dass die Berechnung von XML-Differenzdokumenten algorithmisch komplex ist und sich damit nur bedingt für die Implementierung auf ressourcenbeschränkten Geräten eignet. Obwohl der Differenzcodierungsansatz also konzeptionell durchaus interessant ist, eignet er sich gleichwohl nur eingeschränkt für die Anwendungsbereiche, die von einer Datenkompression am meisten profitieren könnten.

Ausgehend von dieser Erkenntnis stellte der Autor in Kapitel 5 einen weiteren neuartigen Kompressionsansatz vor: die SOAP-Kompression mittels Kellerautomaten. Die WSDL-Beschreibung eines Web-Services wird hier als Grammatik aufgefasst, die festlegt, wie gültige SOAP-Nachrichten für die vorliegende Anwendung aufgebaut sind. Aus dieser Grammatik wird schrittweise ein Kellerautomat konstruiert, welcher als validierender Parser für die vorliegende XML-Sprache dient. Die Kernidee des Autors besteht nun darin, diesen Parserautomaten auch für die Datenkompression einzusetzen.

Sender und Empfänger konstruieren jeweils einen gleichartigen Parserautomaten aus der WSDL-Beschreibung des Web Services. Der Sender codiert das zu sendende Dokument, indem er es mit seinem Automaten verarbeitet und den Pfad durch den Automaten in Form eines Binärcodes an den Empfänger übermittelt. Der Empfänger kann mit Kenntnis der Codewortfolge und unter Zuhilfenahme seines (gleichartigen) Automaten den Pfad nachvollziehen und damit das XML-Dokument rekonstruieren.

Im weiteren Verlauf des Kapitels stellte der Autor seine Implementierung dieses Kompressionsansatzes vor – genannt Xenia. Anhand von Messungen konnte er zeigen, dass Xenia besonders bei kleinen Datensätzen äußerst effektiv arbeitet. So konnte das Datenvolumen bei kleinen SOAP-Nachrichten auf ein Hundertstel reduziert werden – Xenia arbeitet damit noch effektiver als die Differenzcodierung. Zudem eignet sich dieser Kompressionsansatz vor allem auch für den Einsatz auf ressourcenbeschränkten Geräten; zur Laufzeit muss lediglich ein Kellerautomat ausgeführt werden, was auch auf leistungsschwachen Geräten problemlos möglich ist.

In Kapitel 6 dehnte der Autor seine Betrachtungen auch auf die unteren Protokollschichten aus: Zwar lässt sich durch XML-Kompressionsverfahren der Overhead von SOAP wirksam reduzieren, Messungen ergaben aber, dass der Overhead, der durch die darunter liegenden Protokolle HTTP, TCP und IP beim Nachrichtentransport verursacht wird, dabei weitestgehend unverändert bleibt. Werden die SOAP-Nachrichten stark komprimiert, übersteigt das Datenvolumen der für den Nachrichtentransport benötigten Protokoll-Header die Größe der Nachrichten um ein Mehrfaches.

Ausgehend von dieser Beobachtung hat der Autor die zurzeit verfügbaren Mechanismen für den SOAP-Nachrichtentransport (Transport-Bindings) mit Blick auf den jeweils verursachten Overhead untersucht. Dabei zeigte sich, dass ein SOAP-über-UDP-Binding den mit Abstand geringsten Overhead verursacht. Allerdings bringt der SOAP-Transport über UDP auch den Nachteil mit sich, dass damit nur solche Nachrichten transportiert werden können, die kleiner als 64 kBytes sind. Diese Einschränkung ist – besonders bei der Weiterleitung von Nachrichten über SOAP-Zwischenknoten – nicht akzeptabel.

Daher hat der Autor die Idee eines UDP-basierten Nachrichtentransports aufgegriffen und in wesentlichen Punkten weiterentwickelt: Er hat ein Anwendungsprotokoll namens PURE entworfen, das auf UDP aufsetzt und vier zusätzliche Dienste bereitstellt: Nachrichtenfragmentierung, Duplikaterkennung, negative Bestätigungsnachrichten in Verbindung mit einer automatischen Wiederholung des Sendevorgangs sowie optionale positive Bestätigungsnachrichten. Der PURE-Header nimmt dabei lediglich fünf Bytes ein. Praktische Messungen zeigten, dass PURE im Vergleich zu HTTP den Overhead bei sehr kleinen Nachrichten auf etwa ein Sechzehntel reduzieren kann, bei größeren Nachrichten auf etwa ein Drittel. Somit stellt PURE für Anwendungsfelder, in denen Datenflusssteuerung und Staukontrolle entbehrlich sind, eine interessante Alternative dar.

In den Kapiteln 4, 5 und 6 wurden damit jeweils komplementäre Techniken vorgestellt, um das Datenaufkommen bei der SOAP-Kommunikation zu reduzieren. Obwohl PURE und auch die XML-Differenzcodierung aus Kapitel 4 konzeptionell wie praktisch durchaus interessant sind, besteht nach Auffassung des Autors der wissenschaftliche Hauptbeitrag dieser Arbeit im Datenkompressionsverfahren aus Kapitel 5. Dieses ist für sämtliche XML-Anwendungen auf mobilen Geräten äußerst vorteilhaft, denn es begegnet nicht nur sehr wirksam dem Problem des hohen Datenaufkommens,

sondern ermöglicht gleichzeitig auch das Parsen von XML-Daten auf ressourcenbeschränkten Geräten. Betrachtet man die Darstellung des komprimierten Binärstroms in Abbildung 5.10 auf Seite 125, sieht man unschwer, dass eine noch effizientere Codierung des Markups eines XML-Dokuments kaum möglich ist. Zwar sind sicherlich noch einige Detailverbesserungen denkbar – beispielsweise indem man bei der Codierung Heuristiken über die Häufigkeiten möglicher Zustandsübergänge berücksichtigt. Allerdings ist nicht zu erwarten, dass solche Verbesserungen in der Praxis zu einer signifikanten Verbesserung der Kompressionsleistung führen.

Besonders interessant ist dagegen die Frage, wie sich ein Kompressionsautomat effizient als Hardwarestruktur implementieren lässt – so könnten selbst kleinste Geräte mit einem „XML-Coprozessor“ ausgestattet werden, der gleichzeitig für das Parsen und auch für eine kompakte Binärrepräsentation der Daten sorgt.

Auch im Bereich der SOAP-Transport-Bindings ergeben sich viele Perspektiven für weitere Arbeiten, denn alternative Bindings werden in der Literatur bisher kaum behandelt. Für spezielle Einsatzgebiete wie mobile Anwendungen oder auch Grid-Computing-Anwendungen sind angepasste Transportmechanismen aber vorteilhaft. Daher hält der Autor diesen Teilbereich der Web-Service-Forschung für äußerst zukunftsträchtig.

Ebenfalls noch offen ist die Frage, wie sich Kompressionsstrategien und alternative Protokolle für den Nachrichtentransport auf Kenngrößen wie Energiebedarf, Latenz oder Durchsatz auswirken. Zwar ist zu erwarten, dass die im Rahmen dieser Arbeit vorgestellten Lösungen einen positiven oder wenigstens neutralen Einfluss auf diese Kenngrößen haben, experimentelle Messungen hierzu stehen allerdings noch aus.

Zusammenfassend ist festzustellen, dass sich mit Hilfe der in dieser Arbeit vorgestellten Ansätze das Datenaufkommen effektiv reduzieren lässt – und zwar sowohl in Bezug auf SOAP als auch die darunter liegenden Protokollschichten. Der Autor ist zuversichtlich, dass seine hier vorgestellten Verfahren und Techniken wesentlich dazu beitragen können, Web Services auch in den Bereichen der Informationstechnologie zu etablieren, in denen die zur Verfügung stehende Datenrate einen Engpass bildet.

Anhang

Anhang A

Grundlagen von XML

Im Rahmen dieser Arbeit werden die Begriffe aus dem Themengebiet XML durchgehend so verwendet, wie es in den Standarddokumenten [165, 167, 178, 179] vorgesehen ist. Im Folgenden liefert der Autor einen kurzen Abriss über die wichtigsten Aspekte, so dass der Leser einen Überblick über die für das Verständnis dieser Arbeit wesentlichen Begriffe gewinnt.

A.1 Markup und Zeichendaten

XML ist eine Sprache zur Beschreibung von Daten beliebiger Art. Einen XML Datensatz bezeichnet man auch als *XML-Dokument*. XML sieht für die Beschreibung von Daten zwei grundlegende Sprachkonstrukte vor: *Markup* und *Zeichendaten* (engl.: character data).

Unter Markup verstehen wir Angaben, welche die *Struktur* eines Datensatzes festlegen. Syntaktisch wird das Markup im Wesentlichen durch spezielle Bezeichner in spitzen Klammern – so genannte *Tags* – ausgedrückt. Neben Tags gibt es noch eine ganze Reihe weiterer Syntax-Konstrukte, die ebenfalls dem Markup zugerechnet werden – beispielsweise Kommentare oder Entity-Referenzen. Im Rahmen dieser Arbeit sind diese Konstrukte jedoch nicht relevant und werden daher auch nicht näher erläutert. Eine genaue Übersicht über die verschiedenen Markup-Konstrukte findet der Leser in [165].

Der Begriff *Zeichendaten* bezeichnet die Teile eines XML-Dokuments, die nicht zum Markup gehören.

Betrachten wir ein einfaches Beispiel für ein XML-Dokument:

```
<mitarbeiter>
    <nachname>Mustermann</nachname>
    <vorname>Hans</vorname>
    <personalnummer>123456</personalnummer>
</mitarbeiter>
```

Dieses Beispiel zeigt einen Datensatz „Mitarbeiter“ – sämtliche Zeichendaten sind zur Verdeutlichung im Fettdruck dargestellt.

Es gibt zwei Typen von XML-Tags: öffnende und schließende. Erstere haben die Form `<tag-name>` und letztere die Form `</tag-name>`. Zu beachten ist dabei, dass es immer Paare von gleichnamigen öffnenden und schließenden Tags geben muss. Ein solches Tag-Paar kennzeichnet einen Abschnitt im XML-Dokument. Beispielsweise kennzeichnen die Tags `<nachname>` und `</nachname>` den Abschnitt im oben angegebenen Dokument, der den Nachnamen des im Datensatz beschriebenen Mitarbeiters enthält.

Weiterhin gibt es noch eine abkürzende Schreibweise für ein öffnendes Tag, auf das unmittelbar das zugehörige schließende Tag folgt: `<tag-name></tag-name>` ist äquivalent zu `<tag-name/>`.

Wie man im Beispiel erkennt, drückt die Anordnung der Tags eine hierarchische Struktur aus. Der Bereich zwischen `<mitarbeiter>` und `</mitarbeiter>` bildet die oberste Hierarchieebene und die darin enthaltenen Abschnitte sind eine Hierarchieebene unterhalb angeordnet.

Sowohl die Tag-Namen als auch die Zeichendaten können prinzipiell aus beliebigen Unicode-Zeichen [151] bestehen. Allerdings schließt die XML-Spezifikation einige Zeichen explizit aus, beispielsweise Steuerzeichen wie *End of Text (EOT)*. Anderen Zeichen weist sie eine besondere Bedeutung zu, z. B. die Zeichen < und >, die zur Kennzeichnung des Markups verwendet werden.

