

Verteilte Systeme

Vorlesungsskript

Clemens Westerkamp, Heinz-Josef Eikerling
Laborbereich Technische Informatik
Fakultät für Ingenieurwissenschaften und Informatik der
Hochschule Osnabrück

Inhaltsverzeichnis

<i>Abbildungsverzeichnis:</i>	5
<i>Verzeichnis der Beispiele:</i>	7
1 Einleitung	8
1.1 Skriptinhalt und Bezug zu Vorlesungen	8
1.2 Überblick	8
1.3 Quellenverzeichnis	9
2 Einführung	10
2.1 Begriffsdefinitionen	10
2.2 Verteilte Systeme	10
2.3 Verteilte Anwendungen	11
2.4 Kommunikation	11
2.5 Implementierung Verteilter Anwendungen	11
2.6 Architekturen Verteilter Anwendungen	12
2.6.1 Client-Server	12
2.6.2 Peer-to-Peer	13
3 Das OSI-Modell und die TCP/IP Netzwerkarchitektur	14
3.1 Grundlagen / Protokolle	14
3.2 Das OSI-Referenzmodell	14
3.3 Einbettung des OSI-Modells	16
3.4 Dienstgüte	17
3.5 Grundlagen TCP/IP-Protokollfamilie	17
3.5.1 Vergleich zwischen OSI-Modell und TCP/IP	18
3.5.2 IP-Adressen und deren Netzwerk-/Hostdarstellung	18
3.5.3 Transport Layer (UDP/TCP)	18
3.5.4 Betriebssysteme: Unterstützung für Verteilte Systeme	19
3.5.5 Client-/Servermodell	19
3.5.6 Synchrone vs. Asynchrone Aufrufe	20
3.5.7 Port-Nummern	21
3.5.8 Allgemeine Anforderungen an die Programmierung verteilter Systeme	23
4 Datenorientierte Programmierung verteilter Systeme	24
4.1 Allgemeine Funktion von Sockets	24
4.2 Socketadressen und Netzwerkverbindungen	24
4.3 Funktionen zur Socketprogrammierung	25
4.3.1 Verbindungsmanagement – Übersicht	25
4.3.2 Darstellung von Socketadressen in der Programmierung verteilter Systeme	26
4.3.3 Socketfunktionen (<code>socket</code> , <code>bind</code> , <code>listen</code> , <code>accept</code> , <code>connect</code> , <code>close</code>)	27
4.3.4 Datenübertragung (<code>read</code> , <code>write</code> , <code>send</code> , <code>sendto</code> , <code>recv</code> , <code>recvfrom</code>)	30
4.3.5 Gefahr von Buffer-Overflows	31
4.3.6 Konvertierung zwischen Host- und Netzwerksdarstellung (<code>htons</code> etc.)	31
4.3.7 Operationen auf Speicherbereichen (<code>memcpy</code> etc.)	31
4.3.8 Adressumwandlungen	32
4.4 Beispiele	32
4.4.1 <code>tcp_server</code> – Mehrstufiger TCP-Server	32

4.4.2 TCP-Client	34
4.4.3 Beispiel <code>daytime.c</code>	36
4.4.4 Besonderheiten bei verbindungslosen Socketanwendungen mit UDP	38
4.4.5 Beispiel <code>udp_server</code> Echo-Server mit UDP	38
4.5 Ein/Ausgabe-Multiplexing	42
4.6 Hilfs- und Systemfunktionen (gethostbyname etc.)	43
4.7 Socket-Programmierung unter Windows	43
4.8 Socket-Programmierung unter Java	44
4.8.1 Java TCP-Sockets	44
4.8.2 Beispiel <code>JavaTCP_server</code> - einfacher TCP-Echo-Server unter Java	44
4.8.3 Beispiel <code>JavaTCP_client</code> - einfacher TCP-Echo-Client unter Java	45
4.8.4 Beispiel <code>JavaTCP_server</code> - mehrstufiger TCP-Echo-Server unter Java	46
4.8.5 Socket-Programmierung mit dem .net-Framework unter C#	46
4.8.6 Web-Sockets	46
4.9 Stärken / Schwächen datenorientierter Programmierung verteilter Systeme mit Sockets	47
5 Funktionsorientierte Programmierung vert. Systeme - Remote Procedure Calls (RPCs)	48
5.1 Grundlagen	48
5.2 Programmentwicklung mit SUN ONC	49
5.2.1 Beispielprogramm <code>ls</code> (list directory contents) Verzeichnisliste lokale Version	49
5.2.2 Beispielprogramm <code>rls</code> für Low-Level-RPC-Programmierung	51
5.2.3 Bedeutung von program, version und procedure	52
5.2.4 Beispiel <code>math</code> - entfernter Aufruf mathematischer Funktionen	55
5.2.5 Beispielprogramm <code>RPC_No_Seg_Fault</code> für High-Level-RPC-Programmierung	57
5.3 Low-Level-RPC-Programmierung	58
5.3.1 Der RPC-Protokollcompiler	58
5.3.2 Die RPC Definitionssprache RPCL	59
5.3.3 Grundregeln für Datentypen an der Netzwerkschnittstelle	62
5.3.4 Notwendige Low-Level-Funktionen	62
5.3.5 Beispiel <code>rls</code> Remote Listing Service - Low-Level-RPC-Programmierung	63
5.4 RPC-Programmierung unter Solaris und Windows	67
5.5 Asynchrone RPC-Verarbeitung	67
5.5.1 One-Way-Verfahren	68
5.5.2 Beispiel zur One-Way-RPC-Programmierung	68
5.5.3 Follow-Up-Verfahren	70
5.6 Ausblick zu RPCs	71
5.7 Stärken / Schwächen funktionsorientierter Programmierung verteilter Systeme mit RPCs	71
6 Objektorientierte Programmierung verteilter Systeme	72
6.1 Architekturbetrachtungen	72
6.2 CORBA	73
6.2.1 Grundlagen von CORBA	74
6.2.2 Die Interface Definition Language (IDL) von CORBA	76
6.3 Objektorientierte Programmierung verteilter Systeme mit CORBA in Beispielen	80
6.3.1 Beispiel <code>account</code> zur lokalen Kontoverwaltung	80
6.3.2 Beispiel <code>account_loc</code> mit lokaler CORBA-Nutzung	81
6.3.3 Beispiel <code>account_fileref</code> mit Client/Server und Referenzaustausch über IOR-Datei	84
6.3.4 Beispiel <code>account_nameserv</code> mit Naming Service und variablen Listen strukturierter Daten	88

6.4 Einige Hinweise beim Einsatz von CORBA	93
6.4.1 Nutzung von CORBA-Datentypen	93
6.5 Weitere Bestandteile der CORBA-Architektur	93
6.6 Praktischer Einsatz von CORBA	96
6.7 Bewertung von CORBA	98
6.8 Ausblick zu CORBA	98
6.9 Entfernter Methodenaufruf unter Java RMI – Remote Method Invocation)	98
6.9.1 Serialisieren von Objekten	99
6.9.2 Einführung in Java RMI	99
6.9.3 Packages für RMI-Aufrufe	100
6.9.4 Beispiel TimeServer	102
6.9.5 Java RMI Daemon	104
6.9.6 Java RMI und Sicherheit	104
6.9.7 Performance und Portabilität	104
6.10 Interaktion zwischen Java und CORBA – Beispiel nsd_sequence_jclient	104
6.10.1 CORBA-Unterstützung in Java-Versionen	105
6.10.2 RMI-IIOP, Java IDL und das Mapping	105
6.10.3 Java-Client JClient für das C++-CORBA-Beispiel account_nameserv	106
6.11 Nutzung von Callbacks mit Java RMI	107
6.12 Bewertung von Java RMI	109
7 Weborientierte Programmierung verteilter Systeme	110
7.1.1 Aufbau der HTTP-Requests und Responses	112
7.1.2 Adressierung von Objekten in URIs und URLs	112
7.2 Statische Webseiten	113
7.3 Dynamische Webseiten (server-seitige Programmierung mit Java Servlets, php etc.)	113
7.4 Java Servlets	114
7.4.1 Beispiel RequestHeader - Servlet zur Ausgabe des Response-Headers	116
7.4.2 Beispiel FormExample - Servlet zum Verarbeiten eines Formulars	117
7.4.3 Beispiel SessionExample - Servlet zum Verwalten einer Sitzung	120
7.4.4 Beispiel CookieExample - Servlet zur dauerhaften Speicherung des Sitzungskontextes	122
7.4.5 Servlets und Datenbanken	124
7.4.6 Sicherheit bei Web-Anwendungen	125
7.4.7 Server-seitige Web-Applikationen mit Java Server Pages (JSP)	126
7.4.8 PHP	129
7.4.9 Serverseitige Web-Programmierung und das MVC-Muster	131
7.5 Aktive Web-Seiten (clientseitige Programmierung mit JavaScript etc.)	131
7.5.1 JavaScript	131
7.5.2 Java-Applets	133
7.5.3 Beispiel Fibo – Applets mit Java-RMI	134
7.6 Zusammenfassung Web-Applikationen	137
7.7 Web Services	138
7.7.1 XML (Extended Markup Language)	139
7.7.2 Validierung von XML-Daten	140
7.7.3 XML Schema	140
7.7.4 XSL - Automatisierte Verarbeitung von XML-Daten in XML-Parsern	143
7.7.5 Extensible Stylesheet Language	146
7.7.6 AJAX - Ajax: Erstellung von Rich Internet Clients mit JavaScript & XML	147
7.7.7 JavaScript Object Notation - JSON	153
7.7.8 Websockets	153

7.8 Web Services	155
7.8.1 SOAP für den Nachrichtenaustausch	156
7.8.2 WSDL – Web Service Description Language	160
7.8.3 RESTful Web Services	165
7.8.4 UDDI	166
7.8.5 Nachteile von Web Services	166
7.8.6 Vergleich zwischen CORBA und Web Services	166
7.9 Ausblick	167
Anhang	168
Anhang A.1 Index	168

Abbildungsverzeichnis:

Abbildung 1: Optionen der Realisierung Verteilter Anwendungen	12
Abbildung 2: Geschichtetes Netzwerk-System	14
Abbildung 3: OSI-7-Schichtenmodell.....	15
Abbildung 4: Header-Einbettung im OSI-7-Schichtenmodell	16
Abbildung 5: Vergleich zwischen TCP/IP-Protokollfamilie und OSI-Modell	18
Abbildung 6: Varianten synchroner und asynchroner Client-/Server-Kommunikation [Beng04].....	21
Abbildung 7: Well-Known-Ports eines Unix-Systems in der Datei <code>/etc/services</code>	21
Abbildung 8: Sockets (engl. Steckdose) in der TCP/IP-Protokollfamilie	24
Abbildung 9: Web-Server-Verbindung	25
Abbildung 10: Socketfunktionen bei verbindungsorientierter Datenübertragung (TCP).....	27
Abbildung 11: Zusammenhang zwischen Verbindungsparametern und Socketfunktionen bei TCP.....	30
Abbildung 12: Socketfunktionen bei verbindungsloser Datenübertragung mit UDP	38
Abbildung 13: Lokaler und entfernter Prozeduraufruf.....	49
Abbildung 14: Ablauf beim entfernten Prozederaufruf	51
Abbildung 15: Dreifach verschachtelte Strukturierung in SUN ONC	52
Abbildung 16: Zeitlicher Ablauf des Prozederaufrufs	53
Abbildung 17: Automatische Dateigenerierung bei Low-Level-RPC-Programmierung	58
Abbildung 18: Durchlaufen verschiedener Software-Abschnitte beim RPC-Aufruf	67
Abbildung 19: Ablauf des Oneway- (links) und des Follow-Up-Verfahrens (rechts) zur Bearbeitung asynchroner RPC-Anfragen	68
Abbildung 20: Von monolithischen zu verteilten Dreischichtenarchitekturen	72
Abbildung 21: Anwendungsintegration zwischen heterogenen Anwendungen (Beispiel)	73
Abbildung 22: Verteilte Objekte kommunizieren über den Object Request Broker(ORB)	74
Abbildung 23: ORB- Kommunikation und statische/dynamische Methodenaufrufe.....	75
Abbildung 24: Proxys und Servants in CORBA (Beispiel)	75
Abbildung 25: Erzeugung neuer Objekte beim Server durch new-Aufruf bei Client	75
Abbildung 26: Ablauf bei Server-Startup, Registrierung und Nutzung von Serverobjekten	76
Abbildung 27: Interface Definition Language (IDL) in verschiedenen Programmiersprachen	76
Abbildung 28: Interface Definition Language (IDL) in verschiedenen Programmiersprachen	77
Abbildung 29: Aufbau einer IDL mit Interfaces in einem Modul	77
Abbildung 30: Konstanten, Type-Definitionen, Ausnahmen und Interfaces in einer IDL	78
Abbildung 31: IDL-/C++-Datentypen, Wertebereiche und Speicherplatzbedarf	78
Abbildung 32: Interface mit Vererbung	79
Abbildung 33: Mehrfachvererbung von Interfaces analog zu C++	79
Abbildung 34: Aufwand des Marshalings verschiedener Typen [Slama/Garbis/Russell, 1999].....	80
Abbildung 35: Vom Entwickler zu schreibende (links) und daraus erzeugte Dateien (Mitte).....	82
Abbildung 36: Aufbau einer IIOP IOR	84
Abbildung 37: Vollständige CORBA-Architektur.....	94
Abbildung 38: Aktivierung von Objekten.....	94
Abbildung 39: Kontextverwaltung durch den CORBA-Naming-Service	95

Abbildung 40: CORBA-Architektur	95
Abbildung 41: CORBA-Sicherheit: IIOP over SSL.....	96
Abbildung 42: Agrarmeteorologischer Arbeitsplatz	96
Abbildung 43: CORBA-bsierende Architektur (Quelle siehe Fußnote)	97
Abbildung 44: Beispiel eines CORBA-basierenden Enterprise Service Bus [Dunk08]	97
Abbildung 45: Einordnung von Java RMI in die Middleware-Welt.....	99
Abbildung 46: Vollständige RMI-Architektur	100
Abbildung 47: Vererbungshierarchie der Server-Klassen für RMI-Aufrufe.....	101
Abbildung 48: Reihenfolge bei RMI-Aufrufen.....	102
Abbildung 49: Anwendungsintegration von C++/Java-Lösungen mittels IIOP	105
Abbildung 50: Prinzip der Zusammenarbeit zwischen Java und CORBA.....	105
Abbildung 51: Klassenbild des Calculator-Callback-Beispiels.....	108
Abbildung 52: Sequenzdiagramm des Ablaufs beim Registrieren von zwei Clients	108
Abbildung 53: Sequenzdiagramm zur Addition und Benachrichtigung registrierter Clients.....	109
Abbildung 54: Historischer Verlauf der Einführung verschiedener Web-Technologien.....	110
Abbildung 55: Einordnung von HTTP im TCP/IP-Schichtenmodell.....	111
Abbildung 56: Ablauf eines Request-/Response-Vorgangs bei Client und Server	111
Abbildung 57: Prinzipieller Aufbau von HTTP-Requests und -Responses	112
Abbildung 58: Evolution von XHTML.....	113
Abbildung 59: 4-Schichten-Architektur für Web-Anwendungen z.B. Java Enterprise Edition.....	114
Abbildung 60: Lebenszyklus von Servlets	115
Abbildung 61: Einfaches HTTP-Servlet	116
Abbildung 62: Antwort von RequestHeader auf Anfrage aus Web-Browser	116
Abbildung 63: HTML-Formular	117
Abbildung 64: HTML-Code und Browser-Darstellung eines einfachen Formulars	118
Abbildung 65: Do-Post-Routine und Ausgabe im Browser	119
Abbildung 66: Eingabefeld in Formular.....	120
Abbildung 67: Personalisierte Abfrage mit Radio-Buttons.....	121
Abbildung 68: Tabelle mit Statistik	122
Abbildung 69: Meldung bei ungültiger Session.....	122
Abbildung 70: Ablauf einer Cookie-basierten Sitzungsverwaltung	123
Abbildung 71: Varianten des Datenbankzugriffs für Servlets.....	124
Abbildung 72: Digest based Authentication.....	126
Abbildung 73: HTTPS-Verschlüsselung mittels Secure Socket Layer (SSL).....	126
Abbildung 74: JSP: Implizite Objekte und Scoping.....	127
Abbildung 75: Lebenszyklus einer Java Server Page (JSP)	128
Abbildung 76: Anzeige eines Java-Applets mit InputArea im Web-Browser.....	131
Abbildung 77: Zugriff auf HTML-Dokument.....	132
Abbildung 78: Anzeige eines Java-Applets im Web-Browser.....	134
Abbildung 79: Anzeige eines Java-Applets mit InputArea im Web-Browser.....	137
Abbildung 80: Web Services repräsentieren den service-orientierten Ansatz von Middleware	138
Abbildung 81: Service-basierter Ansatz für Web Services	138
Abbildung 82: Rollen bei Web Services	139
Abbildung 83: Datentypen in XML Schemata.....	142
Abbildung 84: Nutzung eines XML-Parsers zur Überprüfung der Einhaltung von Schemata	143
Abbildung 85: XML-Verarbeitung mittels SAX-Parser und Event-Handler	144
Abbildung 86: XML-Verarbeitung mit DOM-Parser und vollständiger Speicherung	145
Abbildung 87: XML-Verarbeitung mit StAX in JAXP	145
Abbildung 88: Transformation/Formatierung von XML mit XSL (Extensible Stylesheet Language) .	146
Abbildung 89: Beispiel der Transformation mittels XSLT	147
Abbildung 90: Formular mit AJAX-Nutzung	148
Abbildung 91: Formular mit AJAX-Nutzung (ausgefüllt)	151
Abbildung 92: Debugging von AJAX im Internet Explorer	152
Abbildung 93: Genereller Ablauf bei AJAX-Anwendungen	152
Abbildung 94: Erweiterung einer HTTP-Verbindung für Websocket-Kommunikation.....	154
Abbildung 95: Nutzung von Websockets in Javascript.....	154

Abbildung 96: Verallgemeinerte Grundidee der Web Services	155
Abbildung 97: Anwendungsbeispiel für mehrere kombinierte Web Services	156
Abbildung 98: SOAP-Kommunikationsmodell	157
Abbildung 99: Aufbau einer SOAP-Nachricht ohne Anhang	157
Abbildung 100: SOAP-Nachrichten mittels SOAP Engine Axis.....	159
Abbildung 101: Handler-Konzept der SOAP Engine Axis	160
Abbildung 102: Struktur eines WSDL-Dokuments	161
Abbildung 103: Temperaturkonvertier-Web-Service-WSDL in Netbeans	162
Abbildung 104: Web Services: Web Service Architecture (W3C)	165

Verzeichnis der Beispiele:

Die folgende Übersicht zeigt, welche Beispiele in den Kapiteln 4 bis 7 verwendet werden. Studierende können schnell selbst ermitteln, welches Beispiel als Grundlage zu einer Praktikumsaufgabe dienen sollte. Im Zweifel sollte man nachfragen.

Kapitel/ Abschnitt	Beispielname bzw. -ordner	Beispieldatei
4	tcp_server_tcp_client daytime udp_server check_host winsocket JavaSockets CS_Sockets	kap4beispiele.zip
5	ls (list directory contents) rls math RPC_No_Seg_Fault (High-Level-RPC) One-Way-RPC-Programmierung	kap5beispiele.zip
6	account_loc account_fileref account_nameserv account_seq time_server.rmi nsd_sequence_jclient	kap6beispiele.zip
7	CookieExample CookieRecExample FormExample RequestHeader SessionEndExample SessionExample	kap7beispiele.zip

1 Einleitung

Dieses Skript stellt den Inhalt der Vorlesung Verteilte Systeme dar, die für die Bachelor-Informatik-Studiengänge an der Hochschule Osnabrück angeboten wird. Außerdem ist das Fach Anpassmodul für Studierende außerhalb der Informatik im ersten Semester des Informatik-Masters Verteilte und Mobile Anwendungen. Das Skript besteht aus verschiedenen Abschnitten, die aus etlichen Lehrbüchern (in den einzelnen Kapiteln und im Quellenverzeichnis referenziert) zusammengestellt wurden. Außerdem haben die Kollegen Timmer und Eikerling wertvolle Hinweise und Ergänzungen geliefert. Bedanken möchte ich mich an dieser Stelle bei allen Kollegen, Mitarbeitern und Studierenden, die durch Anregungen und kritische Fragen dazu beigetragen haben, dass das Skript aktueller und verständlicher wurde.

1.1 Skriptinhalt und Bezug zu Vorlesungen

Ziel der Veranstaltung und des Skriptes ist es, folgende Fähigkeiten bei den Studierenden aufzubauen:

- Anforderungen verteilter Anwendungen analysieren können
- Grundprinzipien der Netzwerkdienste und wichtige Abläufe kennen
- Auswahl der Netzwerkdienste entsprechend der Anforderungen durchführen können
- Entwickeln von Standardbeispielen für wichtige Netzwerkdienste und -anwendungen
- Die Vorlesung setzt die Kenntnisse der Vorlesungen Grundlagen der Programmierung (C), Objektorientierte Programmierung (C++) und Kommunikationsnetze sowie einige Java-Grundkenntnisse voraus.

Der Skripttext stellt also den aktuellen Stand der Vorlesung im Wintersemester dar. Die Beispiele sind an die Linux-Version 14.04 (Ubuntu) im Pool des Laborbereichs Technische Informatik angepasst.

1.2 Überblick

Die Programmierung verteilter Systeme beschäftigt sich mit der Realisierung von verteilten Anwendungen auf der Basis von Datennetzen typischerweise entsprechend einem Client/Server-Prinzip. Beispiele solcher Anwendungen sind

- Standard-Internetanwendungen wie WWW, FTP, Telnet, E-Mail
- Datenbanknutzung / Informationssysteme (häufig auch mobil genutzt)
- Multimedia-Streaming-Server und Clients (vertiefend behandelt im Masterfach Distributed Multimedia Applications)
- Netzwerkmanagement

Die überwiegende Mehrzahl der Netzwerkanwendungen basiert heute auf der TCP/IP-Protokollfamilie. Normalerweise muss, abgesehen von der richtigen Protokollauswahl und –nutzung, keine Rücksicht auf die Abläufe innerhalb der TCP/UDP/ICMP- und IP-Schichten genommen werden. Auch Themen wie Routing und Namensauflösung werden als vorhanden bzw. bekannt vorausgesetzt (siehe Veranstaltung Kommunikationsnetze). Weiterhin geht man davon aus, dass die Parametrierung der Protokolle den Anforderungen für Standardanwendungen entspricht und nicht geändert werden muss.

Für die Programmierung verteilter Systeme haben sich einige Verfahren entwickelt, die innerhalb dieser Vorlesung behandelt werden. Nach der Art der Programmierung verteilter Systeme können fünf Themen-schwerpunkte unterschieden werden, die in den Kapiteln 4 bis 7 dargestellt werden:

Kapitel	Programmieransatz für verteilte Systeme	wichtige Stichworte	Programmiersprachen
4	datenorientiert	Sockets	C, C++, Java, C#
5	funktionsorientiert	RPC	C
6	objektorientiert	CORBA, Java RMI	C++, Java, C#
7	weborientiert	HTTP, HTML, XML, Javascript, Java, SOAP, Java Servlets, Java Server Pages, AJAX, Web Sockets	C, C++, Java, (C#)

Wegen der grundlegenden Bedeutung werden die Socket- und RPC-Programmierung als typische Beispiele ausführlich dargestellt. Alle Abschnitte werden durch einfache Beispielprogramme (in OSCA)

und Anwendungsbeispiele aus der Praxis ergänzt. Zur Vorlesung gibt es ein Praktikum, das aufbauend auf Beispielprogrammen die Entwicklung eigener Lösungen zum Inhalt hat.

1.3 Quellenverzeichnis

Die Quellen- und Literaturangaben sind in alphabetischer Reihenfolge sortiert. In den Einleitungsabschnitten der Kapitel wird beschrieben, welche Literaturstellen jeweils relevant sind.

- [Abts10] Abts, Dietmar: *Client-/Server-Programmierung mit Java*, Vieweg-Verlag Braunschweig/Wiesbaden, 3. Auflage 2010
- [Baue09] Architekturen für Web-Anwendungen. Eine praxisbezogene Konstruktions-Systematik, G. Bauer
- [Beng14] Bengel, Günther: *Verteilte Systeme, Client-Server-Computing für Studenten und Praktiker*, Vieweg-Verlag Braunschweig/Wiesbaden, 3. Auflage 2014.
- [Bloo94] Bloomer, John: *Power Programming with RPC*, O'Reilly, 1994
- [CDK11] Coulouris, George; Dollimore, Jean; Kindberg, Tim: *Distributed Systems - Concepts and Design* 5. Auflage, Addison Wesley/Pearson Education, 2011 <http://www.cdk5.net/wp>
- [Come02] Comer, Douglas E.: Computer Networks And Internets Sixth Edition, 2014. ISBN 0133587932/9780133587937, Prentice Hall 2014
- [DEF08] Dunkel, J., Eberhart, A., Fischer, S., Kleiner, C., Koschel, A.: *Software-Architekturen für verteilte Systeme*. Hanser Fachbuchverlag, 2008
- [DGH03] Sch. Dustdar, H. Gall und M. Hauswirth: *Software Architekturen für Verteilte Systeme*, 264 Seiten, Springer 2003, ISBN: 3540430881
- [Hamm05] Hammerschall , u.: *Verteilte Systeme und Anwendungen : Architekturkonzepte, Standards und Middleware-Technologien*, München [u.a.] : Pearson Studium, 2005
- [Java14a] All about Sockets <http://docs.oracle.com/javase/tutorial/networking/sockets/index.html>
(zuletzt besucht am 23.09.2014)
- [Java14b] Java RMI Tutorial <http://docs.oracle.com/javase/tutorial/rmi/index.html>
(zuletzt besucht am 23.09.2014)
- [Oech14] Oechsle, Reiner (Herausg.): *Parallele und verteilte Anwendungen in Java*, Hanser-Verlag, 4. Auflage, 2014.
- [Poll04] Pollakowski, Martin: Grundkurs Socketprogrammierung mit C unter Linux, Vieweg 2004
- [PRP06] Distributed systems architecture: a middleware approach / Arno Puder; Kay Römer; Frank Pilhofer, Amsterdam [u.a.]: Elsevier/Morgan Kaufmann, 2006.
- [ScSp07] Schill, A.; Springer, T.: *Verteilte Systeme, Grundlagen und Basistechnologien* Springer-Verlag, 2007.
- [Tane08] Tanenbaum , A.: *Verteilte Systeme : Prinzipien und Paradigmen*, 2., aktualisierte Aufl. - München [u.a.]: Pearson Studium, 2008

2 Einführung

2.1 Begriffsdefinitionen

Verteilung ist heute der essentielle Aspekt der Informations- und Kommunikationstechnik. Dies ist durch die zahlreichen Vorteile motiviert, die mit einer Verteilung speziell von Funktionen über die einzelnen Komponenten eines Verteilten Systems einhergehen:

- Verbund von Ressourcen: per Verteilung können Hardware-Komponenten (z.B. Drucker) in einem Verbund von unterschiedlichen Parteien genutzt werden.
- Verbund von Daten: Daten können in einem Verteilten System weitgehend ohne Aufwand ausgetauscht und gemeinsam genutzt werden. Die Aggregation von Daten zur Erzeugung von Information ist auf dieser Basis erst möglich.
- Verbund von Funktionen: bestimmte Funktionen können in dem Verteilten System von speziellen Komponenten übernommen und den anderen Komponenten des Verbunds zur Verfügung gestellt werden. Die Replikation von Funktionen wird damit unterbunden werden.

Dabei ergibt sich zunächst das Problem, genau zu definieren, was mit Verteilung gemeint ist. Leider gibt es diesbezüglich keine verbindliche Definition.

So wird in [Tane08] definiert: „*Ein verteiltes System ist eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen.*“ Etwas stärker gehen Coulouris et al. in [CDK11] auf die technische Realisierung von Verteilten Systemen ein: „*Bei einem verteilten System arbeiten Komponenten zusammen, die – sich auf vernetzten Computern befinden und – die ihre Aktionen durch Austausch von Nachrichten koordinieren.*“ Die Komponenten können dabei alle gleichartig sein (homogenes System) oder verschieden (heterogen).

Beide Definitionen kann man vielleicht so zusammenfassen, dass ein Verteiltes System als solches verstanden wird, bei dem verschiedene Komponenten bestehend aus Prozessoren mit lokalen Speichereinheiten kooperierend zusammenarbeiten, um ein gemeinsames Ziel zu erreichen. Auf den Komponenten laufen dabei Prozesse; über entsprechende Schnittstellen kann auf diese zugegriffen und deren Zustand verändert werden.

Aus dieser Definition folgen einige Charakteristiken Verteilter Systeme, die bei deren Realisierung durchgängig betrachtet werden müssen:

- *Nebenläufigkeit*: die Prozesse auf den einzelnen Komponenten laufen unabhängig und damit nebenläufig voneinander ab.
- *Ausfallsicherheit*: die Komponenten des Verteilten Systems arbeiten in der Regel unabhängig voneinander; sie können damit unabhängig voneinander ausfallen.
- *Synchronität*: in Verteilten Systemen gibt es keine globale Zeitmessung. Die Synchronisation von Ereignissen ist damit nicht trivial.

Mit dem oben Gesagten kann man aber auch Systeme ausschließen, die gemäß der obigen Definition kein verteiltes System darstellen:

- Hardware-Systeme, die mehrere Prozessoren (sog. Cores) beinhalten.
- Systeme, die über gemeinsamen Speicher kommunizieren.
- Lose gekoppelte Systeme, die zwar über ein Kommunikationsnetzwerk miteinander interagieren, aber nicht im Sinne eines gemeinsamen Ziels zusammenarbeiten.

Dabei ist nicht ausgeschlossen, dass bei diesen Systemen verwandte Konzepte im Hinblick auf Datentransfer und Datenverarbeitung zum Einsatz kommen.

2.2 Verteilte Systeme

Nicht unwichtig ist die Unterscheidung zwischen Verteilten *Systemen* und Verteilten *Anwendungen*.

In einem Verteilten System stellen die in der Definition genannten Komponenten die Kommunikationsknoten dar, die über ein Kommunikationsnetzwerk miteinander verbunden sind. Die Kommunikation dieser Knoten untereinander muss auf Basis einer einheitlichen Sprache stattfinden.

Für das bekannteste Verteilte System, das Internet, stellen die einzelnen Rechner die Knoten dar. Die ‘Sprache’ für den Datenaustausch ist durch den standardisierten TCP/IP-Protokollstack gegeben. Prinzipiell ist hier auch die Verwendung anderer Protokolle denkbar, jedoch ist Verwendung von TCP/IP heutzutage dominant.

Im Rahmen dieser Veranstaltung wird demzufolge auch ein Schwerpunkt auf die Internet-Technologien gelegt, die Verteilten Systemen zugrunde liegen. Nicht unerwähnt bleiben sollte allerdings, dass auch in anderen Domänen Verteilte Systeme anzutreffen sind. Eingebettete Verteilte Systeme finden sich z.B. in der Automobiltechnik (z.B. CAN-Bus zur Vernetzung von verschiedenen Sensor- und Aktor-Einheiten im Automobil), in der Gebäudetechnik (z.B. EIB, KNX oder LCN zur Vernetzung von Elektrogeräten) oder im Mobilfunk (GSM ist z.B. ein verteiltes System).

2.3 Verteilte Anwendungen

Verteilte (Software-) Anwendungen nutzen Verteilte Systeme als Infrastrukturen, um den Nutzern der Systeme gewisse Funktionen anzubieten. Die exakte Trennung von Systemen und Anwendungen wird speziell im täglichen Gebrauch der entsprechenden Termini nicht immer klar, weshalb dieses hier einmal herausgearbeitet werden soll.

Bei derartigen Anwendungen verteilt sich die zugrundeliegende Software auf mehrere physikalische Knoten des Verteilten Systems. Die Interaktion zwischen den verteilt im Netz installierte Software-Komponenten der Anwendung erfolgt über das Verteilte System.

Im Hinblick auf das Internet als Verteiltes System stellt z.B. das World Wide Web (WWW) eine der Hauptanwendungen dar. Das WWW ist aber keinesfalls mit dem Internet gleichzusetzen, auch wenn dies heute vielfach geschieht, sondern repräsentiert ein über das Internet implementiertes Hyper-Text-System und besteht aus verschiedenen Software-Komponenten (Web-Client und -Server), die auf Basis von stringent definierten Datenformaten und Anwendungsprotokollen realisiert sind.

2.4 Kommunikation

Bei Verteilten Systemen und Anwendungen sind grundsätzlich zwei Kommunikationsmodelle in Abhängigkeit der Handhabung des Nachrichtenaustauschs zwischen dem aufrufenden (Sender einer Nachricht) und aufgerufenen (Empfänger der Nachricht) Kommunikationspartner zu unterscheiden:

- *Synchrone Kommunikation*: in diesem Modell wird der Sender solange blockiert, bis er die Antwort auf die Nachricht vom Empfänger erhält.
- *Asynchrone Kommunikation*: in diesem Modell wird der Sender nicht blockiert, sondern kann nebenläufig zur Verarbeitung der Nachricht durch den Empfänger weiterarbeiten.

Es ist zu beachten, dass die Kommunikationsmodelle von Verteilter Anwendung und darunter liegendem System nicht übereinstimmen müssen. So kann per *Polling* (mehr hierzu in 4.6) eine synchrone Anwendung auf Basis eines konzeptionell asynchronen Systems realisiert werden.

2.5 Implementierung Verteilter Anwendungen

Standardanwendungen erfordern geringen Aufwand, sind aber inflexibel. Die Realisierung eigener Verteilter Anwendungen kann auf verschiedenen Ebenen ansetzen. Die Anwendung kann direkt auf Basis der durch das Verteilte System bereit gestellten Primitiven zur Kommunikation implementiert werden. Solche Primitiven behandeln u.a. das Verbindungsmanagement (Verbindungsauf und -abbau) und die Fehlerbehandlung, allerdings auf recht niedriger Ebene und ohne Abstraktionen hinsichtlich technischer Details. Man spricht in diesem Fall von *Netzwerkprogrammierung*.

Der Aufwand hierfür ist relativ hoch. In Abbildung 1 ist dies auf der rechten Seite veranschaulicht.

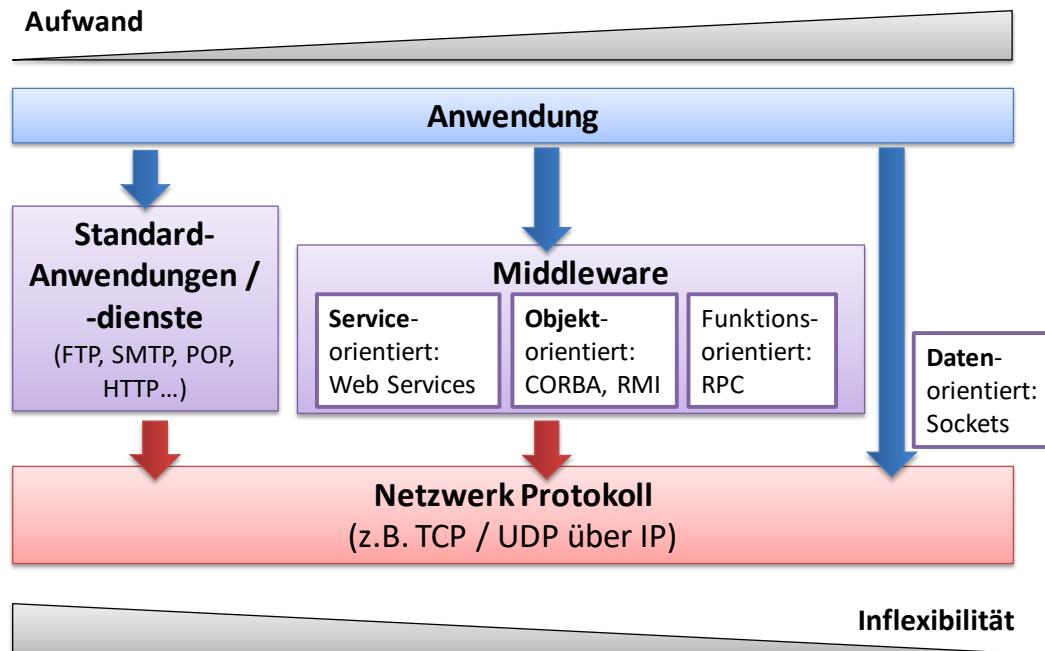


Abbildung 1: Optionen der Realisierung Verteilter Anwendungen

Ökonomischer ist der Einsatz von Middleware, die gewisse Abstraktionen einführt und bestimmte Funktionen in geeigneter Weise kapselt. Ziel derartiger Abstraktionen ist es, gewisse technische Parameter konfigurierbar und austauschbar zu machen. Ein solcher Parameter kann z.B. das Netzwerk-Protokoll sein. Im Idealfall kann das Protokoll per Konfiguration innerhalb der Middleware geändert werden, ohne dass dies Auswirkungen auf die Anwendung oder gar eine Revision der Software zur Folge hat.

Der Aufwand sinkt weiter, wenn auf einer bereits existierenden Anwendung aufgebaut werden kann, d.h. eine solche Anwendung eine Abstraktion von der Netzwerkprogrammierung vorgibt. Viele Internet-basierte Anwendungen (z.B. zum Content Management) bieten Schnittstellen an, um auf deren Funktionalitäten in proprietären Anwendungen und Diensten zugreifen zu können.

Der geringere Aufwand bei der Realisierung von Anwendungen geht mit höheren Abstraktionsebenen aber auch mit einem Verlust an Optimierungspotential einher, so dass hier in der Regel eine Abwägung erfolgen muss. Tendenziell wird heute allerdings der Einsatz von Middleware bei der Realisierung von Verteilten Anwendungen favorisiert. Die Mehrzahl der verteilten Systeme wird heute mit Middleware-Ansätzen entwickelt, die in den Kapiteln 4 bis 7 behandelt werden.

2.6 Architekturen Verteilter Anwendungen

Der Realisierung Verteilter Systeme werden unterschiedliche Architekturmodelle zugrunde gelegt.

2.6.1 Client-Server

In diesem Modell steht der kurzlebige (*transiente*) Prozess des Clients dem langlebigen (*persistenten*) Server-Prozess gegenüber. Der Client führt gewisse Funktionen aus und terminiert dann. Im Gegensatz dazu steht der Server kontinuierlich und auch mehreren Clients zur Verfügung. Das Client-Server-Modell ist das dominierende Paradigma bei Verteilten Anwendungen und wird im Folgenden noch an verschiedenen Stellen referenziert werden.

In einem solchen Modell kann es mehrere Server geben, die unterschiedliche Funktionen übernehmen. Hinsichtlich der Zuordnung von Funktionen zu diesen Server-Komponenten definiert man sog. mehrschichtige Architekturen (*Multi-Tier-Architekturen*). Als Beispiel kann man z.B. eine Portalanwendung betrachten, die typischerweise in Funktionsblöcke zur Handhabung

- der Präsentation der Daten / Information (z.B. HTML-Seiten),
- der Anwendungslogik (z.B. Auswertung von über Formulare eingegebene Daten [PHP]) und
- der Datenhaltung (z.B. Speicherung der Daten)

untergliedert ist.

Diese Funktionsblöcke können nun in unterschiedlicher Weise über die Komponenten des Verteilten Systems verteilt werden:

- in einer 2-Tier-Architektur wird die Präsentation der Daten dem Client und die Datenhaltung dem Server zugeordnet. Die Anwendungslogik wird zwischen Client und Server gleichermaßen aufgeteilt.
- Bei einer 3-Tier-Architektur wird die Präsentation der Daten dem Client und die Datenhaltung dem Server zugeordnet. Man spricht hier dann von Client-Tier und Server-Tier. Die Anwendungslogik wird allerdings nicht aufgeteilt, sondern einem sog. Middle-Tier zugeordnet.
- 4-Tier- und n-Tier-Architekturen: in vielen Fällen ist es sinnvoll, den Middle-Tier weiter aufzuteilen. Bei einem Portal mit vielen Nutzern und zahlreichen unterschiedlichen Anwendungen können die einzelnen Anwendungen auf verschiedenen Servern installiert werden. Diesen Servern ist ein weiterer Server vorgeschaltet, der je nach angefragter Anwendung zum jeweiligen nachgeschalteten Server weiterleitet. Dieses Prinzip lässt sich auf n Ebenen erweitern.

2.6.2 Peer-to-Peer

In einem Peer-to-Peer-Modell sind die Prozesse gleichberechtigt. Komponenten (*Peers*) können sowohl bereitstellen als auch konsumieren. Ein solcher Ansatz wird z.B. bei älteren Tauschbörsen verwendet. In solchen Architekturen wird stets eine Funktion benötigt, die es gestattet, über ein Identifikationsmerkmal nach einem Peer zu suchen, der die mit dem Merkmal assoziierte Funktion anbietet.

In reinen Peer-to-Peer-Architekturen gibt es keine zentrale, koordinierende Komponente. Allerdings gibt es auch Tauschbörsen (das frühere Napster), die ohne eine zentrale Instanz nicht funktionieren.

3 Das OSI-Modell und die TCP/IP Netzwerkarchitektur

Auch wenn in vorhergehenden Vorlesungen (insbes. Kommunikationsnetze) bereits ausführlich auf verschiedene Netzwerkarchitekturen eingegangen wurde, soll im Skript eine kurze Zusammenfassung aus Sicht der Programmierung verteilter Systeme dargestellt werden. In der Vorlesung wird dieser Abschnitt sehr kurz behandelt, da diese Vorkenntnisse vorausgesetzt werden. Vertiefende Literatur findet sich in [Beng14], [StFeRu04] und [Stev94].

3.1 Grundlagen / Protokolle

Ein geschichtetes Netzwerk-System ist grundsätzlich folgendermaßen aufgebaut:

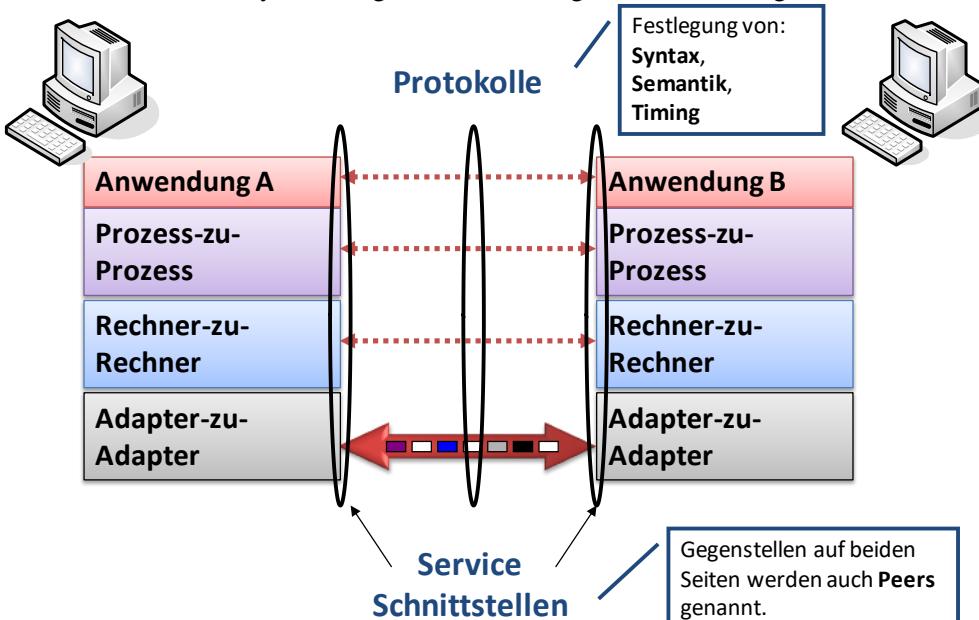


Abbildung 2: Geschichtetes Netzwerk-System

Das Schichtenkonzept lässt sich mit der Situation vergleichen, in der zwei Personen (Peers), die unterschiedliche **Sprachen** (z.B. Deutsch und Spanisch) sprechen, eine Nachricht austauschen wollen. Kommunikation kann nun zum Beispiel dadurch etabliert werden, in dem sich die beiden Personen auf eine **Zwischensprache** (Englisch) einigen und jeweils einen entsprechenden **Übersetzer** engagieren (Deutsch/Englisch bzw. Spanisch/Englisch). Die Übersetzer als Peers kommunizieren dann z.B. über FAX-Dokumente miteinander, was dann der physikalischen Schicht in einem geschichteten Netzwerksystem entsprechen würde.

3.2 Das OSI-Referenzmodell

Folgende Standardisierungsgremien entwickeln Standards und Protokolle für die Programmierung verteilter Systeme:

- Die IETF (Internet Engineering Task Force) konzentriert sich auf TCP/IP und hat viele RFCs (engl. Request For Comment) entwickelt. Sie beschäftigen sich mit den Protokollen der TCP/IP-Familie und darauf basierenden Anwendungen.
- OSI/ISO (Open Systems Interconnection/International Organisation for Standardization) hat neben C und C++, JPEG und MPEG eine allgemeine Darstellung von Mechanismen und Protokollen zur Netzwerkkommunikation entwickelt. Die OSI erkennt die IETF-RFCs an.

Um dem Nutzer und dem Entwickler von verteilten Anwendungen das Leben einfach zu machen, hat die OSI 1981 ein Referenzmodell standardisiert, mit dem Netzwerkarchitekturen beschrieben werden können. Es hat sieben Schichten, die jeweils bestimmte Funktionen mit Hilfe von Diensten der darunter liegenden Schicht realisieren. Einige grundlegende Eigenschaften sollen hier dargestellt und aus Sicht der Programmierung verteilter Systeme zu kommentiert werden:

- Die Schichten sind so entworfen worden, dass der Datenaustausch zwischen ihnen möglichst gering ist und die Funktionen möglichst gut nach außen abgeschirmt werden.
- Die Schicht n einer Maschine kommuniziert mit der (auf gleicher Ebene) Schicht n einer anderen Maschine. Die Regeln für diese Kommunikation werden Protokoll der Schicht n genannt.
- Daten werden nicht direkt von einer Schicht n einer Maschine zur selben Schicht einer anderen Maschine übertragen. Jede Schicht gibt Daten und Steuerinformationen an die direkt darunter liegende Schicht weiter. Die unterste Schicht überträgt die Daten über das physikalische Medium zum entfernten Rechner.
- Schichten und Protokolle bilden die Netzwerkarchitektur.

Das folgende Bild aus [Come02] zeigt den prinzipiellen Aufbau des OSI-Modells und die Behandlung der Schichten in den Vorlesungen an der Hochschule Osnabrück:

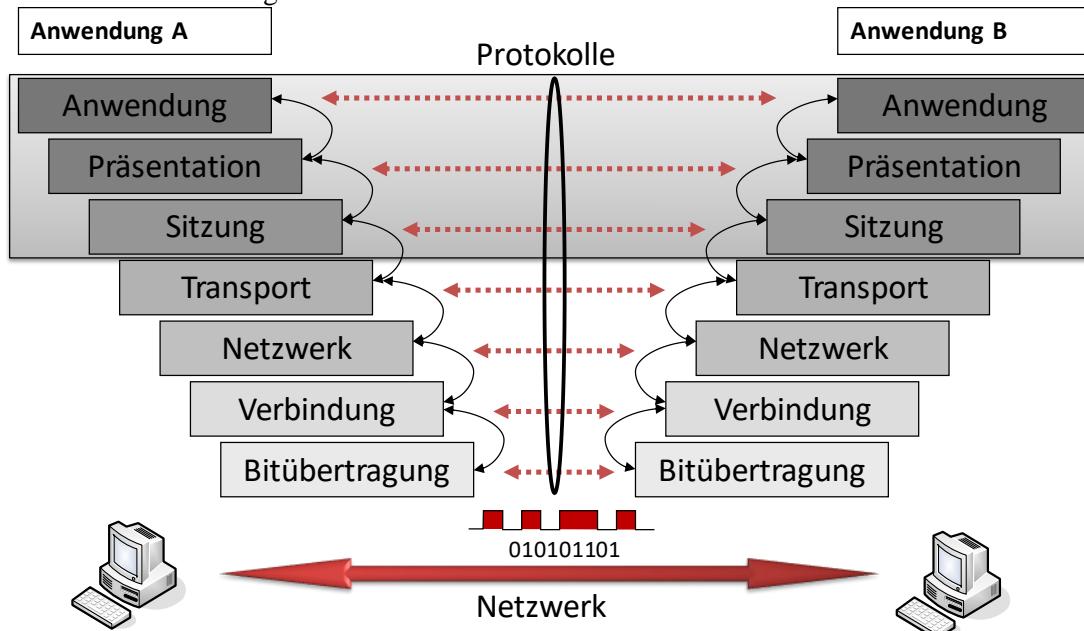


Abbildung 3: OSI-7-Schichtenmodell

Die oberen drei Schichten sind für Verteilte Systeme relevant. Die unteren Schichten sind eher für die Kommunikationsnetze (andere Veranstaltung) relevant.

Die sieben Schichten des OSI Modells haben jeweils bestimmte Funktionen.

Schicht 1	Bitübertragungsschicht (physical layer)	Binärdaten werden über einen Kommunikationskanal mit mechanischen und elektrischen Schnittstellen und dem physikalischen Übertragungsmedium übertragen.
Schicht 2	Sicherungsschicht, (data link layer)	Binärdaten werden zu Übertragungsrahmen (Frames) zusammengefasst und sequenziell übertragen. Die Sicherungsschicht kennzeichnet und erkennt die Rahmengrenzen. Damit können fehlerhafte, verlorene gegangene und doppelte Frames korrigiert werden.
Schicht 3	Vermittlungsschicht, (network layer)	Ermöglicht zwei Kommunikationspartnern ohne direkte Verbindung und ohne gleiches Netz die Vermittlung der Pakete durch Zwischenstationen (Netzwerkelemente). Die Auswahl der Leitwege (Routen) und Behandlung von Engpässen und Unterbrechungen geschieht selbstständig. Unterschiede verschiedener Netzwerktechnologien wie Modem, ADSL, WLAN, Ethernet, ISDN, Mobilfunk werden ausgeglichen.
Schicht 4	Transportschicht, (transport layer)	Für die Programmierung verteilter Systeme wichtigste Schicht. Sie bietet einen universellen Netzzugang für Anwendungsprogramme und schirmt höhere ab, so dass Netzwerkanwendungen transparent und unabhängig von der Hardware arbeiten können. Die Transportschicht baut als Ende-zu-Ende-Schicht Verbindungen zwischen Kommunikationspartnern auf und ab.

Schicht 5	Sitzungsschicht (session layer)	In ihr werden verschiedene parallellaufende Sitzungen verwaltet.
Schicht 6	Darstellungsschicht (presentation layer)	Steuerung einer Sitzung, Kodierungen für Zeichen (ASCII, Unicode), Ganzzahlen (MSB, LSB) und Gleitpunktzahlen werden in dieser Schicht konvertiert. Unterschiedliche Datenformate sowie kryptografische Verfahren und Kompression werden ebenfalls in der Darstellungsschicht behandelt.
Schicht 7	Anwendungsschicht (application layer)	Wichtige Anwendungen sind standardisiert: <ul style="list-style-type: none"> - virtuelle Terminals für interaktive Terminalssitzungen - Dateitransfer mit Konvertierung von Dateinamen, Dateiendezeichen, interner Darstellung der Informationen - Electronic Mail (E-Mail) für den Austausch von Nachrichten - weitere z. B. Informationsanwendungen, Datenbankabfragen usw.

3.3 Einbettung des OSI-Modells

Zwischen den OSI-Schichten werden über die Schnittstellen Daten ausgetauscht. Welche Daten, in welcher Reihenfolge und welcher Form dabei übergeben werden wird durch das Protokoll der entsprechenden Schicht festgelegt. Eine ähnliche Vorgehensweise findet man bei Programmiersprachen. Die Dienste, die eine Schicht der darüber liegenden anbietet, entsprechen einem Satz Funktionen. Wie und wann diese Funktionen aufgerufen werden entscheidet das Protokoll. Analog zu den Funktionen werden auch den entsprechenden Diensten Daten übergeben.

Auf jeder Ebene des OSI-Modells tauschen die Kommunikationspartner ihre Daten in Form von Paketen aus. Der interne Aufbau dieser Pakete wird wieder durch die Protokolle vorgeschrieben. Jede Schicht kann diese Daten verändern (z. B. konvertieren). Dadurch ergibt sich ein Aufbau aus der eigentlichen Nachricht (auch Nutzdaten oder engl. Payload genannt) und mehreren Headern für die verschiedenen Schichten. Dies wird an folgendem Bild deutlich:

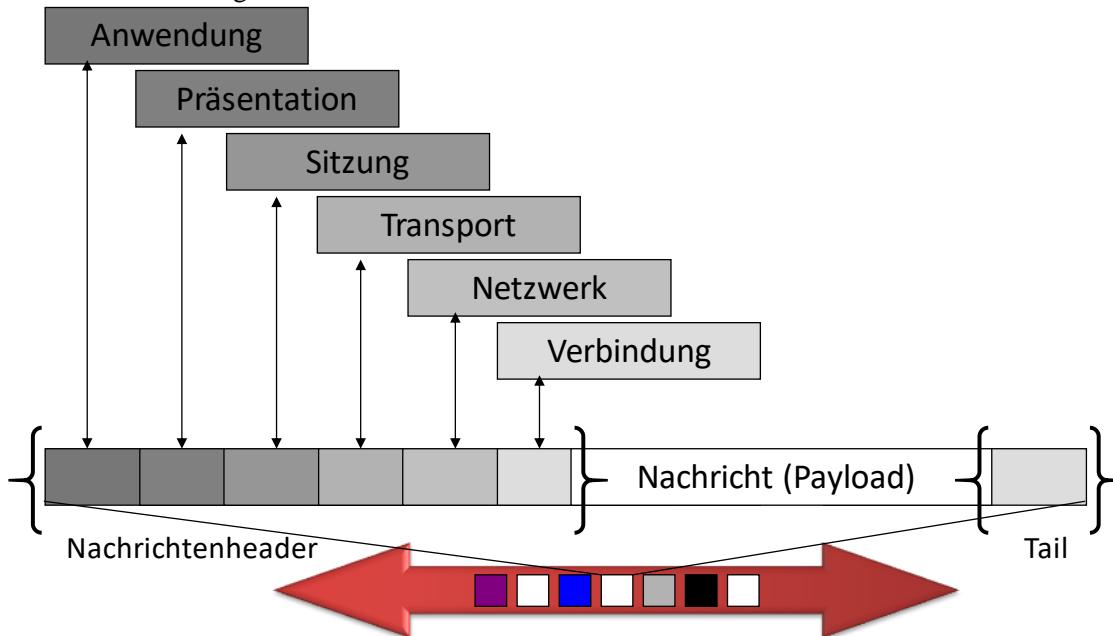


Abbildung 4: Header-Einbettung im OSI-7-Schichtenmodell

Zur Steuerung der Kommunikation müssen protokollspezifische Daten den eigentlichen Nutzdaten vorangestellt werden. Diese finden sich im Nachrichtenheader, der von unten nach oben zusammengesetzt wird. Am nächsten an den Nutzdaten liegt also der Header der Verbindung.

Jede Schicht nimmt das Paket bestehend aus den Nutzdaten und den Headern der darunter liegenden Schichten entgegen, stellt ihren eigenen Header voran und übergibt es der nächst höheren Schicht (im Bild von rechts nach links). Man spricht davon, dass ein Paket der Schicht n+1 in ein Paket der Schicht n

eingebettet (eingekapselt, encapsulated) ist. Diese Methode findet sich bei allen Netzwerkarchitekturen, auch wenn sie nicht OSI konform sind.

3.4 Dienstgüte

Jede Schicht bietet der darüber liegenden Dienste an. Dies können durchaus unterschiedliche Dienste mit unterschiedlicher Qualität sein. So unterscheidet man folgende grundlegenden Merkmale:

- **verbindungsorientierte Dienste (connection-oriented services)**

Sie arbeiten wie ein Telefonsystem. Bevor ein Datenaustausch stattfinden kann, muss eine Verbindung (charakterisiert durch eine Nummer) aufgebaut werden. Über diese Verbindung lässt sich ein serieller Datenstrom übertragen. Die Reihenfolge der Daten ist hierbei festgelegt. Wie beim Telefon werden Verbindungsaufbau, die Verbindungsnutzung und der Verbindungsabbau unterschieden.

- **verbindungslose Dienste (connectionless services)**

Sie arbeiten ähnlich einem Telegrammsystem. Jede Nachricht (jedes Paket) enthält die Bestimmungsadresse und wird unabhängig von den anderen durch das gesamte Netz transportiert. Daher kann es passieren, dass eine später versendete Nachricht früher beim Empfänger ankommt, da sie schneller transportiert wurde.

Diese Dienste können zusätzlich noch weitere Merkmale aufweisen:

- **zuverlässige (gesicherte) Dienste**

Bei zuverlässigen Diensten ist garantiert, dass alle versendeten Daten in der richtigen Reihenfolge beim Empfänger ankommen. Kein einzelnes Byte wird verändert oder geht verloren.

Beispiele für zuverlässige Dienste sind:

- Dateitransfer
- Interaktive Rechnerverbindung.

Bei verbindungsorientierten Diensten werden die Daten meist gesichert übertragen.

- **unzuverlässige (ungesicherte) Dienste**

Bei ihnen kann es passieren, dass Daten während der Übertragung verändert werden oder ganz verloren gehen. Beispiele für diese Dienste sind:

- Audio- und Videodaten
- Einfache Abfragesysteme.

Da die Kommunikation ähnlich E-Mail und SMS nach dem Prinzip „Fire & Forget“ ohne Quittung funktioniert, sind Overhead und Verzögerung etwas kleiner. Verbindungsauflauf und -abbau entfallen.

3.5 Grundlagen TCP/IP-Protokollfamilie

Die TCP/IP Protokolle und die Anfänge des Internets sind älter als das OSI-Modell. Daher ist die Architektur nicht OSI-konform. Das Internet, entstanden aus dem Arpanet, verbindet eine ganze Reihe von physikalischen Netzwerken, die das Internet Protocol (IP) benutzen, um ein großes logisches Netzwerk zu bilden. Das Internet ist also ein Netz von Netzen. Da die zunächst angeschlossenen Universitäten und Forschungsinstitutionen keine Verbindung zu Normungsgremien hatten, sind die Protokolle anders entwickelt worden.

Folgende Merkmale zeichnen die TCP/IP Protokollfamilie aus:

- Sie basiert auf offenen, frei erhältlichen Standards, die unabhängig von der Rechnerhard und -software entwickelt wurden.
- Sie ist unabhängig von der spezifischen Netzwerkhardware und der Netzart.
- Im Internet wird ein einheitliches Adressierungsschema benutzt. Jedes TCP/IP Gerät wird weltweit eindeutig über seine IP-Adresse identifiziert.

Es ist eine große Anzahl von Anwendungsprotokollen spezifiziert worden, um eine einheitliche Benutzeranwendung breit verfügbar zu machen. Internet Protokolle werden seit den ersten Anfängen im Jahr 1969 in den sogenannten Request for Comments (RFC) durch die Internet Engineering Task Force (IETF) [IETF03] veröffentlicht. Die TCP/IP Protokoll Architektur lässt sich analog zum OSI-Modell am besten in einem 4-Schichtenmodell beschreiben.

3.5.1 Vergleich zwischen OSI-Modell und TCP/IP

Das folgende Bild zeigt, welche Schichten zueinander in Beziehung stehen:

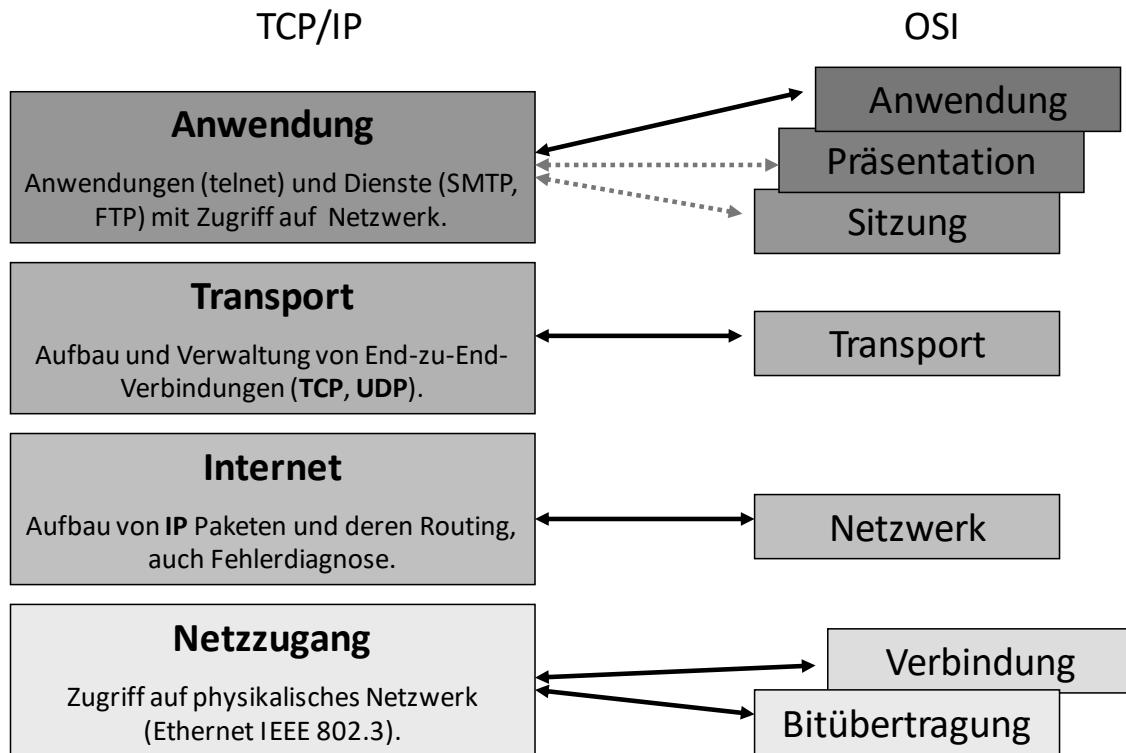


Abbildung 5: Vergleich zwischen TCP/IP-Protokollfamilie und OSI-Modell

Die TCP/IP-Protokollfamilie ermöglicht Netzkomunikation unabhängig von Hardware, Betriebssystem und Netzzugang. Dazu werden das Netzwerk und dessen Physik abstrahiert und die Interna des Netzwerks für die beiden Endpunkte in der Netzwerkschicht verborgen. Die Anwendungsprogramme, die mittels Programmierung verteilter Systeme realisiert werden, greifen auf den Transport layer zu. Für Details der darunter liegenden Schichten wird auf das Fach Kommunikationsnetze bzw. die entsprechenden Lehrbücher verwiesen. In diesem Skript werden nur einige wichtige Details wie z.B. die Darstellung von IP-Adressen näher beschrieben.

3.5.2 IP-Adressen und deren Netzwerk-/Hostdarstellung

In der Version 4 von IP werden die Adressen in vier Bytes untergebracht. Häufig schreibt man diese in der dotted-quad-Notation, also vier Dezimalzahlen (0..255), die durch Punkte getrennt sind. Eine spezielle Adresse ist z. B. 127.0.0.1, mit der auf jedem Rechner der Zugriff auf sich selbst durch das oben erwähnte Loopback-Interface ermöglicht wird. Anstelle der IP-Adressen kann auch ein Rechnername verwendet werden, was eine Erleichterung für das Management bedeutet. Der Rechnername für das Loopback-Interface lautet `localhost`.

Alle Daten in TCP/IP Protokollen und -Paketen werden an im Rechner an 4-Byte-Grenzen aufgespalten. Dabei stehen signifikante Bits links (das höchstwertigste Bit ist das Bit 0 oben). Man bezeichnet diese Ordnung als Network Byte Order oder Big Endian Format, d.h. die Bits 0-7, dann 8-15 usw. werden als erstes übertragen. Sollte ein Rechner ganze Zahlen in einer anderen Reihenfolge verarbeiten, d.h. das hochwertigste Bit ist das Bit 31 oben, muss der Header aus der Rechnerdarstellung (Host-Darstellung) in die Netzwerkdarstellung transformiert werden, bevor er versendet werden kann.

3.5.3 Transport Layer (UDP/TCP)

Die Transportschicht (transport layer) ist die vierte Schicht der TCP/IP-Protokollfamilie und bietet einen universellen Zugang für die Anwendungsprogramme. Dieser wird, je nach Anforderung in einer von zwei Varianten mit den Protokollen UDP und TCP realisiert. Auf Unix-Rechnern finden sich nähere Informationen zu den unterstützten Protokollen unter `/etc/protocols`.

Das User Datagram Protocol (UDP) ist ein einfaches verbindungsloses Transportprotokoll. Ein sendender Prozess verschickt seine Daten als unabhängige UDP-Datagramme, die über einzelne IP Datagramme

transportiert werden. Im Gegensatz dazu versendet ein TCP Prozess Daten als Datenstrom, wobei dieser mit Hilfe vieler IP Datagramme realisiert wird. Das UDP Protokoll ist in RFC 768 beschrieben. Es ist ein ungesichertes Protokoll, d.h. das nicht garantiert ist, dass ein UDP Datagramm jemals sein Ziel erreicht. Trotz dieses offensichtlichen Nachteils gibt es eine Reihe von Anwendungen, für die dieses Protokoll vollkommen ausreicht. Der UDP-Header ist kürzer als ein TCP-Header. Er enthält neben Adressen, Längeninformationen und Prüfsummen die Portnummern, auf die im nächsten Kapitel eingegangen wird. Um die Fragmentierung von UDP-Paketen möglichst zu verhindern, wird in den meisten UDP Anwendungen festgelegt, dass die Maximallänge, der UDP Daten 512 Bytes nicht übersteigt, da IP-Pakete mit maximal 576 Bytes von allen IP-Rechnern empfangen werden können.

Das Transmission Control Protocol TCP ermöglicht eine verbindungsorientierte, gesicherte Kommunikation. Verbindungsaufbau, die Verbindungsnutzung und der Verbindungsabbau werden durch unterschiedliche Paketvarianten im TCP-Header signalisiert. Damit nicht auf die Quittung für jedes einzelne Datenpaket gewartet werden muss, gibt es ein Schiebefensterprotokoll. Diverse Optionen ermöglichen eine situationsbezogene Feinsteuerung der Übertragung. So sorgt z. B. das Push Flag (PSH) für verzögertes Weiterleiten von Daten an die Anwendung (z. B. Tastendruck). Unterschiedliche Timer regeln das Verhalten in Ausnahmesituationen.

Das Internet Control Message Protocol (ICMP) dient dem Versand von Steuer und Kontroll-Daten (z. B. Ping) und wird in Netzwerkanwendungen nicht verwendet.

3.5.4 Betriebssysteme: Unterstützung für Verteilte Systeme

Die folgende Tabelle zeigt eine Reihe interessanter Netzwerkkommandos unter Unix und Windows:

Linux / Unix Kommando	Windows Kommando	Funktion
ifconfig -a	ipconfig /all	Zeigt Netzwerkkonfiguration an oder konfiguriert (Unix) Ethernet-Netzwerk-Interface, PPP etc.
tracepath	tracert	Gibt Route zu einem Zielsystem mit allen beteiligten Netzelementen und Laufzeiten (Unix) aus.
netstat -inet	netstat -a netstat -an	Ausgabe der aktuellen Netzwerkverbindungen mit Rechnernamen. Ausgabe der aktuellen Netzwerkverbindungen mit IP-Adressen (Windows).
arp	arp	Ausgabe der Routing-Tabelle.
ping	ping	Fordert Echo vom angesprochenen Rechner an, liefert Roundtrip-Time.

3.5.5 Client-/Servermodell

Anwendungen der Programmierung verteilter Systeme werden in der Regel entsprechend dem Client-Server-Modell realisiert. Ein Server stellt Dienste zur Verfügung, die von Clients über das Netzwerk beansprucht werden. Bei manchen Diensten kann die Anfrage als Nachricht zusammengefasst werden (Beispiel DNS-Anfrage). Bei anderen Diensten findet der Informationsaustausch in mehreren Schritten statt, was vorteilhaft in einer verbindungsorientierten Kommunikation abgebildet wird.

Zwei Server-Varianten sind hier möglich:

Iterativer Server:	Mehrstufiger Server:
Ablauf: 1. Warte auf Anforderung vom Client 2. Verarbeite Anforderung 3. Sende Antwort an Client 4. Gehe zu 1.	Ablauf: 1. Warte auf Anforderung vom Client 2. Starte einen neuen Prozess oder Thread , der die Anforderung bearbeitet und sich danach beendet. 3. Gehe zu 1.
UDP-Server werden meist als iterative Server realisiert	TCP-Server werden meist als mehrstufige Server realisiert.

Der mehrstufige Server hat den Vorteil, dass jeder Client seinen eigenen Server „sieht“, der seine Anforderung behandelt. Ein Server spaltet sich selbstständig auf mehrere auf. Wenn das Betriebssystem die quasi gleichzeitige Bearbeitung mehrerer Prozess (Threads) vorsieht, ist es möglich mehrere Client gleichzeitig zu bedienen.

In der Regel sind TCP Server mehrstufig und UDP Server iterativ.

3.5.6 Synchrone vs. Asynchrone Aufrufe

Neben der Aufteilung in ein- und mehrstufige Server lohnt sich ein differenzierter Blick in den Ablauf von Aufruf der Clients beim Server. Hier wird zwischen synchronen und asynchronen Aufrufen unterschieden:

Synchron:	Asynchron:
Sender und Empfänger blockieren beim „Senden“ bzw. „Empfangen“ von Nachrichten (Anfragen bzw. Antworten): <ul style="list-style-type: none"> Wenn ein „Senden“ ausgeführt wird, kann der Prozess nur weiterarbeiten, nachdem das zugehörige „Empfangen“ im anderen Prozess ausgeführt wurde. Wenn ein „Empfangen“ ausgeführt wird, wartet der Prozess so lange, bis die Nachricht gesendet wurde. 	Das „Senden“ findet nicht-blockierend statt: <ul style="list-style-type: none"> Der Prozess kann nach dem Senden einer Nachricht sofort weiterarbeiten. Empfangen kann blockierend oder nichtblockierend sein.
Weniger effizient , aber leicht zu implementieren: <ul style="list-style-type: none"> Synchronisation beim Ressourenzzugriff wird gleich mit erledigt 	Effizient , aber komplizierter zu implementieren (Warteschlangen).

Den synchronen Aufruf kennt man aus der prozeduralen und objektorientierten Programmierung. Nachdem eine Funktions- oder Methodenaufruf abgesetzt wurde, warte das aufrufende Modul solange, bis das aufgerufene Modul die Bearbeitung abgeschlossen hat und das Ergebnis zurückgibt. Ein anderes Beispiel ist ein Webseitenaufruf. Auch er wird nach dem Request/Response-Prinzip abgewickelt, wobei das Empfangen und Aufbereiten (Rendern) der Seite sich je nach Randbedingungen über einige Sekunden hinziehen kann.

Gerade im Web-Bereich wird mit zwei Erweiterungen der synchronen bzw. asynchronen Client-/Server-Kommunikation gearbeitet, wie folgendes Bild aus [Beng04] zeigt:

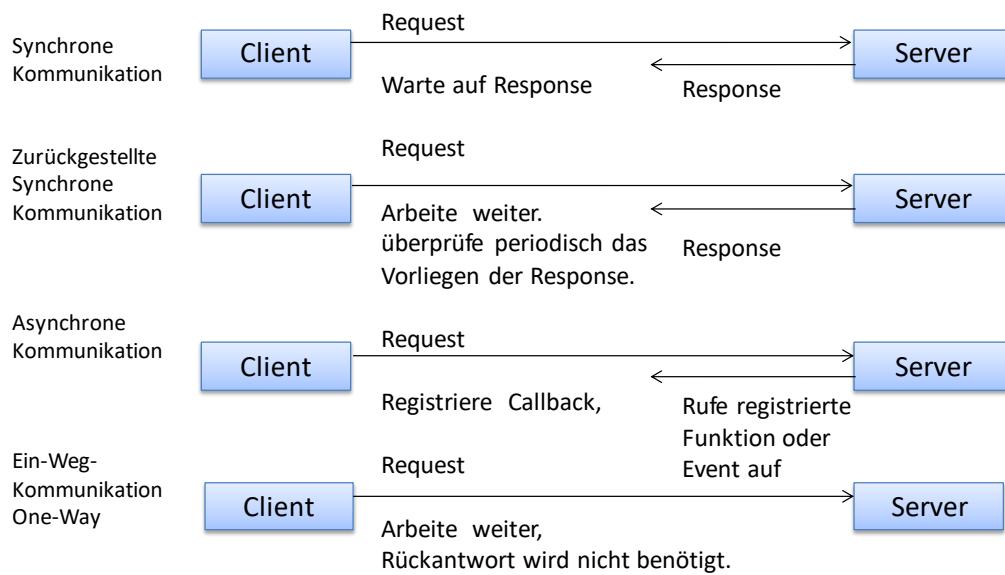


Abbildung 6: Varianten synchroner und asynchroner Client-/Server-Kommunikation [Beng04]

Bei der zurückgestellten synchronen Kommunikation kann der Client weiterarbeiten, fragt aber periodisch durch Polling danach, ob der Server die Bearbeitung der Anfrage bereits abgeschlossen hat.

Die Realisierung der asynchronen Kommunikation wird häufig durch Registrieren eines Callbacks (siehe Kapitel 7 AJAX) realisiert. Man gibt also dem Server-Request eine Callback-Funktion mit, die der Server bei Fertigstellung des Ergebnisses mit der Response aufruft. Bei der Oneway-Kommunikation (siehe auch Kapitel 5) wird hingegen durch den Request lediglich eine Aktion beim Server ausgelöst, ohne dass eine Response benötigt wird.

3.5.7 Port-Nummern

Mit Hilfe der Portnummern werden Anwendungsprozesse identifiziert. Auf Server-Seite werden Standardanwendungen über sogenannte Well-known Ports identifiziert, die von der IANA (Internet Assigned Number Authority) festgelegt werden. So hat ein FTP Server die TCP-Portnummer 21, ein Telnet Server die TCP-Portnummer 23. Zu TFTP (Trivial File Transfer Protocol) gehört die UDP Portnummer 69. Standardisierte Internetanwendungen haben also festgelegte Ports, die sogenannten Well-Known Ports. Diese sind auf einem Unix System in der Datei `/etc/services` verzeichnet z.B.:

```
# @(#)services 1.16 90/01/03 SMI
## Network services, Internet style
# This file is never consulted when the NIS are running
tcpmux    1/tcp      # rfc-1078
echo      7/tcp
echo      7/udp
discard   9/tcp      sink null
discard   9/udp      sink null
sysstat   11/tcp     users
daytime   13/tcp
daytime   13/udp
netstat   15/tcp
chargen   19/tcp      ttyst source
chargen   19/udp      ttyst source
ftp-data  20/tcp
ftp       21/tcp
telnet   23/tcp
smtp     25/tcp      mail
```

Abbildung 7: Well-Known-Ports eines Unix-Systems in der Datei `/etc/services`

Es ist wichtig, dass bei der Programmierung eigener Netzwerkanwendungen auf dem Server nur Ports oberhalb des Bereichs zwischen 0 und 1023 verwendet werden.

Clients hingegen haben kurzlebige Portnummern, die dynamisch vergeben werden. Sie existieren nur für die Laufzeit des Clients (ephemeral ports). Beim Start eines Client-Programmes wird dynamisch von der TCP oder UDP Software eine Quellportnummer vergeben. Diese Nummern liegen normalerweise im Wertebereich oberhalb von 1023 (Solaris wählt z. B. Zahlen größer als 32768 aus).

Viele rechner- und netzunabhängige TCP/IP Standardanwendungen wurden von der IETF (Internet Engineering Task Force) in RFCs (Request for Comments, siehe [IETF03]) standardisiert und zugehörige Standard-Ports aus dem Bereich der Well-Known-Ports festgelegt. Beispiele finden sich in folgender Tabelle:

Anwendung bzw. Dienst	Protokoll	verwendetes Protokoll der Transportschicht, Port(s)	RFC
Dateitransfer	File Transfer Protocol (FTP)	TCP, 20 (Daten), 21 (Steuerung)	959
virtuelles Terminal (telnet)	telnet	TCP, 23	1205, 2877
Versand „kurzer“ Nachrichten	Simple Mail Transfer Protocol (SMTP)	TCP, 25	821
Empfang „kurzer“ Nachrichten	Post Office Protocol (POP3)	TCP, 143	1939
Namensdienst zur Umsetzung zwischen Rechnernamen und IP-Adressen	Domain Name System (DNS)	UDP, 42	1034

3.5.8 Allgemeine Anforderungen an die Programmierung verteilter Systeme

Bevor wir uns mit der eigentlichen Programmierung verteilter Systeme auseinandersetzen, lohnt es sich, die Anforderungen zu betrachten, um später die Verfahren diesbezüglich bewerten zu können. Folgende Wünsche sollten Ansätze zur Programmierung verteilter Systeme erfüllen:

Anforderung	
<i>Netzwerkunabhängigkeit:</i> Clients und Server sollten ohne Änderungen auf unterschiedlichen Netzwerktypen arbeiten können.	Bei TCP/IP sind untere Schichten, u.a. das physikalische Netzwerk, gekapselt. Manche Verfahren der Programmierung verteilter Systeme unterstützen z. B. die Socket-Programmierung sogar weitere Protokollarten z. B. Token Ring
<i>Rechner-/Betriebssystem-Portabilität:</i> Clients und Server sollten ohne Änderungen (außer Neukompilierung) auf verschiedenen Rechnern und Betriebssystemen arbeiten.	Bei Standardverfahren gegeben, beim .net-Framework bevorzugt Windows-Betriebssysteme, aber Open-Source-Implementierungen in Entwicklung
<i>Portabilität bezüglich der Programmiersprache:</i> Clients und Server sollten sich verstehen, auch wenn sie mit unterschiedlichen Programmiersprachen entwickelt wurden.	Häufig gegeben. Eingeschränkt bei funktionsorientierter Programmierung (RPC) und objektorientierter Programmierung verteilter Systeme mit C++, Java und .net Framework, aber hier gibt es passende Übergänge.
<i>Transparente Nutzung:</i> Programmierer sollen Dienste auf entfernten Rechnern wie lokale Operationen nutzen können.	Wichtiges Konstruktionsprinzip, das durch Inkludieren von passenden NWP-Bibliotheken bzw. Packages realisiert wird.
<i>Einfaches Auffinden und Adressieren:</i> Es soll einfach sein, Server und Dienste aufzufinden und zu adressieren.	
<i>Geringerer Protokoll-Overhead:</i> Durch die verwendeten Protokolle sollte möglichst wenig Protokoll-Overhead erzeugt werden, um unnötige Verzögerungen und Kosten, z. B. bei Mobilverbindungen, zu vermeiden.	Hier gibt es größere Unterschiede nach dem Prinzip: Mehr Komfort und Funktionen erfordern größeren Overhead
<i>Erweiterte Funktionen und Dienste</i> Um verteilte Systeme sinnvoll und komfortabel nutzen zu können, sind Basisdienste für Zeit, Koordination, Sicherheit und Persistenz wünschenswert.	

Wir werden nun verschiedene Verfahren der Programmierung verteilter Systeme vorstellen und Empfehlungen für den Einsatz abhängig von den genannten Anforderungen geben.

4 Datenorientierte Programmierung verteilter Systeme

Wir haben uns bisher mit Standardanwendungen beschäftigt und werden nun verschiedene Möglichkeiten kennenlernen, Anwendungen auf Basis des Client-/Server-Prinzips zu entwickeln. Als synonyme Begriffe für solche Anwendungen haben sich verteilte Anwendungen, Distributed Systems und (besonders, wenn es erweiterte Dienste gibt) Middleware eingebürgert.

Wir beginnen nach einer kurzen Übersicht über Hilfs- und Systemfunktionen mit der Socket-Schnittstelle als einer der ersten Programmierschnittstellen für Netzwerke. Sie ist in fast allen Betriebssystemen und selbst in Geräten mit beschränkten Ressourcen wie PDAs und Mobilfunkgeräte verfügbar, sobald diese einen Netzwerkanschluss haben. Alle wichtigen Standardanwendungen, die im vorherigen Kapitel beschrieben wurden, arbeiten socket-basiert. Zwischen Server und Client ist ein Protokoll vereinbart, das beide Seiten einhalten müssen. Möchte man eigene Anwendungen entwickeln, muss man sich ein entsprechendes Protokoll überlegen und einen freien Socket (oberhalb von 1023) festlegen.

Die Socket-Programmierung ist (wie das Betriebssystem Unix) eng mit der Programmiersprache C verknüpft. Das Konzept ist aber so allgemein, dass es in allen wichtigen Programmiersprachen übernommen wurde. Es dient außerdem als Basis für komplexere Methoden der Programmierung verteilter Systeme wie RPC, CORBA, Java RMI etc. (siehe nachfolgende Kapitel). Vertiefende Literatur findet sich in [Beng14], [StFeRu04], [Haro00], [Poll04] und [Java14a].

4.1 Allgemeine Funktion von Sockets

Sockets werden datenorientiert wie Unix-Pipes oder Dateien genutzt. Die Socket-Schnittstelle zur Programmierung verteilter Systeme kann nach Inkludieren der passenden Bibliotheken genutzt werden. Sie besteht aus lokalen Funktionen für den Zugriff auf TCP/IP sowie Erweiterungen einiger Systemfunktionen. Durch die modulare Auslegung dieser Schnittstelle gibt es nicht nur Implementierungen für Internet-, sondern auch für andere OSI-Protokolle. Auch Datei- und Geräte-Ein/Ausgabe und Interprozesskommunikation kann auf Unix-Systemen mit der Socket-Schnittstelle realisiert werden.

Sockets (engl. für Steckdose) werden von Anwendungsprogrammen als Endpunkte für Netzwerkverbindungen verwendet. Sie bilden den Zugangspunkt zur Transportschicht und allen darunter liegenden Funktionen, die das Netzwerk kapseln. Im folgenden Bild werden die Sockets durch liegende Steckdosen symbolisiert:

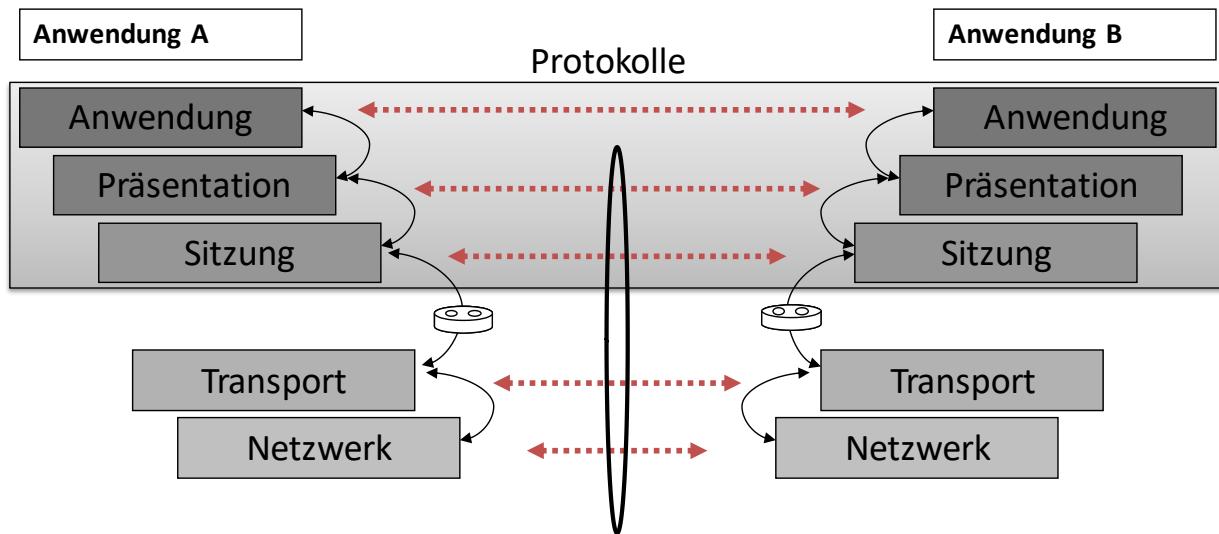


Abbildung 8: Sockets (engl. Steckdose) in der TCP/IP-Protokollfamilie

Die Anwendung greift auf das Netzwerk über den Socket und die zugehörigen Funktionen zu. Die darunter liegenden Schichten bleiben verborgen. Sie sind gekapselt.

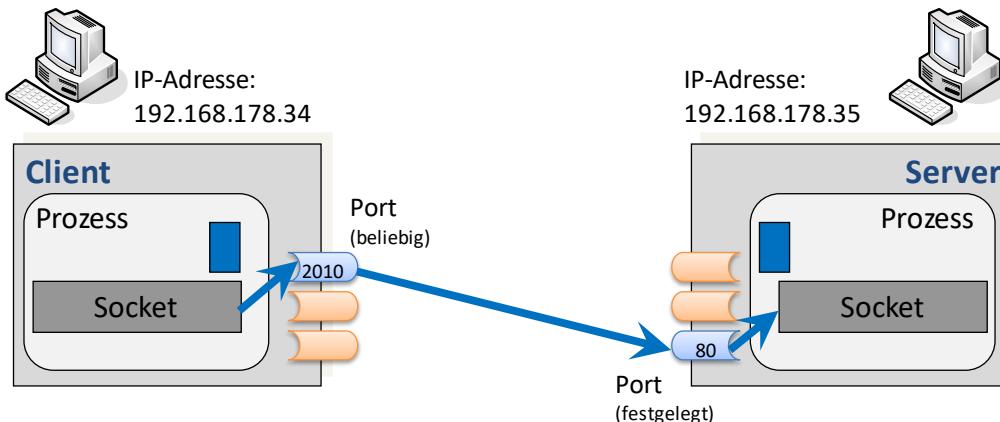
4.2 Socketadressen und Netzwerkverbindungen

Sockets werden eindeutig festgelegt durch IP-Adresse und Port. Bei aufgebauter Verbindung kann ein Socket wie eine Datei oder Pipe verwendet werden (aber bidirektional, Read/Write). Für die Socket-

Programmierung werden nur wenige, einfache Befehle benötigt (schmale Schnittstelle). Diese sind unabhängig von Programmiersprache und Betriebssystem. Dies hat den Vorteil, dass Ein-/Ausgabe über Netzwerk, Dateien und Geräte gleich gehalten werden kann.

Der Socket kann (über einen Deskriptor, der später behandelt wird), wie ein Objekt behandelt werden und die Prozedur zur Datenübertragung (z. B. write) wie eine zugehörige Methode. Je nach Deskriptor wird in der Prozedur entschieden, welche Art der Ein-/Ausgabe verwendet wird.

Eine Netzwerkverbindung besteht aus dem Protokoll und den beiden Sockets, insgesamt also fünf Angaben. Dies wird an folgendem Beispiel einer Webserver-Verbindung deutlich:



Socket: **TCP , 192.168.178.35:80 , 192.168.178.34:2010**

Abbildung 9: Web-Server-Verbindung

Man erkennt den well-known-Port 80 auf der Server-Seite und einen dynamisch vergebenen Port 2010 auf der Client-Seite. Nach Beenden der Terminalsitzung wird der Port wieder geschlossen.

Die meisten Server-Anwendungen erlauben es, mehrere Clients gleichzeitig zu bedienen. Im Beispiel kann der Server-Port 80 Endpunkt mehrere Verbindungen zu verschiedenen Clients sein.

Entscheidend für die Bedienung weiterer Clients ist neben der Parallelverarbeitung (z. B. verschiedene Prozesse) die Unterscheidungsmöglichkeit anhand einer Verbindung. Die IP-Pakete unterschiedlicher Clients können also eindeutig den entsprechenden Server-Instanzen zugeordnet werden, indem die Verbindungen anhand unterschiedlicher Client-Ports unterschieden werden.

4.3 Funktionen zur Socketprogrammierung

Möchten wir eigene datenorientierte Anwendungen entwickeln oder mit selbst geschriebenen Clients entfernte Server-Anwendungen nutzen, muss die Erzeugung und Beseitigung von Sockets sowie der eigentliche Datentransfer programmiert werden. Dazu gibt es einige Socket-Funktionen, die (als Minimalschnittstelle) auf fast allen Rechnern mit TCP/IP-Stack vorhanden sind.

4.3.1 Verbindungsmanagement – Übersicht

Es werden nun die Funktionen für die Vorbereitung der Verbindung und deren Beendigung besprochen. Grob gesprochen, sind folgende Aufgaben zu erledigen:

Aufgabe	Server/ Client	Funktion	Eingabeparameter	Rückgabeparameter
Anfordern der Ressourcen für einen neuen Socket	Server / Client	socket()	Adressfamilie, Protokoll	Socketdeskriptor
Registrierung des Sockets und Belegen eines Ports für eine Anwendung	Server / Client	bind()	Socketdeskriptor, Serveradresse und Serverport	Fehlercode

Einrichten einer Warteschlange für eingehende Aufrufe	Server	<code>listen()</code>	Socketdeskriptor, Länge der Warteschlange	-
Warten auf eingehende Aufrufe	Server	<code>accept()</code>	Socketdeskriptor	Clientadresse und Clientport, neuer Socketdeskriptor
Verbindungsaufbau zu einem Server	Client	<code>connect()</code>	Socketdeskriptor, Serveradresse und Serverport	Fehlercode
Verbindungsabbau	Client (Server)	<code>close()</code>	Socketdeskriptor	-

Die Darstellung von Serveradresse und Serverport wird im nächsten Abschnitt beschrieben.

4.3.2 Darstellung von Socketadressen in der Programmierung verteilter Systeme

Viele Funktionen zur Socketprogrammierung benötigen als Argument einen Pointer auf eine Socket-Adresse. Sie ist als Struktur in `<sys/socket.h>` definiert als:

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
}
```

Dabei ist `sa_family` die Adressfamilie wie `AF_UNIX` (für Dateioperationen) oder `AF_INET` (für Netzwerkoperationen). Die 14 Byte von `sa_data` werden als protokollabhängige Adresse interpretiert.

Für TCP/IP Anwendungen werden die folgenden Strukturen in `<netinet/in.h>` definiert:

```
struct in_addr {
    u_long s_addr;
};

struct sockaddr_in {
    short           sin_family;    // AF_INET
    u_short         sin_port;     // 16-bit port number
    struct in_addr sin_addr;     // 32-bit IP address
    char            sin_zero[8];   // unused
};
```

Die Typen `u_short` usw. sind in `<sys/types.h>` definiert. Für Unix-Domain-Sockets ist folgende Struktur in `<sys/un.h>` definiert:

```
struct sockaddr_un {
    short   sun_family;      // AF_UNIX
    char    sun_path[108];    // path
};
```

Wie man sieht, haben die Strukturen für unterschiedliche Ein-/Ausgabe-Umgebungen unterschiedliche Größen.

Damit die Socket-Funktionen mit den unterschiedlichen Umgebungen zurechtkommen, akzeptieren sie als Argument nur „generische“, also allgemeine Socketadressen vom Typ `sockaddr`. In Programmen muss man daher häufig die protokollspezifischen Adressen durch Typumwandlung (Casting) in generische umwandeln.

Dies zeigt folgender Source-Code-Ausschnitt:

```

...
struct sockaddr_in serv_address;      // Internet Adressenstruktur
                                         // durch _in symbolisiert
// alen enthaelt die Groesse der Adressstruktur, da diese variieren //
// kann und daher explizit angegeben werden muss
alen=sizeof(serv_address);

// Mit dem folgenden Typecast wird von der speziellen Internet- Dar-
// stellung in die allgemeine Socketadressendarstellung umgewandelt.
connect(sockfd, (struct sockaddr *)&serv_address, alen);

...

```

Wir werden uns auf die Nutzung von Internetadressen beschränken, daher fällt das Casting immer gleich aus.

4.3.3 Socketfunktionen (socket, bind, listen, accept, connect, close)

Das folgende Bild zeigt die aufzurufenden Funktionen für einen Verbindungsaufbau:

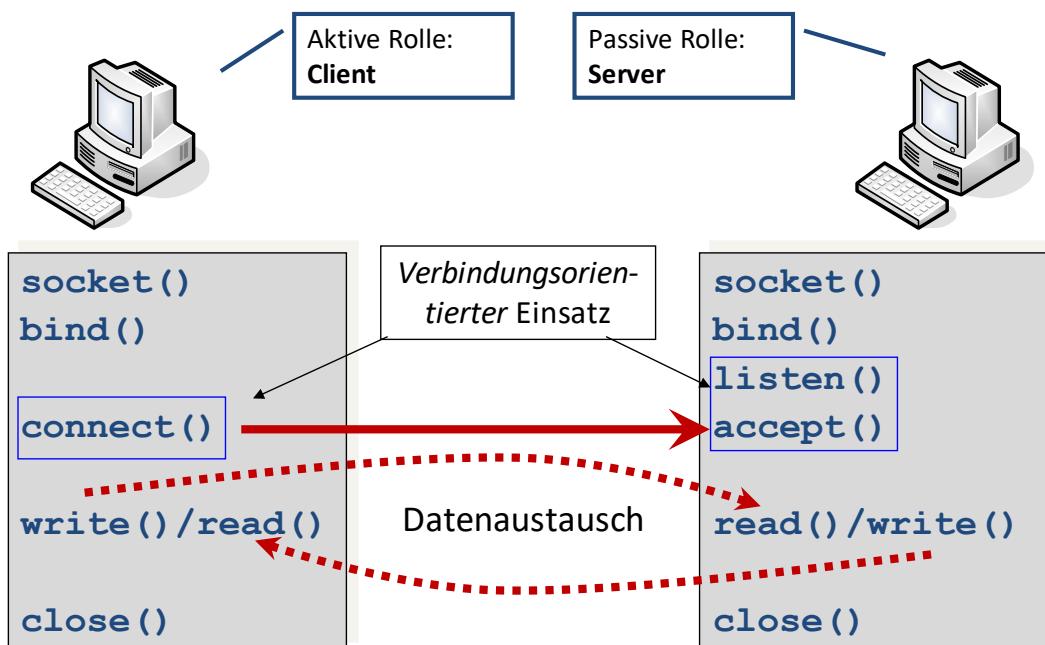


Abbildung 10: Socketfunktionen bei verbindungsorientierter Datenübertragung (TCP)

Man erkennt, dass der Server vier Funktionen aufrufen muss, um für eine eingehende Verbindung zur Verfügung zu stehen. Der Client muss hingegen nur den Socket einrichten kann dann direkt eine Verbindung aufbauen. Die Socketfunktionen werden nun ausführlicher beschrieben:

socket-Befehl

Der **socket**-Befehl richtet einen Socket ein und spezifiziert das zu verwendende Protokoll. Benötigte Datenstrukturen, Puffer usw. werden für den neuen Socket eingerichtet.

Die Funktionsbeschreibung ist:

```
#include <sys/types.h>      // für Sockets zu inkludieren
#include <sys/socket.h>      // für Sockets zu inkludieren
int socket(int family, int type, int protocol);
```

Der Aufruf geschieht folgendermaßen:

```
sockfd = socket(int family, int type, int protocol);
```

Die einzelnen Parameter haben folgende Bedeutung:

sockfd	Socketdeskriptor, wird behandelt wie Datei oder bidirektionale Pipe (wenn < 0, Fehler)
Family	eine der Kommunikationsfamilien AF_UNIX oder AF_INET (normalerweise AF_INET)
type	gewünschter Protokolltyp bei AF_INET: SOCK_STREAM (entspricht dem Protokoll TCP), SOCK_DGRAM (UDP) oder SOCK_RAW (IP)
Protocol	meist Null, außer in Ausnahmen z. B. IPPROTO_ICMP für ICMP mit SOCK_RAW

Bei AF_INET werden die zu verwendenden Übertragungsprotokoll durch folgende type-Angabe festgelegt:

Angabe bei type	Gewähltes Protokoll
SOCK_STREAM	TCP
SOCK_DGRAM	UDP
SOCK_RAW	IP

AF_UNIX wird hier nicht behandelt. Das protocol-Argument ist normalerweise 0. Die Konstanten IPPROTO_xxx sind in <netinet/in.h> definiert.

Der Rückgabewert der socket()-Funktion ist eine ganze Zahl, ähnlich einem Filedeskriptor. Er wird Socketdeskriptor genannt.

bind-Befehl

Der bind-Befehl dient der Registrierung eines Server-Sockets im System. Damit erklärt sich das Anwendungsprogramm für einen Port zuständig. Die Funktionbeschreibung ist:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

Die einzelnen Parameter haben folgende Bedeutung:

sockfd	Socketdeskriptor aus socket()-Aufruf
*myaddr	Pointer auf Socket-Adresse Adressfamilie normalerweise AF_INET
addrlen	Länge der Adresse
protocol	normalerweise Null

Sowohl verbindungsorientierte als auch verbindungslose Server müssen die Registrierung mit bind() durchführen. Ab erfolgreichem bind-Aufruf leitet das Betriebssystem Daten für den angemeldeten Port an den Serverprozess, der den Aufruf durchgeführt hat, weiter.

Wie an den Funktionsargumenten zu sehen ist, füllt die bind()-Funktion im 5-Tupel einer Verbindung die Größen lokale Adresse und lokaler Port. Sie liefert das Ergebnis 0, wenn kein Fehler auftrat bzw. -1 im Fehlerfall. Bei Fehlern wird außerdem die globale Variable errno gesetzt. Fehlercodes sind in <errno.h> definiert.

listen-Befehl (nur beim verbindungsorientierten Aufbau)

Verbindungsorientierte Server sind meistens mehrstufig, d.h. nach einem accept()-Aufruf wird mit fork() ein neuer Prozess erzeugt, der die Verbindung des Client behandelt. Das Erzeugen eines Prozesses benötigt aber etwas Zeit, in der im ungünstigen Fall weitere Verbindungswünsche eingehen können. Deshalb wird mit dem listen-Befehl eine Warteschlange für kurz nacheinander eintreffende Verbindungsanforderungen eingerichtet.

Der Aufruf lautet:

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Die Funktion wird normalerweise nach socket() und bind() und vor accept() ausgeführt.

Die Aufrufparameter haben folgende Bedeutung:

sockfd	Socketdeskriptor aus socket()-Aufruf
backlog	Anzahl der Einträge in Warteschlange (typischerweise maximal 5)

accept-Befehl (nur beim Server beim verbindungsorientierten Aufbau)

Mit dem Einrichten der Warteschlange sind alle Vorbereitungen getroffen. Durch Aufruf der `accept`-Funktion wird nun auf in der Warteschlange eingehende Verbindungen gewartet. Der Server verharrt solange in der `accept`-Funktion, bis eine Anforderung von einem Client eintrifft. Der Aufruf lautet:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *peer, int *addrulen);
```

Die einzelnen Parameter haben folgende Bedeutung:

<code>sockfd</code>	Socketdeskriptor aus <code>socket()</code> -Aufruf
<code>*peer</code>	Eintrag der Adresse des rufenden Client, Zugriff wie bei <code>bind()</code> . Die Funktion trägt die Daten des Partners in die Struktur ein, sofern genügend Platz vorhanden ist und gibt die tatsächlich benötigte Größe zurück. Für Internetanwendungen ist diese Größe typischerweise 16, für Unix-Domain Anwendungen können aber andere Werte auftauchen.
<code>*addrulen</code>	Länge der Struktur für Client-Adresse (Pointer !) Client-Daten werden eingetragen.
Rückgabe	neuer Socketdeskriptor für neue Verbindung

`accept()` blockiert, solange die Warteschlange leer ist. Für die neue Verbindung wird in der Regel ein neuer Prozess (`fork`) gestartet. Der alte Socketdeskriptor wird für weitere Verbindungen freigehalten.

Mit dem neuen Socket-Deskriptor liegen die fünf Verbindungsinformationen (Protokoll, Client-/Server-Socket bestehend aus IP-Adresse und Port) vollständig vor und die Verbindung ist aufgebaut und kann genutzt werden.

Die Funktion `accept()` liefert als Rückgabewert einen neuen Socketdeskriptor, bzw. -1 im Fehlerfall (dann wird auch `errno` gesetzt). Für diesen neuen Socketdeskriptor ist dann auch das 5-Tupel

{Protokoll, lok. Adresse, lok. Port, fr. Adresse, fr. Port}

komplett. Der Socketdeskriptor, der `accept()` übergeben wurde, wird für weitere Aufrufe von `accept()` wiederverwendet.

connect-Befehl (nur beim Client)

Der `connect`-Befehl ist das Gegenstück auf Client-Seite zum `accept`-Befehl des Servers. Er baut mit folgendem Aufruf eine Verbindung zum Server auf:

```
int connect(int sockfd, struct sockaddr *servaddr, int addrulen);
```

Die einzelnen Parameter haben folgende Bedeutung:

<code>sockfd</code>	Socketdeskriptor aus <code>socket()</code> -Aufruf
<code>*servaddr</code>	Pointer auf Socketadresse basierend auf <code>sockaddr</code>
<code>*addrulen</code>	Länge der Struktur für Client-Adresse (Pointer !) Daten des Clients werden in die Struktur eingetragen.

`connect()` blockiert (normalerweise) und wartet auf Antwort oder Timeout. Von `connect()` gibt es auch eine verbindungslose Variante, die die Adresse des gerufenen Servers für mehrere Übertragungsvorgänge festsetzt.

close-Befehl (nur beim verbindungsorientierten Aufbau)

Der `close`-Befehl schließt die Verbindung analog zum Schließen von Dateien. Dies wird normalerweise von der Anwendung, also auf Client-Seite veranlasst. Der Aufruf lautet:

```
int close(int sockfd);
```

Die Bedeutung von `sockfd` entspricht der bei den übrigen Funktionen:

<code>sockfd</code>	Socketdeskriptor aus <code>socket()</code> -Aufruf
---------------------	--

Folgendermaßen werden die fünf Verbindungsparameter in Abbildung 9 durch Socketfunktionen festgelegt:

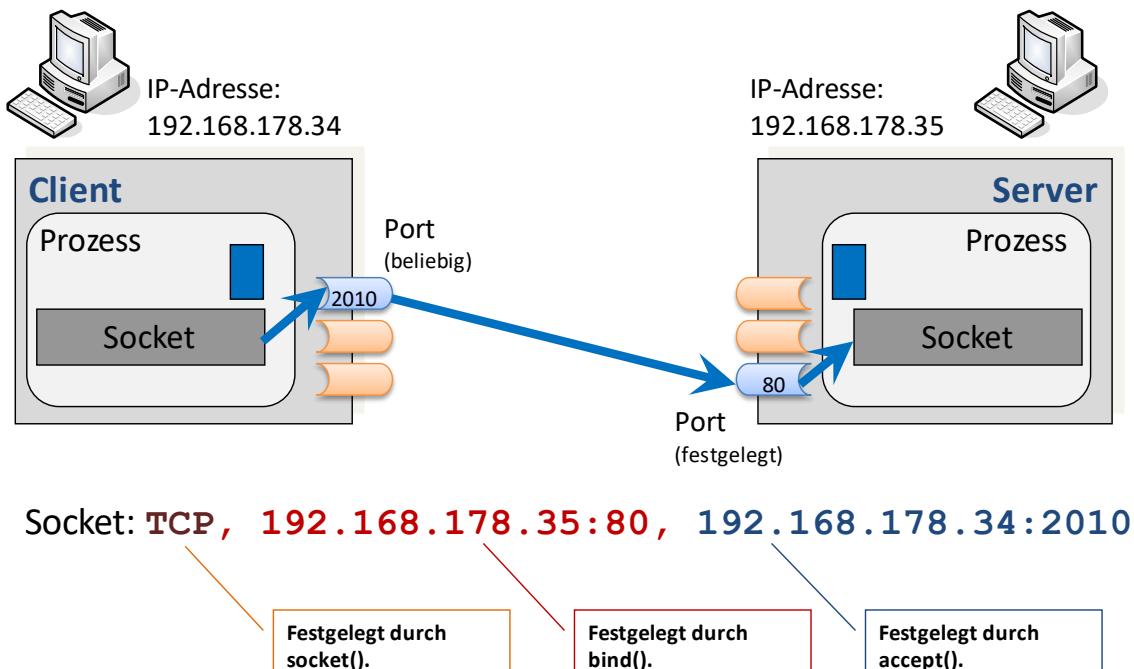


Abbildung 11: Zusammenhang zwischen Verbindungsparametern und Socketfunktionen bei TCP

4.3.4 Datenübertragung (`read`, `write`, `send`, `sendto`, `recv`, `recvfrom`)

Zwischen Verbindungsaubau und –abbau kann die Verbindung zur Datenübertragung genutzt werden. **read()** und **write()**

Diese Systemaufrufe werden normalerweise benutzt um von Dateideskriptoren zu lesen, bzw. auf sie zu schreiben. Sie können aber auch für Socketdeskriptoren benutzt werden. Ihr Aufruf ist:

```
int read(int sockfd, char *buf, int nbytes);
int write(int sockfd, char *buf, int nbytes);
```

Das erste Argument `sockfd` bezeichnet einen offenen Deskriptor, der mit `open()` oder `socket()` erzeugt wurde. Die Funktion `read()` liest dabei bis zu `nbytes` Bytes vom Deskriptor in das Feld `buf` ein und liefert die Anzahl der gelesenen Bytes als Rückgabewert. Es können auch weniger als `nbytes` gelesen werden. Wenn das Ende der Eingabe erreicht ist liefert die Funktion den Wert 0 zurück. Im Fehlerfall ist der Rückgabewert -1.

Die Funktion `write()` schreibt bis zu `nbytes` Bytes aus dem Feld `buf` auf den Deskriptor. Sie liefert als Ergebnis die Anzahl der tatsächlich geschriebenen Bytes zurück, oder -1 im Fehlerfall.