A.2 Wohlgeformtheit von Dokumenten

Erfüllt ein Datensatz sämtliche Syntaxregeln der XML-Spezifikation, so bezeichnet man ihn als *wohlgeformt* (engl.: *well-formed*). Insbesondere müssen hierzu die öffnenden und schließenden Tags korrekt geschachtelt sein, d. h. zu jedem öffnenden Tag gibt es auf der selben Hierarchieebene ein gleichnamiges schließendes Tag (und umgekehrt). Eine weitere wesentliche Anforderung besteht darin, dass genau ein Tag-Paar auf der obersten Hierarchieebene existiert.

Das oben angegebene XML-Dokument ist damit wohlgeformt. Der folgende Datensatz ist hingegen nicht wohlgeformt, weil er das Kriterium der korrekten Tag-Schachtelung nicht erfüllt:

```
<mitarbeiter>
    <nachname>Mustermann</nachname>
    <vorname>Hans</vorname>
    <personalnummer>123456</mitarbeiter>
</personalnummer>
```

Nicht wohlgeformte Datensätze sind laut Spezifikation keine XML-Dokumente.

A.3 Prolog und Zeichencodierung

Vor dem ersten öffnenden Tag kann ein XML-Dokument einen Prolog enthalten. Hier sind technische Informationen abgelegt, die bei der Verarbeitung des Datensatzes relevant sind. Hierzu gehören vor allem Angaben über die verwendete XML-Version (`version`) sowie über die verwendete Zeichencodierung (`encoding`).

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<mitarbeiter>
    <nachname>Mustermann</nachname>
    <vorname>Hans</vorname>
    <personalnummer>123456</personalnummer>
</mitarbeiter>
```

Im Beispiel bedeutet der Wert `UTF-8`, dass die nach dem Prolog folgende Bytesequenz als Zeichenkette in UTF-8-Codierung zu interpretieren ist.

Eine andere mögliche Zeichencodierung für Unicode ist z. B. UTF-16 – hiermit können ebenfalls alle Unicode-Zeichen dargestellt werden, allerdings werden hier Unicode-Zeichen durch andere Bytesequenzen ausgedrückt.

Neben den Unicode-Zeichencodierungen unterstützen die meisten XML-verarbeitenden Programme auch gängige ISO-Zeichencodierungen, wie z. B. ISO-8859-1. Allerdings ist hierbei zu beachten, dass damit nicht alle Unicode-Zeichen ausgedrückt werden können. Somit stehen dem XML-Anwender nicht mehr alle Unicode-Zeichen zur Verfügung, sondern nur noch eine Teilmenge davon.

A.4 Darstellungsformen: Text und Baum

Da sich durch die Schachtelung der Tags eine hierarchische Struktur ergibt, lässt sich jedes XML-Dokument nicht nur als Text, sondern auch als Baum darstellen. Aus den Tag-Paaren und Zeichendaten in der Textdarstellung ergeben sich die Knoten in der Baumdarstellung (vergleiche Abbildung A.1).

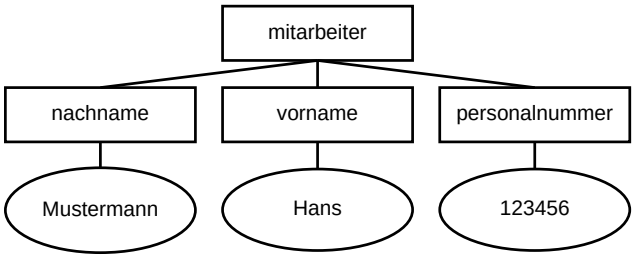


Abbildung A.1: Darstellung eines XML-Dokuments als Baumstruktur

A.5 Elemente

Jedes Tag-Paar eines XML-Dokuments bildet eine „logische“ Einheit, d. h. der Wirkungsbereich eines Tags wird erst dann deutlich, wenn man das öffnende und das korrespondierende schließende Tag gemeinsam betrachtet. Solche Tag-Paare sind damit für die Strukturierung der im XML-Dokument enthaltenen Informationen verantwortlich. In der XML-Spezifikation wird für ein Tag-Paar daher auch der Name *Element* verwendet (von lat.: elementum = Urstoff, Grundstoff).

Anschaulich entspricht ein Element einem Knoten in der Baumdarstellung.

Der Name eines Elements ist gegeben durch den Namen des zugehörigen Tag-Paares. Somit ergibt sich eine Eins-zu-Eins-Zuordnung zwischen Tag- und Elementnamen.

Das Element auf der obersten Hierarchieebene nennt man auch *Wurzelelement*. Elemente, die auf der nächst unteren Hierarchieebene eines Elements E angeordnet sind, bezeichnet man als *Kindelemente* von E. Das Element auf der nächst höheren Hierarchieebene eines Elements E bezeichnet man als *Vaterelement* von E. Im dargestellten Beispiel wäre also *Vorname* Kindelement von *Mitarbeiter*, und *Mitarbeiter* wäre Vaterelement von *Personalnummer*.

A.6 Attribute

Neben Elementen ist in XML noch ein weiteres syntaktisches Konstrukt vorgesehen, um Daten zu strukturieren: *Attribute*. Ein Attribut ist ein Name-Wert-Paar. Im Gegensatz zu Elementen können Attribute nur Zeichendaten enthalten, nicht jedoch Elemente oder andere Attribute.

Ein Attribut ist direkt einem Element zugeordnet und wird direkt im öffnenden Tag in der Syntax `name="wert"` notiert:

```
<?xml version="1.0" encoding="UTF-8"?>
<mitarbeiter status="beurlaubt">
    <nachname>Mustermann</nachname>
    <vorname>Hans</vorname>
    <personalnummer>123456</personalnummer>
</mitarbeiter>
```

A.7 Namespaces

Bei komplexen XML-Applikationen kann es leicht vorkommen, dass Tag- oder Attributnamen in unterschiedlichen Zusammenhängen mehrfach vorkommen.

Beispielsweise könnte ein Element mit dem Namen *adresse* zum einen eine Postanschrift bezeichnen und zum anderen eine E-Mail-Adresse.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<mitarbeiter>
    <nachname>Mustermann</nachname>
    <vorname>Hans</vorname>
    <personalnummer>123456</personalnummer>
    <adresse>
        Hans Mustermann, Breite Straße 7, 12489 Berlin
    </adresse>
    <account>
      <login>hmuster</login>
      <adresse>hmuster@example.com</adresse>
    </account>
</mitarbeiter>
```

Bei der maschinellen XML-Verarbeitung ist es wichtig, solche Namenskonflikte zu vermeiden – im oben angegeben Beispieldokument könnte eine E-Mail-Anwendung nämlich nicht mehr allein anhand des Namens entscheiden, welches der beiden Elemente die E-Mail-Adresse des Mitarbeiters enthält. Daher bietet XML die Möglichkeit Element- und auch Attributnamen mit so genannten *Namespaces* zu kennzeichnen. Ein Namespace hat dabei die Form einer URI.

Bevor ein Namespace verwendet werden kann, muss er deklariert werden – syntaktisch erfolgt diese Deklaration innerhalb eines öffnenden Tags. Im Rahmen der Deklaration wird dem Namespace ein (typischerweise kurzer) Bezeichner zugeordnet, das so genannte *Namespace-Präfix*. Eine Namespace-Deklaration hat dabei die Syntax `<tagname xmlns:präfix="URI">`. Der Gültigkeitsbereich dieser Deklaration erstreckt sich bis zum korrespondierenden schließenden Tag.

Um nun einen Tag- oder Attributnamen mit einem Namespace zu versehen, wird ihm das entsprechende Präfix vorangestellt: `präfix:name`. Einen solchen erweiterten Namen bezeichnet man auch als *Qualified Name* oder kurz *QName*.

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<personal:mitarbeiter
  xmlns:personal="http://company.example.com/personal"
  xmlns:it="http://company.example.com/it">
    <personal:vorname>Hans</personal:vorname>
    <personal:nachname>Mustermann</personal:nachname>
    <personal:personalnummer>123456</personal:personalnummer>
    <personal:adresse>
        Hans Mustermann, Breite Straße 7, 12489 Berlin
    </personal:adresse>
    <it:account>
      <it:login>hmuster</it:login>
      <it:adresse>hmuster@example.com</it:adresse>
    </it:account>
</personal:mitarbeiter>
```

Alle Elemente, die für das Personal-Management-System relevant sind, werden hier mit dem Namespace `http://company.example.com/personal` versehen. Alle für IT-Anwendungen relevanten Elemente tragen dagegen den Namespace `http://company.example.com/it`. Auf diese Weise wird der Namenskonflikt beim Element `adresse` behoben.

A.8 SAX und DOM

Ein Hauptvorteil von XML ist, dass der Programmierer das XML-Dokument nicht als Bytesequenz bzw. als Zeichenkette verarbeiten muss. Vielmehr stehen für alle gängigen Programmiersprachen fertige XML-Parser bereit, die dem Anwendungsentwickler einen komfortablen Zugriff auf das XML-Dokument über eine Programmierschnittstelle (engl.: Application Programming Interface, API) ermöglichen.

Die *Simple API for XML (SAX)* und das *Document Object Model (DOM)* sind die am weitesten verbreiteten APIs zur XML-Verarbeitung.

Bei SAX handelt es sich um einen event-basierten Ansatz, bei dem das XML-Dokument sequentiell verarbeitet wird. Der Parser liest das XML-Dokument als Zeichenkette ein und interpretiert dabei die XML-Syntax. Bei der Verarbeitung von öffnenden und schließenden Tags, Attributen, Zeichendaten usw. löst ein SAX-Parser unterschiedliche Events (also Ereignisnachrichten) aus, die an die Applikation übermittelt werden. Der Programmier wertet nun diese Events aus und steuert so seine Anwendung.

Bei DOM wird das Dokument nicht sequentiell verarbeitet. Es wird zunächst als Ganzes in den Hauptspeicher geladen, und zwar in Form einer baumartigen Datenstruktur (vergleiche Abbildung A.1). Der Programmierer kann nun über die DOM-API in diesem Baum navigieren und an beliebigen Stellen Werte auslesen oder ändern.

Der besondere Vorteil der DOM-API besteht darin, dass der Programmierer wahlfreien Zugriff auf alle Teile des XML-Dokuments hat. Bei der SAX-API werden die Events immer sequentiell erzeugt, d. h. die Daten müssen in der Reihenfolge von der Applikation verarbeitet werden, die im Dokument vorgegeben ist. Allerdings beansprucht ein SAX-Parser deutlich weniger RAM-Speicher im Vergleich zu einem DOM-Parser, denn dieser muss das gesamte Dokument im RAM-Speicher vorhalten.

Anhang B

Grundlagen von XML Schema

Ein XML-Dokument kann prinzipiell aus beliebigem Markup und beliebigen Zeichendaten bestehen. Einem Entwickler steht es also insbesondere frei, welche Attribut- und Tag-Namen er für sein anwendungsspezifisches XML-Datenformat wählt. Auf diese Weise ist es möglich, XML für beliebige Anwendungen einzusetzen.

Allerdings ist es oftmals auch wünschenswert, klare Vereinbarungen über die zu verwendende XML-Syntax festzulegen, d. h. welche Markup-Strukturen und welche Zeichendaten in welchen Sequenzen im XML-Dokument auftauchen können. Für diesen Zweck wurde die Sprache *XML Schema* entwickelt. Mit ihrer Hilfe ist es möglich, eine Teilmenge aller möglichen XML-Dokumente zu definieren, welche für einen konkreten Anwendungsfall relevant ist. Eine solche Teilmenge bezeichnet man auch als *XML-Sprache*. Die Elemente dieser Sprache bezeichnet man als *Instanzdokumente* der zu Grunde liegenden XML-Schema-Beschreibung.

Für technische Anwendungen ist es oftmals wichtig, automatisch überprüfen zu können, ob ein XML-Dokument in der durch ein XML-Schema-Dokument festgelegten XML-Sprache enthalten ist oder nicht. Daher gibt es Werkzeuge, die diese Überprüfung automatisch vornehmen – so genannte *validierende* Parser. Genügt ein XML-Dokument den Regeln der XML-Schema-Beschreibung, bezeichnet man es als *gültig* oder *valide* (engl.: *valid*).

XML Schema ist selbst eine XML-Sprache. Im Folgenden gibt der Autor einen Überblick über die wichtigsten Sprachkonstrukte.

B.1 Lokale und globale Elementdeklarationen

Das Wurzelelement einer jeden XML-Schema-Beschreibung ist `schema`. Es zeigt an, dass eine XML-Schema-Beschreibung folgt. Wie alle Anweisungen und Datentypen eines XML-Schema-Dokuments stammt dieses Element aus dem Namespace `http://www.w3.org/2001/XMLSchema`.

Die wohl wichtigste Anweisung in XML Schema ist eine Elementdeklaration. Sie wird ausgedrückt über das Element `element` und gibt an, dass ein Instanzdokument ein

Element mit dem angegebenen Namen und dem angegebenen Datentyp enthalten darf.

Beispiel:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a" type="xsd:string"/>
</xsd:schema>
```

Im Beispiel müsste ein Instanzdokument also als einziges Element `a` enthalten. Der Datentyp von `a` ist dabei `xsd:string`, d. h. in einem gültigen Instanzdokument darf `a` keinerlei Kindelemente enthalten, sondern nur Zeichendaten.

Ein Beispiel für ein gültiges Instanzdokument der oben angegebenen XML-Schema-Beschreibung:

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

Nicht gültig und damit kein Instanzdokument der obigen XML-Schema-Beschreibung wäre dagegen:

```
<?xml version="1.0" encoding="UTF-8"?>
<a><b/>Text</a>
```

Dieses Dokument enthält ein Element `b`, was aber in der XML-Schema-Beschreibung nicht vorgesehen ist – es ist somit nicht gültig und folglich kein Instanzdokument des oben angegebenen Schemas.

Eine Elementdeklaration, die als direktes Kindelement vom Element `schema` angegeben ist, bezeichnet man als *globale* Elementdeklaration. Elemente, die auf diese Weise deklariert wurden, dürfen in Instanzdokumenten als Wurzelelemente auftreten.

Alle anderen Elementdeklarationen bezeichnet man dagegen als *lokale* Elementdeklarationen. So deklarierte Elemente dürfen in Instanzdokumenten nicht als Wurzelelement auftreten. Beispiele hierfür werden wir im Folgenden noch kennen lernen.

B.2 Typdefinitionen

In XML Schema gibt es zwei Sorten von Datentypen: Zum einen gibt es *einfache Datentypen* (engl.: *simple types*) und zum anderen *komplexe Datentypen* (engl.: *complex types*).

Einfache Datentypen beschreiben die Struktur von Zeichendaten und komplexe die Struktur von Markup.

B.2.1 Einfache Datentypen

Ein Beispiel für einen einfachen Datentyp haben wir bereits oben kennen gelernt: `xsd:string`. XML Schema bietet dem Entwickler eine ganze Reihe einfacher, vordefinierter Datentypen an. Diese brauchen in einem XML-Schema-Dokument nicht neu definiert zu werden; der Entwickler kann diese fest „eingebauten“ Typen direkt verwenden. Die XML-Spezifikation bezeichnet diese Datentypen daher auch als *built-in types*.

Neben `xsd:string` gibt es auch noch `xsd:int`, `xsd:float`, `xsd:boolean` usw. Damit stehen einem Entwickler im Wesentlichen die Datentypen zur Verfügung, die auch in modernen Programmiersprachen als primitive Typen enthalten sind.

Wird beispielsweise in einer Elementdeklaration der Datentyp `xsd:int` verwendet, so darf das Element nur solche Zeichendaten enthalten, die eine ganze Dezimalzahl im Wertebereich $[-2^{31}, \ldots, 2^{31} - 1]$ beschreiben.

Beispiel:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a" type="xsd:int"/>
</xsd:schema>
```

Das folgende Dokument ist nicht gültig, weil `a` keinen numerischen Wert enthält:

```
<?xml version="1.0" encoding="UTF-8"?>
<a>123a21</a>
```

Eine genaue Übersicht über sämtliche eingebauten Typen findet der Leser in [179].

Neben den eingebauten Typen kann man auch selbst einfache Datentypen definieren. Hierzu dient die Anweisung `simpleType`. Hiermit kann man beispielsweise den Wertebereich von bereits existierenden Datentypen weiter einschränken:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a">
  <xsd:simpleType>
   <xsd:restriction base="xsd:int">
    <xsd:minInclusive value="0"/>
    <xsd:maxInclusive value="10"/>
   </xsd:restriction>
  </xsd:simpleType>
 </xsd:element>
</xsd:schema>
```

In Instanzdokumenten dieses Schemas dürfte das Element `a` nur ganzzahlige Dezimalwerte im Bereich $[0, \ldots, 10]$ enthalten. In diesem Beispiel erkennt man weiterhin, dass man den Datentyp eines Elements auch direkt als Kind von `element` angeben kann (anstatt über das Attribut `type`).

Beispiel für ein gültiges Instanzdokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<a>7</a>
```

B.2.2 Komplexe Datentypen

Neben Angaben zum Wertebereich von Zeichendaten kann man mittels XML Schema auch die Struktur von Markup festlegen. Hierzu dient die Anweisung `complexType`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="b" type="xsd:int"/>
        <xsd:element name="c" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Im Beispiel wird dem Element `a` ein komplexer Datentyp zugewiesen. Dieser beschreibt eine Sequenz (Element `sequence`) von zwei Kindelementen `b` und `c`. Bei einer Sequenz ist die Reihenfolge der Kindelemente entscheidend, d. h. `b` und `c` müssen in der richtigen Reihenfolge auftreten. Statt `sequence` könnte man auch die Anweisung `choice` verwenden; hiermit wird eine Alternative beschrieben, d. h. es müsste entweder `b` oder `c` Kind von `a` sein.

Da die Elementdeklarationen innerhalb von `complexType` Elementen keine Kinder von `schema` sind, handelt es sich hierbei um lokale Elementdeklarationen (vergleiche oben). Somit darf in Instanzdokumenten zwar `a` als Wurzelelement vorkommen, nicht jedoch `b` oder `c`.

Beispiel für ein gültiges Instanzdokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<a>
  <b>42</b>
  <c>d</c>
</a>
```

Weiterhin kann man bei komplexen Datentypen auch Entitäten angeben, d. h. wie oft ein Element auftreten darf. Die untere bzw. obere Schranke für die Auftrittshäufigkeit wird über die Attribute `minOccurs` und `maxOccurs` festgelegt.

Beispiel unter Verwendung von `choice`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a">
  <xsd:complexType>
   <xsd:choice>
    <xsd:element name="b" type="xsd:int"    minOccurs="0" maxOccurs="1"/>
    <xsd:element name="c" type="xsd:string" minOccurs="1" maxOccurs="2"/>
   </xsd:choice>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Beispiel für ein gültiges Instanzdokument:

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

Ein weiteres Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<a>
  <c>d</c>
  <c>e</c>
</a>
```

B.3 Namespaces

In XML Schema ist es auch möglich, den Namespace anzugeben, den die Elemente und Attribute im Instanzdokument tragen sollen. Hierfür wird direkt im Wurzelelement `schema` das Attribut `targetNamespace` eingefügt:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema targetNamespace="http://comany.example.com/example"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="a" type="xsd:string"/>
</xsd:schema>
```