Es kann also sein, dass weniger Bytes geschrieben werden, als in den Argumenten verlangt wurde. Dies trifft vor allem auf Sockets zu. In einem solchen Fall müssen die restlichen Bytes im Feld `buf` in einem weiteren Schreibvorgang bearbeitet werden.

send(), **sendto()**, **recv()** und **recvfrom()**

Diese Systemfunktionen ähneln `read()` und `write()`, benötigen aber zusätzliche Argumente. Ihre Aufrufe sind:

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, char *buf, int nbytes, int flags);
int sendto(int sockfd, char *buf, int nbytes, int flags,
           struct sockaddr *to, int addrlen);
int recv(int sockfd, char *buf, int nbytes, int flags);
int recvfrom(int sockfd, char *buf, int nbytes, int flags,
             struct sockaddr *from, int *addrlen);
```

Die ersten drei Argumente entsprechen denen von `read()`/`write()`. Das Argument `flags` ist normalerweise 0.

Die Funktionen `send()` und `recv()` dienen zur Datenübertragung für den Fall, dass zuvor bei einem verbindungslosen Server bzw. Client mit `connect()` die Adressinformationen festgelegt wurden.

4.3.5 Gefahr von Buffer-Overflows

Bei der Socketprogrammierung muss man die Gefahr von Buffer Overflows adressieren. Damit sind Fehler gemeint, bei denen einzelnen Funktionen über den vorgesehenen Speicherbereich hinaus schreiben. Ein einfaches Beispiel in C ist :

```
char input[20];
scanf("%s", input);
```

Da die lokalen Variablen genau wie die Rücksprungadresse von Funktionen auf dem Stack abgelegt werden, ist für Angreifer ein Einschmuggeln und ausführen fremden Codes möglich. Eine einfache Abhilfe wird dadurch ermöglicht, dass durch alle wichtigen zwischenorientierten Funktionen Varianten vorliegen, die eine Kontrolle über die Anzahl der verarbeiteten Bytes ermöglichen. Man sollte also

```
strncpy()      statt strcpy()
fnscanf()      statt fscanf()
snprintf()     statt sprintf()
strncat()      statt strcat()
fgets()        statt gets()
```

usw. verwenden und die diesbezüglichen Warnungen der Compiler ernst nehmen.

4.3.6 Konvertierung zwischen Host- und Netzwerksdarstellung (htons etc.)

Die Reihenfolge der Bytes in zusammengesetzten Daten kann sich innerhalb eines Rechners, je nach Prozessor, von der im Netz unterscheiden. Beispiele sind IP-Adresse (`long`, also 4 Byte) und Port (`short`, also 2 Byte). Zur Konvertierung stehen folgende Funktionen zur Verfügung:

```
#include <sys/types.h>
#include <in/netinet.h>
u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_short netshort);
```

Diese Funktionen wandeln einen `long int`, bzw. `short int` Wert von der Rechner- in die Netzwerkdarstellung bzw. umgekehrt um. Obwohl die Funktionen für vorzeichenlose ganze Zahl definiert sind, arbeiten sie auch mit Zahlen mit Vorzeichen. Implizit wird in diesen Funktionen angenommen, dass ein `short int` 16 Bit und ein `long int` 32 Bit belegt.

4.3.7 Operationen auf Speicherbereichen (memcpy etc.)

Häufig ist es nötig, mit Speicherabschnitten, z. B. Felder mehrerer Bytes in Strukturen umzugehen. Leider handelt es sich nicht immer um Teile reiner C-Konstrukte. Daher gibt es eine Reihe von Funktionen, um diese Daten direkt im Speicher zu manipulieren.

Die wichtigsten sind:

```
#include <string.h>
void *memcpy(void *to, const void *from, size_t nbytes);
void *memset(void *to, int c, size_t nbytes);
int memcmp(const void *s1, const void *s2, int nbytes);
```

Die Funktion `memcpy()` kopiert dabei `nbytes` Bytes von der Stelle `from` an die Stelle `to` im Speicher.

Die Funktion `memset()` schreibt `nbytes` Mal das Zeichen `c` hintereinander beginnend bei der Speicherstelle `to`. Mit

```
memset((void *)&srv_addr, '\0', sizeof(srv_addr));
```

wird z. B. der Speicherbereich ab `srv_addr` mit Nullen belegt. Die Größe des Speicherbereichs wird hier mit `sizeof(srv_addr)` angegeben.

Die Funktion `memcmp()` schließlich vergleicht `nbytes` an den Stellen `s1` und `s2`.

4.3.8 Adressumwandlungen

Normalerweise werden Internetadressen geschrieben als dotted-quad, also z. B. als 131.173.20.138. Für die Socketadressen werden sie aber als 32-Bit-Zahl benötigt. Zur Umwandlung dienen folgende Funktionen:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long inet_addr(const char *dotquad);
char *inet_ntoa(struct in_addr ipaddr);
```

Die erste Funktion wandelt eine Zeichenkette mit dotted-quad Notation in eine 32 Bit Zahl um, während die zweite das umgekehrte tut. Die 32-Bit-Zahlen für die Internetadressen sind in Netzwerkordnung.

4.4 Beispiele

Verbindungsmanagement, Datenübertragung und Hilfsfunktionen sollen nun anhand einiger Beispielprogramme erläutert werden.

4.4.1 tcp_server – Mehrstufiger TCP-Server

Es wird nun beispielhaft ein mehrstufiger TCP-Echo-Server vorgestellt, der für jede Anfrage einen eigenen Prozess erzeugt. Ein Echo-Server nimmt die Eingabe eines Clients entgegen und sendet als Antwort Echo: und die eingegebene Zeile selbst zurück. Es gibt auch Echo-Server, die nur den gesendeten Text „echoen“. Beachten Sie, dass der Server außer einer Startmeldung keine eigenen Ausgaben macht. Dies ist allgemein üblich, da Server normalerweise unbeaufsichtigt laufen und niemand Konsolenmeldungen auswertet. Unbedingt erforderliche Meldungen werden in eine Log-Datei oder die Fehlerausgabe geschrieben. Das Echo der Zeilen an den Client wird solange ausgeführt, bis dieser die Verbindung beendet. Die Socket-Befehle sind unterstrichen.

```
// Beispiel TCP Echo-Server fuer mehrere Clients gekürzt aus
// Stevens: Unix Network Programming getestet unter Ubuntu 14.04
```

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SRV_PORT 8998
#define MAXLINE 512

// Vorwaertsdeklarationen
void str_echo(int);
void err_abort(char *str);

// Explizite Deklaration zur Vermeidung von Warnungen
void exit(int code);
void *memset(void *s, int c, size_t n);

int main(int argc, char *argv[]) {
    // Deskriptoren, Adresslaenge, Prozess-ID
    int sockfd, newsockfd, alen, pid;
    int reuse = 1;
    // Socket Adressen
    struct sockaddr_in cli_addr, srv_addr;

    // TCP-Socket erzeugen
```

```

if((sockfd=socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    err_abort("Kann Stream-Socket nicht öffnen!");
}
// Nur zum Test im Praktikum: Socketoption zur sofortigen Socket-Freigabe
if(setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse))<0){
    err_abort("Kann Socketoption nicht setzen!");
}
// Binden der lokalen Adresse damit Clients uns erreichen
memset((void *)&srv_addr, '\0', sizeof(srv_addr));
srv_addr.sin_family = AF_INET;
srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
srv_addr.sin_port = htons(SRV_PORT);
if(bind(sockfd, (struct sockaddr *)&srv_addr,
        sizeof(srv_addr)) < 0 ) {
    err_abort("Kann lokale Adresse nicht binden, läuft fremder Server?");
}
// Warteschlange für TCP-Socket einrichten
listen(sockfd,5);
printf("TCP Echo-Server: bereit ... \n");

for(;){
    alen = sizeof(cli_addr);

    // Verbindung aufbauen
    newsockfd = accept(sockfd,(struct sockaddr *)&cli_addr,&alen);
    if(newsockfd < 0){
        err_abort("Fehler beim Verbindungsauftbau!");
    }

    // für jede Verbindung einen Kindprozess erzeugen
    if((pid = fork()) < 0){
        err_abort("Fehler beim Erzeugen eines Kindprozesses!");
    }else if(pid == 0){
        close(sockfd);
        str_echo(newsockfd);
        exit(0);
    }
    close(newsockfd);
}

// str_echo: Lesen von Daten vom Socket und Zurückschicken an Client
void str_echo(int sockfd) {
    int n;
    char in[MAXLINE], out[MAXLINE+6];
    memset((void *)in, '\0', MAXLINE);
    for(;{
        // Daten vom Socket lesen
        n = read(sockfd,in,MAXLINE);
        if(n == 0){
            return;
        }else if(n < 0){
            err_abort("Fehler beim Lesen des Sockets!");
        }
    }
}

```

```

        sprintf(out, "Echo: %s", in);
        // Daten schreiben
        if(write(sockfd, out, n+6) != n+6){
            err_abort("Fehler beim Schreiben des Sockets!");
        }
    }

// Ausgabe von Fehlermeldungen
void err_abort(char *str){
    fprintf(stderr, " TCP Echo-Server: %s\n", str);
    fflush(stdout);
    fflush(stderr);
    exit(1);
}

```

Man sieht, wie mit `socket()` ein Socket `listenfd` angefordert wird. Danach werden die einzelnen Elemente der Adressstruktur `srvaddr` folgendermaßen gefüllt:

- `AF_INET` ist die Adressfamilie
- `INADDR_ANY` eine Konstante, die dem Empfang auf allen Netzwerkschnittstellen des Servers entspricht
- `TCP_PORT` ist der zuvor als Konstante definierte Port für diese spezielle Anwendung

`bind()` und `listen()` werden wie im vorherigen Abschnitt beschrieben verwendet. Nach dem Aufruf von `accept()` verharrt der Server blockierend, bis eine Client-Anfrage eingegangen ist. `accept` liefert einen neuen Socket, der für die eigentliche Datenübertragung verwendet wird. Eines der Argumente von `accept` ist die (umfangreiche) Struktur `cliaddr`, die in `cliaddr.sin_addr` die IP-Adresse des anfragenden Clients enthält und in `cliaddr.sin_port` dessen Port.

Das Schließen der Sockets mit `close()` dekrementiert den Referenzzähler auf den angegebenen Socket. Der Socket selbst wird erst geschlossen, wenn der Referenzzähler den Stand Null erreicht.

Auf der Kommandozeile übersetzt man den TCP-Server mit

```
westerka@si0024-1:~/> gcc -o tcp_srv tcp_server.c -g
```

Zum Testen startet man den Server auf einem Rechner

```
westerka@si0024-1:~/> ./tcp_srv
TCP Echo-Server: bereit ...
```

und kontaktiert diesen dann mit Telnet von einem anderen Rechner aus:

```
westerka@si0024-2:~/> telnet westerka@si0024-1 8998
Trying...
Connected to westerka@si0024-1.
Escape character is '^].
test
Echo: test
```

4.4.2 TCP-Client

Besser als die Nutzung von telnet ist natürlich die Entwicklung eines eigenen Clients `tcp_client`. Er sieht folgendermaßen aus:

```
// Beispiel: TCP Echo-Client Quelle: Stevens, R., Fenner, B., Rudoff,
// A. M.: Unix Network Programming getestet unter Ubuntu 14.04
```

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

#include <arpa/inet.h>

#define SRV_PORT 8998
#define MAXLINE 512

void exit(int code);
void str_client(int);
void err_abort(char *str);

int main(int argc, char *argv[]) {
    // Deskriptor
    int sockfd;
    // Socket Adresse
    struct sockaddr_in srv_addr, cli_addr;

    // Argumente testen
    if( argc != 2 ) {
        err_abort("IP Adresse des Servers fehlt!");
    }
    // TCP Socket erzeugen
    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        err_abort("Kann Stream-Socket nicht oeffnen!");
    }
    // Adress Struktur fuer Server aufbauen
    memset((void *)&srv_addr, '\0', sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    srv_addr.sin_port = htons(SRV_PORT);

    // Verbindung aufbauen
    if(connect(sockfd,(struct sockaddr *)&srv_addr,sizeof(srv_addr)) < 0){
        err_abort("Fehler beim Verbindungsaufbau!");
    }
    printf("TCP Echo-Client: bereit...\n");

    // Daten zum Server senden
    str_client(sockfd);

    close(sockfd);
    exit(0);
}

// str_client: Daten von der Standardeingabe lesen, zum Server senden, auf
// das Echo warten und dieses ausgeben
void str_client(int sockfd){
    int n;
    char out[MAXLINE],in[MAXLINE+6];

    // Lesen bis zum Ende der Eingabe
    while(fgets(out,MAXLINE,stdin)!=NULL){
        n=strlen(out);
        out[n-1]='\0';
        // Zeile zum Server senden
        if(write(sockfd,out,n)!=n){
            err_abort("Fehler beim Schreiben des Sockets!");
        }
    }
}

```

```

        }
        // Echo vom Server lesen
        n=read(sockfd,in,MAXLINE);
        if(n<0){
            err_abort("Fehler beim Lesen des Sockets!");
        }
        // ausgeben
        printf("%s\n",in);
    }
}

// Ausgabe von Fehlern und Beenden des Programms
void err_abort(char *str){ /* wie zuvor */}

```

Auf der Kommandozeile übersetzt man den TCP-Client wie zuvor den Server mit
 westerka@si0024-1:~/> gcc -o tcp_clt tcp_client.c -g

Zum Testen startet man den Server auf einem Rechner

```
westerka@si0024-1:~/> ./tcp_srv
TCP Echo-Server: bereit ...
```

Der Client wird mit der IP-Adresse des Servers als Argument aufgerufen und getestet, ob der eigene Client die gleichen Ergebnisse wie beim telnet-Test liefert:

```
westerka@si0024-2:~/> ./tcp_clt 131.173.110.1 8998
asdfghjk
Echo: asdfghjk
```

4.4.3 Beispiel daytime.c

Das folgende Beispiel dient dazu, einen anderen Serverdienst zu implementieren. Es ist dies der Daytime-Dienst, der laut Abb. 7 den Well-known-Port 13 verwendet.

```

/* File: daytime.c
 * Author: Heinz-Josef Eikerling
 *          Implementierung des Daytime-Service.
 *          Compile: gcc -o daytime daytime.c
 *          Aufruf: daytime oder daytime <host address> <host port>
 * Created on 10. März 2012, 21:54 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/* Fehler ausgeben und Programm beenden. */
static void handleError(const char *cause) {
    if ( errno != 0 ) {
        fputs(strerror(errno), stderr);
        fputs(": ", stderr);
    }
    fputs(cause, stderr);
    fputc('\n', stderr);
    exit(1);
}

```

```

/* main-Funktion */
int main(int argc,char **argv) {
    int z;
    char *srvr_addr = NULL;
    char *srvr_port = "9013";
    struct sockaddr_in adr_srvr; // AF_INET
    struct sockaddr_in adr_clnt; // AF_INET
    int addr_len; // length
    int s; // Socket
    int c; // Client socket
    int n; // bytes
    time_t td; // Current date&time
    char tstamp[128]; // Date/Time info

    /* Server-Adresse von der Kommandozeile lesen, falls vorhanden. */
    if ( argc >= 2 ) {
        srvr_addr = argv[1]; // Addr on cmdline
    }
    else {
        srvr_addr = "127.0.0.1"; // Use default address
    }

    /* Server-Port von der Kommandozeile lesen, falls vorhanden. */
    if ( argc >= 3 ) {
        srvr_port = argv[2];
    }

    /* Socket anlegen. */
    s = socket(PF_INET,SOCK_STREAM,0);
    if ( s == -1 ) {
        handleError("socket()");
    }
    memset(&adr_srvr,0,sizeof adr_srvr);
    adr_srvr.sin_family = AF_INET;
    adr_srvr.sin_port = htons(atoi(srvr_port));
    if ( strcmp(srvr_addr,"*") != 0 ) {
        adr_srvr.sin_addr.s_addr = inet_addr(srvr_addr); // Normal
Address
        if ( adr_srvr.sin_addr.s_addr == INADDR_NONE ) {
            handleError("bad address.");
        }
    }
    else {
        adr_srvr.sin_addr.s_addr = INADDR_ANY; // Wild Address
    }

    /* Adresse an Server binden. */
    addr_len = sizeof adr_srvr;
    z = bind(s,(struct sockaddr *)&adr_srvr,addr_len);
    if ( z == -1 ) {
        handleError("bind(2)");
    }

    z = listen(s,10);
    if ( z == -1 ) {
        handleError("listen(2)");
    }
}

```

```

for (;;) {
    /* Server Loop. Auf Anfragen warten und diese auswerten */
    addr_len = sizeof adr_clnt;
    c = accept(s, (struct sockaddr *)&adr_clnt, &addr_len);

    if (c == -1) {
        handleError("accept(2)");
    }

    /* Zeitstempel erzeugen */
    time(&td);
    n = (int) strftime(tstamp, sizeof tstamp,
                       "%A %b %d %H:%M:%S %Y\n",
                       localtime(&td));

    /* Ergebnis an Client geben */
    z = write(c, tstamp, n);
    if (z == -1) {
        handleError("write(2)");
    }

    /* Verbindung schließen */
    close(c);
}
return 0;
}

```

4.4.4 Besonderheiten bei verbindungslosen Socketanwendungen mit UDP

Die bisherigen Aufrufe dienten dem Verbindungsmanagement bei verbindungsorientierten Socketanwendungen mit TCP. Bei verbindungsloser Socket-Nutzung (UDP) entfallen die Funktionen `listen()` und `accept()` wie in folgendem Bild zu sehen ist:

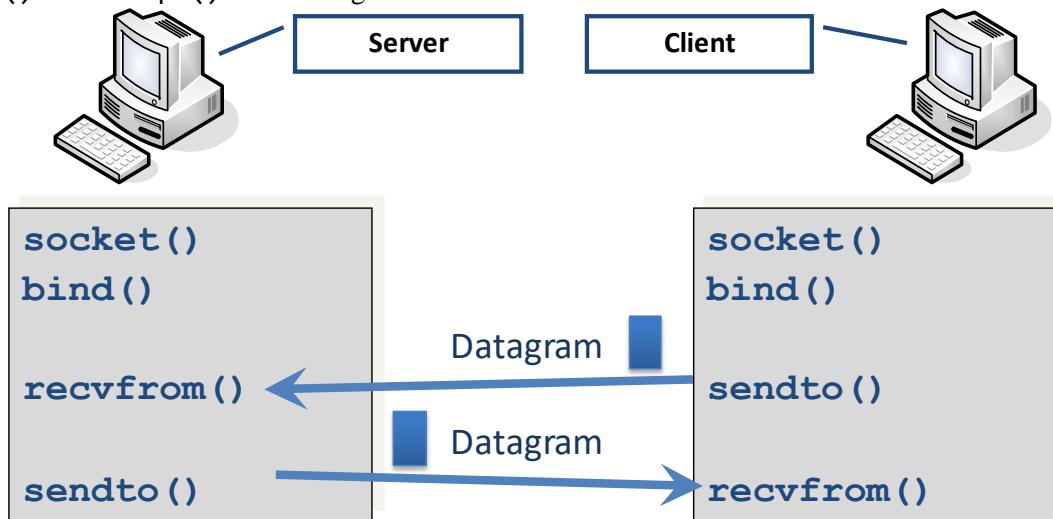


Abbildung 12: Socketfunktionen bei verbindungsloser Datenübertragung mit UDP

Im Gegensatz zur verbindungsorientierten Anwendung baut ein Client keine Verbindung zum Server auf, sondern sendet ihm mit `sendto()` einfach ein Datagramm. Der Server wartet mit dem Aufruf `recvfrom()` auf Datagramme von Client. Die Funktion liefert als Ergebnis das Datagramm und die Adresse des Clients. Sowohl für verbindungslose als auch für verbindungsorientierte Socketkommunikation werden nun die zugehörigen Funktionen zur eigentlichen Datenübertragung betrachtet.

4.4.5 Beispiel udp_server Echo-Server mit UDP

Wenn statt TCP nun UDP als Transportprotokoll benutzt werden soll, müssen Server und Client etwas anders aufgebaut werden. Der Client sendet eine Zeile in einem Datagramm an den Server und der sendet

es auch in einem Datagramm zurück. Die Zeilenlänge muss auf die Datagrammlänge beschränkt bleiben (hier 512 Bytes). Weiterhin wird im Server die lokale Adresse mit Hilfe von `bind()` an den Socket gebunden, da nicht gesichert ist, dass die `sendto()`-Funktion dies immer vornimmt. Festzustellen ist auch, dass Client und Server nicht gesichert sind. Wenn also ein Datagramm verloren geht, wird dies nicht festgestellt.

Der Server `udp_srv.c` ist folgendermaßen implementiert:

```
// Beispiel einfacher UDP Echo-Server
// Gekürzt aus Stevens: Unix Network Programming
// getestet unter Ubuntu 14.04

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SRV_PORT 8998
#define MAXLINE 512

// Vorwaertsdeklarationen
void dg_echo(int);
void err_abort(char *str);

// Explizite Deklaration zur Vermeidung von Warnungen
void exit(int code);
void *memset(void *s, int c, size_t n);

int main(int argc, char *argv[]) {
    // Deskriptor
    int sockfd;
    // Socket Adresse
    struct sockaddr_in srv_addr;

    // TCP-Socket erzeugen
    if((sockfd=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        err_abort("Kann Stream-Socket nicht oeffnen!");
    }

    // Binden der lokalen Adresse damit Clients uns erreichen
    memset((void *)&srv_addr, '\0', sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    srv_addr.sin_port = htons(SRV_PORT);
    if( bind(sockfd, (struct sockaddr *)&srv_addr,
              sizeof(srv_addr)) < 0 ) {
        err_abort("Kann lokale Adresse nicht binden, laeuft fremder
Server?");
    }

    printf("UPD Echo-Server: bereit ...\\n");
    dg_echo(sockfd);
}
```

```

// dg_echo: Lesen von Daten vom Socket und an den Client zuruecksenden
void dg_echo(int sockfd) {
    int alen, n;
    char in[MAXLINE], out[MAXLINE+6];
    struct sockaddr_in cli_addr;

    for(;;) {
        alen = sizeof(cli_addr);
        memset((void *)&in, '\0', sizeof(in));

        // Daten vom Socket lesen
        n=recvfrom(sockfd,in,MAXLINE,0,(struct sockaddr *)&cli_addr, &alen);
        if( n<0 ) {
            err_abort("Fehler beim Lesen des Sockets!");
        }
        sprintf(out, "Echo: %s",in);

        // Daten schreiben
        if(sendto(sockfd,out,n+6,0,(struct sockaddr *)&cli_addr,alen)!=n+6){
            err_abort("Fehler beim Schreiben des Sockets!");
        }
    }
}

// Ausgabe von Fehlermeldungen
void err_abort(char *str){ /* wie zuvor */}

```

Man beachte, dass der Server die Clients der Reihe nach bedient. Es handelt sich also um einen iterativen Server. Da der Test des Servers mit Telnet nicht möglich ist, entwickelt man gleich einen eigenen zugehörigen Client udp_cli:

// Beispiel: Einfacher UDP Echo-Client Gekürzt aus Stevens, R., Fenner, // B., Rudoff, A. M.: Unix Network Programming getestet unter Ubuntu 14.04

```

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SRV_PORT 8998
#define MAXLINE 512

void exit(int code);
void dg_client(int, struct sockaddr*, int);
void err_abort(char *str);

int main(int argc, char *argv[]) {
    // Deskriptor
    int sockfd;
    // Socket Adresse
    struct sockaddr_in srv_addr, cli_addr;

    // Argumente testen
    if( argc != 2 ) {

```

```

        err_abort("IP Adresse des Servers fehlt!");
    }
    // UDP Socket erzeugen
    if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        err_abort("Kann Stream-Socket nicht oeffnen!");
    }
    // lokale Adresse binden
    memset((void *)&cli_addr, '\0', sizeof(cli_addr));
    cli_addr.sin_family = AF_INET;
    cli_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    if(bind(sockfd, (struct sockaddr *)&cli_addr, sizeof(cli_addr))<0){
        err_abort("Fehler beim Binden der lokalen Adresse!");
    }
    // Adress Struktur fuer Server aufbauen
    memset((void *)&srv_addr, '\0', sizeof(srv_addr));
    srv_addr.sin_family = AF_INET;
    srv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    srv_addr.sin_port = htons(SRV_PORT);
    printf("UDP Echo-Client: bereit...\n");
    // Daten zum Server senden
    dg_client(sockfd,(struct sockaddr *)&srv_addr, sizeof(srv_addr));

    close(sockfd);
    exit(0);
}

// dg_client: Daten von der Standardeingabe lesen, zum Server senden,
// auf das Echo warten und dieses ausgeben
void dg_client(int sockfd, struct sockaddr *srv_addr, int srv_len){
    int n;
    char out[MAXLINE],in[MAXLINE+6];

    // Lesen bis zum Ende der Eingabe
    while(fgets(out,MAXLINE,stdin)!=NULL){
        n=strlen(out);
        out[n-1]='\0';
        // Zeile zum Server senden
        if(sendto(sockfd,out,n,0,srv_addr,srv_len)!=n){
            err_abort("Fehler beim Schreiben des Sockets!");
        }
        // Echo vom Server lesen
        n=recvfrom(sockfd,in,MAXLINE,0,(struct sockaddr *)NULL,(int *)NULL);
        if(n<0){
            err_abort("Fehler beim Lesen des Sockets!");
        }
        //rcvline[n-1]='\0';
        // ausgeben
        printf("%s\n",in);
    }
}

```

```
// Ausgabe von Fehlern und Beenden des Programms
void err_abort(char *str){ /* wie zuvor */}
```

Der Test liefert das gleiche Ergebnis wie im TCP-Fall:

```
westerka@westerka@si0024-2:~/> ./udp_cli 131.173.110.14
dfadsf
Echo: dfadsf
```

4.5 Ein/Ausgabe-Multiplexing

Wenn ein Prozess mehrere Sockets zum Lesen geöffnet hat, kann es vorkommen, dass er nicht weiß, von welchem Socket die nächsten Eingabedaten kommen. Normalerweise würde er dann in einem der Prozesse blockiert bleiben, während Daten bei einem anderen vorliegen. Man muss also die Möglichkeit haben, gleichzeitig mehrere Ein/Ausgabekanäle zu bedienen. Dafür gibt es verschiedene Methoden:

1. Man kann Sockets als nichtblockierend einrichten, indem man ihnen das Flag FNDELAY mit der Funktion `fcntl()` gibt, bzw. die FIONBIO Anforderung durch `ioctl()` weitergibt. Dies hat zur Folge, dass Lese- und Schreibvorgänge nicht mehr blockieren. Wenn für den Socket keine Daten vorliegen kehren die Funktionen `read()` und `write()` sofort zum Aufrufer zurück und warten nicht wie sonst auf das Eintreffen von Daten. Dies kann man anwenden, indem man in einer Schleife alle Kanäle der Reihe nach abfragt. Liegen auf einem Kanal Daten vor, können sie verarbeitet werden. Dieses Verfahren nennt man auch Polling. Diese Methode ist nicht sehr beliebt, da sie viel Rechenzeit verschwendet.
2. Der Prozess kann für jeden Kanal einen eigenen Kindprozess erzeugen. In jedem Prozess kann der Kanal mit einem gewöhnlichen `read()`-Aufruf versuchen, Daten zu lesen. Nach Beendigung eines Lesevorganges müssen die Daten über irgendeine Interprozesskommunikation dem Elternprozess übergeben werden. In vielen Fällen scheidet dieser Mechanismus wegen seiner Komplexität und einer möglicherweise hohen Anzahl Prozesse im Hauptspeicher aus.
3. Asynchrone Ein/Ausgabe kann benutzt werden. Dabei wird dem Kernel mit Hilfe der `signal()` Funktion eine Signal-Handler Funktion bekannt gegeben. Diese wird aufgerufen, wenn auf einem Kanal Daten für die Ein- oder Ausgabe vorliegen. Leider hat dieses Verfahren den Nachteil, dass der Signal-Handler nicht weiß, welcher der Kanäle gerade aktiv wurde.
4. Mit Hilfe der `select()`-Funktion wird der Kernel angewiesen, den Prozess bei definierten Ein/Ausgabeaktivitäten aus einem Schlafzustand wieder zu aktivieren. Diese Möglichkeit wollen wir etwas näher betrachten, da sie die zuvor genannten Nachteile vermeidet.

Der Aufruf der Funktion `select()` lautet:

```
#include <sys/types.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Das erste Argument `maxfdp1` gibt die Anzahl der maximal zu beobachtenden Deskriptoren an. Dabei ist `fd_set` ein Bitfeld, in dem jedes Bit für einen Deskriptor steht. Das erste Bit steht für den Deskriptor 0, das zweite für den Deskriptor 1 und so weiter. Die Deskriptoren, deren Bit im Feld gesetzt ist, sollen beobachtet werden. Zur Manipulation des Bitfeldes gibt es folgende Makros:

```
FD_ZERO(fd_set *fdset)           // clear all bits in fdset
FD_SET(int fd, fd_set *fdset)    // turn fd bit on in fdset
FD_CLR(int fd, fd_set *fdset)    // turn fd bit off in fdset
FD_ISSET(int fd, fd_set *fdset)  // test fd bit in fdset
```

Durch den Aufruf der Funktion werden die Deskriptoren `readfds` untersucht, ob sie für die Eingabe bereit sind, bzw. ob die Deskriptoren `writefds` bereit sind für die Ausgabe oder für die Deskriptoren `exceptfds` besondere Bedingungen vorliegen. Mit `timeout` kann man festlegen, ob:

- Die Funktion sofort nach dem Test zurückkehrt. Dies entspricht einem Polling. Dazu muss `timeout` auf eine Struktur zeigen, dessen Zeitwerte `tv_sec` und `tv_usec` null sind.
- Die Funktion sofort zurückkehrt, wenn einer der angegebenen Deskriptoren bereit ist, wobei aber nur über eine Zeitspanne getestet wird. Diese ist in Sekunden und Mikrosekunden in der Struktur, auf die `timeout` zeigt, angegeben.

- Die Funktion nur zurückkehrt, wenn einer der angegebenen Deskriptoren bereit ist. Dazu muss das `timeout` Argument `NULL` sein.

Da es nur sehr wenige besondere Bedingungen für Dateideskriptoren gibt, werden die Bits in `exceptfds` im Folgenden immer auf Null gesetzt. Auf der Beobachtung mehrerer Ports mittels `select()`-Anweisung basiert eine zentrale Komponente des Unix-Systems, der Internet Superdaemon (kurz `inetd` genannt). Er verwirklicht außerdem die ressourcensparende Idee, die Anwendungsprogramme erst zu starten, wenn wirklich eine Anfrage für sie vorliegt. Unter Unix wird der Internet Superdaemon durch die Datei `/etc/inetd.conf` konfiguriert. Eine weitere Ressourcen-Ersparnis besteht darin, dass der `inetd` einfache Basisdienste wie `echo`, `daytime` usw. intern implementiert. Näheres hierzu findet sich in Kapitel 13 von [StFeRu04].

4.6 Hilfs- und Systemfunktionen (`gethostbyname` etc.)

Unix-Systeme bieten eine Reihe Hilfsfunktionen, die im Zusammenhang mit der Socket-Programmierung nützlich sein können. Dazu zählen `gethostbyname`, `getsockname`, `getpeername` und `shutdown`. Die Details der Nutzung sind in den zugehörigen man-Pages zu finden. Als Beispiel dient der Fall, dass man basierend auf dem Rechnernamen Informationen zu einem Rechnersucht. Für die entsprechende Anfrage beim DNS-Server nutzt man die Funktion `gethostbyname()`, die als Ergebnis einen Pointer auf folgende Struktur (in `<netdb.h>` definiert) liefert:

```
struct hostent {
    char *h_name;           // official name of host
    char **h_aliases;       // alias list
    int h_addrtype;         // host address type
    int h_length;            // length of address
    char **h_addr_list;     // address list from name server
#define h_addr h_addr_list[0] // address for backward compatibility
};
```

Die Funktion wird aufgerufen mit

```
struct hostent *gethostbyname(char *hostname);
```

In der Struktur oben ist `h_addrtype` immer `AF_INET` und `h_length` enthält immer den Wert 4. Für Internetadressen ist `h_addr_list` eine Liste von Pointern auf Daten vom Typ `struct in_addr`, die weiter oben definiert wurde. Wie man sehen kann, ist die Struktur `hostent` sehr allgemein ausgelegt. Auch Rechner mit mehr als einer Adresse können damit behandelt werden. Zur Realisierung dieser Funktion wird normalerweise das Domain-Name-System herangezogen.

Kompilierung und Aufruf des Beispielprogramms `check_host` liefern z. B.:

```
westerka@si0024-11:~/> gcc check_host.c -g -ocheck_host
westerka@si0024-11:~/> ./check_host www.google.de
official name of www.google.de is www.google.de
    addr. type = 2, addr. length = 4
    IP address: 173.194.113.152
    IP address: 173.194.113.159
    IP address: 173.194.113.143
    IP address: 173.194.113.151
```

In neueren Unix-Versionen wird `gethostbyname()` als obsolet (veraltet) gekennzeichnet. Dazu steht im Ubuntu-Forum: „P.S. It appears that `gethostbyname()` is an obsoleted function; apparently either `getaddrinfo()` or `getnameinfo()` should be used instead.“

4.7 Socket-Programmierung unter Windows

Die Socket-Programmierung hat unter Unix ihren Ursprung genommen, steht aber auch auf vielen anderen Betriebssystemen und Programmiersprachen zur Verfügung. Unter Windows beispielsweise sind für die Nutzung von Socketfunktionen folgende Schritte erforderlich:

- Mehrere Windows-spezifische Bibliotheken müssen inkludiert werden:


```
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
```

- Eine passende lib-Datei muss beim Linken hinzugefügt werden:
`#pragma comment (lib, "Ws2_32.lib")`
- Sockets müssen vor ihrer Erzeugung initialisiert werden. Dazu dient folgender Code-Abschnitt:

```
// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}
```

Beispielprogramme für Windows finden sich unter (zuletzt aufgerufen am 28.09.2016):

TCP-Server: [http://msdn.microsoft.com/en-us/library/ms737593\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms737593(v=vs.85).aspx)

TCP-Client: [http://msdn.microsoft.com/en-us/library/ms737591\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms737591(v=vs.85).aspx)

4.8 Socket-Programmierung unter Java

Java hat bereits im normalen Sprachumfang die Unterstützung verteilter Anwendungen integriert. Aktiviert werden die Socket-Schnittstellen durch Einbinden des Paketes `java.net`, das verschiedene Klassen für die gängigen Socket-Operationen zur Verfügung stellt:

TCP-bezogene Klassen (verbindungsorientiert):	UDP-bezogene Klassen (Datagramm-orientiert):
<code>Socket</code> <code>ServerSocket</code> <code>URL</code> <code>URLConnection</code>	<code>DatagramPacket</code> <code>DatagramSocket</code> <code>MultiCast</code>

4.8.1 Java TCP-Sockets

Für TCP-Verbindungen gibt es die Klassen `ServerSocket` und `Socket`. `ServerSocket` stellt einen Socket auf der Server-Seite dar. Dieser wird durch Aufrufen des Konstruktors (`new`) gestartet und wartet mit `accept` auf eingehende Verbindungsanforderungen. Sie werden nach Rückkehr aus `accept` durch das Erzeugen einer neuen Socketinstanz bearbeitet. Die `ServerSocket`-Klasse erzeugt auf der Server-Seite einen Socket, der durch Clients gerufen werden kann. Es gibt Methoden, um den Socket empfangsbereit zu machen, ihn zu schließen und den zugehörigen Port zu erfahren. Bei unvorgesehenen Ereignissen gibt es eine `IOException`. Die `Socket`-Klasse wird verwendet, um TCP-Verbindungen von einem Client zu einem Server aufzubauen. Es gibt Methoden für Verbindungsauf-/abbau, Verbindungsinformationen und Stream-Kommunikation.

4.8.2 Beispiel JavaTCP_server - einfacher TCP-Echo-Server unter Java

Die Verwendung des `ServerSocket` in einem einfachen Echo-Server zeigt JavaTCP_Server:

```
package javatcp_server;
import java.io.*; import java.net.*;
public class JavaTCP_Server {
    final static short port = 8998;
    final static int backlog = 10;
    /* Implementiert Echo-Service */
    public static void main (String[] args) {
        ServerSocket lSocket = null;
        try {
            lSocket = new ServerSocket (port, backlog);
        } catch (IOException e) {
            System.err.println("Server Socket auf " + port +
                " kann nicht erzeugt werden.");
            System.exit(1);
        }
        while (true) {
            System.out.println("Warte auf Verbindung... ");
            Socket wSocket;
            try {
                wSocket = lSocket.accept();
```

```

        } catch (IOException e) {
            System.err.println("Eingehende Verbindung nicht möglich.");
            continue;
        }
        handleConnection(wSocket);
    }
}

private static void handleConnection (Socket wSocket) {
    System.out.println("Verbunden mit " + wSocket);
    try {
        PrintStream ps = new PrintStream(wSocket.getOutputStream());
        BufferedReader in = new BufferedReader(new InputStreamReader(
            wSocket.getInputStream()));
        String clientMsg;
        while ((clientMsg = in.readLine()) != null) {
            System.out.println("> " + clientMsg);
            ps.println("Echo: " + clientMsg);
        }
        wSocket.close();
    } catch (IOException e) {
        System.err.println("Verbindung abgebrochen.");
        /* der nächste Client kann nun bedient werden */
    }
}
}

```

4.8.3 Beispiel JavaTCP_client - einfacher TCP-Echo-Client unter Java

Das folgende Programm `JavaTCP_client` realisiert den Echo-Client unter TCP.

```

import java.io.*; import java.net.*;

public class JavaTCP_client {

    public static void main(String[] args) throws IOException {
        PrintWriter out = null; BufferedReader in = null;

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        Socket echoSocket = new Socket(hostName, portNumber);

        try {
            if (args.length != 2) {System.out.println ("Falscher Aufruf");}
            echoSocket = new Socket(args[0], Integer.parseInt(args[1]));

            out = new PrintWriter (echoSocket.getOutputStream(), true);
            in = new BufferedReader (new InputStreamReader(
                echoSocket.getInputStream()));
        } catch (UnknownHostException e) { /*...*/; System.exit(1);
        } catch (IOException e) { /*...*/; System.exit(1); }
        BufferedReader stdIn = new BufferedReader(
            new InputStreamReader(System.in));
        String userInput;
        while ((userInput = stdIn.readLine()) != null) {
            out.println (userInput);
            System.out.println (in.readLine());
        }
        out.close(); in.close();
        stdIn.close(); echoSocket.close();
    }
}

```

Man sieht, dass der Verbindungsaufbau direkt durch den Konstruktorauftrag veranlasst wird. Die in C bzw. Java geschriebenen Server und Clients können in beliebiger Kombination untereinander kommunizieren.

4.8.4 Beispiel JavaTCP_server - mehrstufiger TCP-Echo-Server unter Java

Der in 4.8.2 beschriebene Java-Server ist iterativ. Um einen mehrstufigen Server, der gleichzeitig mehrere Verbindungen bedienen kann, zu erhalten, muss der bisherige Server an wenigen Stellen erweitert werden:

- Anstelle eines Prozesses ein **Thread pro Server** erzeugt.
- Im Server wird die **Runnable**-Schnittstelle implementiert,
- Das Server-Objekt wird mit **start()** in einem separatem Thread gestartet.
- Der Aufruf der **run()**-Methode durch das Laufzeit-System wird nach Thread-Start ausgeführt.

Die Methode **handleConnection()** des Objektes in **run()** kann nebenläufig ausgeführt werden. Der vollständige Code für den mehrstufigen Java-Server ist in **JavaTCP_Server_MT** zu sehen:

```
package javatcp_server_mt;
import java.io.*; import java.net.*;
public class JavaTCP_Server_MT implements Runnable {
    final static short port = 8998; final static int backlog = 10;
    Socket wSocket;
    JavaTCP_Server_MT(Socket s) {
        this.wSocket = s;
    }

    public static void main (String[] args) {
        ServerSocket lSocket = null;
        try {
            lSocket = new ServerSocket (port, backlog);
        } catch (IOException e) {
            System.err.println("Server Socket auf " + port +
                " kann nicht erzeugt werden.");
            System.exit(1);
        }
        while (true) {
            System.out.println("Warte auf Verbindung... ");
            Socket wSocket;
            try {
                wSocket = lSocket.accept();
                /* wSocket in eigenem Thread ausfuehren */
                new Thread (new JavaTCP_Server_MT (wSocket)).start();
            } catch (IOException e) {
                System.err.println("Eingehende Verbindung nicht moeglich.");
                continue;
            }
            handleConnection(wSocket);
        }
    }
    @Override
    public void run() {}

    private static void handleConnection (Socket wSocket) {
        /* wie bisher */
    }
}
```

4.8.5 Socket-Programmierung mit dem .net-Framework unter C#

Das .net-Framework bietet ähnlich Java eine komfortable Laufzeitumgebung, die u.a. die Nutzung von Netzwerkressourcen erleichtert. Anders als in Java ist die Nutzung mit einer Vielzahl von Programmiersprachen (beliebt sind VB#, C++ und C#) möglich. Allerdings gibt es eine offizielle Laufzeitumgebung nur für Windows-Betriebssysteme. Näheres findet sich (zuletzt besucht am 28.09.2016) unter:

[https://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.net.sockets.tcpclient(v=vs.110).aspx)

Ein TCP-Client in C# findet sich bei den Beispielprogrammen unter CS_Sockets.

4.8.6 Web-Sockets

Das W3C führte 2011 in HTML5 WebSockets ein. Auf die Besonderheiten wird in Kapitel 7 eingegangen.

4.9 Stärken / Schwächen datenorientierter Programmierung verteilter Systeme mit Sockets

Wie zum Ende des vorherigen Kapitels angekündigt, wird nun einen Bewertung der Socketprogrammierung anhand der aufgestellten Anforderungen durchgeführt.

Anforderung	Kommentar
<i>Netzwerkunabhängigkeit:</i> Clients und Server sollten auf unterschiedlichen Netzwerktypen arbeiten können.	Stärke: Die Socket-Programmierung unterstützt neben TCP/IP weitere Protokollarten z. B. Token Ring und kann auch mit Dateisystemen umgehen.
<i>Rechner-/Betriebssystem-Portabilität:</i> Clients und Server sollten ohne Änderungen (außer ggf. Neukompilierung) auf verschiedenen Rechnern und Betriebssystemen arbeiten.	Stärke: Uneingeschränkt gegeben, sobald eine Netzwerkanbindung vorhanden ist.
<i>Portabilität bezüglich der Programmiersprache:</i> Clients und Server sollten sich verstehen, auch wenn sie mit unterschiedlichen Programmiersprachen entwickelt wurden.	Stärke: Uneingeschränkt gegeben, sobald eine Netzwerkanbindung vorhanden ist.
<i>Transparente Nutzung:</i> Programmierer sollen Dienste auf entfernten Rechnern wie lokale Operationen nutzen können.	Stärke: Gegeben, wird durch Inkludieren der Socket-Bibliotheken bzw. Packages realisiert. Aber: Daten und Steuerung sind auf dem Datenkanal gemischt.
<i>Einfaches Auffinden und Adressieren:</i> Es soll einfach sein, Server und Dienste aufzufinden und zu adressieren.	Schwäche: höherer Programmieraufwand, denn Server kann nur durch Vereinbarung von Servernamen/IP-Adresse aufgefunden werden. Unterschiedliche Funktionen müssen durch Ports oder eigenes Protokoll implementiert werden.
<i>Geringer Protokoll-Overhead:</i> Durch die verwendeten Protokolle sollte möglichst wenig Protokoll-Overhead erzeugt werden, um unnötige Verzögerungen und Kosten, z. B. bei Mobilverbindungen, zu vermeiden.	Stärke: Minimaler Protokoll-Overhead, wenige kurze Steuerbefehle, Sockets selbst reichen Daten 1:1 durch.

5 Funktionsorientierte Programmierung vert. Systeme - Remote Procedure Calls (RPCs)

Die bisher vorgestellten Methoden der Client-Server-Programmierung hatten zum Ziel, ein Client- und ein Server-Programm zu entwickeln, bei denen die Kommunikation durch ein definiertes Protokoll abgewickelt wird. Über den Socket wurden sowohl Kommandos als auch Daten übermittelt (z. B. bei SMTP in RFC 821 beschrieben) oder es war unmittelbar klar, welche Rückgabedaten erwartet werden (z. B. beim Echo-Server).

Wir wollen nun die Daten mit Funktionen kombinieren. Es soll also für Clients möglich sein, an Funktionen eines Servers Argumente zu übertragen und das Ergebnis der Funktionsbearbeitung als Rückgabewert zu erhalten. Durch diese abstraktere Sichtweise lösen wir uns vom datenorientierten Socket, über den sowohl Protokollbestandteile als auch Daten übertragen werden. Aus Programmiersicht verwenden wir ein Modell, dass sich an die prozedurale Programmierung anlehnt. Die zu RPC gehörigen Protokolle bleiben für den Programmierer weitgehend unsichtbar. In diesem Zusammenhang werden die Begriffe Prozedur und Funktion synonym verwendet.

Außerdem müssen folgende Schwierigkeiten der Socket-Programmierung adressiert werden:

- Bisher wurden die zur jeweiligen Anwendung gehörenden Ports selbst definiert und verwaltet. Dies ist nicht nur lästig, sondern auch fehleranfällig, wenn z. B. die vorgegebenen Portnummern nicht geändert werden.
- Bei einer Aktualisierung der Programme mussten Client und Server gleichzeitig aktualisiert werden, da sonst die Kommunikation nicht mehr konsistent ist. Man würde sich also einen Mechanismus wünschen, der Versionen unterstützt. Der Server könnte dann parallel neue und alte Version anbieten und die Clients sukzessive aktualisiert werden.
- An der Netzwerkschnittstelle soll ein Protokoll realisiert werden, das aufgerufene Funktionen und zugehörige Daten trennt. Dabei müssen wir allerdings darauf achten, dass RPC prinzipiell auch mit anderen Programmiersprachen als ANSI-C genutzt werden soll. Also muss es zur Definition der Protokolle und der Daten eine Abstraktion von ANSI-C geben. Dabei wollen wir natürlich die Vorteile der Typsicherheit erhalten, um bei problematischen Typkonversionen Hinweise vom Compiler zu erhalten. Daher scheiden Zeichenketten als kleinster gemeinsamer Nenner aus.

Vertiefende Literatur zu diesem Kapitel findet sich in [CoDoKi05], [Tane08], [Beng14], [StFeRu04] und [Bloo92].

5.1 Grundlagen

Schon in den 80er Jahren kam man auf die Idee, die Unterprogrammtechnik der Programmiersprachen auf das Netz zu erweitern. Statt wie bisher das aufrufende Programm und das Unterprogramm auf einem Rechner ablaufen zu lassen wurden Techniken entwickelt, die es erlauben, ein Unterprogramm auf einem entfernten Rechner ausführen zu lassen. Man kann Unterprogramme parallel auf vielen Rechnern ablaufen lassen, um Berechnungen zu beschleunigen. Man nennt die Urform dieser Technik Remote Procedure Call (RPC).

Das folgende Bild zeigt den lokalen und den entfernten Funktionsaufruf im Vergleich:

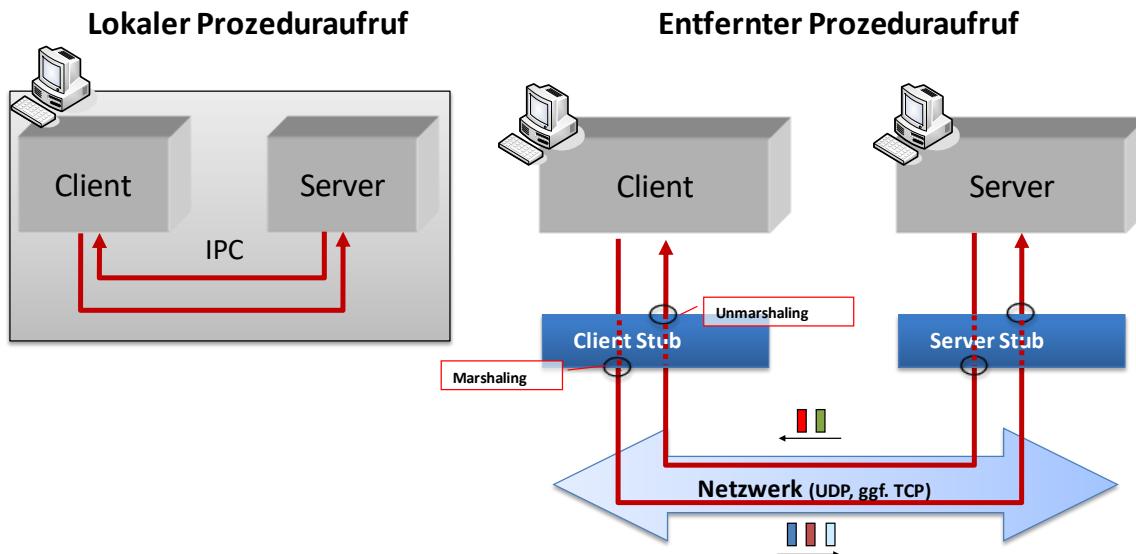


Abbildung 13: Lokaler und entfernter Prozederaufruf

Zur Unterstützung dieser Programmierung sind eine Reihe von Systemen entwickelt worden. Sie haben folgende Eigenschaften:

- Die Darstellung der ausgetauschten Daten ist maschinenunabhängig, um die verteilte Programmierung auf unterschiedlichen Rechnern zu ermöglichen.
- Für den Austausch von Daten und den Aufruf von Funktionen wird ein standardisiertes Protokoll verwendet.
- Bei der High-Level-Programmierung werden RPC-Prozeduren direkt genutzt.
- Bei der in der Praxis vorherrschenden Low-Level-Programmierung werden in einer Definitionsdatei RPC-Prozeduren mit ihren Argumenten und Rückgabewerten beschrieben. Ein Protokoll-Compiler erzeugt daraus Schnittstellen-Stümpfe, die Low-Level-RPC-Prozeduren verwenden.
- Viele RPC-Systeme besitzen Authentisierungsdienste, Verzeichnisdienste (Directory Service, Naming Service), Zeitsynchronisationsdienste und ein verteiltes Dateisystem.

Seit vielen Jahren haben mehrere Firmen RPC-Systeme entwickelt. Bekannt geworden sind u.a. SUN ONC, HP NCS und OSF DCE. Die meisten Unix-basierten Systeme (und die Veranstaltung "Verteilte Systeme") nutzen Sun ONC, während Microsoft OSF DCE favorisierte und daraus objektorientierte Ansätze wie DCOM, OLE etc. entwickelt hat. RPC-Systeme sind auf prozedurale Programmiersprachen ausgerichtet (C, Pascal usw.).

5.2 Programmierung mit SUN ONC

Für die Entwicklung einfacher RPC-Anwendungen werden nur wenige spezielle Anweisungen benötigt. Der Anwender entwickelt sein Programm ähnlich wie lokal ablaufende Programme, übersetzt den Code und linkt ihn mit entsprechenden Bibliotheken für die RPC-Prozeduren, um einen RPC-Client- und einen RPC-Server zu erhalten. Diese Vorgehensweise soll an einem einfachen Beispiel gezeigt werden.

5.2.1 Beispielprogramm ls (list directory contents) Verzeichnisliste lokale Version

Als erstes Beispiel wird ein einfaches Programm entwickelt, das eine Liste der Dateien in einem Verzeichnis auf einem entfernten Rechner ermittelt und überträgt. Dies ist ein praktisch relevantes Beispiel, da einer der wichtigsten Unix-Dienste, das Networked File System (NFS) mit RPC realisiert ist. Alle Informatik-Laborbereiche der Hochschule Osnabrück nutzen (trotz unterschiedlicher Infrastrukturen wie Solaris, Linux und MacOS) NFS-Varianten z. B. für den Zugang zu den Home-Verzeichnissen der Studierenden.

Zunächst wird die lokale Variante `ls` des Programmes entwickelt. Es besteht aus folgenden Dateien

- `ls.c` (Hauptprogramm) und
- `read_dir.c` enthält die Funktion zur Ermittlung der Dateien in einem Verzeichnis (Verzeichnisliste). Sie wird später unverändert bei entfernten Aufrufen (als Server-Funktion eingebunden) benutzt.

Das Hauptprogramm `ls.c` ruft die Funktion `read_dir` auf:

```
// Beispielprogramm ls: Erstellen und Ausgeben einer Verzeichnisliste
// Quelle: Bloomer, Power RPC Programming

#include <stdio.h>
#include <string.h>

#define DIR_SIZE 8192
// C. Westerkamp: Vorwärtsdeklaration, um Warnings zu vermeiden extern
void exit(int);
int main(int argc, char *argv[]) {

    char dir[DIR_SIZE];
    if( argc != 2 ) {
        printf("Fehler, Aufruf: %s <Verzeichnis>\n\n", argv[0]);
        exit(0);
    }
    strcpy(dir, argv[1]);
    // call local procedure
    read_dir(dir);

    // print the result
    printf("%s\n", dir);
}
```

Man sieht, dass die Übergabe der Daten durch Zeiger geschieht. Dies ist ein Grundprinzip, das bei allen RPC-basierenden Programmen eingehalten werden muss. In der Kommandozeile wird der Verzeichnisname übergeben. Er wird der Funktion `read_dir` in der Zeichenkette `dir` übergeben. In derselben Zeichenkette wird bei der Rückgabe der Verzeichnisinhalt zurückgegeben.

`read_dir` ist in der separaten Datei `read_dir.c` definiert:

```
// read_dir.c: list a directory Quelle: Bloomer Power RPC Programming
// adaptiert von C. Westerkamp
#include <dirent.h>
#include <stdio.h>

char *read_dir(char *dir) {
    static DIR *dirp=(DIR *)NULL; // for opendir()
    struct dirent *d;
    // open directory
    dirp = opendir(dir);
    if( dirp == NULL ) return(NULL);

    // pack filenames into dir buffer
    // Verzeichnisnamen ueberschreiben mit Verzeichnisliste
    while( d=readdir(dirp) ) {
        sprintf(dir, "%s%s\n", dir, d->d_name);
    }