Beispiel für ein gültiges Instanzdokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<ex:a xmlns:ex="http://comany.example.com/example">string</ex:a>
```

Anhang C

Konstruktion eines deterministischen Kellerautomaten aus einer regulären Baumgrammatik

Im Folgenden erläutert der Autor die Konstruktion eines deterministischen Kellerautomaten (DKA) aus einer regulären Baumgrammatik (RBG). Zur besseren Verständlichkeit wird das Konstruktionsverfahren leicht vereinfacht: Wir betrachten lediglich das XML-Markup, nicht aber die Zeichendaten. Da sich Zeichendaten eindeutig von Tags unterscheiden lassen, kann ihre Verarbeitung als separater Schritt angesehen werden (siehe hierzu auch Abschnitt C.1 auf Seite 179).

Für eine exakte algorithmische Darstellung und den anschließenden Korrektheitsbeweis benötigen wir zunächst einige Begriffsdefinitionen:

Definition C.1 (XML-Dokument):
Ein XML-Dokument ist ein n-Tupel $x = (t_0, \ldots, t_{n-1})$ bestehend aus endlich vielen Symbolen t_i aus der Menge Σ_t. Σ_t heißt Tag-Alphabet. Alle öffnenden Tags (bezeichnet mit t_i^+) werden in der Menge Σ_t^+ zusammengefasst und alle schließenden (bezeichnet mit t_i^-) in der Menge Σ_t^-. Damit gilt: $\Sigma_t = \Sigma_t^+ \cup \Sigma_t^-$
Anmerkung: Diese Definition umfasst auch nicht wohlgeformte XML-Dokumente.

Definition C.2 (reguläre Baumgrammatik (RBG)):
Eine reguläre Baumgrammatik ist ein 4-Tupel: $G = (N, T, P, S)$

- N ist eine Menge von Symbolen (Nichtterminale).
- T ist eine Menge von Symbolen (Terminale). Es gilt: $N \cap T = \{\}$.
- P ist eine Menge von Produktionsregeln. Jedes $p \in P$ hat die Form: $n \rightarrow t\ \ \varrho$
  Hierbei ist $n \in N$, $t \in T$ und ϱ ein regulärer Ausdruck über N. ϱ heißt Inhaltsmodell von t.
- S ist eine Menge von Symbolen mit $S \subseteq N$ (Startsymbole).

Die Sprache, die G beschreibt, wird mit $\mathcal{L}(G)$ bezeichnet. $\mathcal{L}(G)$ ist dabei eine Menge von Bäumen.

Anmerkung: Nach Murata [101] ist es möglich, aus einem XML-Schema-Dokument eine reguläre Baumgrammatik zu erzeugen. Elementnamen werden hierbei durch Terminale und Datentypen durch Nichtterminale ausgedrückt. Folglich sind die Mengen der öffnenden und schließenden Tags (Σ^+ und Σ^-) jeweils gleich mächtig mit der Menge der Terminalsymbole.

Zum einfacheren Umgang mit den Produktionsregeln $p \in P$ definieren wir die folgenden Funktionen:

- $term(p)$: Liefert das Terminalsymbol von p.
- $type(p)$: Liefert das Nichtterminalsymbol von p.
- $dea(p)$: Liefert das Inhaltsmodell von p (dargestellt als deterministischer endlicher Automat). Das Bild von $dea(p)$ bezeichnen wir mit DEA.
- $state_{init}(dea)$ und $states_{accept}(dea)$ liefern zu einem DEA den Startzustand bzw. die Menge der Endzustände.
- $in(q)$ und $out(q)$ liefern zu einem DEA-Zustand die Menge der ein- bzw. ausgehenden Transitionen.
- $dest(t)$ liefert zu einer DEA-Transition den Folgezustand.
- $type(t)$ liefert zu einer DEA-Transition die Zustandsübergangsmarkierung (d. h. ein Nichtterminal der RBG).
- $tag(type)$ liefert zu einem Nichtterminal das Terminal.[1]
- $dea(type)$ liefert zu einem Nichtterminal das Inhaltsmodell als DEA.[1]

Definition C.3 (Kellerautomat): Ein Kellerautomat K ist ein 6-Tupel:
$K = (Q, \Sigma, \Gamma, \Psi, q_{start}, z)$

- Q ist eine Menge von Zuständen,
- Σ ist das Eingabealphabet,
- Γ ist das Stack-Alphabet,
- Ψ ist eine Menge von Produktionsregeln.
  Jedes $\psi \in \Psi$ hat die Form: $(Q \times \Sigma \times \Gamma) \rightarrow (Q \times \Gamma^*)$,
- $q_{start} \in Q$ ist der Startzustand,
- $z \in \Gamma$ ist das initiale Stack-Symbol.

[1] Diese Zuordung ist bei einer RBG, die aus einem XML-Schema-Dokument erzeugt wurde, stets eindeutig, weil es zu jedem Datentyp (Nichtterminal) genau eine Produktionsregel $p \in P$ gibt.

Der Kellerautomat akzeptiert ein Eingabewort w gdw.: w wurde komplett gelesen, und der Stack ist leer.

Lemma C.1 (vollständige Fallunterscheidung für XML-Dokumente):
Gegeben sei ein XML-Dokument $x = (t_0, \ldots, t_{n-1})$ mit $t_i \in \Sigma_t \cup \{\varepsilon, \#\}$. Die Zeichen ε und # sind zwei reservierte Zeichen ($\varepsilon, \# \notin \Sigma_t$). Sie markieren Anfang und Ende eines jeden Dokuments und dürfen nur einmal im Dokument vorkommen: $t_0 = \varepsilon$ und $t_{n-1} = \#$.

Betrachten wir zwei beliebige Tags t_i, t_{i+1}, die in x direkt aufeinander folgen, so ergeben sich sechs Fälle:

1. $t_i = \varepsilon \wedge t_{i+1} \in \Sigma_t^+$ (z. B. `<root>`) [Root-case]
2. $t_i \in \Sigma_t^- \wedge t_{i+1} = \#$ (z. B. `</root>`) [End-case]
3. $t_i \in \Sigma_t^+ \wedge t_{i+1} \in \Sigma_t^-$ (z. B. `<a></a>`) [Leaf-case]
4. $t_i \in \Sigma_t^+ \wedge t_{i+1} \in \Sigma_t^+$ (z. B. `<a><b>`) [Child-case]
5. $t_i \in \Sigma_t^- \wedge t_{i+1} \in \Sigma_t^-$ (z. B. `</b></a>`) [Parent-case]
6. $t_i \in \Sigma_t^- \wedge t_{i+1} \in \Sigma_t^+$ (z. B. `</b><b>`) [Sibling-case]

Ohne Beweis.

Definition C.4 (deterministischer Parser-Kellerautomat (DKA)):
Sei $G = (N, T, P, S)$ eine reguläre Baumgrammatik.

Der Parser-Kellerautomat $K = (Q, \Sigma, \Gamma, \Psi, q_{start}, z)$ wird wie folgt konstruiert:

- q_{start}: Der Startzustand wird mit s_{start} bezeichnet.
- Q: Die Zustandsmenge Q enthält s_{start}. Weiterhin enthält sie für jedes $n \in N$ zwei Zustände: einen öffnenden (bezeichnet mit $state(tag^+)$) und einen schließenden (bezeichnet mit $state(tag^-)$). (Wie eingangs erläutert, werden Zustände zur Verarbeitung von Zeichendaten – `xsd:int` usw. – mit Blick auf eine kompaktere Darstellung in diesem Anhang nicht betrachtet.)
- Σ: Das Eingabealphabet enthält alle öffnenden und schließenden Tags sowie die Markierungszeichen: $\Sigma_t \cup \{\epsilon, \#\}$.
- Γ: Das Stack-Alphabet enthält die Zustände aller $dea \in DEA$ und das initiale Stack-Symbol z.
- Ψ: Die Produktionsregeln werden nach Algorithmus 2 konstruiert.

Algorithmus 2 Erzeugung der Transitionen

for all $p \in P$ **do**
  $tag \leftarrow term(p)$
  $dea \leftarrow dea(p)$
  $s_i \leftarrow state_{init}(dea)$
  $S_a \leftarrow states_{accept}(dea)$

  if $type(p) \in S$ **then**
    AddTransition: $(s_{start}, tag^+, z) \rightarrow (state(tag^+), [s_i, z])$; ▷ **Root-case**
    AddTransition: $(state(tag^-), \#, z) \rightarrow (state(tag^-), [\,])$; ▷ **End-case**
  end if

  for all $q \in states(dea)$ **do**

    if $q = s_i \wedge q \in S_a$ **then** ▷ **Leaf-case**
      AddTransition: $(state(tag^+), tag^-, q) \rightarrow (state(tag^-), [\,])$;
    end if

    if $q = s_i$ **then** ▷ **Child-case**
      for all $t \in out(q)$ **do**
        $type_t \leftarrow type(t)$
        $tag_t \leftarrow tag(type_t)$
        $q_{dest} \leftarrow dest(t)$
        AddTransition: $(state(tag^+),\ tag_t^+,\ q) \rightarrow (state(tag_t^+), [state_{init}(dea(type_t)),\ q_{dest}])$;
      end for
    end if

    if $q \in S_a$ **then** ▷ **Parent-case**
      for all $t \in in(q)$ **do**
        $type_t \leftarrow type(t)$
        $tag_t \leftarrow tag(type_t)$
        AddTransition: $(state(tag_t^-), tag^-, q) \rightarrow (state(tag^-), [\,])$;
      end for
    end if

    for all $t_{in} \in in(q)$ **do**
      for all $t_{out} \in out(q)$ **do** ▷ **Sibling-case**
        $type_{in} \leftarrow type(t_{in})$
        $type_{out} \leftarrow type(t_{out})$
        $tag_{out} \leftarrow tag(type_{out})$
        $tag_{in} \leftarrow tag(type_{in})$
        AddTransition: $(state(tag_{in}^-), tag_{out}^+, q) \rightarrow (state(tag_{out}^+), [state_{init}(dea(type_{out})), dest(t_{out})])$;
      end for
    end for
  end for
end for

Satz C.1: Der konstruierte Parser-Kellerautomat akzeptiert $x \Leftrightarrow x \in \mathcal{L}(G)$.

Beweis C.1:

Zeige: $x \in \mathcal{L}(G) \Rightarrow$ DKA akzeptiert

Wegen $x \in \mathcal{L}(G)$ ist x wohlgeformt und enthält nur Tags aus Σ_t. Zu jedem Tag t_i existiert ein Typ/Nichtterminal $type(t_i)$ und ein passender endlicher Automat $dea \in DEA$.

Betrachten wir zunächst nur die Stack-Operationen bei der Verarbeitung von öffnenden bzw. schließenden Tags: Beim Lesen eines öffnenden Tags schreibt der DKA ein zusätzliches Element auf den Stack. Analog wird beim Lesen eines schließenden Tags ein Stack-Element entfernt. Da x wohlgefomt ist, existieren genau so viele öffnende wie schließende Tags, und öffnende Tags erscheinen vor ihren schließenden Pendants. Der Stack enthält folglich nach dem Lesen von x nur das Symbol z, das den leeren Stack anzeigt.

Es bleibt zu zeigen, dass es für jedes t_i eine ausführbare Transition im DKA gibt. Wir beweisen dies mit einer vollständigen Induktion über die Tagsequenz:

1. **Induktionsanfang:** Wegen $x \in \mathcal{L}(G)$ gibt es vom Startzustand des DKA q_{start} eine Transition zu einem Zustand q, welche das öffnende Tag t_0 eines Top-Level-Elements (Root-Element) liest. Der Zustand q repräsentiert das gerade gelesene öffnende Tag t_0. Auf den Stack γ wird der Startzustand des zu t_0 gehörenden Automaten gelegt. (Root-Case, Zeile 7)

2. **Induktionsvoraussetzung:** Von x wurden die ersten i Tags gelesen. Der DKA befindet sich im Zustand q und besitzt den Stack γ mit dem obersten Element $top(\gamma)$. Nach Lesen des nächsten Tags t_{i+1} befindet sich der DKA im Zustand q'.

3. **Induktionsschritt:** Wir betrachten nun die beiden Tags t_i und t_{i+1} : Nach Lemma C.1 gilt genau einer der folgenden 4 Fälle:

   - $t_i, t_{i+1} \in \Sigma_t^+$ (Child-case, Zeilen 14–21):
     T_i und T_{i+1} seien die zu t_i und t_{i+1} gehörenden Typen/Nichtterminale und $dea(T_i)$ der zu T_i gehörende endliche Automat. Da $x \in \mathcal{L}(G)$, ist t_{i+1} ein gültiges Kind von t_i. Folglich gibt es in $dea(T_i)$ eine mit T_{i+1} beschriftete Transition, die vom Startzustand ausgeht. Im DKA existiert eine Transition $trans'$ vom Zustand $state(t_i)$ zum Zustand $state(t_{i+1})$. Diese kann ausgeführt werden, da t_{i+1} aus x gelesen wird und der Startzustand des zu t_i gehörenden Automaten dea zuvor als letztes Element auf den Stack geschrieben wurde.
   - $t_i \in \Sigma_t^+, t_{i+1} \in \Sigma_t^-$ (Leaf-case, Zeilen 11–13):
     Wegen $x \in \mathcal{L}(G)$ gehören t_i und t_{i+1} zum selben Element und damit auch

zum selben Typ $type(t_i)$. Das Inhaltsmodell von $type(t_i)$ darf leer sein. Folglich besitzt der dazugehörende DEA einen akzeptierenden Anfangszustand. Dadurch besitzt der DKA eine Transition $trans'$, die vom Zustand $state(t_i)$ zu $state(t_{i+1})$ geht und das schließende Tag t_{i+1} liest. Da der Zustand $state(t_i)$ des zuletzt gelesenen Tags als oberstes Symbol auf dem Stack liegt, kann $trans'$ ausgeführt werden.

- $t_i \in \Sigma_t^-, t_{i+1} \in \Sigma_t^-$ (Parent-case, Zeilen 22–28):
  Wegen $x \in \mathcal{L}(G)$ ist das zu t_i gehörige Element Vater von t_{i+1}. Folglich besitzt der zu $type(t_{i+1})$ gehörende DEA einen Endzustand mit mindestens einer eingehenden Transition, die das zu t_i gehörende Nichtterminal liest. Für jede dieser eingehenden Transitionen existiert im DKA eine Transition, die von $state(t_i)$ ausgeht, t_{i+1} liest und im Zustand $state(t_{i+1})$ mündet. Da der Zustand $state(t_i)$ des zuletzt gelesenen Tags als oberstes Symbol auf dem Stack liegt, kann $trans'$ ausgeführt werden.
- $t_i \in \Sigma_t^-, t_{i+1} \in \Sigma_t^+$ (Sibling-case, Zeilen 29–37):
  Die Tags t_i und t_{i+1} gehören zu zwei aufeinander folgenden Elementen, die dasselbe Vaterelement besitzen. Das Vaterelement ist nicht durch t_i und t_{i+1} identifizierbar; da jedoch $x \in \mathcal{L}(G)$ muss es existieren – wir bezeichnen es mit $parent$. Wir betrachten nun das Inhaltsmodell und den dazugehörenden endlichen Automaten $dea(parent)$. Wegen $x \in \mathcal{L}(G)$ muss es im $dea(parent)$ einen Zustand q_x geben, der sowohl eine eingehende als auch ausgehende Transition besitzt – in der eingehenden Transition wird $type(t_i)$ verarbeitet, in der ausgehenden $type(t_{i+1})$. Für jedes Paar von ein- und ausgehenden Transitionen von q_x existiert im DKA eine Transition. Folglich existiert auch eine Transition $trans'$ von $state(t_i)$ zu $state(t_{i+1})$, die das öffnende Tag t_{i+1} von der Eingabe und q_x vom Stack liest.
- $t_i \in \Sigma_t^-, t_{i+1} = \#$ (End-case, Zeile 8):
  Nach dem Lesen des letzten Tags von x ($i = n-1$) befindet sich der DKA in einem Zustand q und $top(x) = z$. Wegen x wohlgeformt, ist q der zum schließenden Wurzelelement gehörende Zustand. Hier existiert die finale Transition, die das Ende-Symbol # aus x und z vom Stack liest. Der DKA akzeptiert, da die Eingabe komplett gelesen wurde und der Stack leer ist.

□

Zeige: $x \notin \mathcal{L}(G) \Rightarrow$ DKA akzeptiert nicht

Da es trivial ist, x mit einem DKA auf Wohlgeformtheit zu untersuchen, sei im Folgenden angenommen, dass x wohlgeformt ist. Ebenfalls trivial ist die Überprüfung der Wurzel-Tags. Wir nehmen deshalb an, dass mindestens das erste Tag gelesen wurde. Der DKA befindet sich in einem Zustand q, das oberste Stack-Symbol ist $top(\gamma)$. Damit $x \notin \mathcal{L}(G)$ muss für x genau eine der beiden Bedingungen gelten:

1. $\exists t_i \in \{t_0, t_1, \ldots, t_{n-1}\}$ mit $t_i \notin \Sigma_t \cup \{\varepsilon, \#\}$: Für t_i kann keine geeignete Transition existieren, weil der DKA nur Tags aus $\Sigma_t \cup \{\epsilon, \#\}$ verarbeitet.

2. $\forall t_i \in \{t_0, t_1, \ldots, t_{n-1}\} : t_i \in \Sigma_t \cup \{\varepsilon, \#\}$. Folglich ist die Reihenfolge der Tags nicht gültig. O.B.d.A.: Sei t_{i+1} das auf t_i folgende Tag, welches die Gültigkeit von x verletzt. Damit der DKA nicht akzeptiert, darf es keine Transition für das Tag t_{i+1} geben. Wir zeigen dies durch einen Widerspruch:

   Annahme: Es existiert eine Transition $trans'$ von q nach q', die t_{i+1} von der Eingabe und $top(\gamma)$ vom Stack liest.

   Der Wert $top(\gamma)$ kennzeichnet einen DEA-Zustand in einem $dea \in DEA$. Da nach Annahme im DKA eine Transition zur Verarbeitung von t_{i+1} existiert, muss diese durch einen der vier Fälle im Algorithmus 2 entstanden sein. (Da wir davon ausgehen, dass das Wurzel-Element korrekt verarbeitet wird, kommen die beiden Fälle „Root-case“ und „End-case“ nicht in Frage.) Der Algorithmus wertet hierzu den DEA *dea* aus. Folglich akzeptiert *dea* das zu t_{i+1} gehörende Inhaltsmodell. Wegen $x \notin \mathcal{L}(G)$ ist dies ein Widerspruch. $\lightning$

□

Determinismus: Der DKA arbeitet deterministisch, weil jede RTG, die aus einem XML-Schema-Dokument konstruiert wurde, die so genannte Single-Type-Eigenschaft (vgl. [101]) aufweist. Dies bedeutet, dass beim Lesen des nächsten Tags stets klar ist, zu welchem Nichtterminal dieses Tag gehört. Auf diese Weise kann zu jedem Zeitpunkt der Dokumentverarbeitung in den DEA immer nur eine Transition ausgewählt werden. Da der konstruierte DKA die Ausführung der DEA simuliert, arbeitet auch der DKA deterministisch.

□

C.1 Inhaltsmodelle für einfache Datentypen

Die DKA-Konstruktion, wie wir sie bisher betrachtet haben, sieht noch keine Funktionalitäten für die Verarbeitung von einfachen Datentypen (bzw. Zeichendaten) vor. Denn in einer RBG gibt es kein Konzept für Inhaltsmodelle, die nicht zu Elementen abgeleitet werden. Entweder ein Inhaltsmodell ist leer (ϵ) oder nicht leer. Falls es nicht leer ist, werden die Nichtterminale des Inhaltsmodells zu Terminalen (also Tags) abgeleitet.

Um dennoch XML-Daten mit Zeichendaten verarbeiten zu können, hat der Autor bei der Implementierung des Algorithmus Inhaltsmodelle für einfache Datentypen als Spezialfall des leeren Inhaltsmodells behandelt. In der RBG sind in den Inhaltsmo-

dellen spezielle Nichtterminale für einfache Datentypen wie `xsd:int` erlaubt, welche nicht zu Elementen abgeleitet werden (vgl. Beispiel auf Seite 119). Für solche Nichtterminalsymbole werden im DKA spezielle „Simple-Type-Zustände“ vorgesehen – diese sorgen dafür, dass die von der Eingabe gelesenen Zeichendaten entsprechend den Regeln des einfachen Datentyps validiert werden.

Literaturverzeichnis

[1] Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey D.: *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988

[2] Akyildiz, Ian F.; Su, Weilian; Sankarasubramaniam, Yogesh; Cayirci, Erdal: Wireless sensor networks: a survey. In: *Comput. Networks* 38 (2002), Nr. 4, S. 393–422. http://dx.doi.org/http://dx.doi.org/10.1016/S1389-1286(01)00302-4. – DOI http://dx.doi.org/10.1016/S1389–1286(01)00302–4. – ISSN 1389–1286

[3] Amazon.com, Inc.: *Amazon E-Commerce Service*. http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl. Version: November 2005

[4] Arion, Andrei; Bonifati, Angela; Costa, Gianni; D'Aguanno, Sandra; Manolescu, Iona; Pugliese, Andrea: Efficient query evaluation over compressed XML data. In: *Proceedings of the Ninth International Conference on Extending Database Technology*, 2004, S. 200–218

[5] AT&T Labs: *XMill Download*. http://www.research.att.com/sw/tools/xmill/download.html. Version: März 2006

[6] Barr, Graham: *ASN.1 Encode/Decode library (Perl)*. http://search.cpan.org/~gbarr/Convert-ASN1-0.20/lib/Convert/ASN1.pod. Version: Oktober 2003

[7] Bayardo, Roberto J.; Gruhl, Daniel; Josifovski, Vanja; Myllymaki, Jussi: An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. In: *Proceedings of the 13th international conference on World Wide Web*, 2004, S. 345–354

[8] Birrell, Andrew D.; Nelson, Bruce J.: Implementing Remote Procedure Calls. In: *ACM Transactions on Computer Systems* 2 (1984), Nr. 1, S. 39–59. – ISSN 0734–2071

[9] Bormann, Carsten; Burmeister, Carsten; Degermark, Mikael; Fukushima, Hideaki; Hannu, Hans; Jonsson, Lars-Erik; Hakenberg, Rolf; Koren, Tmima; Le, Khiem; Liu, Zhigang; Martensson, Anton; Miyazaki, Akihiro; Svanbro, Krister; Wiebke, Thomas; Yoshimura, Takeshi; Zheng, Haihong: *RObust Header Compression (ROHC): Framework*

and four profiles: RTP, UDP, ESP, and uncompressed . RFC 3095 (Proposed Standard). http://www.ietf.org/rfc/rfc3095.txt. Version: Juli 2001 (Request for Comments). – Updated by RFC 3759

[10] Bouchou, Béatrice; Alves, Mirian Halfeld F.; Laurent, Dominique; Duarte, Denio: Extending Tree Automata to Model XML Validation Under Element and Attribute Constraints. In: *Proceedings of the International Conference on Enterprise Information Systems*. Angers, Frankreich, April 2003, S. 184–190

[11] Bradner, Scott: *The Internet Standards Process – Revision 3*. RFC 2026 (Best Current Practice). http://www.ietf.org/rfc/rfc2026.txt. Version: Oktober 1996 (Request for Comments)

[12] Bradner, Scott: *RFC 3979: Intellectual Property Rights in IETF Technology*. RFC 3979 (Best Current Practice). http://www.ietf.org/rfc/rfc3979.txt. Version: März 2005

[13] Brandt, Ylva: *Algorithmen zur Erzeugung optimierter Parserautomaten aus XML-Grammatiken*. Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, Diplomarbeit, September 2005

[14] Brüggemann-Klein, Anne; Wood, Derick: Caterpillars: A Context Specification Technique. In: *Markup Languages* 2 (2000), Nr. 1, S. 81–106

[15] Brüggemann-Klein, Anne; Wood, Derick: Balanced Context-Free Grammars, Hedge Grammars and Pushdown Caterpillar Automata. In: *Proceedings of Extreme Markup Languages*. Montréal, Québec, August 2004

[16] Burrows, Michael; Wheeler, David J.: *A Block Sorting Data Compression Algorithm*. Palo Alto, Californien, USA, Mai 1994. – Technical Report SRC 124, Digital Equipment Corporation

[17] Busatto, Giorgio; Lohrey, Markus; Maneth, Sebastian: Efficient Memory Representation of XML Documents. In: *Proceedings of the Tenth International Symposium on Database Programming Languages (DBPL 2005)*, 2005 (Lecture Notes in Computer Science Nr. 3774), S. 199–216

[18] Buschmann, Arndt; Werner, Christian; Schmidt, Stefan; Fischer, Stefan: Protokollunterstützung für SOAP Web Services auf mobilen Geräten. In: *PIK – Praxis der Informationsverarbeitung und Kommunikation, Themenheft Web Services* 27 (2004), Nr. 3

[19] Cabrera, Luis F.; Copeland, George; Feingold, Max; Freund, Tom; Johnson, Jim; Kaler, Chris; Klein, Johannes; Langworthy, David; Nadalin, Anthony; Orchard, David; Robinson, Ian; Storey, Tony; Thatte, Satish: *Web Services Atomic Transaction (WS-AtomicTransaction)*. http://specs.xmlsoap.org/ws/2004/10/wsat/wsat1104.pdf. Version: November 2004