    // return the result
    closedir(dirp);
```

```
        return(dir);
    }
```

Näheres zu `opendir`, `dirent` etc. findet man auf den entsprechenden man-Pages eines Unix-Systems. Die einzelnen Einträge der Struktur `dirent` werden sequenziell in die Zeichenkette `dir` geschrieben werden. Werden die Programmteile übersetzt und gebunden (z. B. `gcc -o myls ls.c read_dir.c`), so ergibt der Aufruf (je nach Verzeichnisinhalt):

```
westerka@si0024-1:~/> ./myls ..kap4
.
..
ls.c
myls
read_dir.c      usw.
```

Wir wollen nun dieses lokale Beispiel auf entfernten Aufruf mit RPC-Prozeduren umstellen.

5.2.2 Beispielprogramm rls für Low-Level-RPC-Programmierung

Dieses Beispiel soll zeigen, wie ein Client-Programm über RPC ein Server-Programm aufruft, das die Dateien in einem Server-Verzeichnis als Liste zurückliefert. Die Funktion `read_dir` wird also auf dem entfernten Rechner anstatt lokal aufgerufen, ohne dass der Aufruf sich sonderlich unterscheidet (Prinzip der transparenten Nutzung).

Generell sind bei der RPC-Nutzung drei Schritte zu durchlaufen:

1. Auf dem Serverrechner wird die Prozedur angemeldet. Das zur Prozedur gehörende Serverprogramm macht sich dazu bei einem standardisierten RPC-Dämon, dem Portmapper, bekannt. Dabei wird u.a. vereinbart, auf welchem Port es Anfragen entgegen nimmt.
2. Das Client-Programm wendet sich an den portmap-Dämon des Server-Rechners, um zu erfahren, auf welchem Port es das Server-Programm ansprechen kann. Das eigene Verwalten von Ports entfällt also.
3. Anschließend sendet der Client dem Server die Anfrage an den zuständigen Port, um die Prozedur aufzurufen.

Das folgende Bild veranschaulicht die Reihenfolge der drei Schritte:

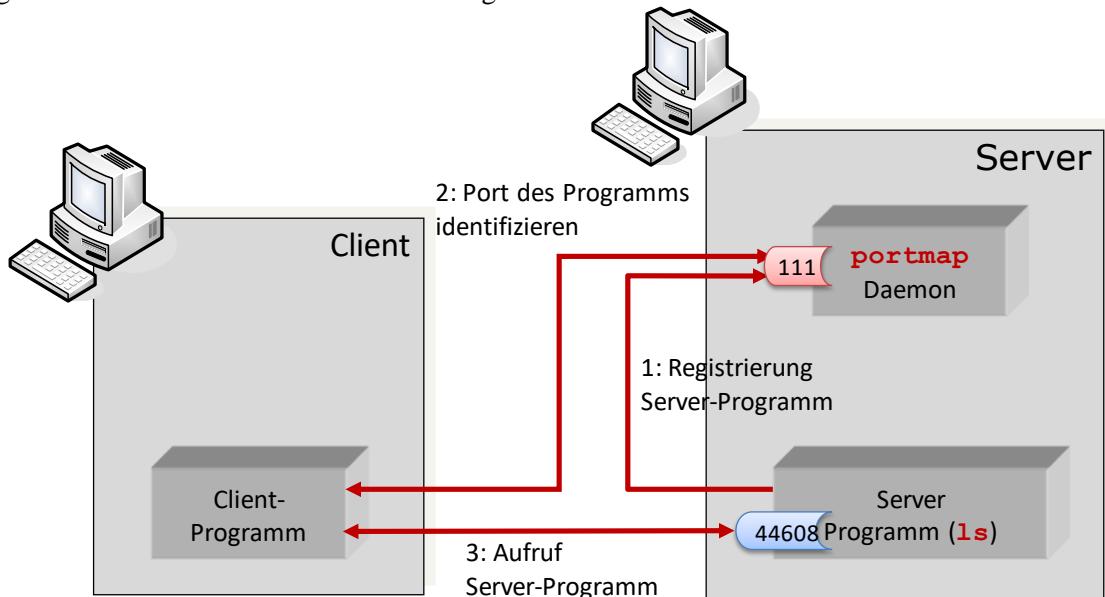


Abbildung 14: Ablauf beim entfernten Prozeduraufruf

Für die Umsetzung als RPC-Programm muss eine spezielle Form der ausgetauschten Daten (External Data Representation - XDR) festgelegt werden, damit das RPC-Prinzip auch zwischen Rechnern unterschiedlicher Bauart funktioniert. Für verschiedene Standarddatentypen (wie z. B. Integer) stehen vorgefertigte XDR-Funktionen zur Verfügung.

Zur Registrierung der RPC-Funktion beim Server (Schritt 1 im ersten Bild auf der vorherigen Seite) dient die Funktion `registerrpc()`. Bei einer Anfrage eines Clients nach diesen Nummern (Schritt 2) teilt der Portmapper dem Client eine entsprechende Port-Nummer für den Server mit. Diese bleibt dem Client-Programm aber verborgen und wird intern von den RPC-Prozeduren verwendet, um den Prozederaufruf (Schritt 3) auszuführen. Mit `program`, `version` und `procedure` ist ein Aufruf eindeutig.

5.2.3 Bedeutung von `program`, `version` und `procedure`

Bei der Registrierung der RPC-Prozeduren ist eine dreifache Hierarchie vorgesehen. Sie besteht auf der obersten Ebene aus „Programmen“, darunter eine oder mehrere „Versionen“, die schließlich mehrere „Prozeduren“ enthalten. Dies ist im folgenden Bild dargestellt:

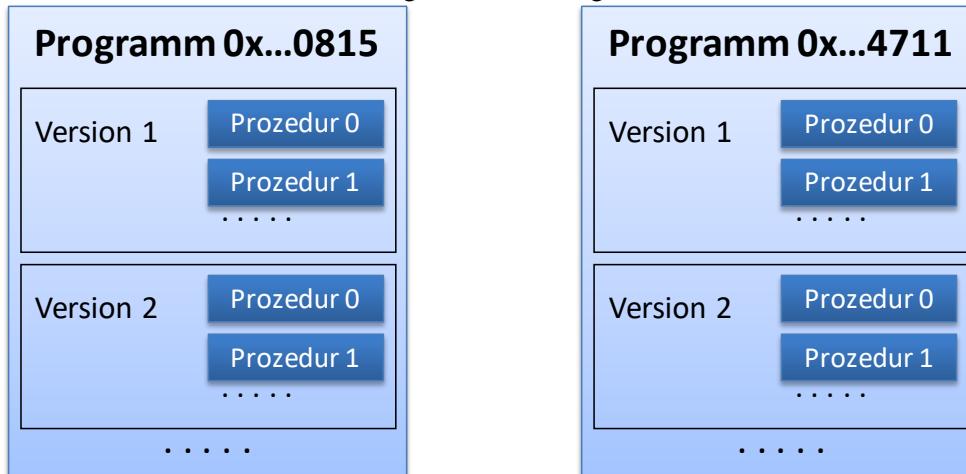


Abbildung 15: Dreifach verschachtelte Strukturierung in SUN ONC

`program` dient dazu, zusammen gehörende Prozeduren zu bündeln. Auf diese Weise kann ein Server unterschiedliche Dienste als "Prozedurbündel" mit passendem Namen anbieten. Bei `registerrpc()` (s.o.) werden die Größen `program`, `version` und `procedure` als `unsigned long int` beim Portmapper registriert, wodurch eine eindeutige Zuordnung möglich ist. SUN hat für einige RPC-Dienste `program`-Nummern vergeben.

Daher muss man sich an das folgende Schema halten:

Bereich	Beschreibung
0x00000000 - 0x1FFFFFFF	Definiert durch SUN
0x20000000 - 0x3FFFFFFF	Benutzerdefiniert
0x40000000 - 0x5FFFFFFF	Transient
0x60000000 - 0xFFFFFFFF	Reserviert

Möchte man einen RPC-Dienst weltweit registrieren lassen, sollte man sich an SUN wenden. Neben der Programmnummer (z. B. 0x20000000) werden Version (z. B. 1) und in je einer Zeile alle aufgerufenen Prozeduren des `program`'s angegeben. Die Version dient dazu, Aktualisierungen und Erweiterungen von der Ursprungsversion zu unterscheiden. Mit jeder neuen Version kann das Procedure-Bündel verändert werden, neue Fähigkeiten erlangen etc.. Möchte man nun eine neue Version "ausrollen", bietet man sie zusätzlich zur alten Version auf dem Server an. Die Clients können nun sukzessive auf die neue Version umgestellt werden und die erweiterten Fähigkeiten nutzen. Sobald alle Clients aktualisiert sind, kann die alte Version abgeschaltet werden. Im Gegensatz zu diesem eleganten Vorgehen ist es bei der Socket-Programmierung notwendig, bei verändertem Server alle Clients gleichzeitig zu aktualisieren.

Für Registrierung der RPC-Funktion wird im Server die Funktion

```
#include <rpc/rpc.h>
int registerrpc(u_long proignum, u_long versnum, u_long procnum, char
*(procname)(), xdrproc_t inproc, xdrproc_t outproc);
```

benötigt. Dabei werden in `proignum`, `versnum` und `procnum` die drei Nummern angegeben, die den Aufruf eindeutig machen. Außerdem wird ein Pointer auf die entsprechende Funktion angegeben. Wenn später der Server eine Anforderung mit den Nummern für Programm, Version und Prozedur erhält, wird er die zugehörige Funktion aufrufen. Die Funktionen `inproc()` und `outproc()` sind XDR-Funktionen, die die Konvertierung der Argumente und Resultate von der XDR-Darstellung in die C-Darstellung bzw. umgekehrt vornehmen. Nach der Registrierung legt sich das Serverprogramm schlafen (es blockiert) und wartet mit der Funktion `svc_run()` auf eingehende Anforderungen. Auf der Seite des Clients wird die RPC-Funktion aufgerufen durch

```
#include <rpc/rpc.h>
int callrpc(char *host, u_long proignum,
u_long versnum, u_long procnum,
xdrproc_t inproc, char *in,
xdrproc_t outproc, char *out);
```

Der zeitliche Ablauf des Aufrufes der Prozedur geschieht wie folgt:

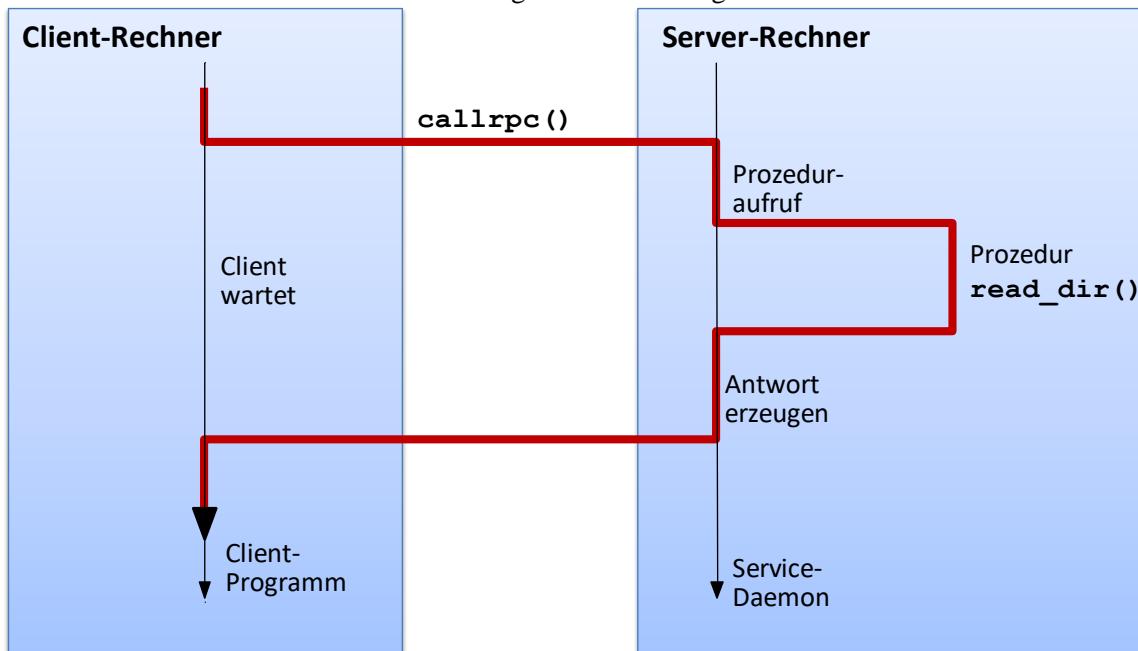


Abbildung 16: Zeitlicher Ablauf des Prozedurauftrufs

Für die Konvertierung der Daten werden nun noch XDR-Funktionen benötigt. In diesem Beispiel kann dies von einer Funktion erledigt werden, da Argumente und Rückgabewerte die gleiche Form haben.

Wir wollen, um die prinzipielle Sichtweise der XDR-Konvertierung zu verstehen, das Verzeichnislistenbeispiel näher betrachten. Später werden wir das Behandeln der XDR-Konvertierung dem Protokoll-Compiler überlassen. Der Name des Verzeichnisses, zu dem die Verzeichnisliste angefordert wird, kann genauso wie die Verzeichnisliste selbst als Zeichenkette übertragen werden. Diese arbeiten ohnehin mit Zeigern. Wir können also für die Funktion `xdr_dir()` direkt die XDR-Bibliotheksfunktion `xdr_string()` nutzen:

```
bool_t xdr_dir(XDR *xdrs, char *objp) {
    register int32_t *buf;
    if (!xdr_string(xdrs, &objp, DIR_SIZE))
        return FALSE;
    else return TRUE;
}
```

So wie in diesem Beispiel lassen sich viele XDR-Funktionen verwenden.

Fast alle Konvertierungsfunktionen haben folgendes Format:

```
bool_t xdrproc(XDR *xdrs, ... *arg, ...);
```

Das erste Argument heißt XDR-Handle. Es ist eine Adresse, an der die XDR-Darstellung der Daten abgelegt wird. Das zweite Argument ist die Adresse, an der die C-Darstellung der Daten abgelegt wird. Hier ist wichtig, dass alle Daten durch Zeiger übergeben werden. Manchmal werden weitere Argumente benötigt, um die Konvertierung durchzuführen. Diese folgen dann und sind abhängig vom zu konvertierenden Datentyp. Die XDR-Funktionen arbeiten immer in beide Richtungen, d.h. sie können von der XDR- in die C-Darstellung umwandeln und umgekehrt.

Welche Richtung genau gemeint ist, wird in einem Feld des XDR-Handle definiert:

- **XDR_ENCODE**
wandelt von der C- in die XDR-Darstellung um. Das Resultat steht anschließend an der Stelle `*xdrs`.
- **XDR_DECODE** wandelt von der XDR- in die C-Darstellung um. Das Resultat steht anschließend an der Stelle `*arg`. Ist der Pointer NULL, so wird auch gleichzeitig Speicher allokiert.
- **XDR_FREE** gibt den von XDR_DECODE angelegten Speicher wieder frei.

Um die verwendeten Server-RPC-Hilfsfunktionen richtig zu verstehen, schauen wir uns Abschnitte eines von Hand geschriebenen Server-Programms (Ansatz der High-Level-Programmierung) näher an:

```
// Ausschnitt aus Server-Programm ohne Protokolldefinition
// (High-Level-Programmierung)
// Deklaration der externen Konvertierungsfunktion:
extern bool_t xdr_dir(XDR *, char *);
// Deklaration der externen RPC-Funktion
extern char *read_dir(char *);
// Registrierung von Programm, Programmversion und RPC-Funktion
// unter Angabe der Hin- und Rückkonvertierungsfunktionen
registerpc(DIRPROG, DIRVERS, REaddir, read_dir, xdr_dir);

// Start der Verarbeitung
svc_run();
}
```

Man sieht, dass beim Registrieren der RPC-Prozedur neben den drei Informationen zu ihrer eindeutigen Identifizierung der Prozedurname selbst und die Konvertierungsfunktionen für Hin- und Rückweg (hier gleich) angegeben werden.

Die Funktion `read_dir()` wurde aus dem lokalen Beispiel 1:1 übernommen. Die Funktion weiß nichts davon, dass der Ausgangspunkt für den Aufruf nicht lokal ist. Diese einfache Programmvariante verzichtet auf eine Fehlerbehandlung. Das Serverprogramm ist iterativ, d.h. es wird jeweils eine Anforderung behandelt und abgeschlossen, bevor die nächste an der Reihe ist. Konkurrerende Server sind aber auch mit RPC möglich. Dann muss allerdings für jede Anforderung ein neuer Prozess oder ein neuer Thread gestartet werden.

Auf der Client-Seite wird eine adaptierte Version der Funktion `read_dir()` verwendet, die anstelle der bisherigen Implementierung den Aufruf `callrpc()` enthält:

```
// Ausschnitt aus High-Level-RPC-Client
// Deklaration der lokalen RPC-Prozedur
void read_dir(char *, char *);

int main(int argc, char *argv[]) {
    char dir[DIR_SIZE];
    // Aufruf der entfernten RPC-Prozedur
    strcpy(dir, argv[2]);
    read_dir(argv[1], dir);

    // Ausgabe des Ergebnisses
    printf("%s\n", dir);
    exit(0);
}
```

```
// Lokale RPC-Prozedur leitet mittels callrpc Aufruf an Server weiter
void read_dir(char *host, char *dir) {
    extern bool_t xdr_dir(XDR *, char *);
    enum clnt_stat clnt_stat;

    clnt_stat = callrpc(host, DIRPROG, DIRVERS, REaddir,
                        (xdrproc_t)xdr_dir, dir, (xdrproc_t)xdr_dir, dir);
    if( clnt_stat != 0 ) clnt_perrno(clnt_stat);
}
```

5.2.4 Beispiel math - entfernter Aufruf mathematischer Funktionen

Eine wichtige Anwendung verteilter Programmierung war und ist die Verteilung komplexer Rechenaufgaben auf mehrere Rechner. Hierzu dienten z. B. Ansätze wie PVM (Parallel Virtual Machine). Der Autor dieses Skripts hat 1992 mit einer DOS-Bibliothek der Firma Sun Rechenaufgaben zur Berechnung von langsamem PCs auf schnellen SparcStations ausgelagert. Heute ist Grid-Computing für viele Bereiche mit Bedarf an hoher Rechenleistung ein wichtiges Thema. Als zugehöriges Beispiel für Low-Level-RPC-Programmierung soll nun ein mathematisches Programm aus [Beng14] betrachtet werden. Darin wird gezeigt, wie Strukturen mit Werten übergeben und verarbeitet werden.

Die wesentlichen Eigenschaften sind wieder in der Definitionsdatei `math.x` ersichtlich:

```
// math.x
// Mathematisches Low-Level-RPC-Beispiel mit Strukturen
// Quelle: Bengel, Verteilte Systeme

struct intpair {
    int a;
    int b;
};

program MATHPROG {
    version MATHVERS {
        int ADD(intpair) = 1;
        int MULTIPLY(intpair) = 2;
        int CUBE(int) = 3;
    }= 1;
} = 536870920;
```

Es gibt drei Prozeduren für Addition, Multiplikation und Berechnung der dritten Potenz der beiden Werte in der Struktur.

Die Serverprozeduren enthalten keine Neuigkeiten:

```
// math_svc_proc.c Mathematisches Low-Level-RPC-Beispiel mit
// Strukturen, Serverprozeduren zum Aufrufen aus dem Server-Stub
// Quelle: Bengel, Verteilte Systeme

#include <rpc/rpc.h>
// math.h wird von rpcgen erzeugt
#include "math.h"

int* add_1_svc(intpair *pair, struct svc_req *req) {
    static int result;
    result = pair->a + pair->b;
    return(&result);
}
```

```

int* multiply_1_svc (intpair *pair, struct svc_req *req) {
    static int result;
    result = pair->a * pair->b;
    return(&result);
}

int* cube_1_svc (int *base, struct svc_req *req) {
    static int result;
    int baseval = *base;
    result = baseval * baseval * baseval;
    return(&result);
}

```

Im Client math_client.c gibt es ebenfalls keine Besonderheiten:

```

// math - Mathematisches Low-Level-RPC-Beispiel mit Strukturen
// Client-Programm mit Aufrufen in den Client-Stub
// Quelle: Bengel, Verteilte Systeme

#include <stdio.h>
#include <rpc/rpc.h>
/* math.h wird von rpcgen erzeugt */
#include "math.h"

int main (int argc, char *argv[]) {
    CLIENT *c1;           /* client handle */
    intpair numbers;
    int *result;

    if (argc != 4) {
        fprintf(stderr,"Aufruf: math_client Zahl1 Zahl2 \n");
        exit(1);
    }

    c1 = clnt_create(argv[1], MATHPROG, MATHVERS, "tcp");
    if (c1 == NULL) {
        clnt_pcreateerror(argv[1]);
        exit(1);
    }
    numbers.a = atoi(argv[2]);
    numbers.b = atoi(argv[3]);

    result = add_1(&numbers, c1);
    if (result == NULL) {
        clnt_perror(c1, "add_1");
        exit(1);
    }
    printf("Die add(%d, %d)-Prozedur lieferte %d\n",
           numbers.a, numbers.b, *result);
    result = multiply_1(&numbers, c1);
    if (result == NULL) {
        clnt_perror(c1, "multiply_1");
        exit(1);
    }
    printf("Die multiply(%d, %d)-Prozedur lieferte %d\n", numbers.a,
           numbers.b, *result);
}

```

```

        result = cube_1(&numbers.a, c1);
        if (result == NULL) {
            clnt_perror(c1, "cube_1");
            exit(1);
        }
        printf("Die cube(%d)-Prozedur lieferte %d\n", numbers.a, *result);
        exit(0);
    }
}

```

Interessant ist ein Detail in der von rpcgen erzeugten Konvertierungsfunktion in `math_xdr.c`.

```

bool_t xdr_intpair (XDR *xdrs, intpair *objp)
{
    register int32_t *buf;
    if (!xdr_int (xdrs, &objp->a))
        return FALSE;
    if (!xdr_int (xdrs, &objp->b))
        return FALSE;
    return TRUE;
}

```

Man sieht, dass bei der Konvertierung der einzelnen Elemente die Standardkonvertierungsfunktionen (hier für Ganzzahlen) `xdr_int` verwendet werden. Nach Kompilierung des Programmes (siehe Makefile) ergibt die Ausführung z. B.

```

westerka@si0024-1:~/> ./math_server
westerka@si0024-2:~/> ./math_client SERVER 3 4
Die add(3, 4)-Prozedur lieferte 7
Die multiply(3, 4)-Prozedur lieferte 12
Die cube(3)-Prozedur lieferte 27

```

Weitere RPC-basierende Programme mit Übergabe von Strukturen können nach dem gleichen Prinzip entwickelt werden.

5.2.5 Beispielprogramm RPC_No_Seg_Fault für High-Level-RPC-Programmierung

In Linux-Versionen seit 9.1 kann `xdr_dir` nicht wie beschrieben implementiert werden, da bei der Freigabe mit `free()` ein Segmentation Fault verursacht wurde. Im Kernel-Patch-Log des 2.6-Kernels gibt es einen entsprechenden Fehlereintrag, die Ursache wurde aber bisher nicht behoben. Ein älterer News-Beitrag zu Sun-Solaris berichtet von einem Fehler in XDR-Funktionen, der wahrscheinlich die Ursache des Fehlers ist. Eine Aufteilung des Hin- und des Rückweges in jeweils eigene unidirektionale Implementierungen behebt den Fehler.

Dies ist in folgenden Programm-Code-Ausschnitt zu sehen ist:

```

// File: rls.h
// Author: Eikerling - Created on 8. April 2010, 21:35

bool_t xdr_arg (XDR *xdrs, char *objp) {
    // Diese Variante funktioniert. xdr_string() oder
    // xdr_wrapstring() fuehrt zu Seg. Fault.
    return (xdr_opaque (xdrs, objp, DIR_SIZE));
}

bool_t xdr_res (XDR *xdrs, char *objp) {
    // Fuer das Resultat kann xdr_string() verwendet werden.
    return (xdr_string (xdrs, &objp, DIR_SIZE));
}

```

Der vollständige Programm-Code ist unter `RPC_No_Seg_Fault` in den Beispielprogrammen zu finden.

Zum Testen des rls-Beispiels muss vor dem Starten des Servers der portmap Daemon gestartet sein. Dies geschieht in der Regel als Superuser (`/sbin/portmap`). Ob der Portmapper läuft, lässt sich mit `rpcinfo -p localhost` überprüfen. Dann wird `rls_server` gestartet und ist unter 536870912 (= 0x2000000) sichtbar. Danach kann der Client `rls_client` aufgerufen werden.

Die bisher beschriebene direkte Verwendung der High-Level Funktionen ohne Protokoll-Compiler ist zum Verständnis der Vorgänge hilfreich. Sie wird aber nur bei einfachen Programmen verwendet. Wenn man das Schreiben der Definitionsdatei einmal erlernt hat, ist die im nächsten Abschnitt beschriebene Low-Level-RPC-Programmierung einfacher und leistungsfähiger.

5.3 Low-Level-RPC-Programmierung

Um die High-Level-Programmierung zu umgehen, kann man einen Protokoll-Compiler nutzen. Dieser erzeugt aus einer Protokollspezifikation in einer Definitionsdatei die notwendigen Dateien automatisch. Da diese Low-Level-RPC-Prozeduren aufrufen, wird die Nutzung des Protokoll-Compilers auch Low-Level-RPC-Programmierung genannt.

5.3.1 Der RPC-Protokollcompiler

Der Protokoll-Compiler `rpcgen` erzeugt aus einer Definitionsdatei Client- und Server-Stümpfe, die ihrerseits Low-Level-RPC-Prozeduren verwenden. Der Anwender muss zwar die Sprache RPCL für die abstrakt beschriebene Protokollspezifikation beherrschen, spart aber das eigene Ausformulieren der Konvertierungsfunktionen.

Der Entwicklungsprozess für eine RPC Anwendung ist in folgender Abbildung gezeigt:

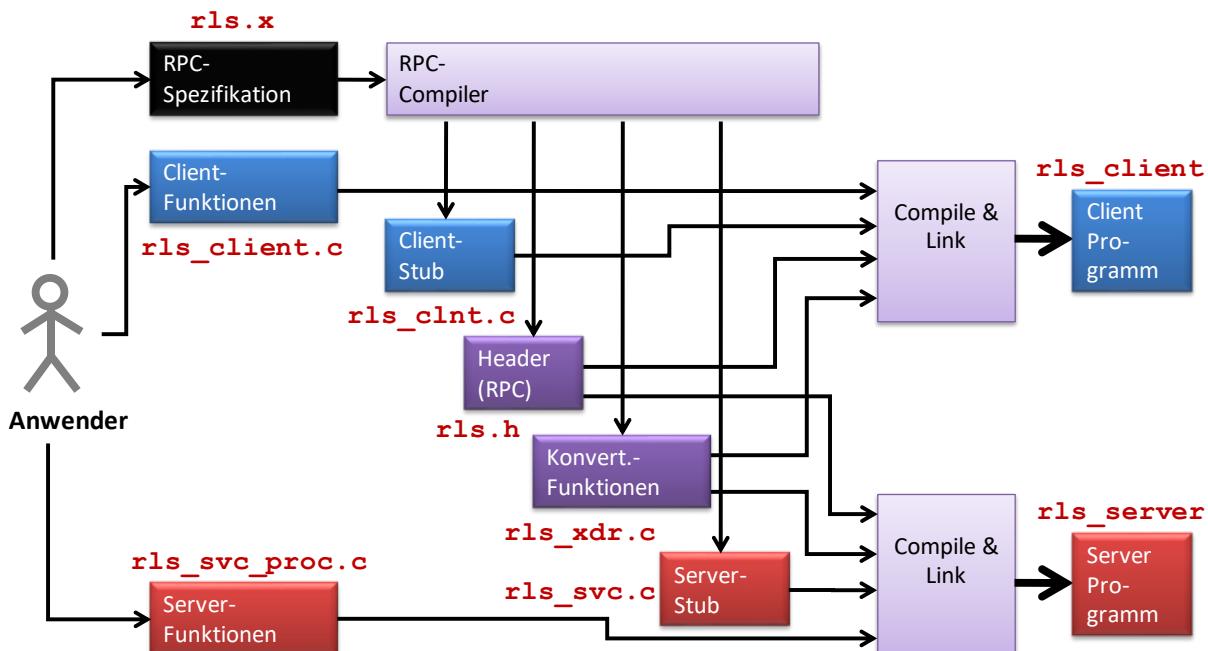


Abbildung 17: Automatische Dateigenerierung bei Low-Level-RPC-Programmierung

Die RPC-Spezifikation beschreibt die aufgerufenen Prozeduren, die übergebenen Argumente und die Rückgabewerte.

Ist die Definitionsdatei z. B. `rls.x`, werden durch folgende Dateien durch `rpcgen` erzeugt:

- XDR-Funktionen in `rls_xdr.c`,
- eine gemeinsame Include-Datei `rls.h`,
- ein Client-Stub `rls_clnt.c` und
- ein Server-Stub `rls_svc.c`.

Die Konventionen für Dateinamen sind fest im RPC-Compiler installiert. Der RPC-Compiler wird aufgerufen durch

`rpcgen infile`

Das Kommando `rpcgen` kennt eine ganze Reihe von Optionen, die sich auf verschiedene Details der erzeugten Dateien beziehen. Natürlich kann ein RPC-Compiler keine Anwendungen erfinden. Neben der Definitionsdatei müssen daher folgende Teile müssen vom Anwender erstellt werden:

- das Server-Programm z. B. in `rls_svc_proc.c` mit den vom Server-Stub aufzurufenden Funktionen.
- das Client-Programm `rls_client.c`. In ihm werden die definierten RPC-Prozeduren angefordert.

Die in der Definitionsdatei verwendete Definitionssprache soll nun näher betrachtet werden.

5.3.2 Die RPC Definitionssprache RPCL

Die Definitionssprache für die Eingangsdateien von `rpcgen` (z. B. `rls.x`) ist RPCL, eine Erweiterung von XDR, das wir bereits als abstrakte Repräsentation von Dateninhalten kennengelernt haben.

Die RPC-Protokolldefinition setzt sich aus Definitionen zusammen, die durch Semikolon getrennt werden. Dabei gibt es sechs Typen, von denen die ersten fünf an Standarddatentypen der Programmiersprache C angelehnt sind:

- `const` für Konstanten
- `enum` für Aufzählungen
- `struct` für Strukturen
- `union` für Unions
- `typedef` für selbstdefinierte Typen
- `program` für Informationen über Programm, Version und Prozeduren

Es müssen, wie sonst auch, alle Größen vor ihrer Benutzung definiert werden. Die unterschiedlichen Definitionen dürfen in beliebiger Reihenfolge in der `.x`-Datei auftauchen.

Konstanten (const)

werden definiert durch eine `const` Definition der Form:

```
const MAX_SIZE = 8192;
```

Bei Konstanten sind nur ganze Zahlen erlaubt. `rpcgen` erzeugt in der Header-Datei eine Zeile der Form:

```
#define MAX_SIZE 8192
```

Aufzählungen (enum)

werden definiert durch eine `enum`-Definition der Form:

```
enum color {  
    RED = 0,  
    GREEN = 1,  
    BLUE = 2  
};
```

`rpcgen` erzeugt hieraus folgenden C-Code in der Header-Datei:

```
enum color {  
    RED = 0,  
    GREEN = 1,  
    BLUE = 2  
};  
typedef enum color color;  
bool_t xdr_color();
```

Typdefinitionen (typedef)

Durch die `typedef`-Anweisung ist es möglich, einen Datentyp z. B. `color` direkt zu verwenden. Außerdem wird automatisch ein XDR-Filter `xdr_color()` der Datei mit den Konvertierungsfunktionen (`_xdr.c`) erzeugt. Strukturen werden definiert wie in C:

```
struct point {  
    int x;  
    int y;  
};
```

`rpcgen` erzeugt daraus in der Header-Datei

```
struct point {
```

```

    int x;
    int y;
};

typedef struct point point;
bool_t xdr_point();

```

Durch die `typedef`-Anweisung kann der Typ `point` später direkt in den Programmen verwendet werden. Gleichzeitig wird wieder eine XDR-Funktion erzeugt.

Durch eine union-Definition wird eine Äquivalenz angegeben, in der Form:

```

union result switch(int error) {
    case 0:
        opaque data[MAX_SIZE];
    default:
        void;
};

```

Es handelt sich hierbei um diskrete Äquivalenzen, die deutliche Unterschiede zu C-Unions haben. In unserem Beispiel bestimmt ein Rückgabewert, ob noch Daten folgen (wenn er 0 ist), oder ob keine Daten folgen. rpcgen erzeugt daraus in der Header-Datei:

```

struct result {
    int error;
    union {
        char data[MAX_SIZE];
    } result_u;
};

typedef struct result result;
bool_t xdr_result();

```

Zusätzliche Typdefinitionen lassen sich mit der `typedef`-Anweisung angeben, wie:

```
typedef point poly[4];
```

Diese Anweisung wird eins-zu-eins in die Header-Datei übernommen.

Programmdefinitionen

sind der einzige Abschnitt, in dem nicht übergebene Daten, sondern Programm und Prozeduren beschrieben werden. Sie erlauben die Spezifikation des Programmnamens, der Nummer, Version und der Prozeduren des Server-Programms wie zuvor beschrieben:

```

program TEST_PROGRAM {
    version TEST_VERSION {
        void TEST_PROC(void) = 1;
    } = 1;
} = 0x20000000;

```

rpcgen macht in der Header-Datei daraus:

```

#define TEST_PROGRAM ((u_long)0x20000000)
#define TEST_VERSION ((u_long)1)
#define TEST_PROC ((u_long)1)
extern void *test_proc_1();

```

Zu beachten ist, dass die C-Funktionen Pointer als Resultate liefern. Die Service-Funktionen werden immer zusammen mit XDR-Filterfunktionen verwendet, die mit Pointern für die Umwandlungen von der XDR- in die C-Darstellung und umgekehrt arbeiten. Wenn man sich über den Datentyp der Service-Funktionen nicht sicher ist, sollte man einen Blick in `p4c_svc.c` werfen, in der diese Funktionen aufgerufen wird. Die 0 ist als Prozedurnummer nicht erlaubt, da rpcgen automatisch eine Prozedur `NULLPROC` mit dieser Nummer generiert. Sie dient zum Test der Erreichbarkeit des Servers. Für die Deklaration von Variablen gibt es insgesamt vier Möglichkeiten:

Einfache Deklarationen werden geschrieben als:

```
color c;
```

Dies wird so in die Header-Datei übernommen. Felder fester Länge werden geschrieben als:

```
color palette[8];
```

Dies wird so in die Header-Datei übernommen. Felder variabler Länge gibt es nicht in C. Sie werden geschrieben:

```
// bis zu MAX_SIZE Elemente
int x<MAX_SIZE>;
// jede Laenge
int y<>;
```

Sie werden für C in Strukturen übersetzt:

```
struct {
    u_int x_len;
    int *v_val;
} x;
```

Pointer werden in XDR genauso deklariert, wie in C. Es macht allerdings keinen Sinn, Speicheradressen über das Netz zu übermitteln. Pointer werden daher nur verwendet als Referenzen auf andere Daten.

```
struct point {
    int x;
    int y;
};

struct poly {
    point *p;
    point *pNext;
};
```

rpcgen definiert für jeden Variablenotyp eine Filterfunktion. Der C-Code in appl.h ist dann:

```
struct point {
    int x;
    int y;
};

typedef struct point point;
bool_t xdr_point();

struct poly {
    point *p;
    point *pNext;
};

typedef struct poly poly;
bool_t xdr_poly();
```

XDR definiert noch einige weitere Datentypen, die es in C so nicht gibt: XDR kennt den Datentyp bool, dessen Werte TRUE oder FALSE sein können.

```
bool busyFlag;
```

Er wird übersetzt in:

```
boot_t busyFlag;
```

Genauso kennt C keinen Datentyp für Strings. Es verwendet hingegen eine Sequenz von Zeichen, die durch '\0' abgeschlossen wird. Dies wird auch von XDR so beibehalten. In RPCL werden Strings aber deklariert als:

```
string buffer<32>;
string longBuff<>;
```

Sie werden übersetzt in:

```
char *buffer;
char *longBuff;
```

Die maximale Länge in buffer zählt das Nullzeichen nicht mit. Sie beeinflusst die Arbeit der entsprechenden XDR-Routine. Daten ohne festen Datentyp lassen sich angeben als:

```
opaque fixData[512];
opaque varData<1024>;
```

Es handelt sich dabei um eine Anzahl Bytes, die nicht weiter umgewandelt werden. Sie werden übersetzt in:

```
char fixData[512];
struct {
    u_int varData_len;
    char *varData_val;
} varData;
```

void-Deklarationen werden nur in Unions und Programm-Definitionen verwendet. Zur Umwandlung wird die Funktion `xdr_void()` verwendet.

5.3.3 Grundregeln für Datentypen an der Netzwerkschnittstelle

Da insbesondere im Web-Bereich viel mit schwach typisierten Programmiersprachen gearbeitet wird, lohnt sich eine prinzipielle Betrachtung der Vorgänge an der Netzwerkschnittstelle.

Viele SW-Entwickler haben schon am eigenen Leib erlebt, dass das verspätete Finden von Fehlern zu einem größeren Aufwand bei der Beseitigung führt. Historisch gesehen hat man daher die Typsicherheit bei C++ deutlich stärker ausgeprägt als bei C. Auf diese Weise erhält man schon bei der Entwicklung Fehlermeldungen wie „Implizite Typkonversion“, die darauf hinweisen, dass man mit Datentypen inkonsistent umgegangen ist. Ein Ignorieren dieser Meldungen kann zu schwer zu findenden Fehlern z. B. bei Wertebereichüberschreitungen führen. Bezogen auf die Programmierung verteilter Systeme führen diese Überlegungen zu folgenden Grundregeln, die im zugehörigen Praktikum einzuhalten sind:

Grundregel 1: An der Schnittstelle sollten aus Gründen der Typsicherheit und –Prüfung immer die Datentypen eingesetzt werden, die auch in der Anwendung verwendet werden. Dies gilt sowohl für Basisdatentypen wie auch für zusammen gesetzte Datentypen (z. B. struct). Fatal und unnötig fehleranfällig wäre es z. B. wenn man die IP-Adresse in eine Zeichenkette wandelt, diese dann über die Schnittstelle übermittelt und am Schluss wieder zurück konvertiert.

In diesem Zusammenhang ist aufgrund der Datensicherheit auch über den Umfang der zu übermittelnden Daten nachzudenken (Datensparsamkeit). Speichert man verschiedene Datensätze z. B. in einer Struktur, so benötigt man häufig nur einen Teil daraus für die weitere Auswertung.

Beispiel: In einem Sportverein werden die Mitgliedsdaten in Datensätzen gespeichert, die u. a. die Kontonummer für den Einzug der Mitgliedsbeiträge, das Geburtsdatum und die Sportgruppe etc. enthalten. Je nach Rolle des Nutzers der Daten wird man nun z. B. den Gruppenleitern Listen mit Namen, Vornamen und Geburtsdatum zur Verfügung stellen. Der Kassenwart (aber nur der) benötigt hingegen die Bankverbindung und der Vorsitzende das Geburtsjahr, um genügend Ehrennadeln für die Ehrung der Jubilare zu bestellen. In allen Fällen wird man aus Gründen der Datensicherheit nie den ganzen Datensatz, sondern nur die tatsächlich benötigten Daten übermitteln. An der funktionsorientierten Netzwerkschnittstelle bedeutet dies für die Datentypen der Argumente und Rückgabewerte, dass zu jeder Auswertungsvariante ein passendes struct mit der passenden Untermenge der übergebenen Daten verwendet wird. Dies führt zu folgender Regel:

Grundregel 2: Man sollte aus Gründen der Datensicherheit und Datensparsamkeit immer nur die Informationen übermitteln, die tatsächlich benötigt werden. Sollte also ein struct zu viele Informationen enthalten, verwendet man an der Schnittstelle ein Übergabe-struct mit der passenden Untermenge.

In der Praxis wird man häufig verkettete Listen für die Übermittlung solcher Auswertungen verwenden. Die zugehörigen Auswertungsstrukturen enthalten dann genau die passende Untermenge zuzüglich Zeiger auf das nächste Glied der verketteten Liste.

5.3.4 Notwendige Low-Level-Funktionen

Auch wenn rpcgen den Hauptteil des Protokolls zwischen Client und Server realisiert, kommt man nicht umhin, einige wenige RPC-Prozeduren selbst aufzurufen. Diese Funktionen sind:

`cInt_create()`

Für jede Client-Server Verbindung, muss der Client eine Struktur (der Client-Handle) verwalten. Dieser Handle wird dann vielen anderen RPC-Prozeduren mit übergeben.

Zur Erzeugung dieses Handles dient die Funktion:

```
CLIENT *clnt_create(char *host, u_long proignum,  
                     u_long versnum, char *protocol);
```

Als Transport-Protokoll kann `tcp` oder `udp` ausgewählt werden, wobei UDP-Messages nicht länger als 8 KByte werden können. Die Funktion liefert einen Client-Handle, wenn das Programm beim Server registriert ist (beim Portmapper). Im Fehlerfall liefert die Funktion `NULL`.

```
clnt_destroy()
```

Die Kommunikation zum Server wird durch die Funktion

```
void clnt_destroy(CLIENT *clnt);
```

abgebrochen. Interner Speicherplatz wird wieder frei gemacht.

```
clnt_control()
```

Nach dem Eröffnen einer Verbindung zum Server können Eigenschaften durch

```
bool_t clnt_control(CLIENT *client, const u_int reg, char *info);
```

geändert werden. Dazu gehört z. B. der Timeout-Wert für die Verbindung.

```
clnt_call()
```

Mit der Funktion

```
enum clnt_stat clnt_call(CLIENT *cl,  
                           u_long procnum,  
                           const xdrproc_t inproc,  
                           const caddr_t in,  
                           const xdrproc_t outproc,  
                           const caddr_t out,  
                           const struct timeval tout);
```

wird der eigentliche RPC-Aufruf durchgeführt. Die Umwandlung von bzw. in die XDR-Darstellung wird durch die Funktionen `inproc` und `outproc` vorgenommen. Auf der Server-Seite sind in aller Regel kaum RPC-Prozeduren innerhalb der eigentlichen Anwendung erforderlich. Die notwendigen Funktionen werden im Server-Stub ausgeführt. Wichtig ist, dass bei der Registrierung eines Dienstes, IP-Sockets sowohl für den TCP- als auch den UDP-Transport angelegt werden.

5.3.5 Beispiel rls Remote Listing Service - Low-Level-RPC-Programmierung

Als Beispiel wird der Remote-Listing Dienst herangezogen, der mittels Protokolldefinition und Code-Generierung durch `rpcgen` realisiert wird. Zunächst muss die Protokolldefinition `rls.x` erzeugt werden:

```
// rls.x: Definitions for remote listing protocol  
const MAXNAMELEN = 512;  
// directory entry  
typedef string nametype<MAXNAMELEN>;  
  
// a linked list for directory entries  
typedef struct namenode *namelist;  
  
// a node in the list  
struct namenode {  
    nametype name;  
    namelist pNext;  
};  
  
// the result of the REaddir operation  
union readdir_res switch(int remoteErrno) {  
    case 0:  
        namelist list;  
    default:  
        void;
```

```
};

// the directory program definition
program DIRPROG {
    version DIRVERS {
        readdir_res REaddir(nametype) = 1;
    } = 1;
} = 0x20000001;
```

Es wird ein Feld variabler Länge für den Verzeichnisnamen, der dem Server gesendet wird, verwendet. Des Weiteren wird eine verkettete Liste von `namenode` Elementen aufgebaut, um den Verzeichnisinhalt bei der Rückgabe vom Server zum Client zu definieren. Jedes Element der Liste enthält den Namen eines Verzeichniseintrages, sowie einen Pointer auf das nächste Element. Für diese verkettete Liste wird des Weiteren der Typ `namelist` eingeführt. Die Server-Prozedur wird in der `program` Definition festgelegt. Sie erhält als Parameter ein Argument vom Typ `nametype` und liefert als Resultat ein Ergebnis vom Typ `readdir_res`. Dieser Typ wiederum ist eine Union, zusammengesetzt aus der Fehlervariablen `errno` und einer `namelist`, sofern beim Aufruf kein Fehler auftauchte.

Als nächstes werden mit `rpcgen` aus dieser Datei die Dateien `rls.h`, `rls_clnt.c`, `rls_svc.c` und `rls_xdr.c` erzeugt. Zum näheren Verständnis des Generierungsvorgangs und der darin generierten Funktionen lohnt sich ein Blick in diese Dateien.

Wie bei allen generierten Quelldateien macht es keinen Sinn, darin zu editieren, da Änderungen beim nächsten Erzeugen überschrieben werden. Als nächstes muss die Server-Prozedur `rls_svc_proc.c` entwickelt werden:

```
// Beispiel rls_svc_proc.c
// Server procedure for directory listing
// Quelle: Bloomer, Power RPC Programming
// adaptiert von C. Westerkamp, getestet unter Ubuntu 14.04
#include <rpc/rpc.h>
#include <dirent.h>
#include <errno.h>
#include "rls.h"

extern int errno;
extern void *malloc();
extern char * strdup();

readdir_res *.readdir_1_svc(nametype *dirname, struct svc_req *request)
{
    namelist nl;
    namelist *nlp;

    // Muss als static vereinbart werden, damit das Resultat
    // weiter verarbeitet werden kann.
    static readdir_res res;
    // static, damit wir sehen können, ob es bereits
    // einen Aufruf gab
    static DIR *dirp = NULL;
    struct dirent *d;
    // Hinweis: struct svc_req *request
    // enthält in req->rq_xprt->xp_raddr.sin_addr.s_addr
    // die IP-Adresse des aufrufenden Clients
    // Verzeichnis öffnen
    dirp = opendir(*dirname);
    if( dirp == NULL ) {
        res.remoteErrno = errno;
```

```

        return(&res);
    }

    // Speicher aus vorheriger Konvertierung freigeben
    if( dirp ) xdr_free((xdrproc_t) xdr_readdir_res, (char *)&res);
    // Durch Verzeichnis laufen
    nlp = &res.readdir_res_u.list;
    while( d = readdir(dirp) ) {
        nl = *nlp = (namenode *)malloc(sizeof(namenode));
        nl->name = strdup(d->d_name);
        nlp = &nl->pNext;
    }
    *nlp = NULL;
    // Rückgabe Ergebnis, Argument wird ueberschrieben
    res.remoteErrno = 0;
    closedir(dirp);
    return(&res);
}

```

Ihr wesentlicher Inhalt ist die Funktion `readdir_1_svc()`. Als Argument wird ein Pointer auf eine Variable vom Typ `nametype` übergeben. Der Rückgabewert ist ein Pointer auf `readdir_res` und muss als static deklariert werden, damit er nach dem Verlassen der Funktion nicht seinen Wert verliert (wie bei automatischen Variablen). Auf diese Weise haben später XDR-Funktionen Zugriff auf dies Daten. Die verkettete Liste für die Verzeichniseinträge wird in der Funktion dynamisch aufgebaut. Da sie später noch von XDR-Funktionen benötigt wird, kann der Speicher dieser Struktur in der Funktion nicht freigegeben werden. Stattdessen wird dies am Anfang durch `xdr_free()` erledigt, wodurch belegter Speicher des letzten Durchlaufs wieder freigegeben wird.

Als letztes wird jetzt noch das Hauptprogramm des Clients `rls_client.c` benötigt.