[20] Casner, Stephen; Jacobson, Van: *Compressing IP/UDP/RTP Headers for Low-Speed Serial Links*. RFC 2508 (Proposed Standard). `http://www.ietf.org/rfc/rfc2508.txt`. Version: Februar 1999 (Request for Comments)

[21] Chawathe, Sudarshan S.; Rajaraman, Anand; Garcia-Molina, Hector; Widom, Jennifer: Change detection in hierarchically structured information. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996, S. 493–504

[22] Cheney, James: Compressing XML with Multiplexed Hierarchical PPM Models. In: *Data Compression Conference*, 2001, S. 163–173

[23] Cheney, James: An Empirical Evaluation of Simple DTD-Conscious Compression Techniques. In: *Proceedings of the Eighth International Workshop on the Web and Databases (WebDB 2005)*, 2005, S. 43–48

[24] Cheney, James: Tradeoffs in XML Database Compression. In: *Proceedings of the 2006 IEEE Data Compression Conference*, 2006

[25] Cheney, James: *xmlppm Project Homepage*. `http://sourceforge.net/projects/xmlppm/`. Version: Mai 2006

[26] Chiu, Kenneth; Govindaraju, Madhusudhan; Bramley, Randall: Investigating the Limits of SOAP Performance for Scientific Computing. In: *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*. Edinburgh, Scotland, Juli 2002, S. 246–254

[27] Clark, James; Makoto, Murata: *Definitive specification for RELAX NG using the XML syntax*. `http://www.relaxng.org/spec-20011203.html`. Version: Dezember 2001

[28] Combs, Gerald: *Ethereal Network Analyzer*. `http://www.ethereal.com`. Version: Mai 2006

[29] Comité Consultatif International Télégraphique et Téléphonique (CCITT): *Recommendation X.409: Message Handling Systems – Presentation Transfer Syntax and Notation*. Juni 1984

[30] Comon, Hubert; Dauchet, Max; Gilleron, Rémi; Jacquemard, Florent; Lugiez, Denis; Tison, Sophie; Tommasi, Marc: *Tree Automata Techniques and Applications*. `http://www.grappa.univ-lille3.fr/tata/tata.pdf`. Version: Oktober 2002

[31] Cormen, Thomas H.; Stein, Clifford; Rivest, Ronald L.; Leiserson, Charles E.: *Introduction to Algorithms*. 2. erweiterte Auflage. McGraw-Hill Higher Education, 2001. – ISBN 0070131511

[32] Crispin, Mark: *Internet Message Access Protocol – Version 4rev1.* RFC 3501 (Proposed Standard). http://www.ietf.org/rfc/rfc3501.txt. Version: März 2003 (Request for Comments)

[33] Crocker, Dave; Overell, Paul: *Augmented BNF for Syntax Specifications: ABNF.* RFC 4234 (Draft Standard). http://www.ietf.org/rfc/rfc4234.txt. Version: Oktober 2005 (Request for Comments)

[34] Davis, Doug: *The hidden impact of WS-Addressing on SOAP.* http://www-106.ibm.com/developerworks/library/ws-address.html. Version: April 2004

[35] Degermark, Mikael; Nordgren, Björn; Pink, Stephen: *IP Header Compression.* RFC 2507 (Proposed Standard). http://www.ietf.org/rfc/rfc2507.txt. Version: Februar 1999 (Request for Comments)

[36] Department of Foreign Affairs and Trade: Final Protocol and Detailed Service Regulations, London, England. In: *International Radiotelegraph Convention.* Canberra, Australia, Juli 1912

[37] Deutsch, Peter: *RFC 1952: GZIP file format specification version 4.3.* http://www.ietf.org/rfc/rfc1952.txt. Version: Mai 1996

[38] Du, Helen; Liu, Jeffrey: *Building a JMS Web service using SOAP over JMS and WebSphere Studio.* http://www-106.ibm.com/developerworks/websphere/library/techarticles/0402_du/0402_du.html. Version: Februar 2004

[39] Eberhart, Andreas; Fischer, Stefan: *Web Services.* Hanser, 2003

[40] Engan, Mathias; Casner, Stephen; Bormann, Carsten; Koren, Tmima: *IP Header Compression over PPP.* RFC 3544 (Proposed Standard). http://www.ietf.org/rfc/rfc3544.txt. Version: Juli 2003 (Request for Comments)

[41] Engelen, Robert A.: Constructing Finite State Automata for High Performance Web Services. In: *Proceedings of the International Symposium on Web Services and Applications (ISWS).* Las Vegas, Nevada, USA, 2004, S. 975–981

[42] Faller, Newton: An Adaptive System for Data Compression. In: *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, 1973, S. 593–597

[43] Fano, Robert M.: *Transmission of Information: A Statistical Theory of Communications.* Wiley, 1961

[44] Fielding, Roy T.; Gettys, Jim; Mogul, Jeffrey C.; Frystyk, Henrey; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim: *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.* http://www.ietf.org/rfc/rfc2616.txt. Version: Juni 1999

[45] Fourthought, Inc.: *XML and RDF toolkit for python: 4suite.* `http://4suite.org/`. Version: März 2006

[46] Fukuhara, Nobutaka: *Tag compressing method for XML.* Oktober 2004. – Patentschrift JP 2004/302868, Anmelder: NRI & NCC Co. Ltd.

[47] Gallager, Robert G.: Variations on a Theme by Huffman. In: *IEEE Transactions on Information Theory* 24 (1978), November, S. 668–674

[48] Giradot, Marc: *Efficient RPC mechanism using XML.* Januar 2003. – Patentschrift US 2003/023628, Anmelder: IBM

[49] Giradot, Marc; Sundaresan, Neelakantan: *System and method for schema-driven compression of extensible mark-up language (XML) documents.* April 2005. – Patentschrift US 6883137, Anmelder: IBM

[50] Girardot, Marc; Sundaresan, Neel: Millau: An Encoding Format for Effcient Representation and Exchange of XML over the Web. In: *9th International World Wide Web Conference.* Amsterdam, Netherlands, Mai 2000, S. 747–765

[51] Goldfarb, Charles F.: A Generalized Approach to Document Markup. In: *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, 1981, S. 68–73

[52] Gudgin, Martin; Combs, Harold; Justice, John; Kakivaya, Gopal; Lindsey, David; Orchard, David; Regnier, Alain; Schlimmer, Jeffrey; Simpson, Stacy; Tamura, Hiroshi; Wright, Don; Wolf, Kenny: *SOAP-over-UDP.* `http://msdn.microsoft.com/library/en-us/dnglobspec/html/soap-over-udp.asp`. Version: September 2004

[53] Hammerschall, Ulrike: *Verteilte Systeme und Anwendungen.* Pearson Studium, 2005

[54] Hamming, Richard W.: *Information und Codierung.* VCH, 1987

[55] Hariharan, Subramanian; Shankar, Priti: Compressing XML documents with finite state automata. In: *In Proceedings of the Tenth International Conference on Implementation and Application of Automata (CIAA 2005)*, 2005, S. 285–296

[56] Hariharan, Subramanian; Shankar, Priti: Evaluating the Role of Context in Syntax Directed Compression of XML Documents. In: *Proceedings of the 2006 IEEE Data Compression Conference*, 2006

[57] Heise, Werner; Quattrocchi, Pasquale: *Informations- und Codierungstheorie.* 3., neubearbeitete Auflage. Springer, 1995

[58] Hind, John R.; Lection, David B.: *Compression and decompression of markup files through xml entity declarations and references.* Oktober 2003. – Patentschrift US 6635088, Anmelder: IBM

[59] Huffman, David A.: A Method for the Construction of Minimum-Redundancy Codes. In: *Proceedings of the Institute of Radio Engineers* 40 (1952), September, Nr. 9, S. 1098–1101

[60] Imaura, Takeshi: *XML data encoding and decoding.* Januar 2003. – Patentschrift US 2003/018466, Anmelder: IBM

[61] Intelligent Compression Technologies, Inc.: *XML Xpress.* `http://www.ictcompress.com/products_xmlxpress.html`. Version: März 2006

[62] International Telecommunication Union (ITU): *Recommendation X.680: Abstract Syntax Notation One (ASN.1) – Specification of Basic Notation.* Juli 2002

[63] International Telecommunication Union (ITU): *Recommendation X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* Juli 2002

[64] International Telecommunication Union (ITU): *Recommendation X.691: Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).* Juli 2002

[65] International Telecommunication Union (ITU): *Recommendation H.323: Packet-based multimedia communications systems.* Juli 2003

[66] International Telecommunication Union (ITU): *Recommendation X.694: Information technology - ASN.1 encoding rules: Mapping W3C XML schema definitions into ASN.1.* Januar 2004

[67] International Telecommunication Union (ITU): *Recommendation X.891: Generic Applications of ASN.1 – Fast Infoset.* Mai 2005

[68] International Telecommunication Union (ITU): *Recommendation X.892: Generic Applications of ASN.1 – Fast Web Services.* Mai 2005

[69] Jäcker, Tobias: *Effiziente SOAP-Transportprotokolle.* Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, Diplomarbeit, Juni 2005

[70] Jehanne, Aymerick: *WBXML Compression Library.* `http://wbxmllib.sourceforge.net/`. Version: November 2003

[71] Kangasharju, Jaakko; Tarkoma, Sasu; Lindholm, Tancred: Xebu: A Binary Format with Schema-Based Optimizations for XML Data. In: *Proceedings of the International Conference on Web Information Systems Engeneering*. New York City, New York, USA, November 2005, S. 528–535

[72] Kangasharju, Jaakko; Tarkoma, Sasu; Lindholm, Tancred: *Xebu Implementierung*. `http://hoslab.cs.helsinki.fi/downloads/xebu/xebu-2.0.tar.gz`. Version: November 2005

[73] Khalaf, Rania; Leymann, Frank: On Web Services Aggregation. In: *Technologies for E-Services, 4th International Workshop*. Berlin, September 2003, S. 1–13

[74] Kieffer, John C.; Yang, En-hui: Grammar-Based Codes: A New Class of Universal Lossless Source Codes. In: *IEEE Transactions on Information Theory* 46 (2000)

[75] Kinno, Akira; Yukitomo, Hideki; Nakayama, Takehiro; Takeshita, Atsushi: Recursive Application of Structural Templates to Efficiently Compress Parsed XML. In: *Lecture Notes in Computer Science* 3579 (2005), Juli, S. 296–301

[76] Kiss, Roman: *SoapMSMQ Transport*. `http://www.codeproject.com/cs/webservices/SoapMSMQ.asp`. Version: Juli 2004

[77] Klensin, John: *Simple Mail Transfer Protocol*. RFC 2821 (Proposed Standard). `http://www.ietf.org/rfc/rfc2821.txt`. Version: April 2001 (Request for Comments)

[78] Kneuer, John M. R.: *Bringing America up to Speed: Devivering on Our Broadband Future Without Sacrificing Local Identity*. `http://commlaw.cua.edu/symposia/2006/2006_symposium_kneuer_presentation.pdf`. Version: März 2006. – Vortrag an der Catholic University Of America, Columbus School of Law, Washington, D.C.

[79] Koberstein, Jochen; Luttenberger, Norbert; Buschmann, Carsten; Fischer, Stefan: Shared Information Spaces for Small Devices: The SWARMS Software Concept. In: *Proceedings of the Workshop on Sensor Networks at Informatik 2004*. Berlin, Deutschland, September 2004

[80] Lee, Ju-Han: *Method of compressing XML data and method of decompressing compressed XML data*. November 2004. – Patentschrift US 2004/225754, Anmelder: Samsung Electronics Co. Ltd.

[81] Leighton, Greg; Diamond, Jim; Müldner, Tomasz: Axechop: a grammar-based compressor for XML. In: *Proceedings of the 2005 IEEE Data Compression Conference*, 2005, S. 467

[82] Leighton, Greg; Müldner, Tomasz; Diamond, Jim: Treechop: a tree-based query-able compressor for XML. In: *Proceedings of the Ninth Canadian Workshop on Information Theory*, 2005, S. 115–118

[83] Lelewer, Debra A.; Hirschberg, Daniel S.: Data Compression. In: *ACM Computing* 19 (1987), Nr. 3

[84] Levene, Mark; Wood, Peter: XML structure compression. In: *Proceedings of the Second International Workshop on Web Dynamics*, 2002

[85] Liefke, Hartmut; Suciu, Dan: XMill: an efficient compressor for XML data. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, 2000, S. 153–164

[86] Lin, Yongjing; Zhang, Youtao; Li, Quanzhong; Yang, Jun: Supporting efficient query processing on compressed XML files. In: *Proceedings of the 2005 ACM Symposium on Applied Computing*, 2005, S. 660–665

[87] Linnhoff-Popien, Claudia: *Corba – Kommunikation und Management*. Springer, 1998

[88] Lottor, Mark K.: *Internet Growth (1981-1991)*. RFC 1296 (Informational). `http://www.ietf.org/rfc/rfc1296.txt`. Version: Januar 1992 (Request for Comments)

[89] Maine, Steve: *SOAP-über-SMTP Implementierung für .NET*. `http://hyperthink.net/blog/content/binary/Net.Hyperthink.Samples.SoapSmtp.zip`. Version: August 2005

[90] Mansuripur, Masud: *Introduction to Information Theory*. Prentice-Hall, 1987

[91] Marahrens, Ingo: *Performance- und Effizienz-Analyse verschiedener RPC-Varianten*. Institut für Betriebssysteme und Rechnerverbund, TU Braunschweig, Diplomarbeit, August 2003

[92] Mathar, Rudolf: *Informationstheorie: diskrete Modelle und Verfahren*. Teubner, 1996

[93] Mende, Tobias: *Generierung validierender LL(1)-Parser für XML-Dokumente*. Institut für Informationssysteme, Universität zu Lübeck, Studienarbeit, 2004

[94] Microsoft Corporation: *Web Service Enhancements 2.0*. `http://www.microsoft.com/downloads/thankyou.aspx?FamilyID=d3c8f18b-7bbf-489d-90e1-e8d4147205b8&displaylang=en`. Version: Mai 2006

[95] Micrsoft Corporation: *.NET Framework Developer Center*. `http://msdn.microsoft.com/netframework/`. Version: Mai 2006

[96] Mildenberger, Otto: *Informationstheorie und Codierung.* Vieweg, 1990

[97] Min, Rex; Bhardwaj, Manish; Cho, Seong-Hwan; Shih, Eugene; Sinha, Amit; Wang, Alice; Chandrakasan, Anantha: Low-Power Wireless Sensor Networks. In: *VLSID '01: Proceedings of the The 14th International Conference on VLSI Design (VLSID '01).* Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7695–0831–6, S. 205

[98] Mouat, Adrian: *XML Diff and Patch Utilities*, Heriot-Watt University, Edinburgh, Scottland, CS4 Dissertation, Juni 2002. http://prdownloads.sourceforge.net/diffxml/dissertation.ps?download

[99] Mouat, Adrian: *diffxml Project Homepage.* http://diffxml.sourceforge.net/. Version: Mai 2006

[100] Mountain, Highland M.; Kopecky, Jacek; Williams, Stuart; Daniels, Glen; Mendelsohn, Noah: *SOAP Version 1.2 Email Binding.* http://www.w3.org/TR/2002/NOTE-soap12-email-20020626. Version: Juni 2002

[101] Murata, Makoto; Lee, Dongwon; Mani, Murali: Taxonomy of XML Schema Languages using Formal Language Theory. In: *Proceedings of Extreme Markup Languages.* Montreal, Canada, August 2001

[102] Myers, John G.; Rose, Marshall T.: *Post Office Protocol - Version 3.* RFC 1939 (Standard). http://www.ietf.org/rfc/rfc1939.txt. Version: Mai 1996 (Request for Comments)

[103] Nadalin, Anthony; Kaler, Chris; Hallam-Baker, Phillip; Monzillo, Ronald u. a.: *Web Service Security: SOAP Message Security 1.0 (WS-Security 2004).* http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf. Version: März 2004

[104] Nakayama, Kojiro; Ueda, Ryoichi: *Method and device for compressing structured documents.* Dezember 2004. – Patentschrift JP 2004/342029, Anmelder: Hitachi Ltd.

[105] Niedermeier, Ulrich; Heuer, Jörg; Hutter, Andreas; Stechele, Walter; Kaup, Andre: An MPEG-7 tool for compression and streaming of XML data. In: *Proceedings of the IEEE International Conference on Multimedia and Expo.* Lusanne, Switzerland, August 2002, S. 521–524

[106] OASIS Open: *Joining Oasis.* http://www.oasis-open.org/who/data_sheets/OASIS-join-datasht-A4-05-04-13.pdf

[107] OASIS Open: *OASIS Intellectual Property Rights (IPR) Policy.* http://www.oasis-open.org/who/intellectualproperty.php

[108] OASIS Open: *UDDI Version 3.0.2*. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm. Version: Oktober 2004

[109] OASIS Open: *Web Services Reliable Messaging (WS-Reliability 1.1)*. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf. Version: November 2004

[110] OASIS Open: *Management of Web Services (MOWS 1.0)*. http://docs.oasis-open.org/wsdm/2004/12/wsdm-mows-1.0.pdf. Version: März 2005

[111] OASIS Open: *Management Using Web Services (MUWS 1.0) Part 1*. http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part1-1.0.pdf. Version: März 2005

[112] OASIS Open: *Management Using Web Services (MUWS 1.0) Part 2*. http://docs.oasis-open.org/wsdm/2004/12/wsdm-muws-part2-1.0.pdf. Version: März 2005

[113] OASIS Open: *UDDI Homepage*. http://www.uddi.org/. Version: Mai 2006

[114] Object Management Group, Inc.: *Common Object Request Broker Architecture: Core Specification v3.0.3*. http://www.omg.org/cgi-bin/apps/doc?formal/04-03-12.pdf

[115] Objective Systems, Inc.: *ASN.1 to C, C++, C#, or Java compiler*. http://www.obj-sys.com/products_asn1c.shtml. Version: Februar 2006

[116] Open Mobile Alliance: *SyncML Specification, Version 1.01*. http://www.openmobilealliance.org/tech/affiliates/LicenseAgreement.asp?DocName=/syncml/spec1-0-1.zip. Version: Juni 2001

[117] Open Mobile Alliance: *Rights Expression Language, Version 1.0*. http://www.openmobilealliance.org/release_program/docs/drm/v1_0-20021104-c/oma-download-drmrel-v1_0-20020913-c.pdf. Version: September 2002

[118] O'Tuathail, Eamon; Rose, Marshall T.: *RFC 3288: Using the Simple Object Access Protocol (SOAP) in Blocks Extensible Exchange Protocol (BEEP)*. http://www.ietf.org/rfc/rfc3288.txt. Version: Juni 2002

[119] Petersen, Peter; Orto, David D.: *XML schema token extension for XML document compression*. Juni 2005. – Patentschrift US 2005/144556

[120] Postel, Jon: *User Datagram Protocol*. RFC 768 (Standard). http://www.ietf.org/rfc/rfc768.txt. Version: August 1980 (Request for Comments)

[121] Postel, Jon: *Internet Protocol*. RFC 791 (Standard). http://www.ietf.org/rfc/rfc791.txt. Version: September 1981 (Request for Comments)

[122] Postel, Jon: *Transmission Control Protocol*. RFC 793 (Standard). http://www.ietf.org/rfc/rfc793.txt. Version: September 1981 (Request for Comments). – Updated by RFC 3168

[123] Postel, Jon; Reynolds, Joyce K.: *File Transfer Protocol*. RFC 959 (Standard). http://www.ietf.org/rfc/rfc959.txt. Version: Oktober 1985 (Request for Comments)

[124] Raghunathan, Vijay; Schurgers, Curt; Park, Sung; Srivastava, Mani B.: Energy-Aware Wireless Microsensor Networks. In: *IEEE Signal Processing Magazine* 1 (2002), Nr. 2, S. 40–50

[125] Resnick, Pete: *Internet Message Format*. RFC 2822 (Proposed Standard). http://www.ietf.org/rfc/rfc2822.txt. Version: April 2001 (Request for Comments)

[126] Reuter, Florian; Luttenberger, Nobert: Cardinality Constraint Automata: A Core Technology for Efficient XML Schema-aware Parsers. In: *WWW2003 Conference Proceedings*, ACM Press, 2003

[127] Roman, Steven: *Coding and Information Theory*. Springer, 1992

[128] Rose, Marshall: *The Blocks Extensible Exchange Protocol Core*. RFC 3080 (Proposed Standard). http://www.ietf.org/rfc/rfc3080.txt. Version: März 2001 (Request for Comments)

[129] Sandoz, Paul: *Fast Infoset Implementierung*. http://fi.dev.java.net/files/documents/2634/21595/FastInfosetPackage_dist_1.0.1.zip. Version: November 2005

[130] Sandoz, Paul; Pericas-Geertsen, Santiago; Kawaguchi, Kohuske; Hadley, Marc; Pelegri-Llopart, Eduardo: *Fast Web Services*. http://java.sun.com/developer/technicalArticles/WebServices/fastWS/. Version: August 2003

[131] Sandoz, Paul; Triglia, Alessando; Pericas-Geertsen, Santiago: *Fast Infoset*. http://java.sun.com/developer/technicalArticles/xml/fastinfoset/. Version: Juni 2004

[132] Sayood, Khalid: *Introduction to data compression (2nd ed.)*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2000. – ISBN 1–55860–558–4

[133] Schöning, Uwe: *Theoretische Informatik kurzgefaßt*. 3. Auflage. Spektrum, 1997

[134] Segoufin, Luc; Vianu, Victor: Validating Streaming XML Documents. In: *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA : ACM Press, 2002, S. 53–64

[135] Shannon, Claude E.: A mathematical theory of communication. In: *Bell System Technical Journal* 20 (1948), Juli and Oktober, S. 379–423, 623–656

[136] Shannon, Colleen; Moore, David; Claffy, Kimberly C.: Beyond folklore: observations on fragmented traffic. In: *IEEE/ACM Trans. Netw.* 10 (2002), Nr. 6, S. 709–720. – ISSN 1063–6692

[137] Shepler, Spencer; Callaghan, Brent; Robinson, David; Thurlow, Robert; Beame, Carl; Eisler, Michael; Noveck, David: *Network File System (NFS) version 4 Protocol*. RFC 3530 (Proposed Standard). `http://www.ietf.org/rfc/rfc3530.txt`. Version: April 2003 (Request for Comments)

[138] Streit, Achim; Erwin, Dietmar; Lippert, Thomas; Mallmann, Daniel; Menday, Roger; Rambadt, Michael; Riedel, Morris; Romberg, Mathilde; Schuller, Bernd; Wieder, Philipp: UNICORE - From Project Results to Production Grids. In: *Grid Computing and New Frontiers of High Performance Processing*, Elsevier, 2005

[139] Sun Microsystems, Inc.: *RFC 1014: XDR: External Data Representation standard*. Juni 1987

[140] Sun Microsystems, Inc.: *RFC 1057: RPC: Remote Procedure Call Protocol specification: Version 2*. Juni 1988

[141] Sun Microsystems, Inc.: *Java Message Service, Version 1.1*. `http://java.sun.com/products/jms/docs.html`. Version: März 2002

[142] Sun Microsystems, Inc.: *Sun Relax NG Converter*. `http://www.sun.com/software/xml/developers/relaxngconverter/`. Version: Mai 2006