```

// rls_client.c - remote directory-listing client
// Quelle: Bloomer, Power RPC Programming

#include <stdio.h>
#include <rpc/rpc.h>
#include <errno.h>
#include "rls.h"

extern int errno;
int main(int argc, char *argv[])
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if( argc != 3 ) {
        fprintf(stderr, "usage: %s host directory\n", argv[0]);
        exit(1);
    }

    // Command line arguments
    server = argv[1];
    dir = argv[2];

    // Client handle erzeugen (mit TCP Protokoll)
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");

```

```

if( cl == NULL ) {
    // Verbindungsauftbau erfolglos
    clnt_pcreateerror(server);
    exit(1);
}
// Eigentlicher Aufruf der entfernten Prozedur
result = readdir_1(&dir, cl);
if( result == NULL ) {
    // Fehler beim RPC-Aufruf
    clnt_perror(cl, server);
    exit(1);
}
// RPC-Aufruf erfolgreich, Resultat auswerten
if( result->remoteErrno != 0 ) {
    // RPC-Fehler aus Server
    errno = result->remoteErrno;
    perror(dir);
    exit(1);
}
// Resultat ohne Fehler, Verzeichnisliste ausgeben
for( nl=result->readdir_res_u.list;
     nl != NULL;
     nl = nl->pNext) {

    printf("%s\n", nl->name);
}
exit(0);
}

```

Mit `clnt_create()` wird ein Client-Handle erzeugt, wobei auch gleich das Transport-Protokoll festgelegt wird. Außerdem hat man nach dem Aufruf die Gewissheit, dass eine Verbindung zum Server besteht. Für die Behandlung von Fehlern werden die Funktionen `clnt_pcreateerror()` für RPC-Fehler, `clnt_perror()` für fehlerhafte RPC-Funktionsaufrufe und `perror()` für Unix-Fehler verwendet. Anschließend folgen Übersetzen und Linken, wie z. B. in Makefile im Verzeichnis `rls` definiert:

```

all: rls_client rls_server
    rls_client: rls_client.c rls_xdr.c rls_clnt.c rls.h
        gcc -o rls_client rls_xdr.c rls_clnt.c rls_client.c
    rls_server: rls_svc_proc.c rls_svc.c rls_xdr.c rls.h
        gcc -o rls_server rls_xdr.c rls_svc_proc.c rls_svc.c
clean:
    rm rls_client rls_server

```

Denken Sie daran, dass im Makefile hinter dem Doppelpunkt und dem Zeilenumbruch ein Tabulator (und nicht mehrere Leerzeichen) stehen müssen.¹

Beschwert sich das Makefile über fehlende Dateien, hat man wahrscheinlich den rpcgen-Lauf vergessen.

¹ Bei der Entwicklung eigener RPC-Programme kann es bei der Nutzung der dynamischen Speicherverwaltung (z.B. für die verketteten Listen) zu Speicherzugriffsfehlern kommen. Bei Unsicherheit in diesem Themenbereich lohnt es sich, die entsprechenden Skriptseiten aus den ersten beiden Semestern zu konsultieren. Außerdem ist das Hilfsprogramm valgrind bei der Fehlersuche hilfreich. Nutzung: `valgrind -v <ausführbare Datei>` bzw. `/usr/bin/valgrind ...`

Der Test zeigt, dass wieder der Inhalt des als Argument übergebenen Verzeichnisses ausgegeben wird:

```
westerka@si0024-1:~/> ./rls_server
westerka@si0024-2:~/> ./rls_client 131.173.110.1 ../rls
.
..
rls.h
rls.x
rls_client.c      usw.2
```

Das folgende Bild veranschaulicht, welche Software-Abschnitte in welchen Dateien bei Ablauf eines RPC-Aufrufs durchlaufen werden:

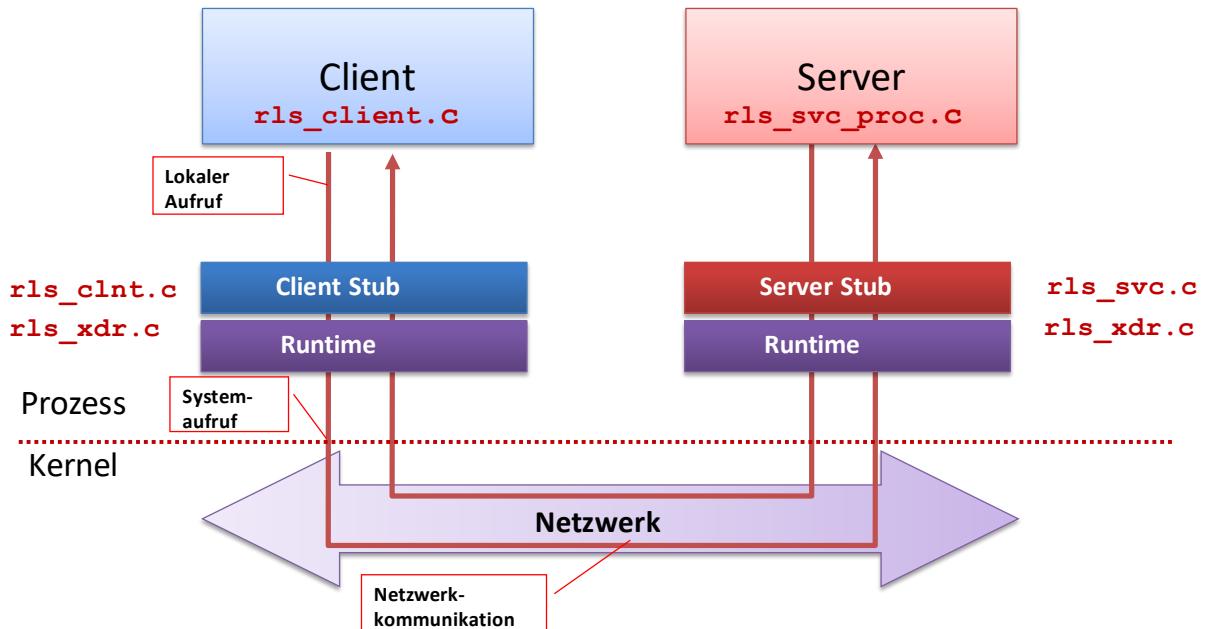


Abbildung 18: Durchlaufen verschiedener Software-Abschnitte beim RPC-Aufruf

5.4 RPC-Programmierung unter Solaris und Windows

Das unter Solaris nutzbare RPC (entspricht dem Original-Sun RPC) unterscheidet sich von der Linux-Variante nur durch etwas verkürzte Parameterlisten bei den Aufrufen. Das in Windows eingebaute Microsoft RPC (MS RPC) hingegen basiert auf OSF/DCE RPC und unterscheidet sich von ONC RPC grundlegend. MS RPC kann mit anderen OSF/DCE RPC-Varianten zusammenarbeiten, aber nicht mit ONC RPC.

5.5 Asynchrone RPC-Verarbeitung

Im bisherigen Beispiel wurde die entfernte Prozedur synchron durch den Client aufgerufen, d.h. dass der Client nach dem Aufruf der entfernten Prozedur auf dessen Beendigung wartet. Dies ist in vielen Fällen nicht wünschenswert, denn der Vorteil von RPC ist ja gerade die Parallelverarbeitung. Der Client soll also seine Arbeit fortsetzen, um später die Ergebnisse des Servers zu verarbeiten. Zum Erreichen dieses Verhaltens gibt es zwei Ansätze:

- One-Way-Verfahren
- Follow-Up-Verfahren.

² Ob der portmap-Daemon gestartet ist, lässt sich durch `ps -edf | grep portmap` überprüfen
Ob der Server läuft, kontrolliert man `rpcinfo -p localhost`.

Das folgende Bild zeigt die prinzipiellen Abläufe der Verfahren, die in den nachfolgenden Abschnitten näher erläutert werden:

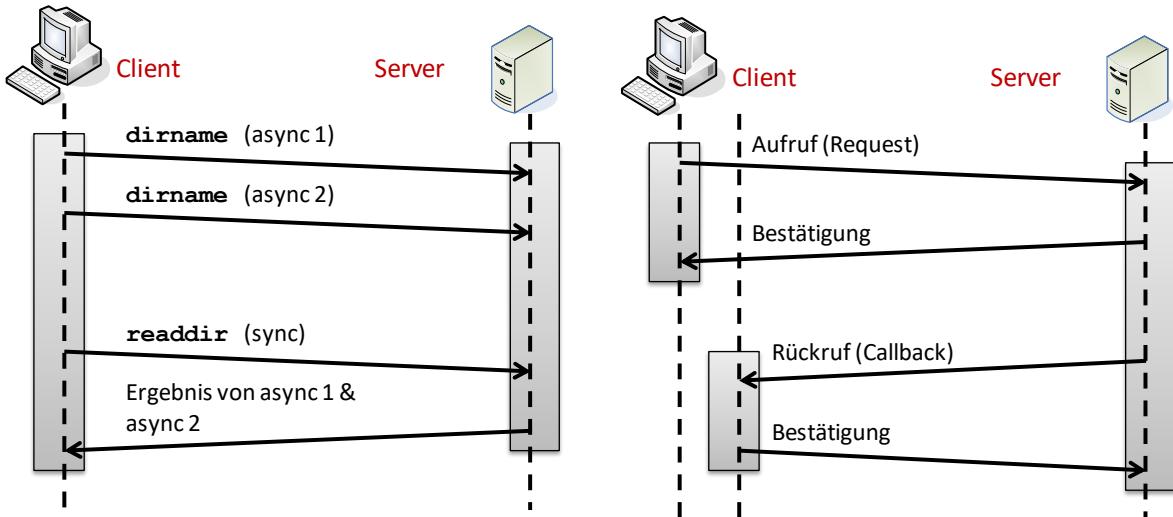


Abbildung 19: Ablauf des One-way- (links) und des Follow-Up-Verfahrens (rechts) zur Bearbeitung asynchroner RPC-Anfragen

5.5.1 One-Way-Verfahren

Man kann das Warten (Blockierung) des Clients mit-RPC Mitteln vermeiden, indem man einen sogenannten One-Way-Request absetzt. Dazu wird beim Aufruf der RPC-Prozedur ein vom Standardwert abweichender Timeout-Wert von 0 angegeben. Der RPC-Aufruf erfolgt, aber die Funktion kehrt sofort zum Client zurück. Der Server erhält die Anforderung, ruft die Prozedur auf, sendet aber keine Resultate zurück, weil der Client diese zunächst nicht verarbeiten kann. Erst später wird das Resultat durch einen synchronen Prozedurauftrag des Client abgerufen.

Man sieht, dass mehrere Anfragen schnell hintereinander abgesetzt werden können, ohne dass auf das Ergebnis der ersten Anfrage gewartet werden muss. Die Ergebnisse der Anfragen werden durch synchronen RPC-Aufruf abgeholt. Da die Anfragen nicht sequenziell bearbeitet werden, können sie schneller erledigt werden.

5.5.2 Beispiel zur One-Way-RPC-Programmierung

Dieses Vorgehen soll am Beispiel des entfernten Verzeichnisdienstes verdeutlicht werden. Dieser soll so erweitert werden, dass in der Kommandozeile mehrere Verzeichnisse angegeben werden können. Zunächst wird dazu eine kleine Veränderung an der Protokolldefinition vorgenommen. Es wird eine zusätzliche Prozedur eingeführt, die nur die Aufgabe hat, im Server ein Verzeichnis zu lesen und das Ergebnis im Hauptspeicher abzulegen. Diese Prozedur `dirname_1()` liefert kein Resultat. Erst die Prozedur `readdir_1()`, die der Client später aufruft, fasst durch lokalen `dirname_1`-Aufruf die Ergebnisse zusammen und sendet sie zurück. Das Originalbeispiel aus [Blo92] und eine an Linux adaptierte Version sind in den Beispieldateien zu diesem Kapitel zu finden. Interessant ist dabei folgender Abschnitt aus `one-way.x`

```
program DIRPROG {
    version DIRVERS {
        void          DIRNAME(nametype) = 1;
        readdir_res  READDIR(nametype) = 2;
    } =
        1;
} =
    0x20000001;
```

rpcgen muss zweimal ausgeführt werden:

```
westerka@si0024-1:~/> rpcgen one-way.x
westerka@si0024-1:~/> rpcgen -s tcp -o one-way_svc.c one-way.x
file `one-way_svc.c' already exists and may be overwritten
```

Das zweite Kommando erzeugt einen neuen Server-Stub `one-way_svc.c`, der nur TCP als Transportprotokoll akzeptiert.

Jetzt muss noch die von rpcgen erzeugte Datei `one-way_svc.c` editiert werden. rpcgen erzeugt einen Standard-Timeout Wert, der für die One-Way-Variante nicht passt. Folgender Abschnitt wird in `one-way_svc.c` geändert:

```
// Default timeout can be changed using clnt_control()
static struct timeval TIMEOUT = { 25, 0 };
```

in

```
// Default timeout can be changed using clnt_control()
extern struct timeval TIMEOUT;
```

Die beiden Service-Prozeduren zeigen folgender Ausschnitt:

```
static readdir_res res;           // must be static!
static int      beginCycleFlag = TRUE;

void *dirname_1_svc(nametype *dirname, struct svc_req *req) {
// wie zuvor
readdir_res *readdir_1_svc(nametype *dirname, struct svc_req *req)
{
    // Record this directory's contents.
    dirname_1_svc(dirname, req);

    // Start the cycle over and return the result.
    beginCycleFlag = TRUE;
    return (&res); }
```

Es folgt der Code für den Client. Beim Aufruf der Prozedur `dirname_1()` wird der Timeout auf Null gesetzt, so dass der Aufruf sofort zurückkehrt (asynchroner Aufruf).

Erst die Prozedur `readdir_1()` holt die Resultate synchron vom Server.

```
// One-way-Client Quelle: Bloomer Power RPC Programming
#include <stdio.h>
#include <rpc/rpc.h>
#include "one-way.h"

// extern int errno;
// For clnt_call() Timeouts
struct timeval TIMEOUT = {0, 0};

int main(int argc, char *argv[]) {
    CLIENT          *cl;
    char            *server;
    char            *dir;
    readdir_res    *result;
    namelist       nl;
    int             i;

    if (argc < 3) {
        fprintf(stderr, "Usage: one-way_client directories\n");
        exit(1);
    }
    server = argv[1];
    // Create client "handle" We use the "tcp" protocol to assure
    // our one-way requests get there.
    if ((cl=clnt_create(server,DIRPROG,DIRVERS,"tcp"))==NULL) {
        clnt_pcreateerror(server);
        exit(1);
    }
    // Call the remote procedure DIRNAME on the server each time for
```

```

// all but the last directory specified on the command line.
// Set the time-out for this client handle to zero to tell
// RPC not to wait for a reply.      */
TIMEOUT.tv_sec = TIMEOUT.tv_usec = 0;
if (clnt_control(cl, CLSET_TIMEOUT, (char *)&TIMEOUT)==FALSE){
    fputs("can't zero timeout\n", stderr);
    exit(1);
}
for (i = 2; i < argc - 1; i++) {
    dirname_1(&(argv[i]), cl); // One Argument per dirname call
    clnt_perror(cl, server); // ignore time-out errors
}

// Now send a normal sync. RPC to signal the server to send the
// reply. First you must set the time-out back
// to some realistic non-zero value.
TIMEOUT.tv_sec = 25;
if (clnt_control(cl, CLSET_TIMEOUT, (char *)&TIMEOUT)==FALSE){
    fputs("can't delay timeout\n", stderr);
    exit(1);
}

if ((result = readdir_1(&(argv[i]), cl)) == NULL) {
    clnt_perror(cl, server);
    exit(1);
}
// Successfully called the remote procedures.
if (result->errno != 0) {
    // a remote system error occurred,
    // errno = result->errno;
    perror(dir);
    exit(1);
}
// Got a directory listing, print it out
for (nl=result->readdir_res_u.list; nl!=NULL; nl=nl->pNext){
    printf("%s\n", nl->name);
}
exit(0);
}

```

Diese noch verhältnismäßig einfache Anwendung hat noch Schwächen. Es wird angenommen, dass der Client zunächst RPC-Anforderungen an den Server sendet und später die relevanten Daten vom Server abruft. Wenn allerdings der Client nach der Anforderung abbricht oder ein anderer Client eine Anforderung stellt, kommt der Server in Schwierigkeiten. Ein Ausweg ist, beim Server eine Verwaltung der Anfragen zu implementieren, die z. B. Timeouts berücksichtigt. Ein anderes Verfahren, nicht-blockierende Prozedur-Aufrufe zu realisieren, ist das Follow-Up-RPC.

5.5.3 Follow-Up-Verfahren

Ein weiteres Verfahren für die asynchrone Verarbeitung von Anforderungen ist das in Abbildung 18 prinzipiell gezeigte Follow-Up-RPC:

1. Ein Client sendet eine Anforderung an den Server. Der Server sendet als Antwort eine Bestätigung, bevor er mit der Verarbeitung beginnt. Der Client setzt nach der Bestätigung seine weiteren Arbeiten fort.
2. Das Resultat der Anforderung wird vom Server an den Client übermittelt. Dort wird eine RPC-Prozedur bereitgehalten, die die Daten entgegen nimmt. Das Eintreffen einer RPC-Anforderung muss signalisiert werden.

Zu Details des Follow-Up-Verfahrens wird auf [Bloo92] verwiesen.

5.6 Ausblick zu RPCs

Über mehrere Jahre war XML RPC, eine Weiterentwicklung der RPCs sehr populär. Daraus haben sich die Web Services entwickelt, die in Kapitel 7 näher besprochen werden. Facebook hat unter dem Namen Apache Thrift (wörtliche Übersetzung: Sparsamkeit) eine RPC-Alternative entwickelt, die (u.a.) in C++, C#, Java, JavaScript, Python, Ruby und PHP unterstützt wird. Es wird gehostet durch Apache und hat experimentellen Status. Ziele waren ein sprach-unabhängiges Typsystem, bidirektonaler Datentransport, ein unabhängiger Protokoll-Layer und Weitere unterstützte Eigenschaften sind Versionierung und Generatoren zur Erzeugung von Routinen (Prozessoren) zur Verarbeitung von RPCs.

Näheres findet sich unter <http://incubator.apache.org/thrift/> (zuletzt besucht am 28.09.2016).

5.7 Stärken / Schwächen funktionsorientierter Programmierung verteilter Systeme mit RPCs

Es soll nun eine Bewertung der RPC-Programmierung in Bezug auf die in Kapitel 2 formulierten Anforderungen durchgeführt werden.

Anforderung	Kommentar
<i>Netzwerkunabhängigkeit:</i> Clients und Server sollten ohne Änderungen auf unterschiedlichen Netzwerktypen arbeiten können.	Die RPC-Programmierung beruht auf der Socketprogrammierung und kann mit verschiedenen Protokollarten umgehen.
<i>Rechner-/Betriebssystem-Portabilität:</i> Clients und Server sollten ohne Änderungen (außer ggf. Neukompilierung) auf verschiedenen Rechnern und Betriebssystemen arbeiten.	Hier gibt es Einschränkungen, da z. B. auf Windows-Systemen eine andere RPC-Variante als auf Linux- und anderen Unix-Systemen verwendet wird.
<i>Portabilität bezüglich der Programmiersprache:</i> Clients und Server sollten sich verstehen, auch wenn sie mit unterschiedlichen Programmiersprachen entwickelt wurden.	RPCs sind auf die Programmiersprache C beschränkt, in C++ und Java gibt es ähnliche Ansätze, die aber objektorientiert arbeiten (siehe Kapitel 6).
<i>Transparente Nutzung:</i> Programmierer sollen Dienste auf entfernten Rechnern wie lokale Operationen nutzen können.	Gegeben, wird durch Inkludieren der entsprechenden Bibliotheken und automatisierte Generierung von Konvertierungsfunktionen und Header-Dateien realisiert.
<i>Einfaches Auffinden und Adressieren:</i> Es soll einfach sein, Server und Dienste aufzufinden und zu adressieren.	Stärken: Protokoll-Overhead sehr gering (Konvertierung), Steuerungsbefehle kurz, Unterschiedliche Funktionen lassen sich aber anders als bei Sockets durch passende Funktionsnamen implementieren; Daten (=Übergabeargumente und Rückgabewerte) und Steuerung (Funktionsaufrufe) sind getrennt und die Kommunikation an der Netzchnittstelle erfolgt typsicher. Schwäche: Immer noch hoher Programmieraufwand, denn Server kann nur durch Servernamen bzw. IP-Adresse aufgefunden werden.
<i>Geringer Protokoll-Overhead:</i> Durch die verwendeten Protokolle sollte möglichst wenig Protokoll-Overhead erzeugt werden, um unnötige Verzögerungen und Kosten, z. B. bei Mobilverbindungen, zu vermeiden.	

6 Objektorientierte Programmierung verteilter Systeme

Wir haben bisher die daten- und die funktionsorientierte Programmierung verteilter Systeme kennen gelernt. In beiden wird ein prozeduraler Programmieransatz verfolgt. Dieser ist für das Grundverständnis von hoher Bedeutung und wird weiterhin dort verfolgt, wo es auf gute Performance, geringen Overhead und transparentes Verhalten an der Netzwerksschnittstelle ankommt. Mit C++ und Java hat sich in der SW-Entwicklung inzwischen objektorientierte Sprachen durchgesetzt. Diese bieten entweder durch leistungsfähige Spracherweiterungen (C++) oder in der Programmiersprache integrierte Packages (Java) eine gute Unterstützung verteilter Programmierung auf mittlerer Abstraktionsschicht. Naturgemäß hat man auf die eigentlichen Vorgänge weniger Einfluss als bei den bisher betrachteten Methoden der Programmierung verteilter Systeme.

Für die Methoden der Programmierung verteilter Systeme auf mittlerem Abstraktionsniveau hat sich auch der Begriff **Middleware** eingebürgert.

6.1 Architekturbetrachtungen

Bei der Entwicklung von Middleware für objekt-orientierte verteilte Systeme wurden einige Architekturaspekte berücksichtigt. Bei mehrschichtigen Architekturen (Multi-Tier) hat sich generell eine Systemeinteilung in (mindestens) folgende Schichten eingebürgert:

- Präsentation
- (Geschäfts-) Logik → Geschäftsobjekte
- Datenverwaltung

Man kann die Middleware-Entwicklung auch aus der Perspektive der Software betrachten. Die Aufteilung von Software ist bei Multi-Tier-Architekturen wesentlich modularer, denn unterschiedliche Module für Datenhaltung, Business-Logik und das Frontend können nahezu beliebig miteinander kombiniert werden. Dies wird anhand des folgenden Bildes veranschaulicht:

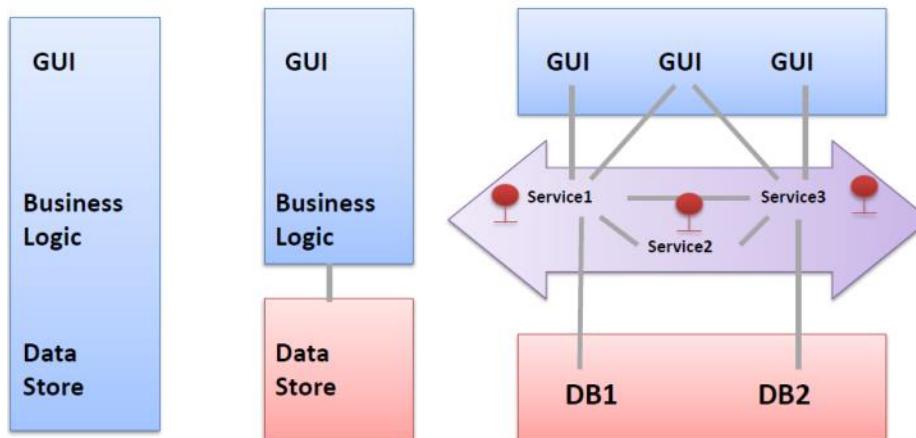


Abbildung 20: Von monolithischen zu verteilten Dreischichtenarchitekturen

Die erste Generation enthielt die drei Schichten in monolithischen Applikationen. In der zweiten wurde die Datenhaltung abgetrennt und in eigenen Datenbankprogrammen (auf eigenen Servern) mit optimierten Eigenschaften implementiert. Die dritte Generation schließlich ermöglicht eine Modularität durch eine Middleware, die verschiedene Dienste beliebiger Datenhaltungs-Anbieter für unterschiedlicher Präsentations-Module (GUIs) anbietet.

Unabhängig von Architekturanforderungen sollen bei Entwicklung eines Middleware-Ansatzes folgende Eigenschaften erreicht werden:

- Integration: Plug & Play
- Portabilität: über Systemgrenzen hinweg
- Intelligenz: z.B. bei Replikation
- Interoperabilität & Koexistenz - Unabhängigkeit von Programmiersprache, Betriebssystem (Hardware-)Plattform.

Der letzte Punkt ist wichtig bei der Anwendungsintegration, bei der Anwendungen auf unterschiedlichen Plattformen gekoppelt werden. Ein typisches Beispiel sieht folgendermaßen aus:

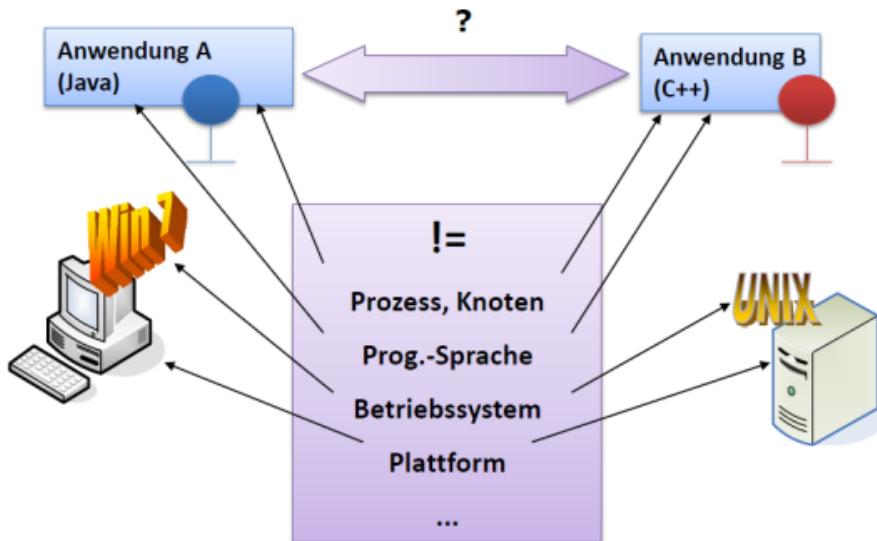


Abbildung 21: Anwendungsintegration zwischen heterogenen Anwendungen (Beispiel)

Es sollen also folgende Fragen geklärt werden:

- Wie können Anwendungen gekoppelt werden?
- Wie können Anwendungen verteilt werden?

Bei der Einführung von Middleware sollten außerdem folgende Vorteile der Objektorientierung auf verteilte Objekte übertragen werden:

- Intuitive Programmierung
Reale Welt = Ansammlung von Objekten (z.B. Bank, Account, Statement, GUI)
Klassen = Gruppen von Objekten
Kommunikation über Nachrichten
Nachrichtenarten werden in Schnittstelle der Klasse festgelegt
- Unterstützung durch OO Infrastruktur
OO Datenbanken (ODMG Standard)
OO Sprachen (C++, Java, SmallTalk,...)

Die Umsetzung der zuvor gewünschten Eigenschaften wird zunächst anhand von CORBA erläutert.

6.2 CORBA

Die Object Management Group (OMG) hat seit 1989 eine Spezifikation mit Namen Common Object Request Broker Architecture (CORBA) entwickelt. Die OMG ist ein Firmenkonsortium (Sun, 3COM, IBM etc.). Microsoft ist passives Mitglied, verfolgte aber mit DCOM lange Zeit einen Konkurrenzvorschlag, der in die Entwicklung von .NET einmündete.

Neben OMG CORBA (Common Object Request Broker Architecture) wurden eine Zeit lang SUN DOE (Distributed Objects Everywhere) und IBM Opencod (Open Distributed Object Computing) parallel entwickelt. Inzwischen hat sich aber CORBA durchgesetzt und wird durch die gute CORBA-Unterstützung in Java so schnell nicht aussterben.

6.2.1 Grundlagen von CORBA

Der Kern von CORBA ist eine verteilte Architektur, bei der Objekte über einen Object Request Broker (ORB) miteinander kommunizieren:

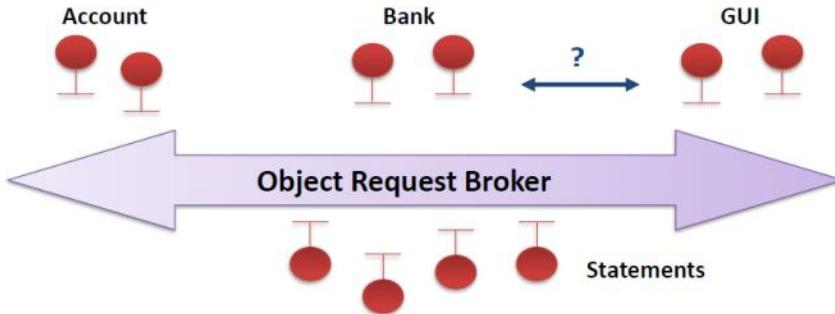


Abbildung 22: Verteilte Objekte kommunizieren über den Object Request Broker(ORB)

Alle miteinander kommunizierenden Objekte und Services wickeln ihre Kommunikation mit dem Object Request Broker (ORB) über ein einheitliches Interface ab. Der ORB ist der zentrale Teil der Architektur. Er ist in jedem Client und in jedem Server notwendig und verbindet die Objekte über das Netzwerk. Auf der Clientseite bietet der ORB eine Schnittstelle zum Aufruf von Operationen. Auf der Serverseite stellt der ORB eine Schnittstelle zum Aufruf von Operationen im Server zur Verfügung. Die Aufgabe des ORB sind die Lokalisierung eines geeigneten Servers und die Ausführung einer Operation auf diesem Server. Im Bild sind Account, Bank und GUI verteilte Applikationsobjekte, also die vom Entwickler geschriebenen Client- und Server-Programme, die verteilte Dienste und Objekte anbieten und nutzen. Außerdem sollte nicht nur der direkte Zugriff auf verteilte Objekte, sondern ihr gesamter Lebenszyklus, also das Management abgebildet werden. Dabei sind die Hauptaufgaben:

- (a) die Überwachung von Schnittstellen
- (b) die Bereitstellung eines Kommunikations-Protokolls

Für das weitere Verständnis sind folgende Begriffe des verteilten Objekt-Managements wichtig:

- Interfaces:
Objekte können nur über Methoden in der Schnittstelle manipuliert werden
- Portable Datentypen:
Standard- und benutzerdefinierte Datentypen müssen unterstützt werden
- Marshaling/Unmarshaling:
Datentypen müssen über Systemgrenzen (Plattform, Prozess) hinweg transferierbar sein;
Datenintegrität muss erhalten bleiben
- Proxy-Objekte:
Stellvertreter zur Abstraktion vom Aufenthaltsort eines Objektes
- Statische & Dynamische Methodenaufrufe:
Aufrufe mit bzw. ohne genaue Kenntnis der Methodensignaturen.
- Objekterzeugung:
Verwaltung des Lebenszyklus eines Objektes zum Beispiel über Factories.

Von der OMG werden folgende CORBA-Bestandteile normiert:

- IDL = Interface Definition Language
- GIOP = General-Inter-ORB-Protocol
- IIOP = Internet-Inter-ORB-Protocol (GIOP für IP)

In folgendem Bild sieht man das vom ORB genutzte Protokoll IIOP und Aspekte wie statische/dynamische Methodenaufrufe und das Marshaling von übergebenen Parametern

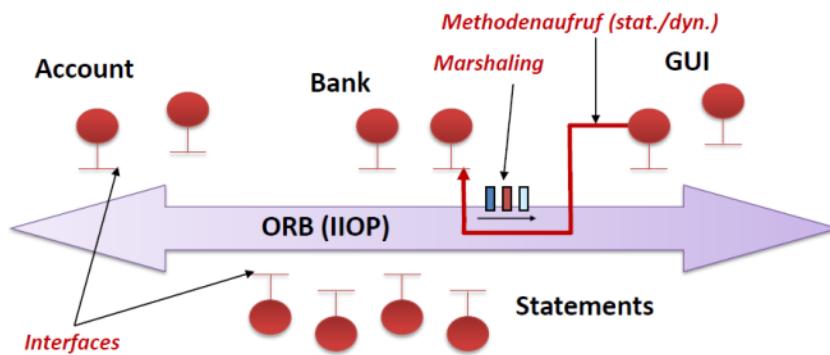


Abbildung 23: ORB- Kommunikation und statische/dynamische Methodenaufrufe

Ein wichtiges Konzept, dem in OOAD ein eigenes Design Pattern gewidmet ist, ist der Proxy. Auch in CORBA übernehmen Proxys die wichtige Rolle der Stellvertreter externer Objekte. Proxys bilden lokal die Eigenschaften eines entfernt genutzten Objekts ab. Sie übernehmen also in der objektorientierten Welt die Rolle, die bei RPCs die Stubs übernommen haben.

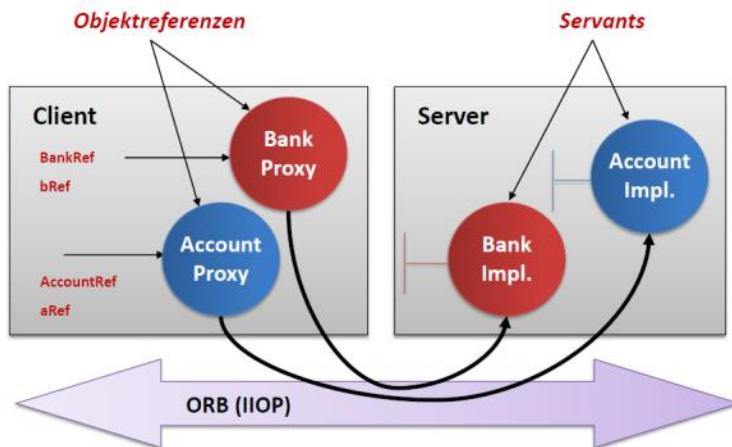


Abbildung 24: Proxys und Servants in CORBA (Beispiel)

Im Client ermöglichen ein Bank und ein Account_Proxy über Objektreferenzen den Zugriff auf die entfernten Objekte beim Server. Dort gibt es die tatsächlichen Implementierungen der Objekte, auf die die Client-Proxys durch sogenannte Servants zugreifen können.

CORBA regelt aber nicht nur den Zugriff, sondern auch das Aktivieren und Neuerzeugen von Objekten, denn man möchte bei komplexen Systemen nicht alle irgendwann benötigten Objekte ständig im Server vorhalten. Wird also auf Client-Seite ein neues Konto erzeugt, so geschieht dies ganz normal über ein new-Statement beim Proxy, das die Erzeugung des neuen Objektes beim Server auslöst.

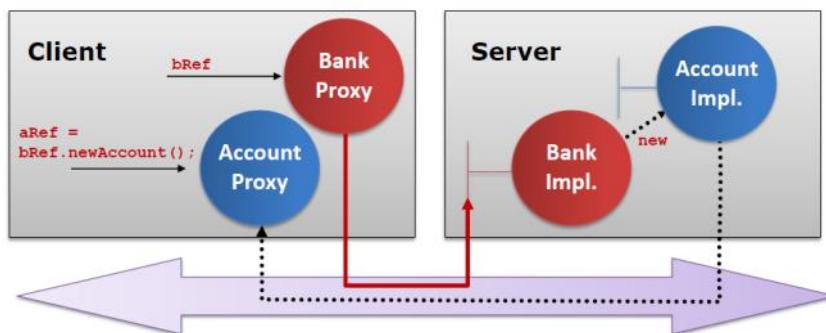


Abbildung 25: Erzeugung neuer Objekte beim Server durch new-Aufruf bei Client

Auch wenn es technisch möglich wäre, Objekte zwischen Client und Server zu kopieren, wird dies so weit wie möglich vermieden und ist aufgrund des Proxy/Servant-Konzepts unnötig.

Wenn Objekte beim Server gestartet werden, registrieren sie sich beim Server wie in folgendem Sequenzdiagramm in den ersten beiden Methodenaufrufen zu sehen ist:

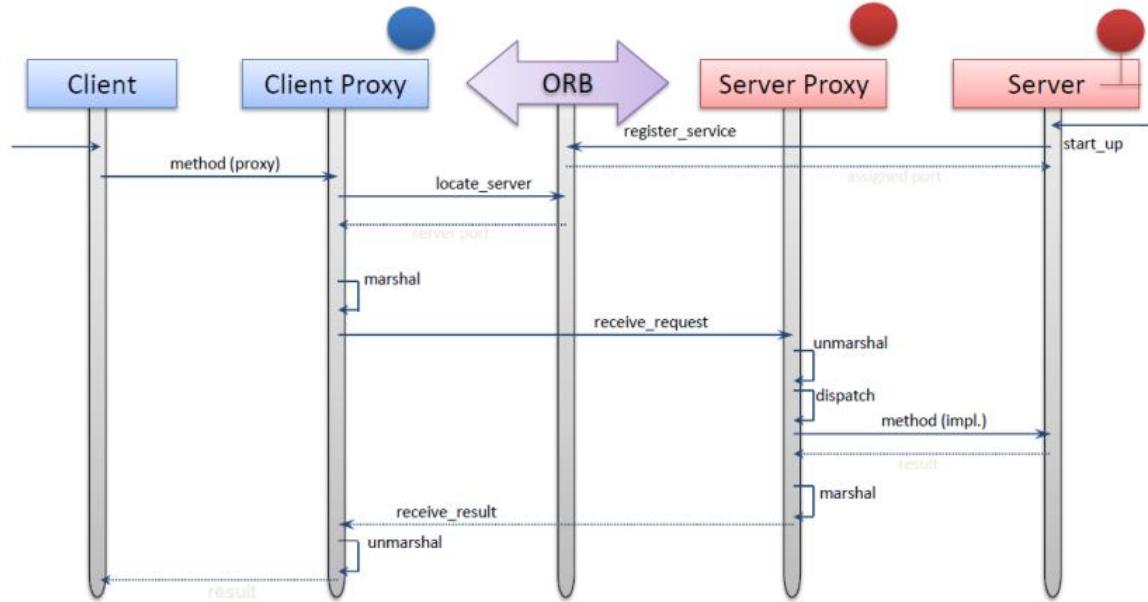


Abbildung 26: Ablauf bei Server-Startup, Registrierung und Nutzung von Serverobjekten

Auf Client-Seite übernimmt der Proxy bei einem Methodenaufruf die Lokalisierung des registrierten Objektes beim Server und transformiert die Client-Daten von der lokalen in die CORBA-Darstellung (Marshaling). Nach dem Transport durch das Netzwerk wandelt der Server die Daten in seine lokale Darstellung um (Unmarshaling) und führt den lokalen Methodenaufruf bei der implementierten Instanz des Objektes durch.

6.2.2 Die Interface Definition Language (IDL) von CORBA

Das Kernkonzept der Interface Definition Language (IDL) von CORBA ist in folgendem Bild veranschaulicht:

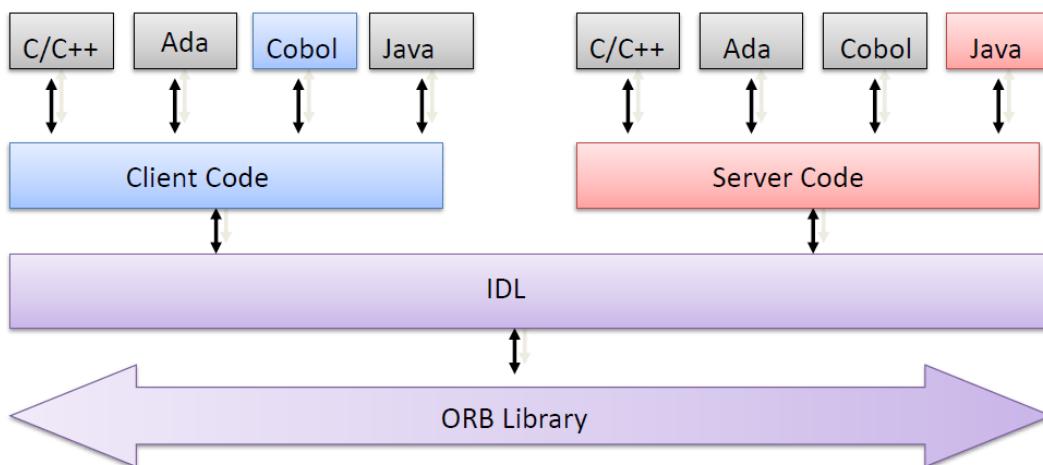


Abbildung 27: Interface Definition Language (IDL) in verschiedenen Programmiersprachen

Da CORBA sprachunabhängig ist, muss die Spezifikation des Objekt Interfaces von der Implementation getrennt werden. Die Implementation wird in einer entsprechenden Programmiersprache vorgenommen. Das Interface wird mit der IDL spezifiziert. Durch ein Language Mapping ist die Umsetzung in verschiedene Programmiersprachen (C++, Java), die auch gemischt genutzt werden können, festgelegt.

Die folgende Abbildung veranschaulicht den selbst zu schreibenden Teil (`Account.idl`, Server- und Client-Code) und die generierten gemeinsamen Klassen in einem C++-Kontobeispiel:

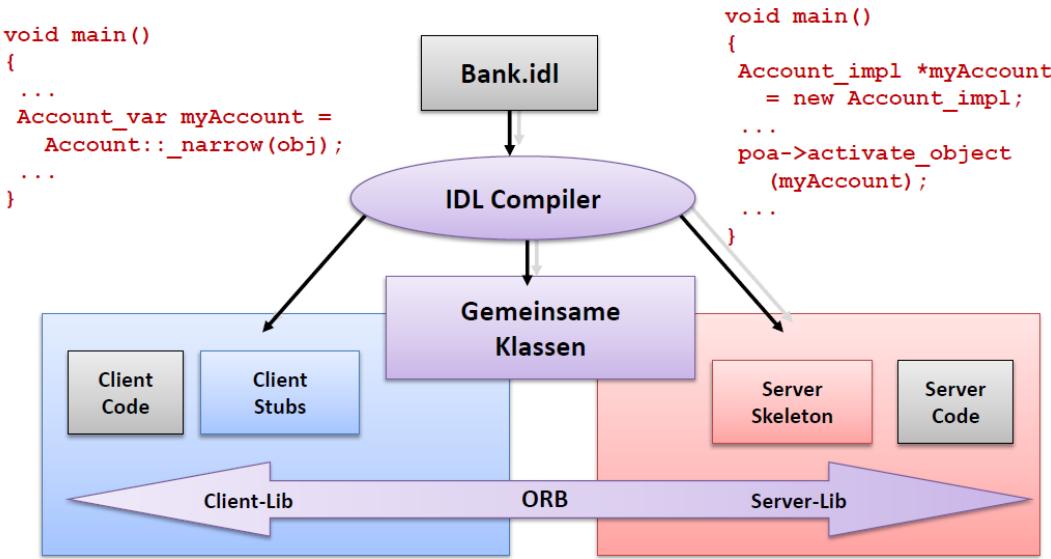


Abbildung 28: Interface Definition Language (IDL) in verschiedenen Programmiersprachen

Ausgehend von der idl-Datei erzeugt der IDL-Compiler (vergleichbar dem rpcgen bei der RPC-Programmierung) gemeinsame Klassen. Diese enthalten u.a. den Client-Stub und das Server-Skeleton (vergleichbar dem Server-Stub bei RPC). Der Client-Stub ist Vermittler zwischen Client-Programm und ORB. Das Skeleton ist verantwortlich für die Steuerung der Methodenaufrufe der eigentlichen Implementation im Server.

In Client- und Server-Code gibt es wieder einen Vorspann, der den Zugriff auf die verteilten Objekte regelt. Im Server wird dynamisch eine Instanz der `Account_impl`-Klasse erzeugt. `_impl` weist immer auf die konkrete Implementierung, also die Server-Seite hin. Um das erzeugte Objekt von außen erreichbar zu machen, muss man den POA (Portable Object Adapter) anweisen, es zu aktivieren. Der vollständige Server-/Client-Code wird bei den Beispielprogrammen näher erläutert.

Am besten versteht man den Aufbau einer IDL-Beschreibung anhand eines kleinen Beispiels:

```

module Finance {
    interface Bank {
        exception Reject { string reason; };
        exception TooMany {}; // Too many accounts.

        Account newAccount(in string name) raises (Reject, TooMany);
        void deleteAccount(in Account a);
    };

    interface Account {
        // Attributes to hold the balance and the name
        readonly attribute string owner;
        attribute long balance;
        // The operations defined on the interface.
        void deposit (in long amount, out long newBalance);
        void withdraw (in long amount, out long newBalance);
        long balance();
    };
}

```

Abbildung 29: Aufbau einer IDL mit Interfaces in einem Module

CORBA hat ein module-Konzept, dass in C++ den Namespaces entspricht. Es erleichtert die Arbeit, wenn man mit Software aus unterschiedlichen Quellen arbeitet und die Namen konsistent halten möchte. In der zweiten Zeile erkennt man das Bank-Interface mit Methoden zur Verwaltung von Konten. Das `Account`-Interface lässt typische Kontoaktionen wie `deposit()` (Einzahlen) und `withdraw()`

(Abhebung) erkennen. Man beachte, dass Rückabeparameter explizit im Aufruf (z.B. `out long newBalance`) übertragen werden können. Ein `in` oder `out` vor dem jeweiligen Parameter kennzeichnet die Übertragungsrichtung (Übergabe oder Rückgabe). Die übliche C/C++-Rückgabe ist ebenfalls möglich, wie die `balance()`-Methode zeigt, die den Kontostand zurückgibt.

Im interface-Abschnitt der IDL ist die Vereinbarung von Konstanten, Typen und Ausnahmen möglich, wie folgendes Bild zeigt:

```
module Finance {
    ....
    const boolean true = TRUE;
    typedef string FiString;
    exception FiException {};
    ....
    interface Bank {
        ....
    };
    ....
};
```

Abbildung 30: Konstanten, Type-Definitionen, Ausnahmen und Interfaces in einer IDL

Bei der Vereinbarung von Konstanten, Typen und Übergabe-/Rückgabeparametern sind folgende Basis-Datentypen möglich:

- `char, octet`
- `short, long, unsigned ... , float, double`
(vgl. C/C++; kein Integer-Typ wegen dessen Plattformabhängigkeit)
- `string`

Es gibt außerdem folgende benutzerdefinierte bzw. zusammengesetzte Datentypen:

- `enum, struct, array, union,`
- `sequence (dyn. Feld),`
- `any (beliebiger IDL-Typ)`

Das Mapping von IDL-Datentypen auf C++-Datentypen sowie die Wertebereiche und den Speicherplatzbedarf zeigt folgende Tabelle:

IDL Type	C++ Type	Range	Size
<code>short</code>	<code>CORBA::Short</code>	$-2^{15} \dots 2^{15} - 1$	≥ 16 bits
<code>long</code>	<code>CORBA::Long</code>	$-2^{31} \dots 2^{31} - 1$	≥ 32 bits
<code>unsigned long</code>	<code>CORBA::ULong</code>	$0 \dots 2^{32} - 1$	≥ 32 bits
<code>unsigned short</code>	<code>CORBA::UShort</code>	$0 \dots 2^{16} - 1$	≥ 16 bits
<code>float</code>	<code>CORBA::Float</code>	IEEE Single Prec.	≥ 32 bits
<code>double</code>	<code>CORBA::Double</code>	IEEE Double Prec.	≥ 64 bits
<code>char</code>	<code>CORBA::Char</code>	ISO Latin 1	≥ 8 bits
<code>octet</code>	<code>CORBA::Octet</code>	0-255	≥ 8 bits
<code>boolean</code>	<code>CORBA::Boolean</code>	TRUE, FALSE	Unspecified
<code>any</code>	<code>CORBA::Any</code>	Run-time type	Variable

Abbildung 31: IDL-/C++-Datentypen, Wertebereiche und Speicherplatzbedarf

Auch Vererbung kann in der IDL angegeben werden. Der folgende Ausschnitt einer IDL zeigt beispielhaft ein CheckingAccount, das von der Basisklasse Account abgeleitet ist.

```
....  
interface Bank {  
    exception Reject {  
        string reason;  
    };  
    exception TooMany {};  
    Account newAccount(in string name)  
        raises (Reject, TooMany);  
};  
  
interface Account; // forward  
declaration  
  
interface CheckingAccount : Account  
{  
    const boolean maxAccountNo = 25;  
    typedef string FiString;  
    readonly attribute overdraftLimit;  
    boolean orderChequeBook();  
};  
....
```

Abbildung 32: Interface mit Vererbung

Auch Mehrfachvererbung ist möglich:

```
....  
  
interface CheckingAccount : Account {  
    readonly attribute overdraftLimit;  
    boolean orderChequeBook();  
};  
  
interface SavingsAccount : Account {  
    float calculateInterest();  
};  
  
interface PremiumAccount :  
    CheckingAccount, SavingsAccount {  
};  
  
....
```

Abbildung 33: Mehrfachvererbung von Interfaces analog zu C++

CORBA-Objekte können auch als Parameter übergeben werden. Die empfangende Instanz (Client oder Server) muss dabei eine Kopie erzeugen. Clients verwalten dabei die in-Objekte, Server die out-Objekte. Wenn als Rückgabewert ein void angegeben wird, arbeitet der Client asynchron weiter.

Das folgende Bild zeigt den Aufwand bei verschiedenen Übergabeansätzen:

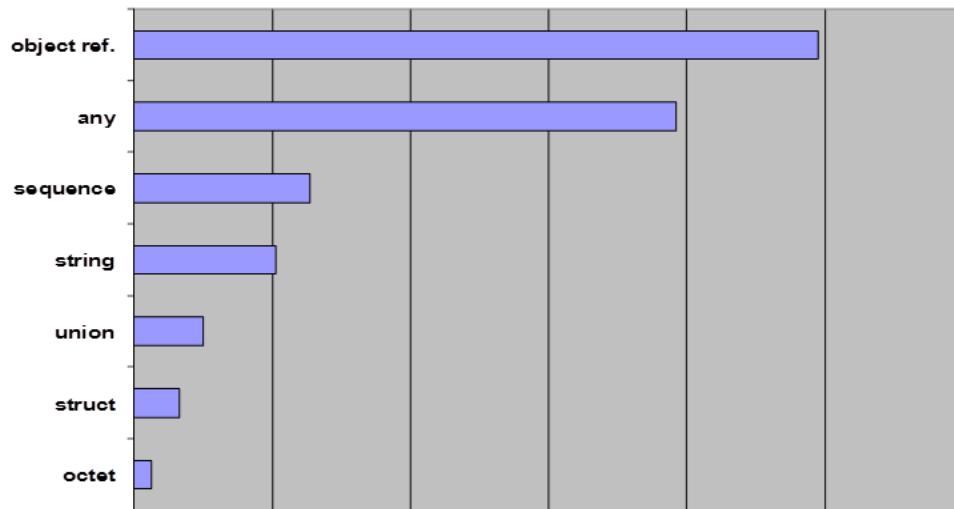


Abbildung 34: Aufwand des Marshalings verschiedener Typen [Slama/Garbis/Russell, 1999]

Man sieht, dass der Aufwand für die Übertragung von Objektreferenzen und bei Verwendung von any sehr hoch ist und daher vermieden werden sollte [Slama/Garbis/Russell, 1999].

6.3 Objektorientierte Programmierung verteilter Systeme mit CORBA in Beispielen

Die folgenden Beispiele werden mit Omnidorb 2.3.13, einer freien CORBA-Implementierung, die an vielen Hochschulen im Einsatz ist, realisiert. Installationstipps finden sich in Unterkapitel 6.4.

6.3.1 Beispiel account zur lokalen Kontoverwaltung

Im lokalen Beispiel account beschreibt eine Account_Klasse ein simples Bankkonto:

```
// account.cc Einfaches Kontobeispiel ohne Netzwerk (alles in einer Datei)
#include <iostream>
using namespace std;

class Account {
    long _current_balance;
public:
    Account();
    void deposit( unsigned long amount );
    void withdraw( unsigned long amount );
    long balance();
};

// method definitions, will later form server part
Account::Account() { _current_balance = 0; }
void Account::deposit( unsigned long amount ) { _current_balance += amount; }
void Account::withdraw( unsigned long amount ) { _current_balance -= amount; }
long Account::balance() { return _current_balance; }

// Hauptprogramm, wird spaeter zum Client,
int main( int argc, char *argv[] ) {
    cout << "Beispiel Account - Kontoverwaltung lokal " << endl;
    Account acc;
```

```

    acc.deposit( 700 );
    acc.withdraw( 250 );
    cout << "Kontostand nach Einzahlung und Abhebung ist ";
    cout << acc.balance() << "." << endl;
    return 0;
}

```

Denken Sie daran, in diesem Kapitel mit g++ als Compiler zu arbeiten. Nach

```
westerka@si0024-1:~/> g++ -o account account.cc
```

und Ausführen des Programms wird nach einer Einzahlung (`deposit`) von 700 und einer Abhebung von 250 (`withdraw`) der Kontostand (`balance`) angezeigt:

```
westerka@si0024-1:~/> ./account
Kontostand nach Einzahlung und Abhebung ist 450.
```

Die Methodendefinitionen werden Teil der Server-Implementierung. so wie dies bei RPC mit den Prozeduren geschehen ist. Das Hauptprogramm mit den Methodenaufrufen stellt den späteren Client-Teil dar.

Um das Entwickeln einer CORBA-Lösung schrittweise zu verdeutlichen, werden drei unterschiedliche Versionen mit jeweils zusätzlicher CORBA-Funktionalität entwickelt:

<code>account_loc</code>	Lokale Version innerhalb eines gemeinsamen Prozesses mit Steuerung des Aufrufs über die Omnidb-ORB-Funktionalität
<code>account_fileref</code>	In zwei Programme getrennte Version, die die Objektreferenz über eine Datei austauscht. Diese ist z. B. in einem Intranet, in dem beide Rechner Zugriff auf eine Netzlaufwerk haben, nutzbar.
<code>account_nameserv</code>	Vollständig verteilte Version mit Übertragung dynamischer Listen strukturierter Daten, die auf unterschiedlichen Rechnern läuft und über einen sogenannten Naming-Service (dazu später mehr) erreichbar ist

6.3.2 Beispiel `account_loc` mit lokaler CORBA-Nutzung

Zunächst befindet sich das Programm weiter in einer gemeinsamen Datei. Die Methodenaufrufe werden nun über CORBA abgewickelt. Der bisherige Programmcode erhält durch eine Header-Datei Zugriff auf die generierte Programmteile, die aus der IDL-Spezifikation `account_loc.idl` erzeugt werden:

```
// account.idl
// account.idl Interface-Definition fuer alle drei Beispiele
interface account {
    void deposit( in unsigned long amount );
    void withdraw( in unsigned long amount );
    long balance();
};
```

Das Schlüsselwort `in` deklariert dabei `amount` als Input-Parameter. Die Interface-Deklaration wird durch den IDL-Compiler `omniidl` des Omnidb-Pakets in C++ Code umgewandelt:

```
westerka@si0024-1:~/> omniidl -bcxx account.idl
```

Das folgende Bild zeigt die vom Entwickler zu schreibenden Dateien (links) und die vom IDL-Compiler erzeugten Dateien sowie als ausführbares Endergebnis, den Client und den Server (ganz rechts).

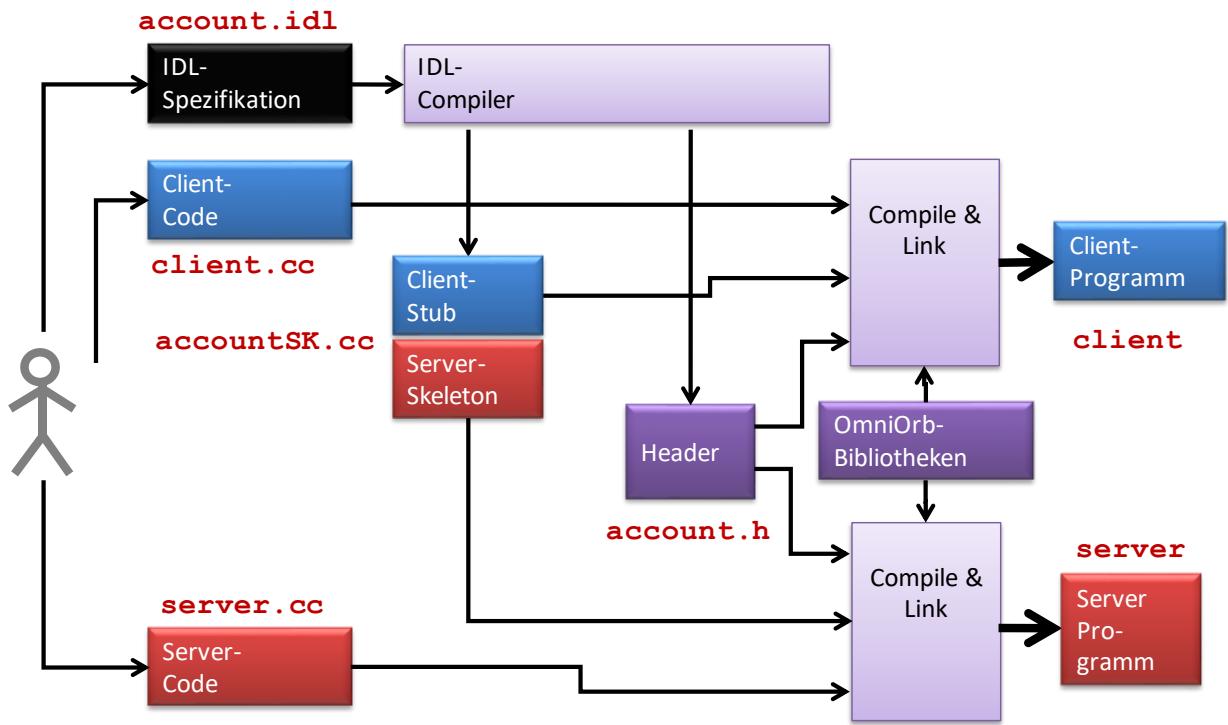


Abbildung 35: Vom Entwickler zu schreibende (links) und daraus erzeugte Dateien (Mitte)

Aus der Schnittstellendefinitionsdatei `account.idl` werden die für Server und Client gemeinsamen Dateien `account.hh` und `accountSK.cc` erzeugt. Die Header-Datei enthält Klassendeklarationen für die Basisklasse `POA_Account` mit dem Server-Skeleton. Später wird davon die Implementierung der eigentlichen Klasse abgeleitet. Weiterhin gibt es im Header die Stubklasse, die vom Client verwendet wird, um Methoden auf entfernten Konten-Objekten auszuführen. `AccountSK.cc` enthält die Implementierung dieser Klassen und zusätzlichen Code.

Für jedes Interface im IDL-File werden drei C++-Klassen erzeugt. In diesem Beispiel wird eine Konto-Klasse als Basisklasse generiert. `Account` enthält alle Definitionen, die zum Interface `Account` gehören. Diese Klasse definiert für jede Operation im Interface eine virtuelle Funktion. Die Klasse `Account_stub` ist abgeleitet von `Account`. Sie enthält einen Dispatcher für die Operationen, die in `Account` definiert wurden. Beide Klassen sind im C++-Sinne abstrakte Basisklassen. Um das Interface zu implementieren, muss eine Subklasse von `Account_stub` gebildet werden, die Funktionen für die virtuellen Methoden `deposit()`, `withdraw()` und `balance()` implementieren. Allerdings wird die Klasse nie direkt verwendet, der Zugang geschieht immer über die Klasse `Account`.

Bisher wurde nur das Interface des Kontenobjektes entwickelt.

Als nächstes wird ein Gesamtprogramm `account_loc.cc` mit Methoden und einem kleinen Testabschnitt definiert:

```
// account_loc: Lokale Account_Implementierung unter Nutzung
// von CORBA-Referenzen
#include "account.h"
#include <iostream>

using namespace std;
// Implementierung der Accountklasse
class Account_impl : virtual public POA_Account
{
    CORBA::Long _current_balance;
public:
    Account_impl ();
    void deposit(CORBA::Long amount);
    void withdraw(CORBA::Long amount);
    CORBA::Long balance();
};
```

```

void deposit (CORBA::ULong);
void withdraw (CORBA::ULong);
CORBA::Long balance();

};

Account_Impl::Account_Impl() : _current_balance(0){};

void Account_Impl::deposit(CORBA::ULong amount) {_current_balance += amount; }

void Account_Impl::withdraw(CORBA::ULong amount) {_current_balance -= amount; }

CORBA::Long Account_Impl::balance() { return _current_balance; }

int main (int argc, char *argv[])
{
    cout<<"Beispiel account_loc - Kontoverw. lokal mit omniORB-
Referenzen"<<endl;
    Account_Impl *my_Account = new Account_Impl;
    // deposit und withdraw
    my_Account->deposit (700);
    my_Account->withdraw (250);
    cout << "Kontostand nach Einzahlung und Abhebung ist ";
    cout << my_Account->balance() << "." << endl;
    delete my_Account;
    return 0;
}

```

Das Hauptprogramm besteht aus dem Server-Teil, der die `Account`-Implementierung enthält. Die Testanweisungen werden später zum Client-Code in einem separaten Programm. Die Objektverwaltung geschieht unter Nutzung der `Account_Impl`-Klasse. Die Referenz hierauf wird im Speicher ausgetauscht. Für diesen Referenzaustausch werden wir später bei Aufteilung auf Client und Server zwei separate Mechanismen kennen lernen.

Die selbst geschriebenen und durch `omniidl` erzeugten Dateien werden nun zu einem lauffähigen Programm `account_loc` kompiliert:

```

westerka@si0024-1:~/> g++ -c account_loc.cc

westerka@si0024-1:~/> g++ -c accountSK.cc

westerka@si0024-1:~/> g++ -o account_loc account_loc.o accountSK.o
-lomnithread -lomniORB4

```

Zu diesem und allen folgenden Beispielprogrammen gibt es entsprechende Makefiles, die durch `make all` die Testprogramme erzeugen. Beim Ausführen von `account_loc` wird der Kontostand nach Ausführen der Testbuchungen ausgegeben:

```

westerka@si0024-1:~/> ./account_loc
Kontostand nach Einzahlung und Abhebung ist 450.

```

Noch kommt man wegen der Verwendung des Zeigers im Programm ohne separate Referenz auf das entfernte Objekt aus.

6.3.3 Beispiel account_fileref mit Client/Server und Referenzaustausch über IOR-Datei

Im vorherigen Beispiel waren Client und Server-Abschnitt durch Objektreferenzen verbunden. Sollen sie getrennt werden, muss ein Naming Service zur Verfügung stehen, mit dem der Client den richtigen Server und das richtige Objekt findet.

Generell verwendet CORBA für eindeutige Referenzen lange Hex-Zahlen mit der Bezeichnung Interoperable Object Referenz (IOR). Sie enthält alle Informationen, die ein Client benötigt, um den Server zu kontaktieren.

Das folgende Bild zeigt den prinzipiellen Aufbau einer IOR:

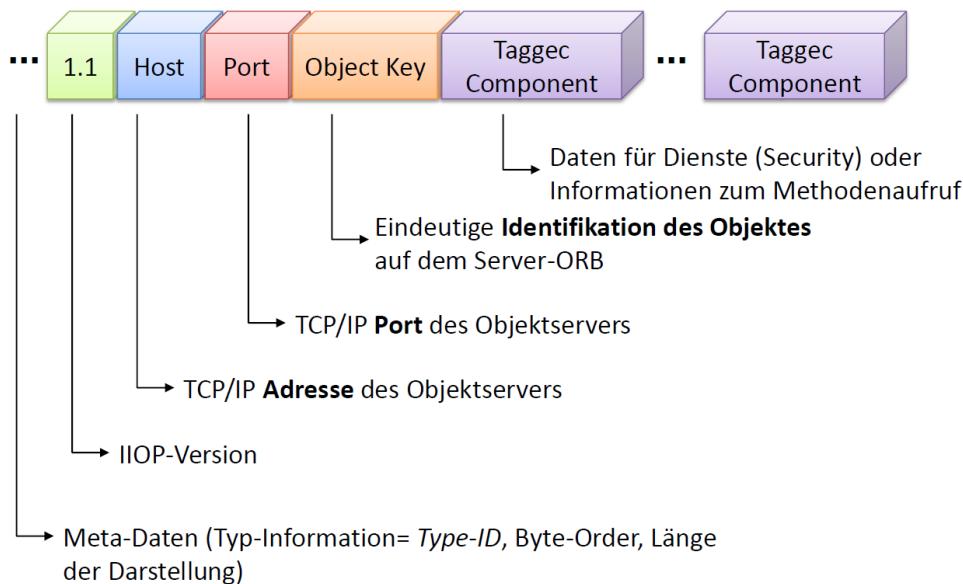


Abbildung 36: Aufbau einer IIOP IOR

Eine IIOP IOR sieht z. B. so aus:

```
010000001000000049444c3a4163636f756e743a312e3000
02000000000000003400000001010000190000006f64616c2e65742e66682d6f736e61
62727565636b2e6465000082130c000000424f4183ad7ba40000734e03010000002400
000001000000010000000100000014000000010000000100010000000000901010000
000000
```

Mit dem Hilfsprogramm iordump erhält man eine lesbare Darstellung der IOR:

```
westerka@si0024-1:~/> iordump
IOR:010000001000000049444c3a4163636f756e743a312e3000200000000000000048
00000001010000260000006c6273742d6e7063612d312e6c6273742e6563732e66682d
6f736e6162727565636b2e6465005ba9140000002f33303231312f313235363339334
35342f5f30010000002400000001000000010000000100000001400000001000000010000000100
0100000000000901010000000000
    Repo Id: IDL:Account:1.0
    IIOP Profile
        Version: 1.0
        Address: inet:westerka@si0024-1.lbst.ecs.fh-osnabrueck.de:43355
        Location: corbaloc::westerka@si0024-1.lbst.ecs.fh-
osnabrueck.de:43355//30211/1256393454/%5f0
        Key: 2f 33 30 32 31 31 2f 31 32 35 36 33 39 33 34 35
/30211/125639345
            34 2f 5f 30
Multiple Components Profile
Components: Native Codesets:
normal: ISO 8859-1:1987; Latin Alphabet No. 1
4/_0
```

```
wide: ISO/IEC 10646-1:1993; UTF-16, UCS Transformation
Format 16-bit form
Key: (empty)
```

Beim Registrieren eines Objektes beim Server wird eine solche eindeutige IOR erzeugt, die der Client erhalten muss. Eine einfache (nur im Intranet realisierbare) Möglichkeit besteht in der Verwendung des verteilten Dateisystems (z. B. eines gemeinsamen Netzverzeichnisses bzw. -laufwerkes).

Dies wird in `server.cc` gezeigt, wobei sich nur die `main()`-Funktion ändert:

```
// server.cc      - Beispiel account_fileser
// Client mit Referenzaustausch ueber IOR-Datei

// Hauptprogramm startet POA und macht Account_Objekt erreichbar
int main (int argc, char *argv[]) {
    cout << "server laeuft." << endl << endl;
    // ORB initialisieren
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
    // Referenz zum RootPOA und seinem Manager holen
    CORBA::Object_var poaobj = orb->resolve_initial_references
("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
    PortableServer::POAManager_var mgr = poa->the_POAManager();
    // Konto-Objekt erzeugen
    Account_impl *myAccount = new Account_impl;
    // Objekt innerhalb POA aktivieren und oid zur Objektverwaltung
    holen
    PortableServer::ObjectId_var oid = poa->activate_object (myAccount);

    ofstream of ("Account.ref");
    if (!of) return -1;
    CORBA::Object_var ref = poa->id_to_reference (oid.in());
    CORBA::String_var str = orb->object_to_string (ref.in());
    of << str.in() << endl;
    of.close ();

    mgr->activate ();
    orb->run();

    // Einzahlen/Abheben verlagert in Client
    delete myAccount;
    poa->destroy (TRUE, TRUE); // Server herunter fahren
    return 0;
}
```

Diese Version unterscheidet sich von der vorherigen durch Speicherung der String-Darstellung der Objektreferenz in einer Datei. Die Wirkungsweise des POA, der die Objektreferenz innerhalb eines Rechners verwaltet, soll nun detailliert betrachtet werden.

Mit folgender Zeile wird ein ORB-Objekt erzeugt und dabei der ORB initialisiert:

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
```

Beim Start des ORBs wird ein RootPOA erzeugt. Von ihm können noch weitere POA-Instanzen abgeleitet werden, wenn z. B. mit unterschiedlichen Verhaltensweisen (Policies) des POA benötigt werden (hier nicht verwendet). Man erhält die Referenz zum RootPOA bei gestartetem ORB mit:

```
CORBA::Object_var poaobj=orb->resolve_initial_references ("RootPOA");
```

Das gelieferte `poaobj` ist vom allgemeinen Typ `Object_var`. Zur weiteren Arbeit benötigen wir allerdings ein Objekt vom Typ `POA_var`. Für die entsprechende Typumwandlung wird die `_narrow`-Methode mit dem erzeugten `poaobj` als Argument aufgerufen:

```
PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
```

Das Verhalten des POA wird durch einen Zustandsautomaten in POA-Manager gesteuert der mit

```
PortableServer::POAManager_var mgr = poa->the_POAManager();
```

gestartet wird. Wir erzeugen wieder ein `Account_Objekt` und müssen es nun durch Aufruf der Methode `activate_object` mit der lokalen Objektreferenz beim POA bekannt machen. Diese Methode liefert eine `oid`, die als Referenz beim POA fungiert. Nun wird die IOR zur `oid` erzeugt und an eine Umwandlungsmethode übergeben, die daraus einen String macht. Diesen String kann man jetzt in eine Datei (z.B. auf einem Netzverzeichnis) schreiben, damit Clients ihn nutzen können.

Der Client liest die String-Version der IOR aus `Account.ref` und kann damit das Objekt nutzen:

```
// client.cc - Beispiel account_fileserv
// Client mit Referenzaustausch ueber IOR-Datei
#include "account.h"
#ifndef HAVE_UNISTD_H
#include <unistd.h>
#endif
#include <iostream>
#include <fstream>

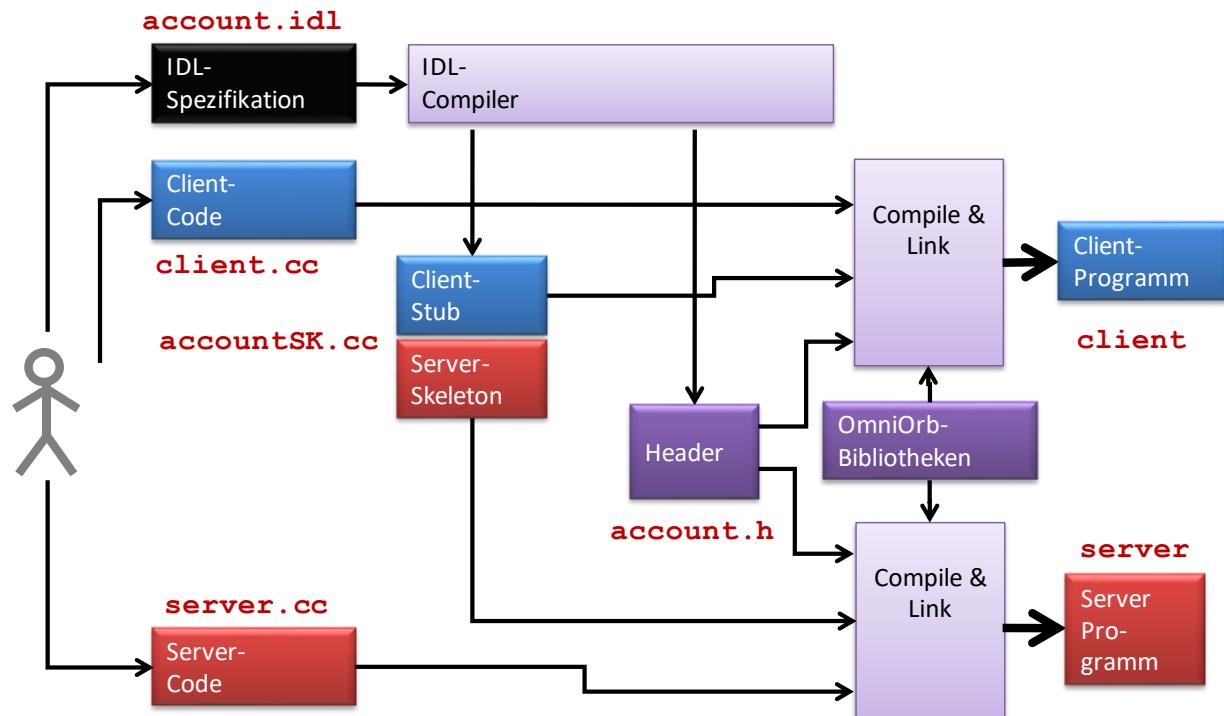
using namespace std;
int main (int argc, char *argv[]) {
    cout << "Beispiel account_fileref - Kontoverwaltung mit IOR in
Datei" << endl;
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
    // IOR befindet sich in Konto.ref im lokalen Verzeichnis
    char pwd[256], uri[300];
    // getcwd liefert momentanes Arbeitsverzeichnis (current working
directory)
    sprintf (uri, "file:///%s/Account.ref", getcwd(pwd, 256));
    // Binden an Konto-Server
    CORBA::Object_var obj = orb->string_to_object(uri);
    Account_var myAccount= Account::_narrow(obj);
    if (CORBA::is_nil(myAccount)) {
        cout << "Fehler: Konnte Konto-Server nicht finden!" << endl;
        return -1;
    }

    // deposit und withdraw
    myAccount->deposit (700);
    myAccount->withdraw (250);
    cout << "Kontostand nach Einzahlung und Abhebung ist ";
    cout << myAccount->balance() << "." << endl;
    return 0;
}
```

Wir werden jetzt auf zwei verschiedenen Rechnern arbeiten und gehen von folgender Zuordnung aus:

si0024-16a Server

si0024-15a Client



Die Quellen werden mit folgendem makefile übersetzt und gelinkt:

```

all: server client
client: accountSK.o client.o
        g++ -o client accountSK.o client.o -lomnithread -lomniORB4

server: accountSK.o server.o
        g++ -o server accountSK.o server.o -lomnithread -lomniORB4

server.o: server.cc account.hh
        g++ -c server.cc

client.o: client.cc account.hh
        g++ -c client.cc

accountSK.o: accountSK.cc account.hh
        g++ -c accountSK.cc

clean:
        rm client.o server.o accountSK.o client server
    
```

Zunächst wird der Server gestartet:

```
westerka@si0024-1:~/> ./server
```

Bei jedem Aufruf des Clients wird ein neuer Wert des Kontos ausgegeben, außer der Server wird neu gestartet:

```
westerka@si0024-1:~/> ./client
Beispiel account_fileref - Kontoverwaltung mit IOR in Datei
client laeuft.
```

Lesen der Server-IOR aus
 '/home/westerka/vts/2014/kap6/account_fileref/Account.ref'
 Kontostand nach Einzahlung und Abhebung ist 450.

```
westerka@si0024-1:~/> ./client
```

Beispiel account_fileref - Kontoverwaltung mit IOR in Datei client laeuft.

```
Lesen der Server-IOR aus  
'/home/westerka/vts/2014/kap6/account_fileref/Account.ref'  
Kontostand nach Einzahlung und Abhebung ist 900.  
westerka@si0024-1:~/> ./client  
Beispiel account_fileref - Kontoverwaltung mit IOR in Datei  
client laeuft.
```

```
Lesen der Server-IOR aus  
'/home/westerka/vts/2014/kap6/account_fileref/Account.ref'  
Kontostand nach Einzahlung und Abhebung ist 1350.
```

6.3.4 Beispiel account_nameserv mit Naming Service und variablen Listen strukturierter Daten

Die allgemeinste Möglichkeit, CORBA-Objekte miteinander arbeiten zu lassen, ist die Verwendung eines Naming Service. Damit entfällt die Notwendigkeit eines gemeinsamen Netzverzeichnisses. Der Naming Service ist in unserem Fall der **omniNames**. Er wird typischerweise auf irgendeinem (dritten) Rechner und nicht unbedingt auf dem Server betrieben. CORBA bietet durch diese Entkopplung die Möglichkeit, Dienste durch mehrere Server anzubieten und erst bei einer konkreten Anfrage zu entscheiden, welcher Server angesprochen werden soll (Redundanz, Load-Balancing, hier nicht behandelt).

Um den Naming Service zu nutzen, wird bei IP-Adresse 127.0.0.1 und Port 4334 folgendermaßen verfahren:

1. Den Naming Service in einem eigenen Terminalfenster starten mit
`omniNames -start 4334 -always -logdir . -errlog omniNamesErrors.txt`
2. Server und Client erhalten die Adresse des Naming Service durch die Standardoption
`-ORBInitRef NameService=corbaloc::127.0.0.1:4334/NameService`
3. Der Server registriert die von ihm angebotenen Objekte beim Naming Service.
4. Der Client befragt den Naming Service nach dem richtigen Server.

Die Referenz zum Naming Service wird ähnlich wie beim RootPOA durch **resolve_initial_references** ermittelt. Auch die Typumwandlung ist vergleichbar. Der Name des Objektes wird erzeugt und mittels **rebind**-Methode an den Naming Service übergeben. **rebind()** überschreibt eine bereits vorhandene Referenz gleichen Namens, wohingegen der **bind()**-Aufruf in diesem Fall abbrechen würde.

Für den Test sehen wir nun folgende Zuordnung vor:

```
si0024-16a    Naming Service  
si0024-15a    Server  
lbst-npca:-3  Client
```

Es wird übersetzt und gebunden (**makefile** wie im vorherigen Unterkapitel)

Nun wird der Namensdienst gestartet (Shell-Skript **startomnинames.sh**):

```
westerka@si0024-1:~/> omniNames -start 4334 -always -logdir .  
-errlog omniNamesErrors.txt &
```

Der Server **server_nameserv** startet mit (Shell-Skript **startserver.sh**):

```
westerka@si0024-2:~/> ./server_nameserv-ORBInitRef  
NameService=corbaloc::localhost:4334/NameService  
myBank wird an Naming Service gebunden ... done.
```

Nun wird der Client **client.cc** gestartet. Er erfragt das Konto-Objekt beim Naming-Service und greift auf dieses beim Server zu (Shell-Skript **start_client.sh**):

```
lbst-npca:~3> ./client -ORBInitRef  
NameService=corbaloc::localhost:4334/NameService
```

Um das volle Potenzial von CORBA demonstrieren zu können, wird das Konto-Beispiel um dynamisch erzeugte Listen mit strukturierten Daten erweitert.

Zu einem Konto gehören nun:

1. eine Kontonummer (Ganzzahl mit dem Datentyp `long`)
2. ein Kontoinhaber (Zeichenkette mit dem Datentyp `string`)
3. der Kontostand (Ganzzahl mit dem Datentyp `long`)

Dies ist in der neuen `account.idl`-Datei zu sehen:

```
// -*- c++ -*-
exception NullValue{};

struct Account {
    long id;
    string owner;
    long balance;
};

typedef sequence<Account> AccountSeq;

interface Bank {
    void createAccount(in long balance, in string owner);
    Account getAccount(in long id) raises(NullValue);
    AccountSeq getAccountSeq();
};
```

Da wir die Möglichkeiten der dynamischen Objektverwaltung durch einen Servant-Manager nicht nutzen werden, reicht es aus Effizienzgründen, die Werte der Attribute eines Objektes in einer Struktur zu übertragen. In Abbildung 34 ist erkennbar, dass damit eine Zeitsparnis um den Faktor 4 erreicht wird.

Das Beispiel besteht aus folgenden Dateien:

- `account.idl`: enthält die Beschreibung der CORBA Interfaces.
- `server.cc`: enthält die Implementation der Server-Klassen.
- `client.cc`: enthält ein Programm zum testen der einzelnen Funktionalitäten des Servers.
- `makefile`: dient zum Kompilieren der Programme, bzw. zum Löschen erzeugter Dateien.
- `startnsd.sh`: ermittelt die lokale IP Adresse und startet den Naming-Service unter Verwendung dieser IP.
- `startserver.sh`: ermittelt die lokale IP Adresse und startet den Server unter der Annahme, dass der Naming-Service ebenfalls unter dieser IP erreichbar ist.
- `startclient.sh`: ermittelt die lokale IP Adresse und führt den Client unter der Annahme aus, dass der Naming-Service ebenfalls unter dieser IP erreichbar ist.

Das folgende Listing zeigt den Server:

```
/*
* Autor: Dipl.-Inf.(FH) R.Koenig           Datum: 12.11.2014
* (adaptiert aus MICO Beispiel)
*
* Beispiel fuer eine Bank Klasse.
*
* In diesem Beispiel wird ein NamingService,
* und aus Effizienzgründen eine Sequence mit Strukturen verwendet.
* Damit ist eine dynamische Objektverwaltung mittels
* Servant-Manager nicht möglich.
*/
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <assert.h>
#include "account.hh"
```

```

using namespace std;

#define MAX_ACCOUNTS 5

class Bank_impl : virtual public POA_Bank {
public:
    Bank_impl();
    void createAccount(CORBA::Long balance, const char* owner);
    Account* getAccount(CORBA::Long id);
    AccountSeq* getAccountSeq();

private:
    AccountSeq* accounts;
};

Bank_impl::Bank_impl() {
    accounts = new AccountSeq();
}

void Bank_impl::createAccount(CORBA::Long balance, const char* owner){
    cout << "[createAccount]" << endl;
    cout << "[arguments] balance:" << balance << " owner:" << owner <<
endl;
    Account newAccount;
    newAccount.balance = balance;
    int count = accounts->length();
    count++;
    accounts->length(count);
    newAccount.id = count + 1000;
    newAccount.owner = strdup(owner);
    (*accounts)[count - 1] = newAccount;
    cout << "done" << endl << endl;
}

Account* Bank_impl::getAccount(CORBA::Long id) {
    cout << "[getAccount]" << endl;
    cout << "[arguments] id:" << id << endl;
    int count = accounts->length();
    if (id >= count){
        cout << "no valid account found" << endl;
        NullValue e;
        throw e;
    }
    Account* result = new Account();
    result->id = (*accounts)[id].id;
    result->owner = strdup((*accounts)[id].owner);
    result->balance = (*accounts)[id].balance;
    cout << "done" << endl << endl;
    return result;
}

AccountSeq* Bank_impl::getAccountSeq() {
    cout << "[getAccountSeq]" << endl;
    int count = accounts->length();
    AccountSeq* result = new AccountSeq();
    result->length(count);
    for (int i = 0; i<count; i++){
        Account tmp;
        tmp.id = (*accounts)[i].id;
        tmp.owner = strdup((*accounts)[i].owner);
        tmp.balance = (*accounts)[i].balance;
        (*result)[i] = tmp;
    }
    cout << "done" << endl << endl;
}

```

```

        return result;
    }

int main(int argc, char *argv[]){
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var poaobj = orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var poa = PortableServer::POA::_narrow(poaobj);
    PortableServer::POAManager_var mgr = poa->the_POAManager();

    // Bank erzeugen
    Bank_impl * bankcash = new Bank_impl();

    // bankcash aktivieren
    PortableServer::ObjectId_var oid = poa->activate_object(bankcash);

    // Referenz zum Naming Service holen
    CORBA::Object_var nsobj =
        orb->resolve_initial_references("NameService");
    CORBA::Object_var ref = poa->id_to_reference(oid.in());

    CosNaming::NamingContext_var nc =
        CosNaming::NamingContext::_narrow(nsobj);

    if (CORBA::is_nil(nc)) {
        cerr << "Fehler: Naming Service nicht erreichbar!" << endl;
        exit(1);
    }

    // Naming Service fuer unsere Bank einrichten
    CosNaming::Name name;
    name.length(1);
    name[0].id = CORBA::string_dup("myBank");
    name[0].kind = CORBA::string_dup("");

    // Referenz zum Bank-Server im Naming Service speichern.
    // 'rebind' anstelle von 'bind', um vorhandene Bindungen zu
    // ueberschreiben.
    cout << "myBank wird an Naming Service gebunden ... " << flush;
    nc->rebind(name, ref);
    cout << "done." << endl;

    // Activate both POAs and start serving requests
    mgr->activate();
    orb->run();

    /* Shutdown (never reached). This would call etherealize() for
    /* all our accounts. */
    //poa->destroy (TRUE, TRUE); //edit (omniORB)
    poa->destroy(true, true); //edit (omniORB)
    delete bankcash;

    return 0;
}

```

Die Server-Methoden haben folgende Funktion:

- `createAccount(CORBA::Long balance, const char* owner)` erzeugt ein neues Kontobjekt und füllt es mit den als Argumenten übergebenen Daten. Die Kontobjekte werden im Bank-Objekt erzeugt und in einer Liste verwaltet. Der Zähler wird für jedes neue Account Objekt um eins erhöht.
- `getAccount(CORBA::Long id)` gibt die Referenz auf ein einzelnes Kontoobjekt zurück. Das Ergebnisobjekt wird dabei schrittweise mit den Elementwerten des gewünschten Kontos gefüllt (entspricht einer tiefen Kopie bei C++, man beachte `strdup`)

- `getAccountSeq()` liefert eine Liste mit den Inhalten der Kontobjekte zurück. Aus Effizienzgründen wird diese als Sequenz von Strukturen realisiert.

Der Client `client.cc` ändert sich gegenüber dem vorherigen Beispiel nur im Bereich der Testfunktionen, die bei Server aufgerufen werden:

```
// bis hierher wie im vorherigen Beispiel
Bank_var bank = Bank::_narrow(obj);

if (CORBA::is_nil(bank)) {
    cout << "Error: Bank object not found" << endl;
    exit(1);
}

// create accounts
bank->createAccount(100, "Max Mustermann");
bank->createAccount(300, "Willi Schmidt");

cout << "-----" << endl << endl;

// test: getting accounts by id
Account* account = bank->getAccount(0);
char* owner = (*account).owner;
long balance = (*account).balance;
cout << "[" << owner << "] [" << balance << "]" << endl;

account = bank->getAccount(1);
owner = (*account).owner;
balance = (*account).balance;
cout << "[" << owner << "] [" << balance << "]" << endl;

cout << "-----" << endl << endl;

// test: get accounts by sequence
AccountSeq* accounts = bank->getAccountSeq();
for (int i = 0; i < accounts->length(); i++){
    owner = (*accounts)[i].owner;
    balance = (*accounts)[i].balance;
    cout << "[" << owner << "] [" << balance << "]" << endl;
}

cout << "-----" << endl << endl;

// test: get account which does not exist
try{
    account = bank->getAccount(3);
}
catch (NullValue& e){
    cout << "Nullvalue exception" << endl;
}
```

Nacheinander werden

- die Erzeugung von zwei Konten mit `bank->createAccount`,
- der Abruf der Kontodaten beider Konten mit `bank->getAccount`,
- die Liste aller Konten mit `bank->getAccountSeq` und
- der Zugriff auf ein nicht existierendes Konto

getestet.

Für den Test sehen wir folgende Zuordnung vor:

<code>si0024-16a</code>	Naming Service
<code>si0024-15a</code>	Server
<code>lbst-npca:-3</code>	Client

Mit folgenden Kommandos startet man Name-Server, Server und Client auf drei verschiedenen Rechnern:

```
westerka@si0024-1:~/> ./startomnianames.sh
```

```
westerka@si0024-2:~/> ./startserver.sh
```

```
westerka@si0024-3:~/> ./startclient.sh
```

Die Ausgabe des Test-Client sieht folgendermaßen aus:

```
Get reference of Bank object ... done.
```

```
[Max Mustermann] [100]  
[Willi Schmidt] [300]
```

```
[Max Mustermann] [100]  
[Willi Schmidt] [300]
```

Nullvalue exception

Man sieht, dass alle Tests erfolgreich durchlaufen werden. Wir haben nun auch die Übertragung von Listen benutzerdefinierter Objektinhalte demonstriert und können damit komplexe verteilte CORBA-Projekte entwickeln. In der Omnidb-Dokumentation sind weitere Möglichkeiten wie Redundanz, Migration von Objekten, Persistenz etc. beschrieben ist.

6.4 Einige Hinweise beim Einsatz von CORBA

6.4.1 Nutzung von CORBA-Datentypen

Beim Einsatz von CORBA an der Schnittstelle zwischen Server und Client müssen bei der Übergabe von Daten CORBA-Datentypen eingesetzt werden. Für Variablen, die nur lokal im Server oder Client verwendet werden, gilt dies nicht und sie müssen nicht in der idl definiert werden. Für Zeichenketten gelten Besonderheiten:

Der OMG IDL string typ, wird in C++ auf char* umgesetzt und ist zeroterminated. Um dynamisch mit Strings arbeiten zu können, müssen an den Schnittstellen folgende Funktionen verwendet werden:

```
char *string_alloc( ULong len );  
char *string_dup( const char* );  
void string_free( char * );
```

string_alloc allokiert dynamischen Speicher für len+1 Zeichen(also mit Zerotermination). Im Fehlerfrei wird NULL zurückgegeben. string_dup allokiert die Menge Speicher, die für die übergebene Zeichenkette benötigt wird. Im Fehlerfall wird NULL zurückgegeben. char *-Zeichenketten können damit in eine schnittstellenkonforme Darstellung übertragen werden. Es kann auch direkt mit CORBA::string gearbeitet werden. Beispiel:

```
class Hello_impl : virtual public Hello_skel {  
public:  
    char *hello (const char *s) {  
        cout << s << endl;  
        return CORBA::string_dup (s);  
    }  
};
```

6.5 Weitere Bestandteile der CORBA-Architektur

In der Einführung wurde bereits der Object Request Broker erläutert. Weitere Bestandteile der CORBA-Architektur sind in folgender Abbildung sichtbar:

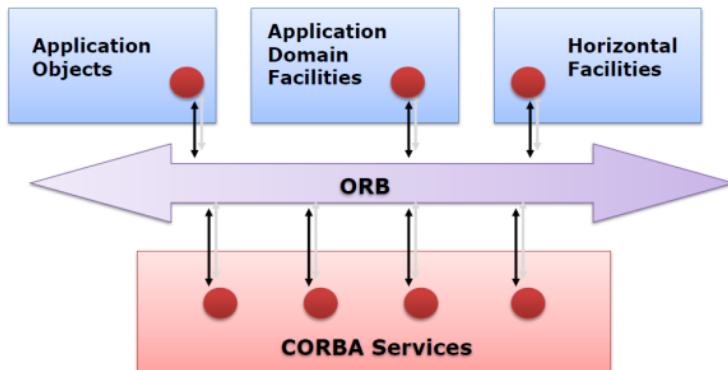


Abbildung 37: Vollständige CORBA-Architektur

CORBA-Services decken alle Dienste, die bei verteilten Anwendungen benötigt werden könnten, ab. Hierzu gehören:

- Instanzenmanagement (Objekte erzeugen, zerstören, kopieren, verschieben)
- Namensdienst (Naming Service) zur eindeutigen Identifikation und Lokalisierung
- Ereignis- und Zeitdienst
- Transaktionsdienst
- Anfragendienst (Obermenge von SQL) etc.

CORBAfacilities fassen bestimmte Dienste zusammen oder erweitern sie. CORBAdomains enthalten anwendungsspezifische Rahmenwerke z. B. für Produktion, Gesundheitswesen, Telekommunikation etc. Die Server-Aktivierung mit dem POA kann manuell oder dynamisch bei Anfrage erfolgen

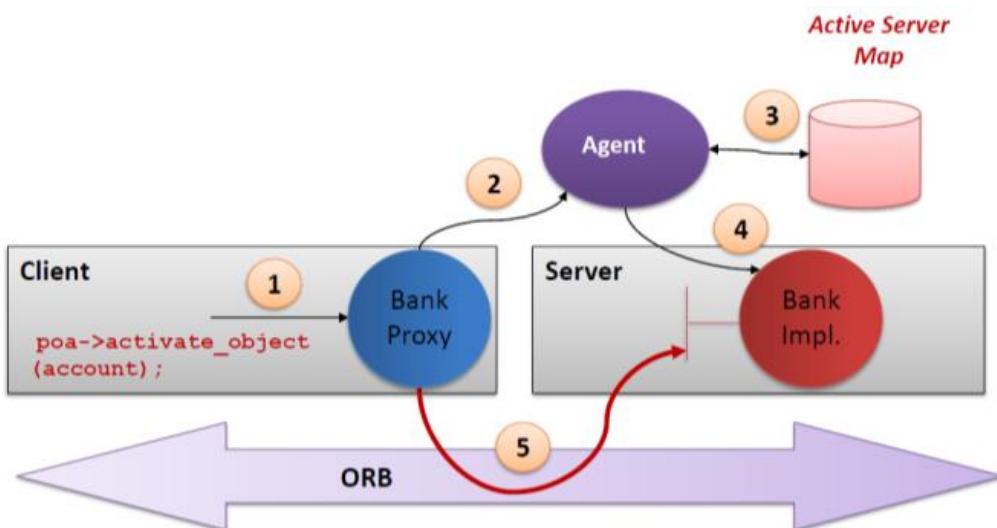


Abbildung 38: Aktivierung von Objekten

Dabei laufen folgende Schritte ab:

1. Der Client aktiviert ein entferntes Objekt beim Proxy.
2. Der Proxy leitet die Aktivierung an den Agenten (POA-Manager) weiter.
3. Dieser ermittelt in der Active Object Map, wo das Objekt zu finden ist.
4. Beim Servant (anbietende Instanz) wird die Account_Impl. (mit _Impl abgekürzt) aktiviert.
5. Der Proxy kann über den ORB direkt auf das entfernte Objekt zugreifen.

Der CORBA Naming Service ermöglicht das Auffinden von Single-Entry-Point-Objekten (z.B. Bank-Objekt), den Startpunkt z.B. über URL und die Zuordnung von Namen (z.B. IOR) zu Objekten innerhalb von Kontexten.

Dies zeigt folgendes Bild:

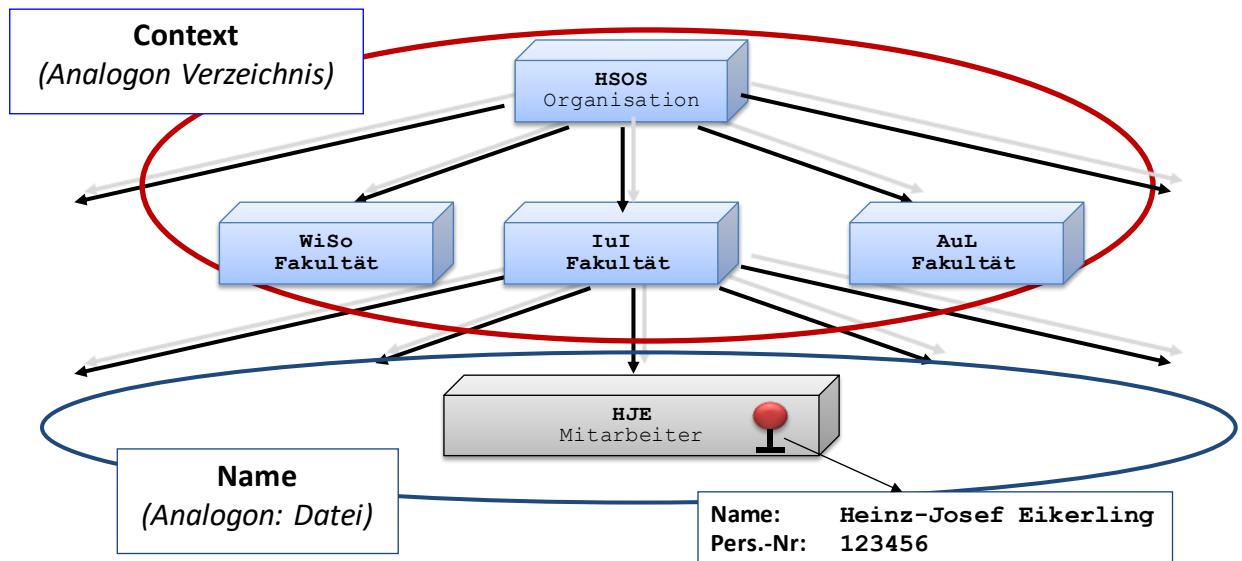


Abbildung 39: Kontextverwaltung durch den CORBA-Naming-Service

Einige weitere Komponenten der CORBA-Architektur sollen etwas ausführlicher besprochen werden:

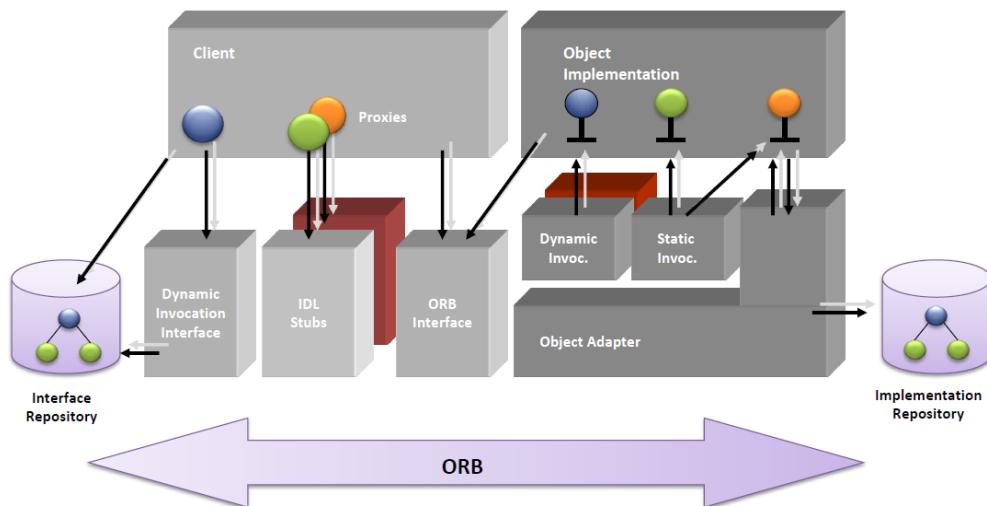


Abbildung 40: CORBA-Architektur

Folgende Teile der CORBA-Architektur sind wichtig:

Dynamic Invocation Interface (DII)

Auf der Clientseite wird eine generische Programmierschnittstelle angeboten, das Dynamic Invocation Interface (DII). Über diese Schnittstelle werden Methodenaufrufe initiiert, die auf der Serverseite über das Gegenstück, das **Dynamic Skeleton Interface (DSI)** weitergeleitet werden. Für statisch gebundene Objekte gibt es entsprechende Static Interfaces (SSI und SII).

Object Adapter (OA)

Die eigentliche Kommunikation geschieht über den Object Adapter (OA), genauer den Portable Object Adapter (POA). Er organisiert die Ausführung einer Operation auf dem Server und steuert die Methodenaufrufe in den Skeletons. Außerdem regelt er die Unterstützung der Lebensdauer von Objekten (Erzeugung, Vernichtung). CORBA definiert zwei Runtime-Datenbanken:

- Das Interface Repository enthält IDL-Spezifikationen für den Abruf zur Laufzeit.
- Das Implementation Repository enthält Informationen über den Server. Sie wird vom OA benötigt für die Aktivierung des Servers.

CORBA bietet ein Konzept zur Verschlüsselung der Verbindung mittels IIOP over SSL:

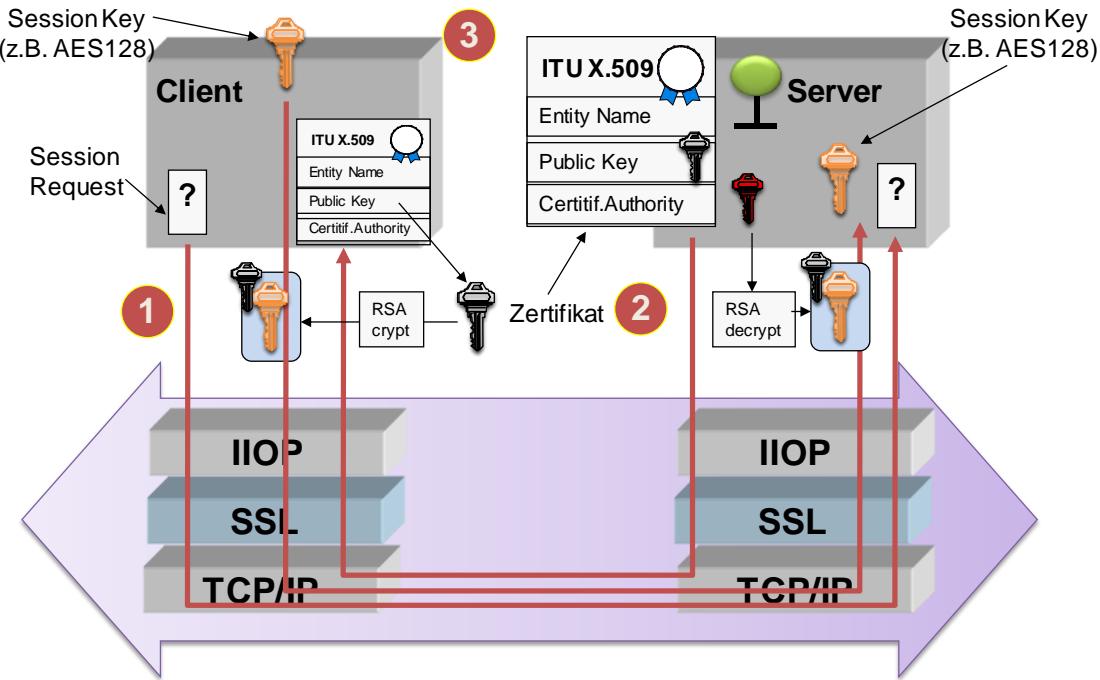


Abbildung 41: CORBA-Sicherheit: IIOP over SSL

6.6 Praktischer Einsatz von CORBA

CORBA ist die Basis aller objektorientierten Ansätze zur Programmierung verteilter Systeme und in einigen Branchen wie Medizin, Banken, Wetterdienste und Telekommunikation sehr beliebt. Der Deutsche Wetterdienst nutzt CORBA intensiv, wie in folgenden beiden Bildern zu sehen ist:

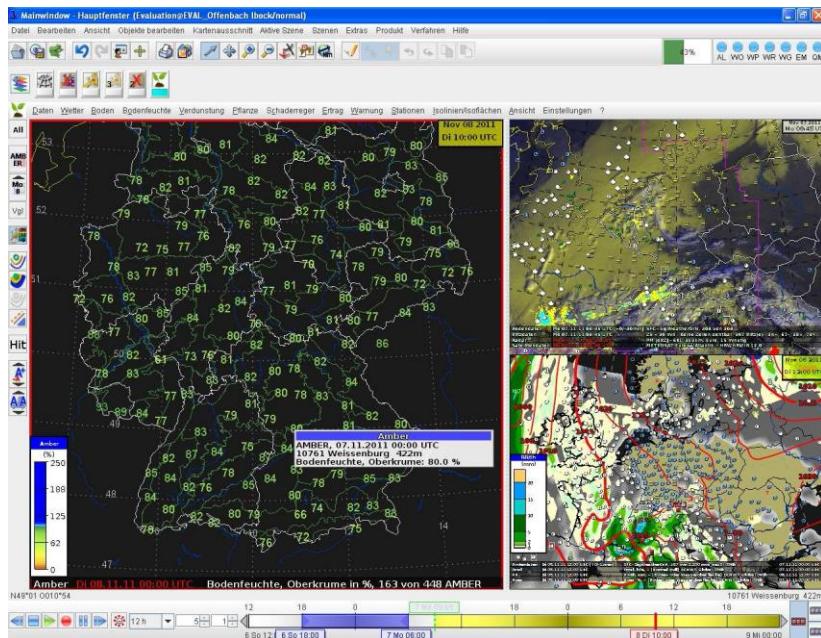


Abbildung 42: Agrarmeteorologischer Arbeitsplatz ³

³ Quelle: [He09] Heizenreder, D. et. al., 2009: Das meteorologische Visualisierungs- und Produktionssystem NinJo, Promet 35, Heft 1-3, Deutscher Wetterdienst, Offenbach 2009

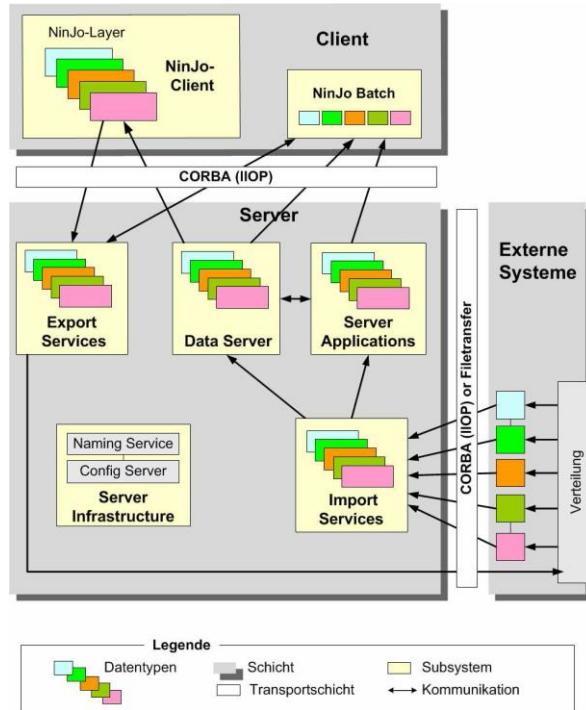


Abbildung 43: CORBA-basierende Architektur (Quelle siehe Fußnote)

Die Integration mit vielen heterogenen Diensten über einen CORBA-basiierenden Enterprise Service Bus (ESB) zeigt folgendes Bild eines großen Bankenrechenzentrums in Münster:

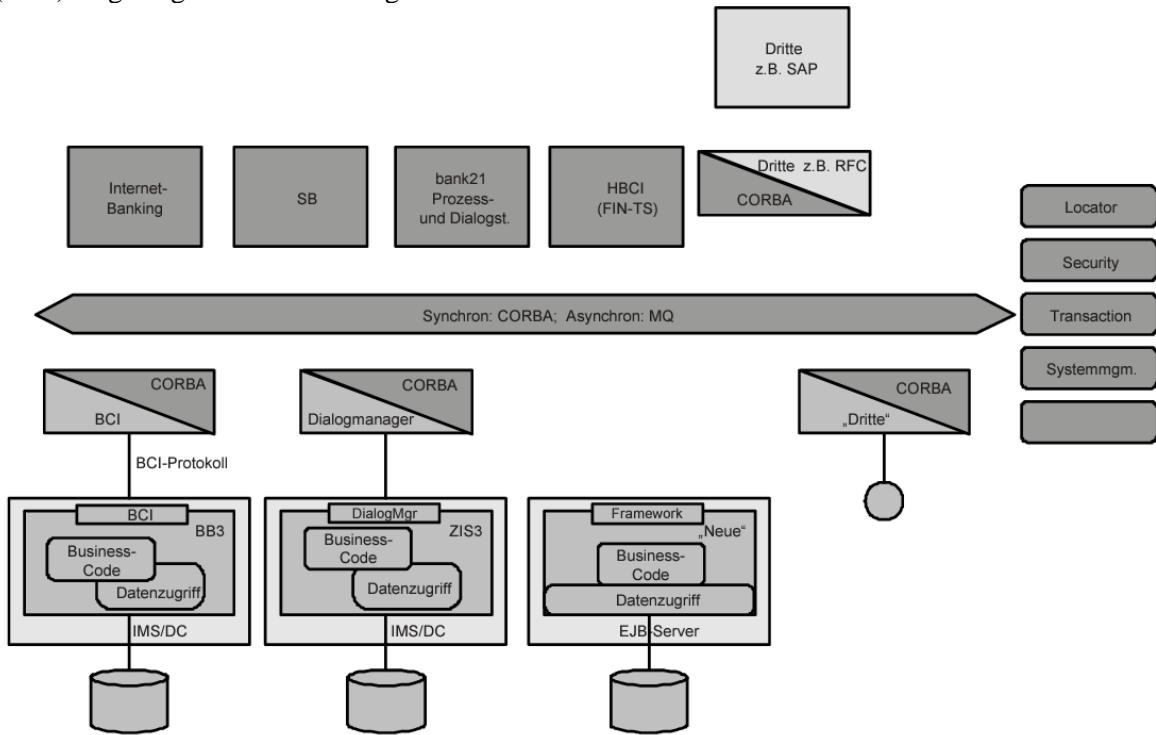


Abbildung 44: Beispiel eines CORBA-basierten Enterprise Service Bus [Dunk08]

6.7 Bewertung von CORBA

CORBA wird nun in Bezug auf die in Kapitel 3 formulierten Anforderungen bewertet:

Anforderung	Kommentar
<i>Netzwerkunabhängigkeit:</i> Clients und Server sollten ohne Änderungen auf unterschiedlichen Netzwerktypen arbeiten können.	CORBA beruht auf Socket-Programmierung und kann daher mit TCP/IP und Protokollarten wie z.B. Token Ring umgehen.
<i>Rechner-/Betriebssystem-Portabilität:</i> Clients und Server sollten ohne Änderungen (außer ggf. Neukompilierung) auf verschiedenen Rechnern und Betriebssystemen arbeiten.	Hier kommt es auf das verwendete CORBA-System an. Prinzipiell (z.B. bei Omniorb) gibt es einen hohen Abdeckungsgrad unterschiedlicher Plattformen.
<i>Portabilität bezüglich der Programmiersprache:</i> Clients und Server sollten sich verstehen, auch wenn sie mit unterschiedlichen Programmiersprachen entwickelt wurden.	Wieder kommt es auf das CORBA-Produkt an. Meist werden mehrere Sprachen unterstützt. Java hat CORBA-kompatiblen Modus (eigenen ORB).
<i>Transparente Nutzung:</i> Programmierer sollen Dienste auf entfernten Rechnern wie lokale Operationen nutzen können.	Ist gegeben, wird durch Inkludieren der entsprechenden Bibliotheken und automatisierte Generierung von Konvertierungsfunktionen und Header-Dateien realisiert.
<i>Einfaches Auffinden und Adressieren:</i> Es soll einfach sein, Server und Dienste aufzufinden und zu adressieren.	Ist gegeben, aktiver Server für Dienstnamen wird beim Naming Service erfragt. Eine Dienstmigration und Lastverteilung sind möglich. Die Objekt-/Methodennutzung für C++-Programmierer intuitiv.
<i>Geringer Protokoll-Overhead:</i> Durch die verwendeten Protokolle sollte möglichst wenig Protokoll-Overhead erzeugt werden, um unnötige Verzögerungen und Kosten, z. B. bei Mobilverbindungen, zu vermeiden.	Stärken: Protokoll-Overhead sehr gering (Konvertierung), die verwendeten Steuerungsbefehle sind kurz.

6.8 Ausblick zu CORBA

Die neue CORBA-Version 3.0 ist inzwischen kommerziell erhältlich und bietet z. B. Firewall-Unterstützung, Quality of Service und vieles mehr. Ein freies komponentenorientiertes CORBA 3.0 wird von Neubauer, B., Ritter, T. und Stinski, F. in ihrem Buch *CORBA-Komponenten* (TXG 135699) [NRS04] beschrieben.

6.9 Entfernter Methodenaufruf unter Java RMI – Remote Method Invocation)

Zum folgenden Abschnitt finden sich weitere Informationen in [Beng14] und [Suns03a]. Im Gegensatz zu CORBA ist der entfernte Methodenaufruf (Remote Method Invocation, RMI) integraler Bestandteil der Programmiersprache ab der Standard Edition aufwärts (nicht in der Micro Edition und nicht in Android). Methoden und der lokale und entfernte Zugriff werden durch JVM (Java Virtual Machine) verwaltet.