[143] Sun Microsystems, Inc.: *Trang: Multi-format schema converter based on RELAX NG*. `http://www.thaiopensource.com/relaxng/trang.html`. Version: Mai 2006

[144] T-Mobile Deutschland GmbH: *Preisliste für Laufzeitverträge von T-Mobile*. `http://www.t-mobile.de/downloads/tarife/laufzeittarife_04_06.pdf`. Version: April 2006

[145] Tanenbaum, Andrew; van Steen, Marten: *Verteilte Systeme – Grundlagen und Paradigmen*. Pearson Studium, 2003

[146] The Apache Software Foundation: *Apache Axis Homepage*. `http://ws.apache.org/axis/`. Version: Mai 2006

[147] The Apache Software Foundation: *Xerces Homepage*. `http://xerces.apache.org/`. Version: Mai 2006

[148] Tian, Min; Voigt, Thiemo; Naumowicz, T.; Ritter, Hartmut; Schiller, Jochen H.: Performance considerations for mobile web services. In: *Computer Communications* 27 (2004), Nr. 11, S. 1097–1105

[149] Tolani, Pankaj; Haritsa, Jayant R.: XGRIND: A Query-friendly XML Compressor. In: *Proceedings of the International Conference on Data Engineering*. San Jose, California, USA, Februar 2002, S. 225–234

[150] Toman, Vojtech: Syntactical Compression of XML Data. In: *Proceedings of the International Conference on Advanced Information Systems Engineering*. Riga, Lettland, Juni 2004

[151] Unicode, Inc.: *Unicode Homepage*. `http://www.unicode.org/`. Version: Mai 2006

[152] UserLand Software, Inc: *XML-RPC Homepage*. `http://www.xmlrpc.com/`. Version: Mai 2006

[153] Verma, Kunal; Sivashanmugam, Kaarthik; Sheth, Amit; Patil, Abhijit; Oundhakar, Swapna; Miller, John: METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. In: *Journal of Information Technology and Management, Special Issue on Universal Global Integration* 6 (2005), Nr. 1, S. 17–39

[154] Walkin, Lev: *Open Source ASN.1 Compiler*. `http://lionet.info/asn1c/`. Version: März 2006

[155] Wang, Hongzhi; Li, Jianzhong; Luo, Jizhou; He, Zhenying: XCpaqs: compression of XML document with XPath query support. In: *Proceedings of the 2004 International Conference on Information Technology: Coding and Computing*, 2004, S. 354–358

[156] Werner, Christian: *Patentanmeldung – Aktenzeichen: DE 10 2005 056 122.5*. November 2005. – Deutsches Patent- und Markenamt, München

[157] Werner, Christian; Buschmann, Carsten; Brandt, Ylva; Fischer, Stefan: Compressing SOAP Messages by using Pushdown Automata. In: *Proceedings of the IEEE International Conference on Web Services*. Chicago, USA, September 2006

[158] Werner, Christian; Buschmann, Carsten; Fischer, Stefan: Compressing SOAP Messages by using Differential Encoding. In: *Proceedings of the IEEE International Conference on Web Services*. San Diego, USA, Juli 2004

[159] Werner, Christian; Buschmann, Carsten; Fischer, Stefan: WSDL-Driven SOAP Compression. In: *International Journal of Web Services Research* 2 (2005), Nr. 1

[160] Werner, Christian; Buschmann, Carsten; Jäcker, Tobias; Fischer, Stefan: Enhanced Transport Bindings for Efficient SOAP Messaging. In: *Proceedings of the IEEE International Conference on Web Services*. Orlando, USA, Juli 2005

[161] Werner, Christian; Buschmann, Carsten; Jäcker, Tobias; Fischer, Stefan: Bandwidth and Latency Considerations for Efficient SOAP Messaging. In: *International Journal of Web Services Research* 3 (2006), Nr. 1

[162] Werner, Christian; Fischer, Stefan: Web Service Technology – Architecture and Standardization. In: Studer, Rudi (Hrsg.); Abecker, Andreas (Hrsg.); Grimm, Stephan (Hrsg.): *Semantic Web Enabled Web Services*. Springer, 2006, S. 11–29. – (Veröffentlichung in Vorbereitung)

[163] Wilson, Hervey: *WSE 2.0 transport supporting SOAP-over-UDP v0.1*. http://www.dynamic-cast.com/downloads/WseTransports_0_1.zip. Version: Mai 2004

[164] Wireless Application Protocol Forum: *Wireless Markup Language, Version 2.0*. http://www.openmobilealliance.org/release_program/docs/Browsing/V2_1-20050614-C/WAP-238-WML-20010911-a.pdf. Version: September 2001

[165] World Wide Web Consortium (W3C): *Recommendation: Extensible Markup Language (XML) 1.0*. http://www.w3.org/TR/1998/REC-xml-19980210. Version: Februar 1998

[166] World Wide Web Consortium (W3C): *Member Submission: WAP Binary XML Content Format*. http://www.w3.org/1999/06/NOTE-wbxml-19990624/. Version: Juni 1999

[167] World Wide Web Consortium (W3C): *Recommendation: Namespaces in XML*. http://www.w3.org/TR/1999/REC-xml-names-19990114/. Version: Januar 1999

[168] World Wide Web Consortium (W3C): *Recommendation: XML Path Language (XPath)*. http://www.w3.org/TR/1999/REC-xpath-19991116.html. Version: November 1999

[169] World Wide Web Consortium (W3C): *Recommendation: XSL Transformations (XSLT)*. http://www.w3.org/TR/1999/REC-xslt-19991116. Version: November 1999

[170] World Wide Web Consortium (W3C): *Member Submission: Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl. Version: März 2001

[171] World Wide Web Consortium (W3C): *Recommendation: XML Information Set*. http://www.w3.org/TR/xml-infoset/. Version: Oktober 2001

[172] World Wide Web Consortium (W3C): *Recommendation: XML Encryption Syntax and Processing*. http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/. Version: Dezember 2002

[173] World Wide Web Consortium (W3C): *Recommendation: XML-Signature Syntax and Processing*. http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/. Version: Februar 2002

[174] World Wide Web Consortium (W3C) ; Champion, Michael (Hrsg.); Ferris, Chris (Hrsg.); Newcomer, Eric (Hrsg.); Orchard, David (Hrsg.): *Working Group Note: Web Services Architecture*. http://www.w3.org/TR/2002/WD-ws-arch-20021114/. Version: November 2002

[175] World Wide Web Consortium (W3C) ; Haas, Hugo (Hrsg.); Orchard, David (Hrsg.): *Working Group Note: Web Services Architecture Usage Scenarios*. http://www.w3.org/TR/2002/WD-ws-arch-scenarios-20020730/. Version: Juli 2002

[176] World Wide Web Consortium (W3C): *Recommendation: SOAP Version 1.2 Part 1: Messaging Framework*. http://www.w3.org/TR/soap12-part1/. Version: Juni 2003

[177] World Wide Web Consortium (W3C): *Recommendation: SOAP Version 1.2 Part 2: Adjuncts*. http://www.w3.org/TR/soap12-part2/. Version: Juni 2003

[178] World Wide Web Consortium (W3C): *Recommendation: XML Schema Part 1 – Structures, Second Edition*. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. Version: Oktober 2004

[179] World Wide Web Consortium (W3C): *Recommendation: XML Schema Part 2 – Datatypes, Second Edition*. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/. Version: Oktober 2004

[180] World Wide Web Consortium (W3C) ; Weitzner, Daniel J. (Hrsg.): *W3C Patent Policy*. http://www.w3.org/Consortium/Patent-Policy-20040205/. Version: Februar 2004

[181] World Wide Web Consortium (W3C) ; Booth, David (Hrsg.); Haas, Hugo (Hrsg.); McCabe, Francis (Hrsg.); Newcomer, Eric (Hrsg.); Champion, Michael (Hrsg.); Ferris, Chris (Hrsg.); Orchard, David (Hrsg.): *Working Group Note: Web Services Architecture*. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. Version: Februar 2004

[182] World Wide Web Consortium (W3C) ; Goldman, Oliver (Hrsg.); Berjon, Robin (Hrsg.); Bournez, Carine (Hrsg.): *Charter of the Efficient XML Interchange Working Group*. http://www.w3.org/2005/09/exi-charter-final.html. Version: November 2005

[183] World Wide Web Consortium (W3C) ; Cokus, Mike (Hrsg.); Pericas-Geertsen, Santiago (Hrsg.): *Working Group Note: XML Binary Characterization Use Cases*. http://www.w3.org/TR/2005/NOTE-xbc-use-cases-20050331/. Version: März 2005

[184] World Wide Web Consortium (W3C): *World Wide Web Consortium Process Document*. http://www.w3.org/2005/10/Process-20051014/. Version: Oktober 2005

[185] World Wide Web Consortium (W3C): *Proposed Recommendation: Web Services Addressing (WS-Addressing)*. http://www.w3.org/TR/2006/PR-ws-addr-core-20060321/. Version: März 2006

[186] World Wide Web Consortium (W3C): *XML Interchange Working Group – Status Update*. http://www.w3.org/XML/EXI/public-status-200603.html. Version: März 2006

[187] XML:DB Initiative: *Working Draft: XML Update Language (XUpdate)*. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html. Version: September 2000

[188] Xu, Yingyue; Qi, Hairong: Distributed computing paradigms for collaborative signal and information processing in sensor networks. In: *J. Parallel Distrib. Comput.* 64 (2004), Nr. 8, S. 945–959. http://dx.doi.org/http://dx.doi.org/10.1016/j.jpdc.2004.04.002. – DOI http://dx.doi.org/10.1016/j.jpdc.2004.04.002. – ISSN 0743–7315

[189] Zhang, Kaizhong; Wang, Jason T. L.; Shasha, Dennis: On the editing distance between undirected acyclic graphs and related problems. In: Galil, Z. (Hrsg.); Ukkonen, E. (Hrsg.): *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*. Espoo, Finnland : Springer-Verlag, Berlin, 1995, S. 395–407

[190] Ziv, Jacob; Lempel, Abraham: A Universal Algorithm for Sequential Data Compression. In: *IEEE Transactions on Information Theory* 23 (1977), Nr. 3, S. 337–343

Index

Lebenslauf des Autors

Christian Werner		geboren am 15.06.1977 in Salzgitter-Bad
Schulen:	1984 – 1988	Grundschule Am Ziesberg in Salzgitter-Bad
	1988 – 1990	Orientierungsstufe Am Eikel in Salzgitter-Bad
	1990 – 1997	Gymnasium Salzgitter-Bad
Wehrdienst:		befreit wegen einer Allergie gegen Wespengift
Hochschul-studium:	1997 – 2002	Studium an der mathematisch-naturwissenschaftlichen Fakultät der Humboldt-Universität zu Berlin Studiengang: Informatik
Berufliche Entwicklung:	Februar 1998 bis Dezember 2001	IT-Dienstleistungen für die Steuerberatungs- und Wirtschaftprüfungsgesellschaft mbH Tober & Co., Berlin
	August 1998 bis Dezember 2002	Führung eines eigenen IT-Unternehmens „SkyTec – Software-Entwicklung, professionelle EDV-Systeme, Internet-Dienstleistungen“
	August 2002 bis Dezember 2004	Wissenschaftlicher Mitarbeiter am Forschungszentrum L3S in Hannover sowie Forschungs- und Lehrtätigkeit am Institut für Betriebssysteme und Rechnerverbund an der Technischen Universität Braunschweig
	November bis Dezember 2004	Lehrauftrag an der Universität zu Lübeck „Übung: Betriebs- und Kommunikationssysteme“
	seit Januar 2005	Wissenschaftlicher Mitarbeiter am Institut für Telematik der Universität zu Lübeck