Das folgende Bild zeigt eine Einordnung von Java RMI in die Middleware-Welt:

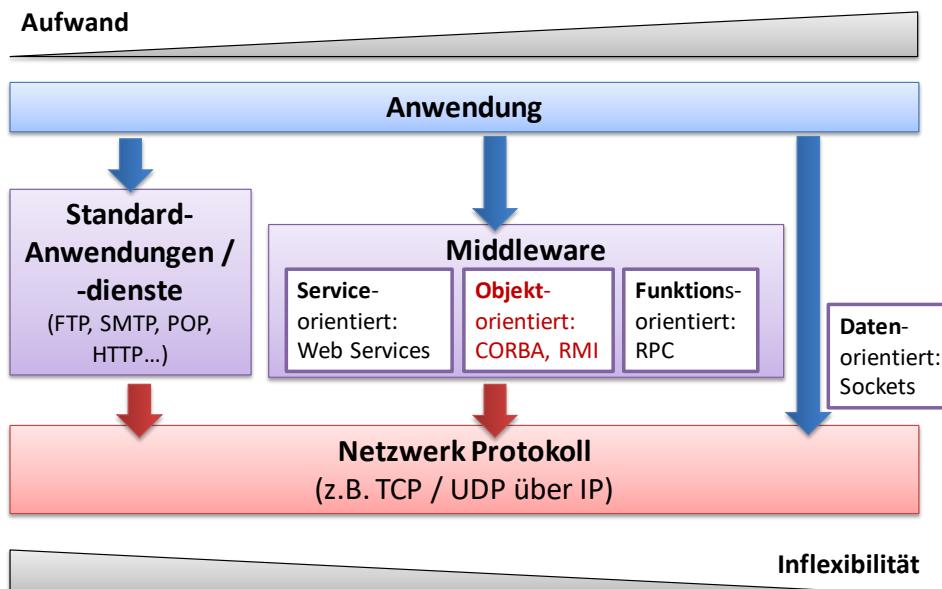


Abbildung 45: Einordnung von Java RMI in die Middleware-Welt

Am linken Ende des Spektrums stehen die Standard-Anwendungen, deren Einsatz wenig Aufwand verursacht, die aber durch Beschränkung auf ihre Aufgabe wenig flexibel sind. Am anderen Ende des Spektrums steht die Entwicklung datenorientierter verteilter Anwendungen, die zwar maximal flexibel ist, aber einen hohen Entwicklungsaufwand verursacht. Dies ist vor allem in vielen Verwaltungs- und Dienstfunktionen begründet, die bei Sockets selbst realisiert werden müssen. Bei Middleware-Ansätzen (auch Java RMI) versucht man die jeweiligen Vorteile durch ein Angebot von unterstützenden Funktionen so zu kombinieren, dass sich die Anwendungsentwickler auf die Kernaufgabe konzentrieren können.

6.9.1 Serialisieren von Objekten

Bei Java/RMI können (wie auch bei CORBA) ganze Java-Objekte mit ihren Methoden als Kopien versendet werden. Dazu wird das Objekt in einen Byte-Strom umgewandelt (Object Serialization). Object Serialization wird auch für die Speicherung des Zustandes eines Objektes in einer Datei verwendet. Bei Nutzung des Stubs wird dieser Mechanismus verwendet, um die Parameter:

- beim Client einzupacken und zu versenden (Marshaling)
- beim Server zu empfangen und wieder auszupacken (Unmarshaling)

Aus Sicherheitsgründen können nur primitive Typen und entfernte Objekte serialisiert werden. Aus dem Package `java.rmi.server` wird die Klasse `UnicastRemoteObject` für Punkt-zu-Punkt-Verbindungen verwendet. Das Anlegen eines entfernten Objektes beim Server geschieht durch Erweiterung des Interfaces `java.rmi.Remote`. Die dadurch entstehende Unterklasse von `Remote` beschreibt die durch den Client aufrufbaren Methoden des Objektes.

6.9.2 Einführung in Java RMI

Um Entwicklern das Selbstschreiben von Stubs zu ersparen, bietet Java einen eigenen Mechanismus zur einfachen Nutzung entfernter Objekte an. Der entfernte Zugriff erfolgt über einen Binder mit der Bezeichnung Registry (`rmiregistry`). Der Standard-Port ist 1099. Bei ihm werden die entfernt nutzbaren Methoden der Objekte registriert. Ein Naming Service ermöglicht dem Client, Referenzen auf eine entfernte Methode zu erhalten.

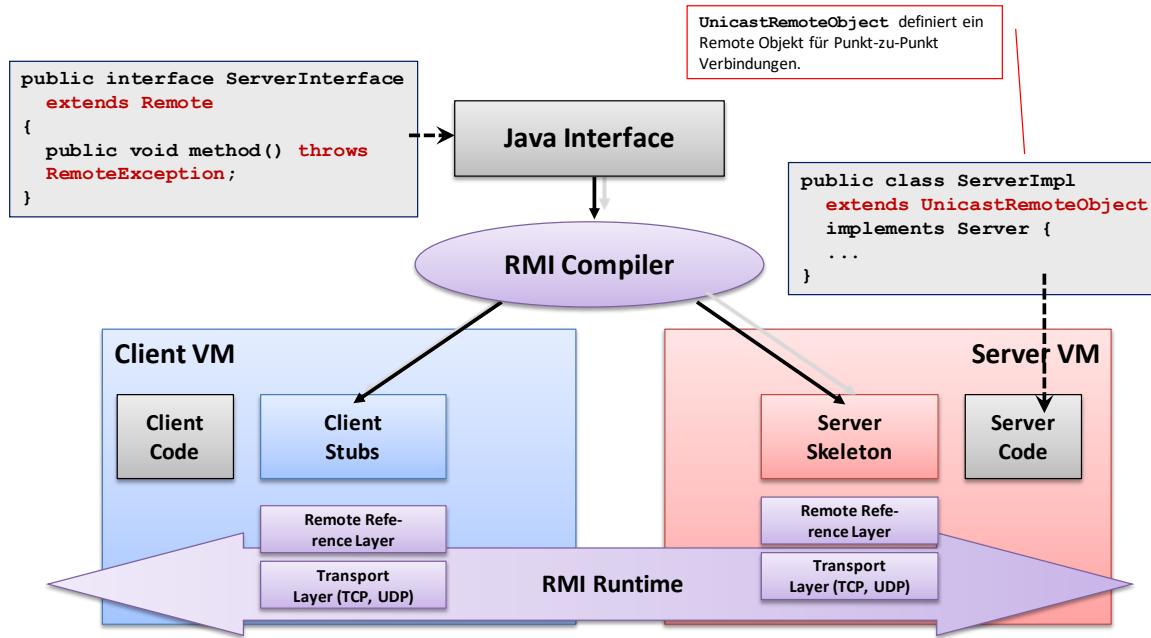


Abbildung 46: Vollständige RMI-Architektur

Transport und Reference Layer sind als Bibliothek in der JVM enthalten. Der Remote Reference Layer bildet entfernte Referenzen (Referenzen auf entfernte Objekte) auf Rechnernamen und Objekte ab. Die Stub-Klasse übernimmt die Funktion eines Proxies (Proxy-Pattern wie in Gamma et al. "Design Pattern" beschrieben). Der Proxy repräsentiert das entfernte Objekt und kümmert sich um die Objektverwaltung und die Methodenaufrufe. Das Skeleton ist das Gegenstück zum Stub und nimmt den Aufruf entgegen, sorgt ggf. für die Aktivierung des Objektes und führt den lokalen Methodenaufruf durch. Nach dem Ausführen der Methode sendet er das Ergebnis an den Stub des Clients zurück. Eine automatische Objektaktivierung sorgt dafür, dass erst zum Zeitpunkt der Dienstnutzung das jeweilige Objekt aktiviert wird, um Ressourcen zu sparen. Eigene Skeletons sind nicht erforderlich, da die notwendigen Informationen durch Reflexion gewonnen werden können. Für die Koppelung mit CORBA (siehe „Interaktion zwischen Java und CORBA“ später in diesem Kapitel) werden CORBA TIEs benötigt, zu deren Erzeugung Skeletons notwendig sind.

6.9.3 Packages für RMI-Aufrufe

Folgende Java-Packages werden zur Nutzung von RMI genutzt:

<code>java.rmi</code>	Klassen, Interfaces und Exceptions (Ausnahmen) für die Client-Seite des Aufrufs
<code>java.registry</code>	Klassen usw. für Kennzeichnung, Registrierung und Auffinden von entfernt aufzurufenden Objekten
<code>java.rmi.server</code>	Klassen usw. für die Server-Seite
<code>java.rmi.dgc</code>	Klassen usw. für die entfernte Speicherbereinigung (distributed Garbage Collection), RMI-Interfaces und -Klassen für den Server

Java regelt den Zugriff auf Klassen eines Packages über Interfaces. Das Interface `Remote` erlaubt Zugriff auf entfernte Schnittstellen. Auf Objekte, die das Interface `Remote` implementieren, können alle JVM, die Netzzugriff auf das Objekt haben, zugreifen, nachdem sie auf der Server-Seite registriert wurden. Das folgende Bild zeigt die Interfaces und Klassen für den Server:

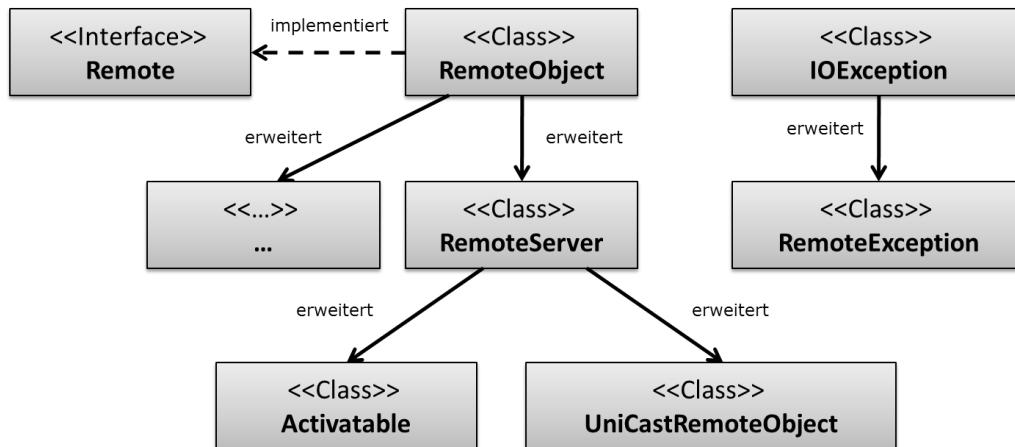


Abbildung 47: Vererbungshierarchie der Server-Klassen für RMI-Aufrufe

Die Klasse `java.rmi.naming` stellt eine Implementierung der Registry dar. Diese enthält die eindeutige Referenz auf ein entferntes Objekt in Form einer URL (Uniform Resource Locator) mit Hostname, Port (normalerweise 1099) und Objekt-Name z. B.

`rmi://test.hs-osnabrueck.de:1234/Hello`

Clients nutzen die Methode `lookup()` mit der URL als Argument und erhalten die zugehörige Objektreferenz. Der mit der URL implementierte Namensraum ist flach (nicht hierarchisch). Ein Beispiel für eine Remote-Interface-Definition zeigt ein Ausschnitt aus dem später beschriebenen Beispiel `TimeServer`:

```

package de.hsos.vs.rmi;
import java.rmi.Remote; import java.rmi.RemoteException;
public interface TimeServer extends Remote {
    public long getTime() throws RemoteException;
}
  
```

Wichtige Methoden (fett gedruckt) von `java.rmi.naming` sind:

```

public static void bind (String url, Remote ro)
    throws RemoteException, AccessException,
           AlreadyBoundException, UnknownHostException
  
```

Mit `bind()` registriert der Server die URL für ein entferntes Objekt. Damit wird das Ziel für den `lookup()`-Zugriff auf Client-Seite zur Verfügung gestellt.

```

public static void rebind (String url, Remote obj)
    throws RemoteException, AccessException,
           UnknownHostException
  
```

`rebind()` bindet einen neuen Namen an eine bereits gebundene URL. `rebind()` wird häufig anstelle von `bind()` verwendet, um sicherzustellen, dass auch bei vorher gebundener URL erneut gebunden wird.

```

public static Remote lookup (String url)
    throws RemoteException, NotBoundException,
           AccessException, UnknownHostException
  
```

Der Client benutzt `lookup()`, um zu einer URL die Referenz zu erhalten und damit das entfernte Objekt nutzen zu können.

```

public static void unbind (String url)
    throws NotBoundException, AccessException, UnknownHostException
  
```

`unbind()` entfernt ein Objekt aus der Registry

```

public static String[] list (String url)
    throws RemoteException, AccessException, UnknownHostException
  
```

`list()` liefert alle Namensbindungen an eine URL.

Der Ablauf eines RMI-Aufrufs ist in folgendem Bild am Beispiel `TimeServer` zu sehen:

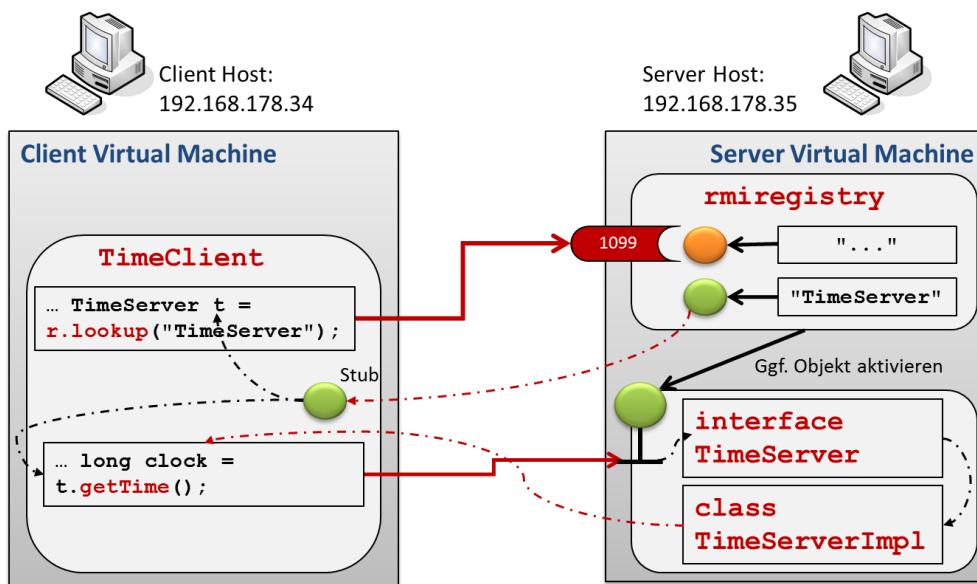


Abbildung 48: Reihenfolge bei RMI-Aufrufen

1. rebind meldet Obj (durch URL gekennzeichnet) bei der rmiregistry an. Diese muss zuvor gestartet worden sein.
2. Der Client erfragt mit lookup(URL) eine Referenz auf eine entfernte Methode. Die Registry aktiviert das Server-Objekt, falls es noch nicht aktiv ist, und gibt die Stub-Referenz zurück.
3. Ist das Stub-Objekt beim Client nicht vorhanden, wird es beim Server serialisiert und zum Client übertragen.
4. Der lokale Client-Aufruf `getTime()` führt über den Stub zum Ausführen der Server-Methode.

Bei der Parameterübergabe von Remote-Methoden können primitive Datentypen aber auch serialisierte Objekte übergeben werden. Dazu müssen diese das `java.io.Serializable`-Interface implementieren. Fehlende Klassen für Parameter oder Rückgabewerte können auf der Empfängerseite ggf. nachgeladen werden. Die Übergabe von Nicht-Remote-Objekten erfolgt durch Erzeugen einer Kopie und Verschicken durch Java-Serialisierung. Im Empfänger wird das Objekt dann mit denselben Daten deserialisiert. Remote-Objekte, die von `UnicastRemoteObject` abgeleitet sind, werden per Stub an den Empfänger verschickt (Call by Reference). Das übergebene Remote-Objekt kann nur Remote Interfaces implementieren.

6.9.4 Beispiel TimeServer

Im RMI-Beispiel `TimeServer` soll ein simpler Zeitdienst zur Veranschaulichung von Java RMI dienen. `TimeServer.java` enthält die Serververwaltung:

```
package de.hso.vs.rmi;
import java.rmi.*;
public class TimeServerImpl extends UnicastRemoteObject implements
    TimeServer {
    public long getTime() throws RemoteException
        return System.currentTimeMillis();
    }
    public static void main(String[] args) {
        try {
            // Man kann entweder eine lokale Registry erzeugen ...
            LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            // ... oder verwendet die Standard-RMI Registry auf dem Lokalen
            // Host unter Port 1099. In diesem Fall müssen aber Policies
            // konfiguriert werden. Siehe timer.policy in diesem Verzeichnis
            // fuer Hinweise.
            // Registry registry = LocateRegistry.getRegistry();
```

```

        Naming.rebind("TimeServer", new TimeServerImpl());
    } catch (Exception ex) {}
}
}

```

Ein Timeserver-Objekt wird mittels Aufruf des Konstruktors der Superklasse `TimeServerImpl()` erzeugt und mit `rebind()` bei der RMI-Registry angemeldet. Das entspricht dem Schritt 1 im Bild oben. Wie bereits bei RPC und CORBA gesehen, sehen die Methoden selbst wie eine lokale Methode aus. Der entfernte Aufruf kommt durch die Deklaration in einem entfernten Interface `TimeServer` zustande. Der Ort der Deklaration, nicht aber das Aussehen der Methode, entscheidet über den lokalen oder entfernten Aufruf. Entscheidend ist die Zeile

```
public class TimeServerImpl extends UnicastRemoteObject implements
TimeServer
```

Die `Impl`-Klasse muss also das `UnicastRemoteObject` erweitern, um entfernt nutzbar zu sein. Welche Methoden dann tatsächlich verwandt werden, wird durch Implementieren des entsprechenden Interfaces (hier `TimeServer`) definiert. Neben den entfernt nutzbaren Methoden könnte es auch lokale geben, die nicht nach außen sichtbar sein sollen.

Alternativ kann man das Server-Objekt direkt in der main-Methode als Stub zu exportieren

```
TimeServerImpl obj = new TimeServerImpl();
TimeServerInterf stub = (TimeServerInterf)
        UnicastRemoteObject.exportObject(obj, 0);
```

Der Client `TimeServerClient.java` erzeugt ein Objekt, das über die `lookup`-Methode des Naming-Dienstes beim Server gefunden wird. Danach können Methoden wie beim lokalen Aufruf genutzt werden.

```
package de.hsos.vs.rmi;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.registry.LocateRegistry;
public class TimeClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry();
            TimeServer timer = TimeServer)registry.lookup("TimeServer");
            System.out.println("TimeServer: " + timer.getTime());
        } catch (AccessException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
}
```

Zur Ausführung des Beispiels verwenden wir die im Server aufgerufene `rmiregistry` und gehen von folgender Arbeitsteilung aus:

```
westerka@si0024-1:      Server
westerka@si0024-2:      Client
```

Zunächst müssen die Java-Dateien übersetzt werden:

```
westerka@si0024-1:~/> javac *.java
westerka@si0024-2:~/> javac *.java
```

Nun wird der Server mit der integrierten RMI-Registry gestartet und in Gang gesetzt:

```
westerka@si0024-1:~/> java de.hsos.vs.rmi.TimeServerImpl
TimeServerImpl created...
TimeServerImpl registered as 'TimeServer' ...
```

Nun kann der Client den Server nutzen:

```
westerka@si0024-1:~/> java java de.hsos.vs.rmi.TimeClient  
TimeServer: 1476882009531 (Wed Oct 19 15:00:09 CEST 2016)
```

6.9.5 Java RMI Daemon

Alternativ zur RMI-Registry gibt es auch einen RMI Daemon (`rmid`). Er ist vergleichbar mit dem Object Adapter bei CORBA und aktiviert die Objekte dynamisch.

6.9.6 Java RMI und Sicherheit

In den gezeigten Beispielen wurde auf die Implementierung eines Sicherheitskonzepts verzichtet, wie dies in nach außen abgeschotteten Netzen (Firmen-Intranet) möglich ist. Für die Absicherung von Aufrufen in heterogenen Netzen bietet Java RMI ein Sicherheitskonzept, dass auf einem sogenannten Securitymanager basiert. Zur Nutzung wird im Hauptprogramm des Servers und des Clients als erste Anweisung der Start des Securitymanagers durchgeführt:

```
public static void main (String args[]) throws Exception {  
    System.setSecurityManager(new RMISecurityManager());  
    // Danach Erzeugen eines Serverobjekts und rebind (Server)  
    // bzw. lookup-Aufruf (Client)
```

Er kontrolliert das Verhalten von nachgeladenen Klassen. Das genaue Verhalten des Sicherheitsmanagers lässt sich durch ein Policy-File steuern. Auch eine moderne Zertifikatverwaltung steht zur Verfügung.⁴

6.9.7 Performance und Portabilität

RMI beruht auf Java-Sockets und dem darunter liegenden Netzwerk, teilt also die entsprechenden Performance-Eigenschaften. Wichtige Einflussgrößen für die Performance sind z. B. die Window-Größen (LAN <-> Internet) und die Garbage Collection, die je nach Heap-Füllstand verzögert werden kann. Es kann sinnvoll sein, RMI-Methodenaufrufe zu kombinieren (Chaining), um deren sequenzielle Bearbeitung zu umgehen.

Die Serialisierung hat einen erheblichen Einfluss auf die Performance. Durch Nutzung der Externalized-Klasse (anstelle von Serialization) kann dieses Problem auf Kosten einer aufwändigeren Programmierung teilweise beseitigt werden. Möchte man andere Software und Systeme in eine verteilte Anwendung integrieren, kann die Beschränkung von RMI auf die Programmiersprache Java störend wirken. Um programmiersprachenunabhängig arbeiten zu können, muss z. B. CORBA in Erwägung gezogen werden. Außerdem wurden viele Middleware-Systeme neuerdings mit SOAP-Schnittstellen zur Nutzung als Web Services erweitert (siehe Kapitel 7).

Mit Cajo gibt es einen freien Ansatz, der gegenüber Java RMI vereinfacht ist und ohne explizite Schnittstellen und Schnittstellen-Compile rauskommt (<https://github.com/ravn/cajo>). Einen ähnlichen Ansatz verfolgt SIMON <http://dev.root1.de/projects/simon/wiki>.

6.10 Interaktion zwischen Java und CORBA – Beispiel nsd_sequence_jclient

Java ist später als CORBA entstanden und integriert in eine Programmiersprache die Möglichkeit, verteilte Anwendungen oberhalb des Socket- und Methodenansatzes zu entwickeln.

Neben den im nächsten Kapitel besprochenen komponentenbasierten Ansätzen war und ist es Entwicklungsziel von Java, mit bestehenden CORBA-basierten Netzwerkanwendungen zu kooperieren. Dazu wurden sukzessive Möglichkeiten zur bequemen Kommunikation zwischen CORBA- und Java-Anwendungen geschaffen.

4

http://openbook.galileocomputing.de/javainsel8/javainsel_26_001.htm#mj9db5f1554c352606233f8d7e8b6796eb

6.10.1 CORBA-Unterstützung in Java-Versionen

Auch wenn Java anfangs von einem Inseldasein mit bescheiden unterstützter Verbindung zur übrigen verteilten Welt ausging, wurde bald klar, dass Koppelungsmöglichkeiten zum etablierten CORBA und seinen Protokollen sinnvoll waren. Seit JDK 1.4 (ca. 2002) gibt es eine Unterstützung für die CORBA-Version 2.3.1 mit Portablem Object Adapter (POA) im ORB.

6.10.2 RMI-IIOP, Java IDL und das Mapping

Normalerweise nutzt Java zur Realisierung von RMI mit anderen Java-Applikationen das eigene Protokoll Java Remote Method Protocol (JRMP). Um mit CORBA-Applikationen zusammenzuarbeiten, muss man das IIOP-Protokoll nutzen.

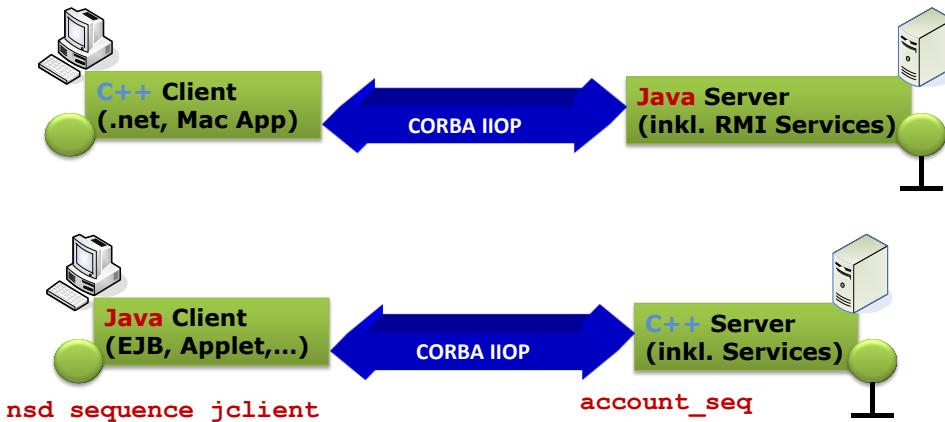


Abbildung 49: Anwendungsintegration von C++/Java-Lösungen mittels IIOP

Das folgende Bild zeigt beispielhaft, wie ein Java-Client über ein Java/CORBA-Mapping eine IIOP-Kommunikation zwischen dem in Java eingebauten ORB und dem Server-ORB aufbaut:

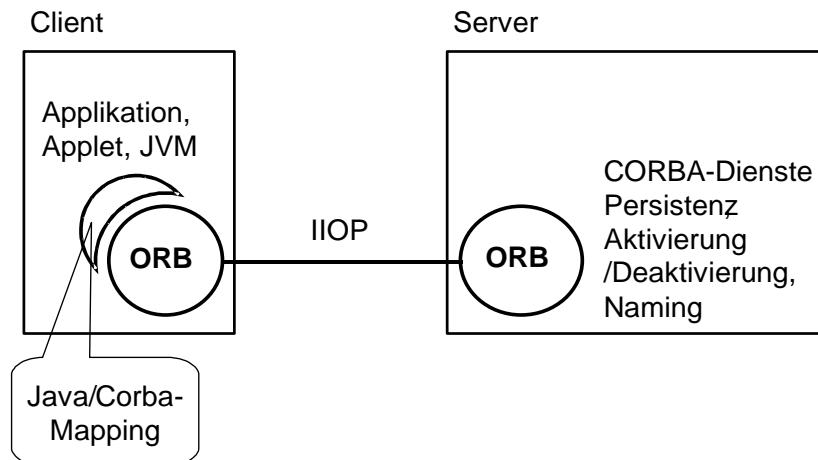


Abbildung 50: Prinzip der Zusammenarbeit zwischen Java und CORBA

Aus einer idl-Spezifikationsdatei werden sprachspezifische Schnittstellen und abstrakte Klassen (Stub etc.) durch Aufruf des idlj-Programms erzeugt. Die weitere Bearbeitung geschieht wie vom Java-eigenem RMI gewohnt.

6.10.3 Java-Client JClient für das C++-CORBA-Beispiel account_nameserv

Die Koppelung von Java-Clients an ein CORBA-System soll am Beispiel eines Java-Clients für den Kontoserver aus Beispiel `account_nameserv` aus Kapitel 6.3.4 illustriert werden. Das Programm `JClient.java` sieht folgendermaßen aus:

```
// JClient.java
// Java-Client fuer den account-nameserv-Server

import java.util.*;
public class account_nam_JavaClient {
    public static void main(String [] args) {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        if (orb == null) System.exit(-1);
        try {
            // obtain service from naming server
            org.omg.CORBA.Object ns_obj =
                orb.resolve_initial_references("NameService");
            org.omg.CosNaming.NamingContext nc
                = org.omg.CosNaming.NamingContextHelper.narrow(ns_obj);
            org.omg.CosNaming.NameComponent [] path
                = { new org.omg.CosNaming.NameComponent("Accountname", "") };
        };
        org.omg.CORBA.Object obj = nc.resolve(path);
        Account myAccount = AccountHelper.narrow (obj);
        // Geld einzahlen und abheben
        myAccount.deposit (700);
        myAccount.withdraw (350);

        // Neuen Kontostand anzeigen
        System.out.println("Neuer Kontostand:"+myAccount.balance());
        // destroy
        orb.destroy();
    } catch (org.omg.CORBA.ORBPackage.InvalidName exception) {
        exception.printStackTrace(System.out);
    } catch (org.omg.CosNaming.NamingContextPackage.NotFound
exception) {
        exception.printStackTrace(System.out);
    } catch (org.omg.CosNaming.NamingContextPackage.CannotProceed
exception){
        exception.printStackTrace(System.out);
    } catch (org.omg.CosNaming.NamingContextPackage.InvalidName
exception){
        exception.printStackTrace(System.out);
    } catch (org.omg.CORBA.COMM_FAILURE exception) {
        exception.printStackTrace(System.out);
    } catch (Exception exception) {
        exception.printStackTrace(System.out);
    }
}
}
```

Der Vergleich mit dem in C++ geschriebenen Client zeigt, dass der Ablauf sehr ähnlich ist. In diesem Programm wird eine differenzierte `try/catch`-Fehlerbehandlung implementiert, die in C++ ebenso

möglich (und sinnvoll) wäre, aber zunächst zur Verbesserung der Übersichtlichkeit weggelassen wurde. Durch

```
idlj account.idl  
javac *.java
```

werden die Helper-Dateien aus der idl-Schnittstellenbeschreibung erzeugt und in Byte-Code übersetzt.

Dabei werden folgende Dateien und ihre zugehörigen .class-Dateien erzeugt:

`AccountHelper.java, AccountHolder.java, AccountOperations.java,
Account.java, _AccountStub.java`

Ein Blick in diese Dateien zeigt, dass der Stub und die für die Konformität mit dem RMI-Konzept notwendigen Dateien erzeugt werden. Nach Start des Naming Servers und des `server_nameserv` führt der Aufruf des Clients mit `StartJavaClient` zu den gleichen Ausgaben, wie dies beim C++-Programm der Fall war:

```
westerka@si0024-1:~/> ./startomnianames.sh  
  
westerka@si0024-2:~/> ./startserver.sh  
  
myBank wird an Naming Service gebunden ... done.
```

```
westerka@si0024-3:~/> ./startclient.sh  
Neuer Kontostand ist 450  
westerka@si0024-3:~/> ./startclient.sh  
Neuer Kontostand ist 900
```

Mit diesem Beispiel ist aufgezeigt, dass auch auf Objektebene eine gute Austauschbarkeit zwischen C++ mit CORBA und Java RMI gegeben ist. Prinzipiell wäre auch ein Java-Server und ein C++Client denkbar. Der gezeigte Fall ist jedoch der häufigere, da CORBA in vielen Firmen in der Server-Landschaft verbreitet ist und auch die Einbindung weiterer IT-Systeme in C++ leichter ist. Ein anderer Begriff für solche vorhandenen, älteren Systeme ist Legacy-Systeme. Bei Banken sind diese häufig in COBOL programmiert und werden üblicherweise über CORBA-Systeme angebunden. Die Vorteile von Java bezüglich der Benutzerschnittstellen und der Einbindung in Web-Applikationen können also ohne Änderungen in der Server-Welt realisiert werden.

Im Buch Enterprise Java Beans (EJB) von Richard Monson-Haefel wird darüber gezeigt, dass sich Java RMI auch über das SOAP-Protokoll, das zur Realisierung von Web-Services (siehe Kap. 6) dient, abwickeln lässt.

6.11 Nutzung von Callbacks mit Java RMI

Solange man mit dem synchronen Request/Response-Prinzip auskommt, kann man Java RMI wie beschrieben verwenden. Häufig hat man aber die Notwendigkeit einer asynchronen Kommunikation nach folgendem Prinzip:

- Ein Server publiziert einen Dienst, der zu zufälligen Zeitpunkten Informationen versendet bei einer Verwaltung (Publish).
- Ein Client registriert sich für diesen Dienst (Subscribe)
- Der Server hat neue Daten und möchte diese allen registrierten Clients schicken (Notify).

Mit den bisherigen Ansätzen musste der Client regelmäßig beim Server anfragen, ob es neue Daten vorliegen (Polling). Mithilfe eines Callback-Verfahrens (das es auch bei der nicht verteilten Programmierung gibt) kann man dies vermeiden.

Beim Callback-Verfahren gibt der Client im Augenblick der Registrierung dem Server eine Callback-Referenz, die bei Benachrichtigungen aufgerufen werden soll. Dies soll am Beispiel eines Java-RMI-Calculator-Servers näher erläutert werden.

Das folgende Bild zeigt das Klassendiagramm des Calculator-Beispiels:

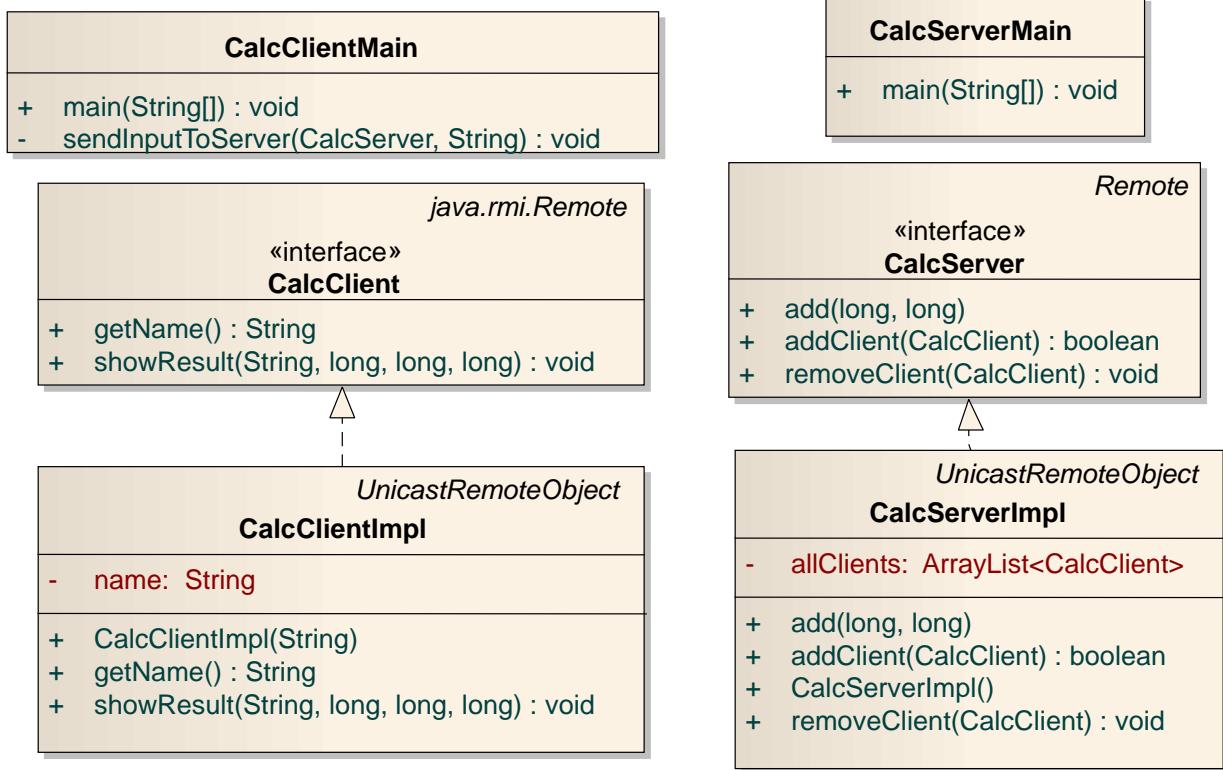


Abbildung 51: Klassenbild des Calculator-Callback-Beispiels

Der Server bietet die `addClient()`- und die `removeClient()`-Methoden zum An- und Abmelden von Clients an. Der eigentliche Dienst ist die `add()`-Methode, die eine Addition durchführt. Die Argumente und das Ergebnis der Addition werden allen registrierten Clients durch Aufruf von deren `showResult`-Methode mitgeteilt. Der Server führt in `allClients` eine Liste der registrierten Clients.

Den Registrierungsablauf für zwei Clients `c1` und `c2` zeigt folgendes Bild:

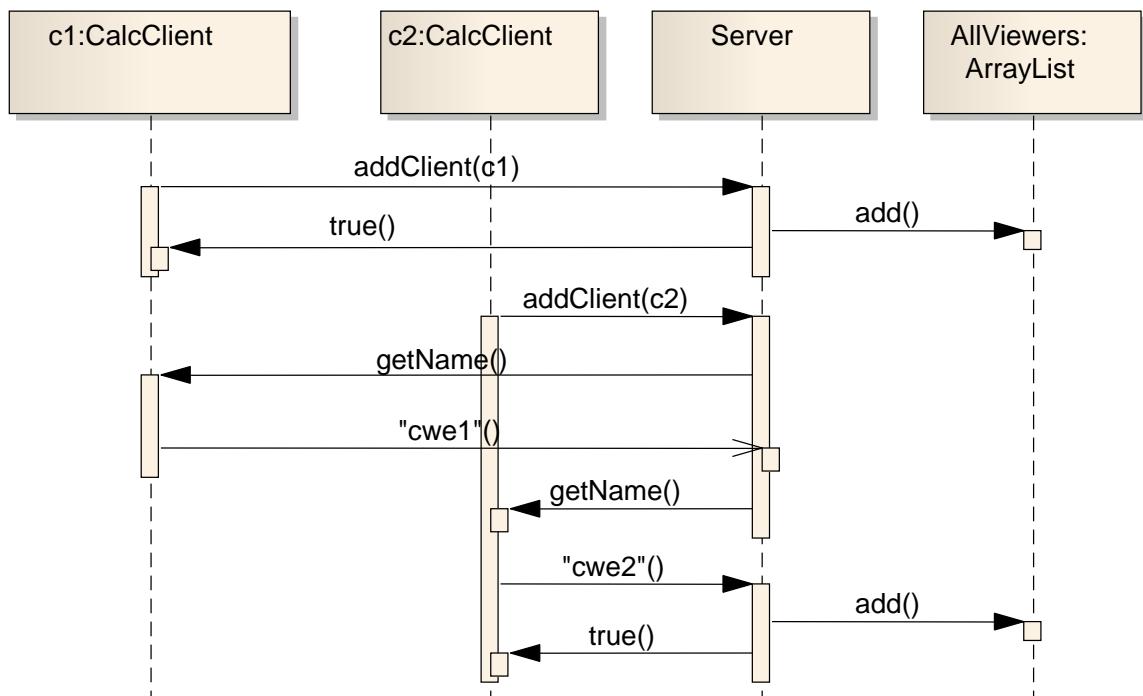


Abbildung 52: Sequenzdiagramm des Ablaufs beim Registrieren von zwei Clients

Nach der Registrierung erfolgt die Nutzung des Servers wie in folgendem Bild:

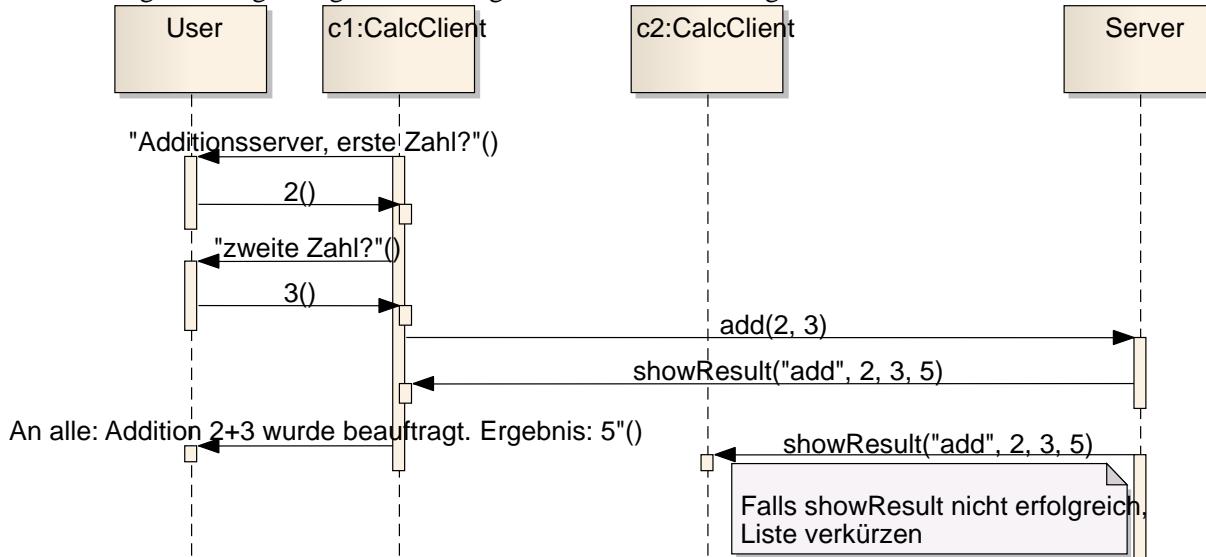


Abbildung 53: Sequenzdiagramm zur Addition und Benachrichtigung registrierter Clients

Der Benutzer des ersten Clients gibt die beiden zu addierenden Zahlen ein und der Client sendet diese an den Server. Dieser benachrichtigt alle registrierten Clients über das Ergebnis (Benutzeranzeige hier nur für c1 abgebildet). Sollten Clients zwischendurch ohne explizite Deregistrierung verschwinden, fällt dies spätestens beim showResult()-Aufruf auf und die Liste kann entsprechend bereinigt werden. Mit diesem Beispiel lassen sich weitere Anwendungen mit Callback-Funktionen implementieren.

6.12 Bewertung von Java RMI

Die Bewertung von Java RMI in Bezug auf die in Kapitel 3 formulierten Anforderungen ergibt:

Anforderung	Kommentar
<i>Netzwerkunabhängigkeit:</i> Clients und Server können ohne Änderungen auf unterschiedlichen Netzwerktypen arbeiten.	Auch Java RMI beruht letztlich auf der Socketprogrammierung und kann daher mit TCP/IP und Protokollarten wie z.B. Token Ring umgehen.
<i>Rechner-/Betriebssystem-Portabilität:</i> Clients und Server können ohne Änderungen (außer ggf. Neukompilierung) auf verschiedenen Rechnern/Betriebssystemen arbeiten.	Solange es eine JVM mit RMI gibt (auf vielen Standard-Desktop-Plattformen, aber nicht auf Smartphones), können Clients und Server damit genutzt werden.
<i>Portabilität bezüglich der Programmiersprache:</i> Clients und Server verstehen sich, auch wenn sie mit unterschiedlichen Programmiersprachen entwickelt wurden.	Java RMI ist auf Java beschränkt, Interoperabilität mit CORBA auf Objektebene ist durch Nutzung passender Naming Server gegeben.
<i>Transparente Nutzung:</i> Programmierer können Dienste auf entfernten Rechnern wie lokale Operationen nutzen.	Ist gegeben, Java RMI ist sogar Bestandteil der Sprache und wird durch Implementierung einer Schnittstelle gelöst.
<i>Einfaches Auffinden und Adressieren:</i> Es soll einfach sein, Server und Dienste aufzufinden und zu adressieren.	Ist gegeben, RMI Registry verwaltet entfernt nutzbare Objekte. Die Objekt-/Methodennutzung für Java-Programmierer intuitiv. Übertragene Daten sind von der Steuerungskommunikation getrennt.
<i>Geringer Protokoll-Overhead:</i> Die verwendeten Protokolle sollen möglichst wenig Protokoll-Overhead verursachen, um unnötige Verzögerungen und Kosten, z. B. bei Mobilverbindungen, zu vermeiden.	Protokoll-Overhead bei Verwendung einfacher Datentypen sehr gering (Konvertierung), die verwendeten Steuerungsbefehle sind kurz. Serialisieren von Objekten ist eine mächtige Zusatzoption, die aber Overhead mit sich bringt.

7 Weborientierte Programmierung verteilter Systeme

Die Web-basierende Nutzung verteilter Systeme stellt intelligente Benutzeroberflächen und Navigationsmöglichkeiten zur Verfügung, die die bisherigen Standalone-Programme mehr und mehr verdrängt. Die Vorteile liegen auf der Hand:

- Entwickler müssen sich (kaum) um Portabilität und Rechner-/Betriebssystemabhängigkeit ihrer Anwendungen kümmern. Um diese Aspekte kümmern sich die Browser-Hersteller.
- Die Aktualisierung von Programmen erfordert keine komplizierten Rollout-/Update-Prozesse.
- Sich ändernde Inhalte können leicht angepasst werden, ohne dass z. B. ein Datenbank-Client oder ein anderes Anwendungsprogramm erforderlich ist.
- Die Web-basierte Programmierung verteilter Systeme nahm ihren Anfang 1992, als das Kernforschungszentrum CERN in der Schweiz ein Informationssystem aufbaute, das sich schnell zum World Wide Web auswuchs. 1992 erstand der Autor dieses Skripts ein Büchlein von knapp 2 cm Dicke, das den heute witzig anmutenden Titel „The Whole Internet“ trägt. Das WWW wurde darin auf 15 Seiten abgehandelt.

Die Grundlagen der Web-Programmierung werden als bekannt angenommen, da sie in vorhergehenden Vorlesungen behandelt werden. Zu den vorausgesetzten Aspekten gehören URIs und URLs, Content-Type, MIME-Type etc.. Zur Wiederholung wird kurz auf HTTP und statische Web-Seiten eingegangen, um die daraus hervorgegangenen aktiven und dynamischen Ansätze (auch als client-/server-seitige Web-basierte Programmierung verteilter Systeme bezeichnet) zu erläutern. Auf die vielen Möglichkeiten, Web Seiten attraktiv und adaptiv zu gestalten, wird in separaten Vorlesungen eingegangen. Die Historie der Web-orientierten Programmierung lässt sich recht gut anhand eines Zeitstrahls aufzeigen, der die Einführung der jeweiligen Technologieerweiterung darstellt:

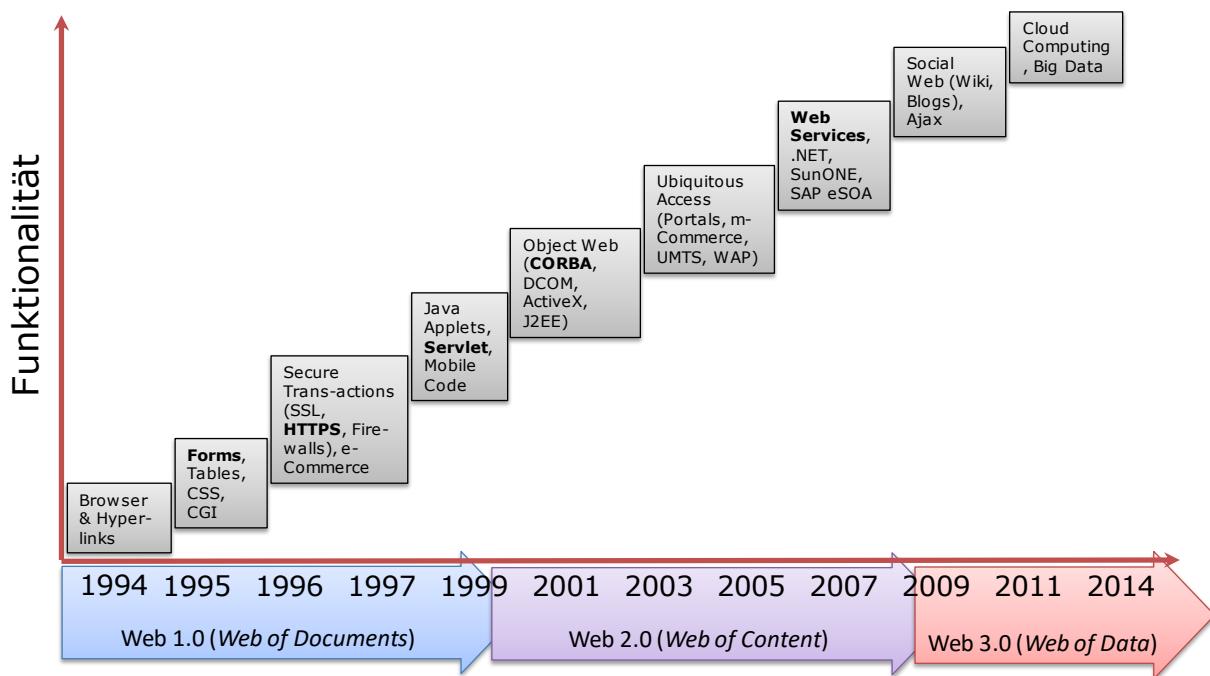


Abbildung 54: Historischer Verlauf der Einführung verschiedener Web-Technologien

Man sieht, dass sich die (bisher drei) Epochen des Web an der Fokussierung auf Dokumente/Web-Seiten (1.0), den Inhalt allgemein (2.0) und Daten (3.0) fokussieren.

HTTPDas Hypertext Transport Protocol wichtige Grundlage von Web-Applikationen und ähnlich wie FTP und Telnet als Anwendungsprotokoll in RFC 2616 (Version 1.1 aus 1999) spezifiziert. Der verwendete Standard-Port ist 80. Das folgende Bild zeigt die Einordnung innerhalb des TCP/IP-Schichtenmodells:

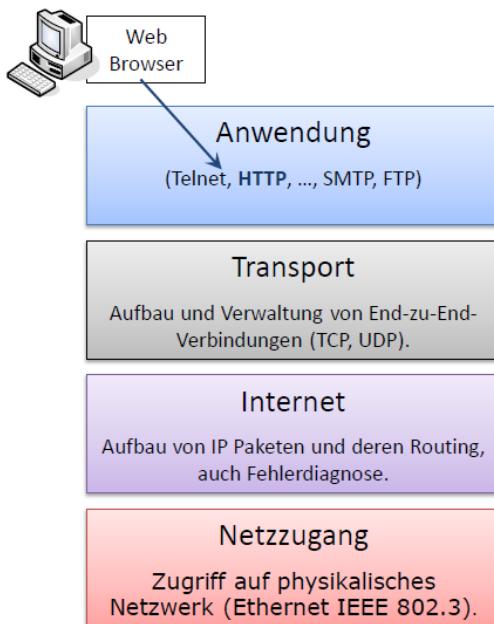


Abbildung 55: Einordnung von HTTP im TCP/IP-Schichtenmodell

HTTP Request-Response-Mechanismus bei einem Aufruf einer URL in einem Browser:

1. Client: Aufbau einer TCP/IP-Verbindung zum entsprechenden Web-Server.
2. Client: Generierung eines HTTP-Requests und Versendung an Server.
3. Server: liest Request, führt die angefragte Aktion (z.B. Zugriff auf HTML-Seite) aus und schickt eine HTTP-Response an Client (inklusive HTML-Seite).
4. Client: liest die Response und zeigt die Datei im Browserfenster an.

Das folgende Bild zeigt die vier Schritte in ihrer Auswirkung beim Client und beim Server:

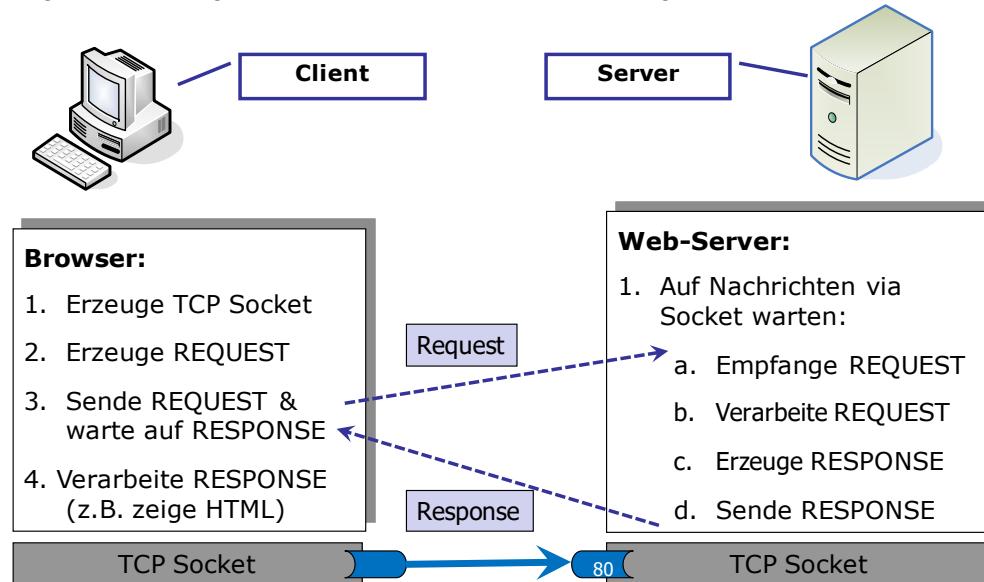


Abbildung 56: Ablauf eines Request-/Response-Vorgangs bei Client und Server

Es geht davon aus, dass im Client ein Browser läuft, der von einem Web-Server Informationen durch Angabe einer URL lädt. Dies geschieht nach dem Request/Response-Schema in vier Schritten:

1. Client: Aufbau einer TCP/IP-Verbindung zum entsprechenden Web-Server.
2. Client: Generierung eines textbasierten HTTP-Requests und Versendung an Server.
3. Server: liest Request, führt die angefragte Aktion (z.B. Zugriff auf HTML-Seite) aus und schickt (über den (bidirektionalen) Socket) eine textbasierte HTTP-Response (Antwort inklusive HTML-Seite) an den Client.
4. Client: liest die Response und zeigt die Datei im Browserfenster an.

Die Mächtigkeit des Ansatzes besteht einerseits aus dem Seitenaufbau, der die zuvor textorientierte Darstellung von netzbasierten Applikationen ablöste. Zum anderen kann man durch URLs und Links von einer Startseite aus beliebig in einem Informationsbestand verzweigen.

Die Internet Engineering Taskforce (IETF) arbeitet momentan an Version 2.0 von http und folgt dabei weitgehend einem Vorschlag von Google mit dem Namen SPDY. Version 15 des Entwurfs wurde am 27.10.2014 veröffentlicht und ist unter <http://tools.ietf.org/html/draft-ietf-httpbis-http2-15> zu finden.

7.1.1 Aufbau der HTTP-Requests und Responses

Das folgende Bild zeigt wie HTTP-Requests und Responses prinzipiell aufgebaut sind:

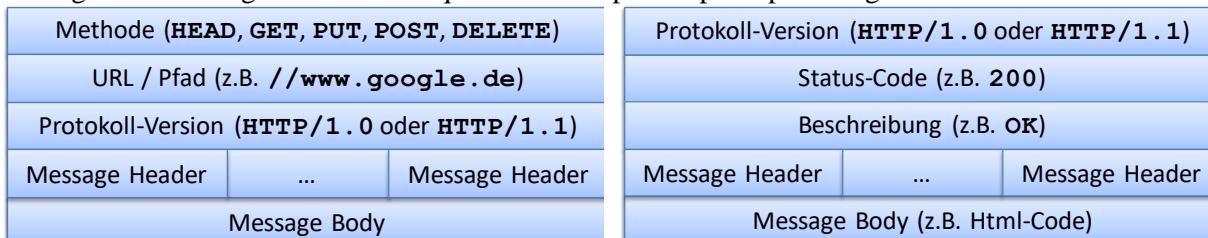


Abbildung 57: Prinzipieller Aufbau von HTTP-Requests und -Responses

Beim Request wird eine Methode angegeben. Die wichtigsten vier sind in folgender Tabelle aufgeführt:

Methode	Bedeutung
GET	fragt eine Ressource per URL an
HEAD	fragt nur den Header einer Ressource an; hiermit kann z.B. die Gültigkeit von Links geprüft werden
POST	wird zur Übertragung von Daten (z.B. interaktives Formular) zwischen Client und Server verwendet
PUT	Übertragene Ressource soll unter dem Namen der URL gespeichert werden

Weitere Methoden sind DELETE, TRACE, OPTIONS und CONNECT. Nach der Methode werden die angeforderte URL und das verwendete Protokoll angegeben. Danach kommen Message Header und – Body.

In der Response wird nach der Protokollversion ein Statuscode als dreistellige Zahl und ein zugehöriger Text ausgegeben. Die Bedeutung ist in folgender Tabelle ersichtlich:

Status Code	Bedeutung
Informational: 1xx	Bearbeitung des Requests läuft z.B. „100 Continue“, „101 Switching Protocols“
Success: 2xx	Request erfolgreich empfangen und verarbeitet z.B. „200 OK“, „204 No Content“
Redirection: 3xx	Weitere Aktionen zur Verarbeitung des Requests nötig z.B. „301 Moved Permanently“, „304 Not Modified“
Client Error: 4xx	Request enthält Fehler oder kann nicht ausgeführt werden z.B. „403 Forbidden“, „404 Page not found“, „405 Method not Allowed“
Server Error: 5xx	Server nicht in der Lage, den Request auszuführen z.B. „500 Internal Server Error“, „505 HTTP Version not supported“

7.1.2 Adressierung von Objekten in URIs und URLs

Die Adressierung von Objekten geschieht allgemein in URI: Universal (Uniform) Resource Identifier. Sie haben als Zeichenkette den allgemeinen Aufbau <Schema>:<schemaspezifischer Teil>.

Ein URL: Uniform Resource Locator ist ein spezielles (aber weit verbreitetes) URI-Schema und sieht folgendermaßen aus:

<Schema>://[Benutzer[:Passwort]@]Server[:Port]/[Pfad][?Anfrage][#Fragmentbezeichner]

Mit URLs werden Dokumente im Netz eindeutig identifiziert, um via HTTP darauf zuzugreifen. URLs haben eine feste Syntax, die das Zugriffsprotokoll und den Ort einer Ressource im Netz identifiziert. Sie können relativ sein (d.h. global nicht eindeutig).

Ein URN: Uniform Resource Name ist wiederum ein spezielles URI-Schema, das eine Ressource mittels eines vorhandenen oder frei zu vergebenden Namens identifiziert z.B. urn:isbn oder urn:sha1.

7.2 Statische Webseiten

HTML entstand als plattform- und anzeigenabhängige Beschreibungssprache. Web-Browser holen sich über das beschriebene HTTP-Protokoll Web-Seiten und stellen diese dar. Dabei wird der Text umgebrochen und weitere Bestandteile wie Bilder und Tabelle so eingebettet, dass sich angepasst an den Rechner und dessen Anzeigmöglichkeiten eine gut lesbare Darstellung ergibt. Die Seiten sind in Head und Body aufgeteilt und durch Tags begrenzt. Näheres findet sich in den einschlägigen Veranstaltungen der Medieninformatik und der Webreferenz <http://de.selfhtml.org>. HTML ist XML-basierend und kann weitere XML-Vokabulare wie SVG etc. integrieren. Zu XML kommen wir später im Kapitel. HTML5 übernimmt viele Aufgaben, die zuvor durch Browser-Plugins wie Flash und Silverlight erledigt wurden. Das folgende Bild zeigt die evolutionäre Entwicklung von HTML:

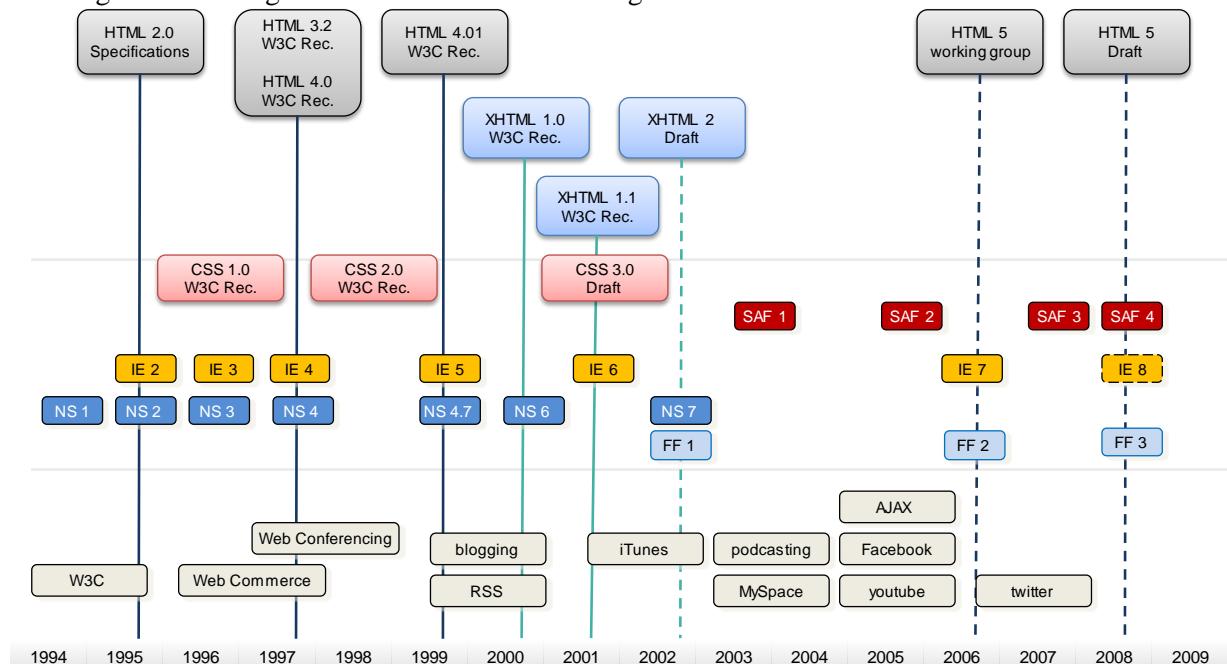


Abbildung 58: Evolution von HTML

HTML5 wird seit einigen Jahren in allen Web-Browsern unterstützt.

7.3 Dynamische Webseiten (server-seitige Programmierung mit Java Servlets, php etc.)

Um Informationen dynamisch in Web-Seiten einzufügen, werden verschiedene Verfahren auf Server-Seite angewendet. Dazu müssen auf der Server-Seite folgende Erweiterungen verfügbar sein:

- Der Web-Server muss Anwendungsprogramme ausführen oder Skriptbefehle (ggf. mit externen Interpretern) interpretieren können.
- Informationen, Berechnungsergebnisse, von weiteren Systemen hinzugefügte Daten wie Börsenkurse, Wetterdaten etc. werden nahtlos in den Seitenrumpf eingefügt, so dass sich beim Web-Browser eine konsistente HTML-Seite ergibt. Dies geschieht, indem die Web-Seite wie bisher in HTML aufgebaut, die dynamischen Informationen aber an den vorgesehenen Stellen einfügt.
- Es muss einen Mechanismus geben, der Übergabeparameter vom Client zum Server überträgt, damit dieser sie dem Anwendungs- oder Skriptprogramm mitteilt.

Folgende Skripttechnologien (in Klammern Dateierweiterung) sind verbreitet, es kommen aber ständig neue hinzu:

- ASP (.asp, .aspx) - Active Server Pages sind eine Entwicklung von Microsoft und werden im IIS (Internet Information Server) durch einen Interpreter ausgeführt.

- JSP (.jsp) - Java Server Pages wurden mit dem gleichen Ziel von Sun eingeführt, aber auf mehreren Web-Servern unterstützt. Mit ihnen werden Java-Programme durch eine Laufzeitumgebung auf dem Server ausgeführt.
- Perl (.pl) ist eine C-ähnliche Skriptsprache, die sich durch zusätzliche Möglichkeiten zur Verarbeitung von Texten auszeichnet.
- PHP (.php3, .php) - Personal Home Page / PHP Hypertext Preprocessor / Perl Helper Pages erfreuen sich aufgrund der Stärken im Umgang mit Datenbanken und der Eignung für schnelle Lösungen zunehmender Beliebtheit.

Das folgende Bild zeigt eine 4-Schichten-Architektur (Java Enterprise Edition als Beispiel), die sich für die Umsetzung dynamischer Web-Anwendungen durchgesetzt hat:

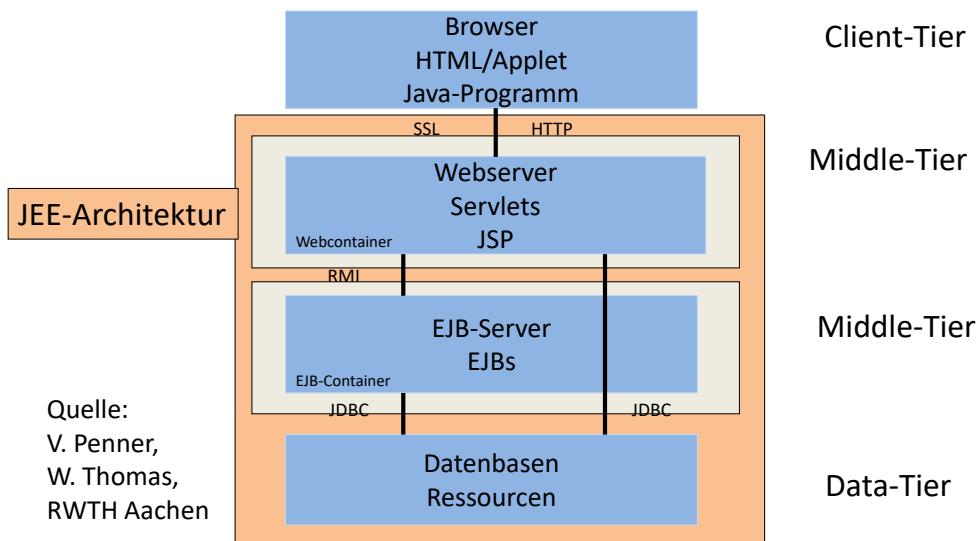


Abbildung 59: 4-Schichten-Architektur für Web-Anwendungen z.B. Java Enterprise Edition

- In der obersten der 4 Schichten (Client-Tier) arbeitet der Web-Browser, der für die Präsentation zuständig ist. Er stellt die HTML-Seiten dar und kann darin client-seitige Programmanteile (z.B. Java-Applets oder JavaScript wie in Kap. 7.4 beschrieben) ausführen
- Der Web-Browser kommuniziert mit dem Web-Server (Mittel-Tier) über HTTP oder (gesichert) über SSL bzw. HTTPS. Im Web-Server geschieht die Aufbereitung der Web-Seiten z.B. in Servlets und JSP.
- Der Zugriff auf die Middle-Tier erfolgt über Java RMI (vergl. Kap. 6). In einem EJB-Server wird die Geschäftslogik z.B. in Enterprise Java Beans abgewickelt. Diese greifen auf die Persistenzschicht (Data Tier) über JDBC zu.

Andere Web-Architekturen sind ähnlich aufgebaut, nutzen aber anstelle des EJB-Servers andere Server-Varianten. Im Fach Komponentenbasierte Software-Entwicklung wird näher auf JEE eingegangen. Die server-seitige Web-Programmierung soll nun anhand von Java-Servlets näher erläutert werden.

7.4 Java Servlets

Eine weit verbreitete Möglichkeit Web-basierender Programmierung verteilter Systeme ist die Nutzung von Java-Servlets, auf der viele weiterführende Java-Technologien basieren. Java Servlets sind Bestandteil der Java Enterprise Edition (JEE). Sie implementieren einen Request/Response-Mechanismus für GET-Requests und können auch für die Verarbeitung von Formularen (POST-Requests) genutzt werden. Aus Java-Sicht geschieht die Kommunikation mit Servlets über die Schnittstelle `javax.servlet.Servlet` mit den Methoden `init()`, `service()` und `destroy()`. Die Abwicklung der Anfragen in `service()` erfolgt im Paket `javax.servlet.http` durch die Methoden: `doGet()`, `doPost()`, `doHead()`, `doDelete()`, ...

Argumente sind:

`HttpServletRequest` und `HttpServletResponse`.

Um Servlets nutzen zu können, muss der Web-Server um einen Servlet-Container erweitert werden. Dies geschieht in Applikationsservern wie Tomcat, Jetty, Resin etc.. Zur Entwicklung von Servlets sind

integrierte Entwicklungsumgebungen, die mit dem Applikationsserver integriert sind, von Vorteil. Im Kontext dieses Skriptes und der Veranstaltung werden Netbeans mit Glassfish bzw. Tomcat eingesetzt. Servlets werden temporär oder permanent eingerichtet (Deployment). Ihr Lebenszyklus sieht folgendermaßen aus:

1. Laden aus Archiv
2. Instantiierung als Objekt
3. Starten des Servlets als Thread oder Prozess / in eigener VM

Wenn das Servlet aktiv ist, behandelt es Anfragen entsprechend folgendem Ablauf:

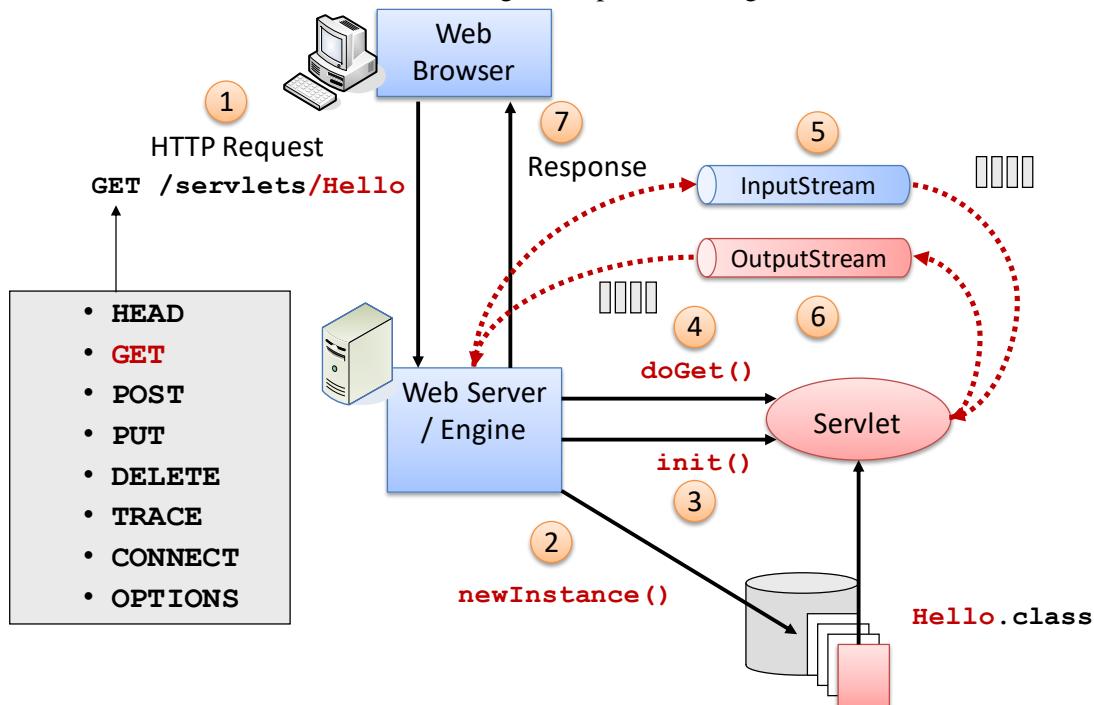


Abbildung 60: Lebenszyklus von Servlets

Die Graphik zeigt die komplette Abarbeitung eines (hier parameterlosen) GET-Requests (Hello):

1. Brower schickt GET-Request an Webserver.
2. Servlet wird aus Java Class-File geladen und ein entsprechendes Objekt angelegt.
3. Methode `init()` wird auf das Servlet angewandt.
4. Request wird durch Abarbeitung von `service()` ausgeführt.
5. Dabei werden die Input-Parameter aus dem Eingabestrom gelesen.
6. Ein Ausgabestrom wird angelegt und mit den Daten der Antwort beschrieben
7. Der Web-Server liefert die Seite über HTTP an den Web-Browser aus.

Voraussetzung ist, dass die Packages `javax.servlet` und `javax.servlet.http` müssen eingebunden sein.

7.4.1 Beispiel RequestHeader - Servlet zur Ausgabe des Response-Headers

Einen Ausschnitt aus einem Beispiel-Servlet zeigt folgendes Listing:

```

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestHeader extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println(name + " = " + value);
        }
    }
}

```

Abbildung 61: Einfaches HTTP-Servlet

In dem Beispiel werden folgende wichtige Konzepte gezeigt:

- Die Ableitungsstruktur, die erfordert, dass Servlets `HTTPServlet` erweitern
- Die Schnittstelle mit der Methode `doGet()` und den zugehörigen Exceptions
- Das Auslesen des Requests (Header-Informationen) und die Aufteilung in einzelne Elemente
- Das Setzen der Response (hier bestehend aus den Namen der Header-Elemente und den jeweiligen Werten)

Bei diesem einfachen Beispiel könnte Antwort z.B. folgendermaßen aussehen:



Abbildung 62: Antwort von RequestHeader auf Anfrage aus Web-Browser

Einige wichtige Eigenschaften von Java-Servlets sind:

- `doPut()`, `doOPTION()`, `doDELETE()` sind generell unüblich
- Die Verarbeitung von `doGet/doPost` kann kombiniert werden.
- `protected`, da von interner `service()`-Methode aufgerufen
- Weitere Auswertemöglichkeiten für übergebene Parameter sind `request.getParameter` für einzelne Parameter bzw. `request.getParameterValues` für mehrere Werte, z. B. bei Mehrfachselektion.

- Weitere Informationen aus Request sind: IP-Adresse, Aussage, ob gesicherte Verbindung, bei Login Username etc.

Die HttpServlet-Response muss enthalten:

- Statuscode z. B. 200=OK oder 404=Seite nicht gefunden
- Inhaltstyp, default ist `text/html`
- Eigentlichen Inhalt, der mittels `PrintWriter` übertragen wird
- Wichtig: Eingaben vor Weiterverarbeitung prüfen, unbekannte Zeichen entfernen, da Eingabeformulare gerne als Einfallstore für eingeschleusten Code verwendet werden.
- Die Verarbeitung von `doGet/doPost` kann kombiniert werden.
- Empfehlung: Servlets selbst nur als Durchgangsstation nutzen und eigentliche Verarbeitung in externen Klassen

7.4.2 Beispiel FormExample - Servlet zum Verarbeiten eines Formulars

Für die Auswertung von Daten aus HTML-Formularen sind besondere Funktionen vorhanden:

- `getInputStream()` ermöglicht es, POST-Daten auszulesen
- `getQueryString()` ermöglicht es, GET-Daten auszulesen

Eine Klasse `HttpUtils` stellt Methoden zum Dekodieren und Enkodieren zur Verfügung.

Das folgende Beispiel zeigt eine durch ein Servlet zu verarbeitende HTML-Seite im Source-Code und im Browser

```
<html>
<head>
<title>Experten Tip</title>
</head>
<body>
<form action="http://localhost:8084/vs\_sample\_web\_apps/FormExample" method="POST">
<h2>Bitte geben Sie Ihren Namen ein:</h2>

<input type="text" name="name">
<h2>Wer wird Fussballweltmeister 2014?</h2>
<input type="radio" name="tip" value="Deutschland">Deutschland
<BR><input type="radio" name="tip" value="Brasilien">Brasilien
<BR><input type="radio" name="tip" value="Frankreich">Frankreich
<h2>Treiben Sie selber Sport?</h2>

<input type="checkbox" name="sport" value="0">Fussball
<BR><input type="checkbox" name="sport" value="1">Nordic Walking
<BR><input type="checkbox" name="sport" value="2">anderer
<BR><BR><input type=submit value="Abgeben"> <input type=reset
value="Verwerfen">
</form>
</body>
</html>
```

Befinden sich die Ressourcen alle im selben **Kontext**, so kann dieser Teil der URI weggelassen werden.

Abbildung 63: HTML-Formular

Mit dem abgebildeten Source-Code wird folgende Formularseite erzeugt:

Bitte geben Sie Ihren Namen ein:
experte

Wer wird Fussballweltmeister 2014?

Deutschland
 Brasilien
 Frankreich

Treiben Sie selber Sport?

Fussball
 Nordic Walking
 anderer

Abgeben Löschen

Abbildung 64: HTML-Code und Browser-Darstellung eines einfachen Formulars

Das folgende Listing zeigt die entscheidenden Ausschnitte des verarbeitenden Servlets:

```
package de.hsoc.vs;
import javax.servlet.*; import javax.servlet.http.*;
import java.io.*;
public class FormExample extends javax.servlet.http.HttpServlet implements
    javax.servlet.Servlet {
    String tips[]={"Deutschland", "Brasilien", "Frankreich"};
    int nrTip[] = new int[tips.length];
    String sports[]={"Fussball", "Nordic Walking", "anderer"};
    int nrSport[] = new int[sports.length];
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/html");
        ServletOutputStream out;
        out = res.getOutputStream();
        String name= req.getParameterValues("name")[0];
        out.println("<html><head><title>Danke, "+name+"!</title></head>");
        out.println("<body>\n<h1>Danke fuer die Mitarbeit, " + name +
                    "</h1>");
        ...
        out.println("</body></html>");
        ...
        // Werte erfassen
        String ft= req.getParameter("tip");
        for (int i=0; i< tips.length;i++) {
            if (ft.equals(tips[i])) {
                nrTip[i]++;
                break;
```

```

        }
    }
    String[] sport=req.getParameterValues("sport");
    if (sport != null)
        for (int i=0; i< sport.length ;i++) {
            try {
                int k=(new Integer(sport[i])).intValue();
                nrSport[k]++;
            }
            catch( NumberFormatException ex) {
                throw new ServletException("uups: " + ex);
            }
        }
    // ....
    // Ausgabe der bisherigen Ergebnisse
    out.println("<h2>Bisherige Tips:</h2>");
    for (int i=0; i< tips.length;i++) {
        out.println("<B>" +tips[i]+ "</B>: " +nrTip[i]+ <BR>");
    }
    out.println("<h2>Statistik zu Sportarten:</h2>");
    for (int i=0; i< sports.length;i++) {
        out.println("<B>" + sports[i] + "</B>: " +
                   nrSport[i] + "<BR>");
    }
}

```

Besonderheiten:

- Das Servlet liest die in das Formular eingegebenen Werte aus dem Request-Objekt.
- Der eingegebene Name wird als Parameter verarbeitet und eine personalisierte „Danke“-Meldung erzeugt.
- Die Zahl der abgegebenen Tipps pro Mannschaft und der Teilnehmer insgesamt als Statistik werden erfasst.

Die erzeugte Antwortseite wird im Browser folgendermaßen angezeigt:



Abbildung 65: Do-Post-Routine und Ausgabe im Browser

7.4.3 Beispiel SessionExample - Servlet zum Verwalten einer Sitzung

Normalerweise erstreckt sich die Verbindung zwischen Web-Browser und Web-Server vom Request bis zur Response. Die kontinuierliche Verbindung mit dem Servlet ist jedoch ebenfalls möglich und wird durch eine Session abgebildet. Die Zuordnung erfolgt auf Serverseite über eine Session-ID, die bei jedem Zugriff übermittelt wird. Diese wird bei jedem Zugriff übermittelt. Grundlage für beide Methoden ist HttpSession.

```
// Die Session wird aus Request-Objekt gewonnen
HttpSession session = request.getSession (true);
// Session-Information lesen über Key (session.errors):
Integer errs = (Integer) session.getAttribute("session.errors");
// Session beenden:
session.invalidate();
```

Die Klasse HttpSession unterstützt u. a. die Methoden getCreationTime() und getLastAccessTime().

Das gezeigte Beispiel wird nun um die Sitzungsverwaltung erweitert. Das HTML-Formular sieht folgendermaßen aus:

```
<html>
<head>
  <title>Session Beispiel</title>
</head>
<body>
  <form action="SessionExample" method="GET">
    <h2>Bitte geben Sie Ihren Namen ein:</h2>
    <input type="text" name="name">
    <br><input type="submit" value="Eingabe">
  </form>
</body>
</html>
```



Abbildung 66: Eingabefeld in Formular

Weil sich die HTML-Seite und das Servlet im gleichen Kontext befinden, kann die Adressierung des Kontextes in der URL weggelassen werden. Ähnlich kann beim Aufrufen von Servlets innerhalb desselben Kontextes vorgegangen werden.

Wir sehen hier ein einfaches HTML-Formular, über das ein User-Name eingegeben wird. Dieser wird durch einen GET-Request an das aufgerufene Servlet übertragen und zur Personalisierung des nachfolgenden Dialoges verwendet. Im Prinzip kann über einen solchen Mechanismus auch eine einfache Authentifizierung vorgenommen werden (in diesem Fall müsste das Formular noch um ein Passwort-Feld erweitert werden), für die Authentifizierung sieht Java jedoch eigene Mechanismen vor.

Die Verarbeitung des Formulars im Servlet erfolgt in der Klasse SessionExample:

```
import javax.servlet.http.HttpSession; ...
public class SessionExample extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        String name=req.getParameterValues("name")[0];
        // Neue Session anlegen

        HttpSession session= req.getSession(true);
        // Objekt ,name' in Session ablegen
        session.setAttribute("name", name);
        res.setContentType("text/html");
        javax.servlet.ServletOutputStream out;
        out = res.getOutputStream();
        out.println("<html><head><title>Abstimmung</title></head>");
        out.println("<body>Wer wird Fussballweltmeister 2010, "
            + name + "?</h1>");
        out.print("<form action=\"");
        out.print(res.encodeURL("SessionEndExample")); // Ermöglicht
```

```
// URL-Rewriting
out.println("\" method=\"GET\"");
out.println("<h2>Abstimmung hier:</h2>" +
    "<input type=\"radio\" name=\"tip\" value=\"Deutschland\">Deutschland" +
    "<BR...><BR><input type=\"submit\" " +
    "value=\"Abgeben\"></form></body></html>");
out.close();
}
}
```

Durch `res.encodeURL` wird URL-Rewriting zugelassen. Bei "`<BR...>
<input type=\"submit\"`" muss man die Auswahl ergänzen. Die Ausführung des Servlets führt zu einer personalisierten Begrüßung



Abbildung 67: Personalisierte Abfrage mit Radio-Buttons

Die Statistik wird nun durch folgende Code erzeugt:

```
public class SessionEndExample extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {
    int nrDeutschland; int nrBrasilien; int nrFrankreich;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        HttpSession session= req.getSession(false); // auf exist. Session
        zugreifen
        res.setContentType("text/html");
        ServletOutputStream out;
        out = res.getOutputStream();
        if (session == null) {
            out.println("<html><head><title>Session Fehler" +
            "</title></head>");
            out.println... out.close(); return;
        }
        String name=(String)session.getAttribute("name");
        String tip= req.getParameterValues("tip")[0];
        if ("Frankreich".equals(tip)) nrFrankreich++;
        else if ...
        out.println("<html><head><title>Abstimmungsergebnisse" +
        "</title></head>");
        out.println("<body>\n<h2>Bisherige Abstimmungsergebnisse, " +
        name + "</h2>");
        out.println("<TABLE BORDER=1><TR><TH>Mannschaft</TH>" +
            "<TH># Stimmen</TH></TR>");
        out.println("<TR><TD>Deutschland</TD><TD>" +
            nrDeutschland+"</TD></TR>"); ...
        out.println("</TABLE></body></html>"); out.close();
        session.invalidate();
    }
}
```

}

Dies führt zu folgender Darstellung der bisherigen Statistik:

A screenshot of a web browser window titled "Abstimmungsergebniss". The URL is "localhost:8084/vs_web_examples/SessionEndExample?tip=Deutschland". The main content is a table titled "Bisherige Abstimmungsergebnisse, experte!" with the following data:

Mannschaft	# Stimmen
Deutschland	1
Frankreich	0
Brasilien	0

Abbildung 68: Tabelle mit Statistik

Ist die Session nicht mehr gültig wird folgende Meldung angezeigt:



Abbildung 69: Meldung bei ungültiger Session

Im aufgerufenen Servlet wird eine **Session** aufgebaut, in die der zuvor eingegebene Name nach dem Auslesen des entsprechenden Parameters im HTTP-Request als Attribut gespeichert wird.

Bei URL-Rewriting wird den URLs einer Session automatisch ein Parameter wie

`...SessionEndExample; jsessionid=DFE3A0E068FFFA14566B38CEB5C5A3E7...`

hinzugefügt.

7.4.4 Beispiel CookieExample - Servlet zur dauerhaften Speicherung des Sitzungskontextes

Möchte man den Sitzungskontext über mehrere Sitzungen hinweg verwalten, nutzt man Cookies, die im Web-Browser dauerhaft gespeichert werden können. Voraussetzung hierfür ist, dass die zu speichernden Daten serialisierbar sind. Für die Nutzung von Cookies dient `javax.Servlet.http.Cookie`. Wichtige Parameter sind Name, Wert (lange Binärzahl) und Ablaufdatum. Wichtige Methoden sind: `set/getName`, `set/getMaxAge`, `set/getValue()`. Vor Nutzung dieser Form der Sitzungsverwaltung muss geprüft werden, ob Cookies beim Client erlaubt sind. Dem Hinzufügen des Cookies dient die `addCookie()`-Methode von `HttpServletResponse`. Beachten Sie, dass keine vertraulichen Informationen in Cookies abgelegt werden dürfen, da Cookies beim Client im Klartext gespeichert werden. Cookie immer an spezielle URL gebunden.

Im Beispiel unseres Tippspiels wollen wir über die laufende Sitzung hinaus eine Statistik über die ausgewählten Länder führen. Im Source-Code der `doGet()`-Methode müssen dazu an drei Stellen Ergänzungen vorgenommen werden:

```

public class SessionEndExample extends javax.servlet.http.HttpServlet
    implements javax.servlet.Servlet {
    int nrDeutschland; int nrBrasilien; int nrFrankreich;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        HttpSession session= req.getSession(false);
        res.setContentType("text/html");
        ServletOutputStream out;
        out = res.getOutputStream();
        if (session == null) {
            out.println("<html><head><title>Session Fehler" + "</title></head>");
            out.println... // Hier ergänzen
            out.close(); return;
        }
        String name=(String)session.getAttribute("name");
        String tip= req.getParameterValues("tip")[0];
        if ("Frankreich".equals(tip)) nrFrankreich++;
        else if ... // Hier ergänzen
        out.println("<html><head><title>Abstimmungsergebnisse" +
        "</title></head>");
        out.println("<body>\n<h2>Bisherige Abstimmungsergebnisse, " + name +
        "</h2>");
        out.println("<TABLE BORDER=1><TR><TH>Mannschaft</TH>" +
        "<TH># Stimmen</TH></TR>");
        out.println("<TR><TD>Deutschland</TD><TD>" +
        nrDeutschland+"</TD></TR>"); ... // Hier ergänzen
        out.println("</TABLE></body></html>"); out.close();
        session.invalidate();
    }
}

```

Das folgende Bild zeigt den Ablauf einer Sitzung mit Cookie-Verwaltung:

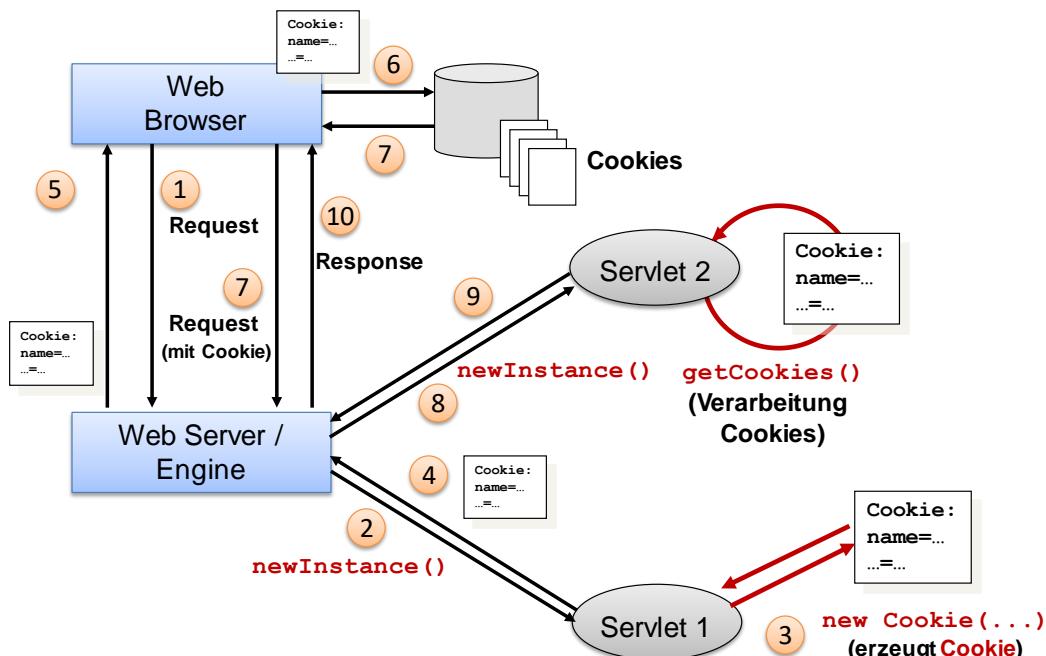


Abbildung 70: Ablauf einer Cookie-basierenden Sitzungsverwaltung

1. Eine Anfrage kommt vom Web-Browser.
2. Ein Servlet 1 wird mit `newInstance()` erzeugt.
3. Ein dazu gehöriger Cookie wird erzeugt.
4. Servlet 1 übergibt mit der Antwort auf `newInstance()` auch den Cookie.
5. Der Cookie wird mit der Antwort an den Web-Browser zurück gegeben
6. Er wird im Web-Browser für spätere Sitzungen aufbewahrt.

7. Beim erneuten Aufruf der Seite wird der Cookie aus dem Cookie-Speicher des Web-Browser gelesen.
8. Er wird mit dem Aufruf mit gegeben und Servlet 2 kann damit die dauerhaft gespeicherten Daten aus dem Cookie verarbeiten (9 und 10).

7.4.5 Servlets und Datenbanken

In Servlets besteht die Möglichkeit, Datenbanken wie in einer typischen Middle-Tier-Architektur als Persistenzschicht über Java Database Connectivity (JDBC) anzubinden. Jeder namhafte Datenbankhersteller bietet zu seiner Datenbank JDBC-Treiber an (ähnlich den bereits zuvor eingeführten ODBC-Treibern von Microsoft). Manche (langsame) Ansätze z.B. für MS Access realisieren den JDBC-Zugriff über eine JDBC/ODBC-Bridge. Die Zugriffe erfolgen über `init()`, `doGet()` und `destroy()`, wie im Fach Datenbanken näher ausgeführt wird.

Je nach Datenbankhersteller kann der Zugriff direkt oder über eine JDBC/ODBC-Bridge erfolgen:

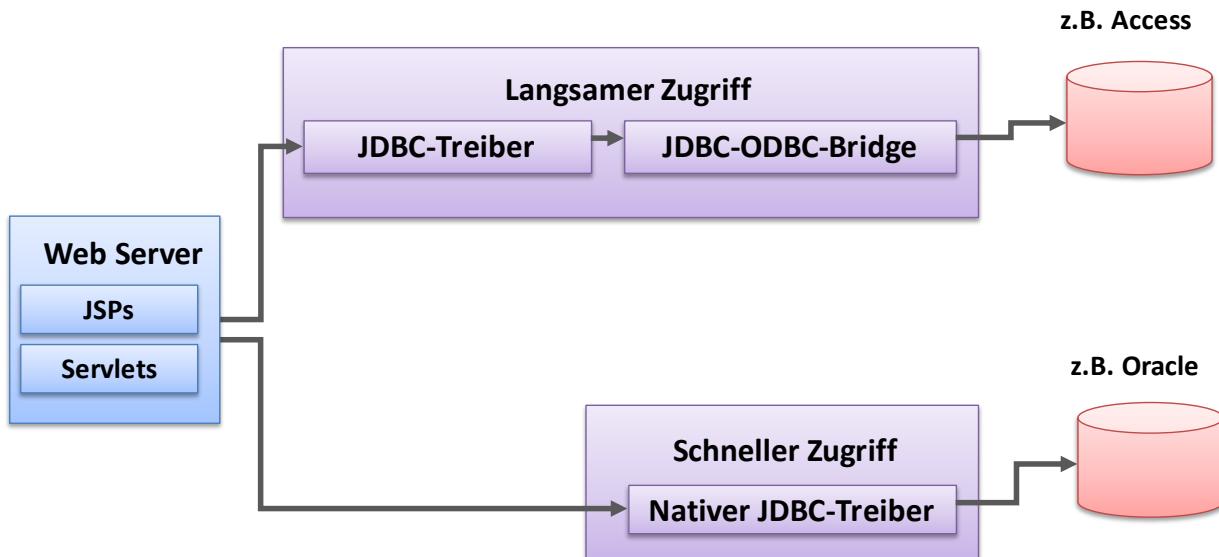


Abbildung 71: Varianten des Datenbankzugriffs für Servlets

Man unterscheidet native Treiber und sog. Bridges. Letztere kapseln eine andere Datenbank-Middleware (z.B. ODBC für Microsoft) und sind aufgrund dieser Eigenschaft in der Regel langsamer als native Treiber.

Plattform- und datenbankunabhängig erfolgt der Zugriff über Standard SQL ANSI 92 Entry Level. JDBC 2.x/3.0 ist eine Low-Level-Schnittstelle, die im Gegensatz zu ODBC (Open Database Connectivity, Microsoft) objektorientiert sind. Sie basiert auf dem X/OPEN SQL-CLI (Call-Level-Interface), ebenso wie ODBC und ist im Paket `javax.sql / java.sql` realisiert.

Das Paket `javax.sql` stellt Erweiterungen für die Serverseite zur Verfügung, z.B. Mechanismen zur Verwaltung von Connection Pools und von Transaktionen.

Neben JDBC gibt es noch weitere Möglichkeiten, Datenbanken von Java aus anzusprechen. Mit den Java Data Objects (JDO) bietet sich eine Möglichkeit der persistenten Speicherung von Objekten. Während JDBC die Sichtweise auf eine beliebige relationale Datenbank definiert, ist bei JDO der Ausgangspunkt eine objektorientierte Sicht. Anwendungsentwickler definieren Business- oder Datenobjekte, welche über eine konkrete JDO-Implementierung persistent gemacht werden. Mittels eines Objekt/Relational-Mapping kann aber auch JDBC zur Speicherung der Daten herangezogen werden. Ein weiterer Ansatz ist die Java Persistence API (JPA).

Das folgende Listing zeigt den Servlet-Zugriff auf eine Datenbank:

```
public class db_example extends HttpServlet {
    private Connection con;
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        // Ausgabe setzen
        res.setContentType("text/html");
        ServletOutputStream out; out = res.getOutputStream();
```

```

out.println("<html><head><title>DB Beispiel</title></head>\n");
out.println("<body>\n");
// Statement erzeugen und abschicken
String query = "SELECT Vorname, Nachname, EmailName " +
    " FROM Kursteilnehmer WHERE KursteilnehmerID > 0";
try {
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
// Auswertung des Resultats
    while (rs.next()) {
        String v = rs.getString("Vorname"); ... // same Nachname, EmailName
        out.println(v + " " + n + ", " + e + "<br>");
    }
    stmt.close(); // Schliessen des Statements
}
catch (SQLException se) {
    out.println ("<B>SQLException doing query: " + se + "</B>");
    return;
}
out.println("</body></html>");
}
}

```

Mit der definierten Abfrage (`query`) wird ein Statement erzeugt (`con.createStatement`) und ausgeführt (`stmt.executeQuery`). Das Resultat wird ausgewertet und das Statement wieder geschlossen (`stmt.close`).

7.4.6 Sicherheit bei Web-Anwendungen

Die Sicherheit ist bei Web-Anwendungen ein sehr wichtiges Thema. Neben der bereits in Kapitel 6 betrachteten Nutzung von Security Policies in Java sind die Authentifizierung, Autorisierung und Abrechnungsmöglichkeiten (englisch: Authentication, Authorization, Accounting (AAA) weitergehende Aspekte, die mit folgenden Ansätzen realisiert werden können:

- NTLM
ist eine Microsoft-spezifische proprietäre Lösung
- Basic Authentication
funktioniert nach folgendem Prinzip:
 1. Der Server beantwortet einen Client-Request mit WWW-Authenticate: Basic realm="..." im Response-Header
 2. Der Browser blendet Formular zur Authentifizierung ein
 3. Der Client schickt ursprünglichen Request ergänzt um Header: Authorization: Basic d2lraTpwZWRpYQ== (base64-kodierte Kombination User-Name und Passwort = Klartext!)
Von dieser Methode wird wegen ihrer Unsicherheit abgeraten.
- Digest-based Authentication:
Die Authentifizierung erfolgt auf Basis eines Challenge-Response-Protokolls, bei dem die Passwörter werden nicht im Klartext verschickt werden.

Bei der Digest-based Authentication sendet der Client einen normalen HTTP-Request. Der Server blendet anstelle der gewünschten Seite ein vorgesetztes Login-Formular zur Authentifizierung ein. Der Client versendet die MD5-Check-Summe der Authentifikationsdaten in Kombination mit einem Zufalls Wert („nonce“, number used once) als Hash-Wert. Der Zufalls Wert kann bei jeder Anfrage variiert werden. Man in the Middle Attacken sind aber möglich.

Das folgende Bild zeigt das Prinzip der Digest based Authentication:

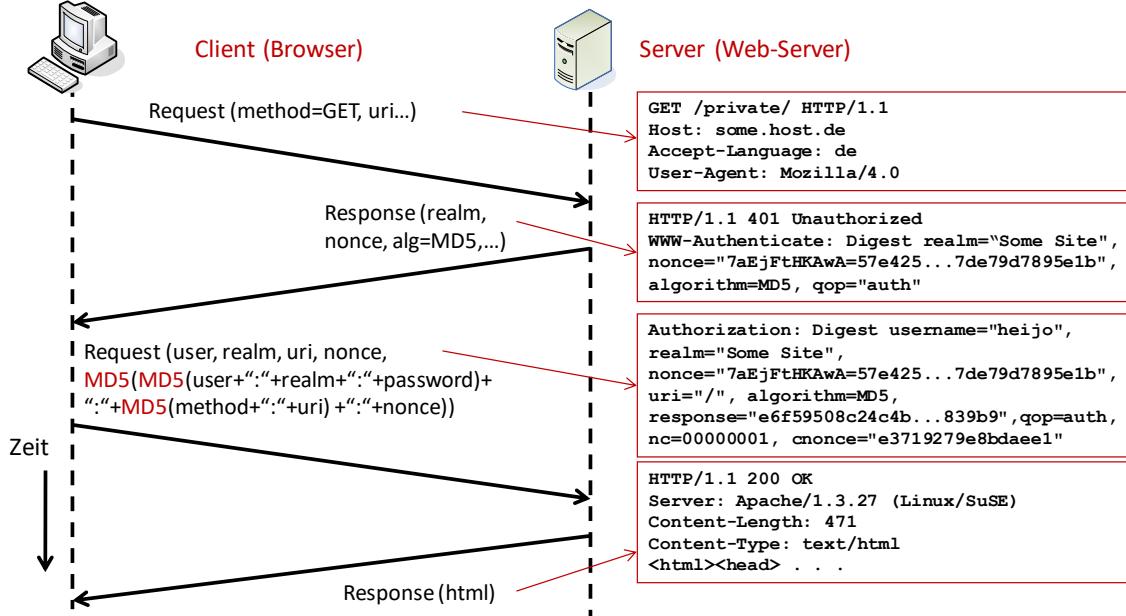


Abbildung 72: Digest based Authentication

Die Verschlüsselung der Verbindung mittels https erfolgt durch Abstimmung des 3DES-Schlüssels mittels einer Public-Key-Infrastruktur (PKI) entsprechend folgendem Bild:

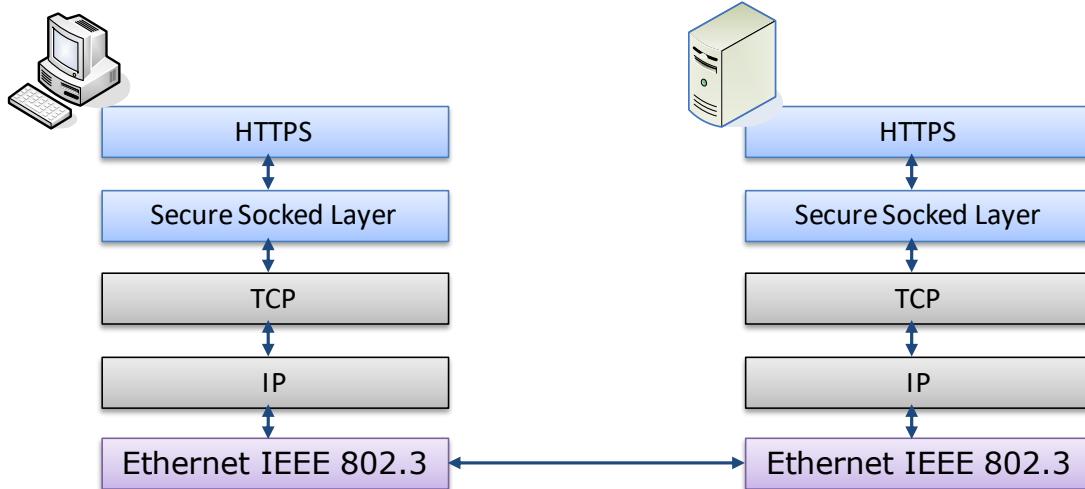


Abbildung 73: HTTPS-Verschlüsselung mittels Secure Socket Layer (SSL)

7.4.7 Server-seitige Web-Applikationen mit Java Server Pages (JSP)

Die gezeigte Servlet-Programmierung ist sehr verbreitet für Anwendungen mit einem höheren Anteil an Geschäftslogik, Interfaces zu weiteren Servern, Datenbanken etc. Manchmal möchte man aber nur eine Web-Seite mit wenigen Daten aus einer Java-Applikation anreichern. Auch für diesen Fall bietet Java die Java Server Pages (JSP). JSP ist eine Technologie, die es einem Webseiten-Entwickler erlaubt, einfache dynamische Webserver-Zugriffe selbst zu realisieren und in die Webseiten einzubetten. Aus den entsprechenden JSP-Konstrukten werden beim Laden der Seite dynamisch Servlets generiert, kompiliert und in den Container geladen.

Vereinfacht gesprochen, werden bei JSP in eine HTML-Seite zeilenweise kurze Java-Programme eingebaut (eingeleitet durch `<%@ page language="java" import=%>` wie auf der nächsten Seite an einem Beispiel gezeigt. Hier überwiegt also der HTML-Anteil gegenüber dem Java-Anteil.

Technisch werden JSPs durch ein spezielles Servlet (ausgeführt in einer Servlet-Engine) realisiert. Der Lebenszyklus geht davon aus, dass die JSP nur ein einziges Mal, spätestens bei der ersten Anforderung kompiliert wird. Es ist natürlich möglich und sinnvoll, eine Präkompilierung durchzuführen.

Aufgrund der dynamischen Kompilation kann es beim Arbeiten mit einem Web-Container zu Problemen kommen, sofern die Java-Umgebung nicht richtig konfiguriert wurde. Bei der Verwendung des Tomcat ist es beispielsweise wichtig, ein JDK (Java Development Kit) und nicht ein Runtime-Environment (JRE) als Java-Umgebung einzustellen, da ansonsten kein Java-Compiler zur Kompilation der aus den JSPs generierten Servlets zur Verfügung steht. Das generierte Servlet erweitert die `HttpJspBase` Klasse und enthält die Methoden: `jspInit()`, `jspDestroy()` und `_jspService()`. Prinzipiell eignet sich der JSP-Code nicht zur Verarbeitung von Anwendungslogik. Aus diesem Grund kann das generierte (wie auch normale Servlets) auf andere Java-Komponenten, Java Beans, etwa für den Datenbankzugriff (Enterprise Information Systems – EIS) zugreifen. Java Beans werden im Fach Komponentenbasierende SW-Entwicklung im fünften Semester des Bachelor näher behandelt. Das folgende Beispiel zeigt den HTML-Code einer JSP-Anwendung, bei der die Quadrate der Zahlen 1 bis 10 in Java berechnet und ausgegeben werden:

```
<html>
<head>
    <title> Demo fuer eine JSP-Seite </title>
</head>
<body>
<H1> Berechnung von Quadraten mit JSP</H1>
<OL>
<%-- Definition von globalen Informationen fuer die Seite --%>
<%@ page language="java" import="java.util.Date" %>
<%-- Deklaration seitenweiten Variablen --%>
<%! int i = 0; int z = 0; private int accessCount = 0; %>
<%-- Scriptlet - Java code --%>
<% for (int i = 1; i <= 10; i++) {
    z=i*i;
%>
<%-- Ergebnis via z.toString() an die HTML-Seite uebergeben --%>
<LI><%= z %></LI>
<%}>
</OL>
Diese Seite wurde am <%= new Date() %><br> zum
<Font color="#F00000"><%= ++accessCount %></Font>. mal besucht.
</body>
</html>
```

Wichtig ist, verschiedene Sichtbarkeitsbereiche (Scopes) zu unterscheiden, wie folgendes Bild zeigt:

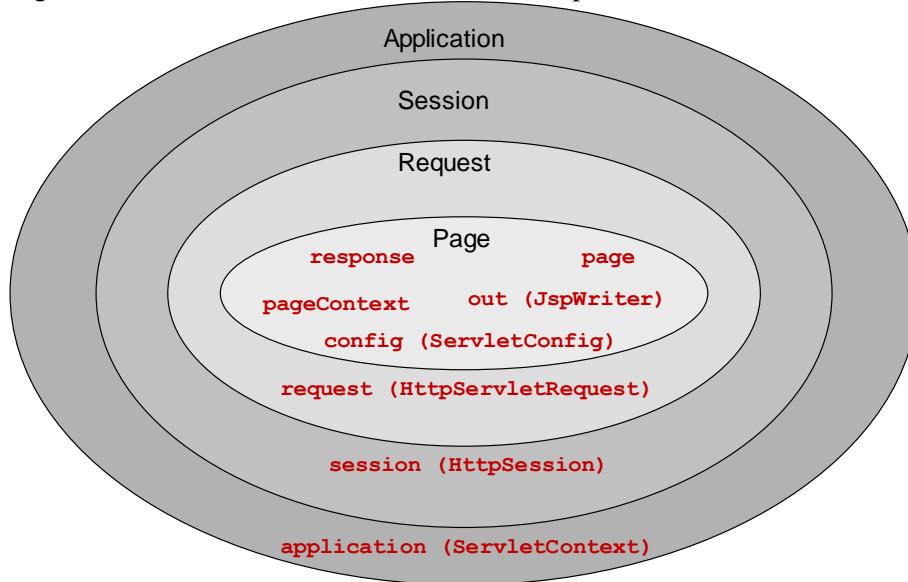


Abbildung 74: JSP: Implizite Objekte und Scoping

Die Session-Verwaltung kann ähnlich wie bei Servlets gehandhabt werden (Kennzeichnung mit eindeutiger ID und Abspeicherung als Cookie sowie Auswertung des HttpSession-Objekts. Per Default nehmen alle JSPs an Sessions teil. Dies kann man abschalten mit:

```
<%@ page session="false" %>
```

Es können auch Java Beans als SW-Komponenten integriert werden. Beispiel:

```
<% SesObject obj = new SesObject(); session.putValue ("Obj", obj) %>
<% SesObjekt obj = (SesObject) session.getValue ("Obj") %>
```

Die Steuerung der Lebensdauer für den Session Heap im Servlet-Container erfolgt über setMaxInvalidationInterval().

Den Lebenszyklus einer JSP kann man in folgendem Bild erkennen:

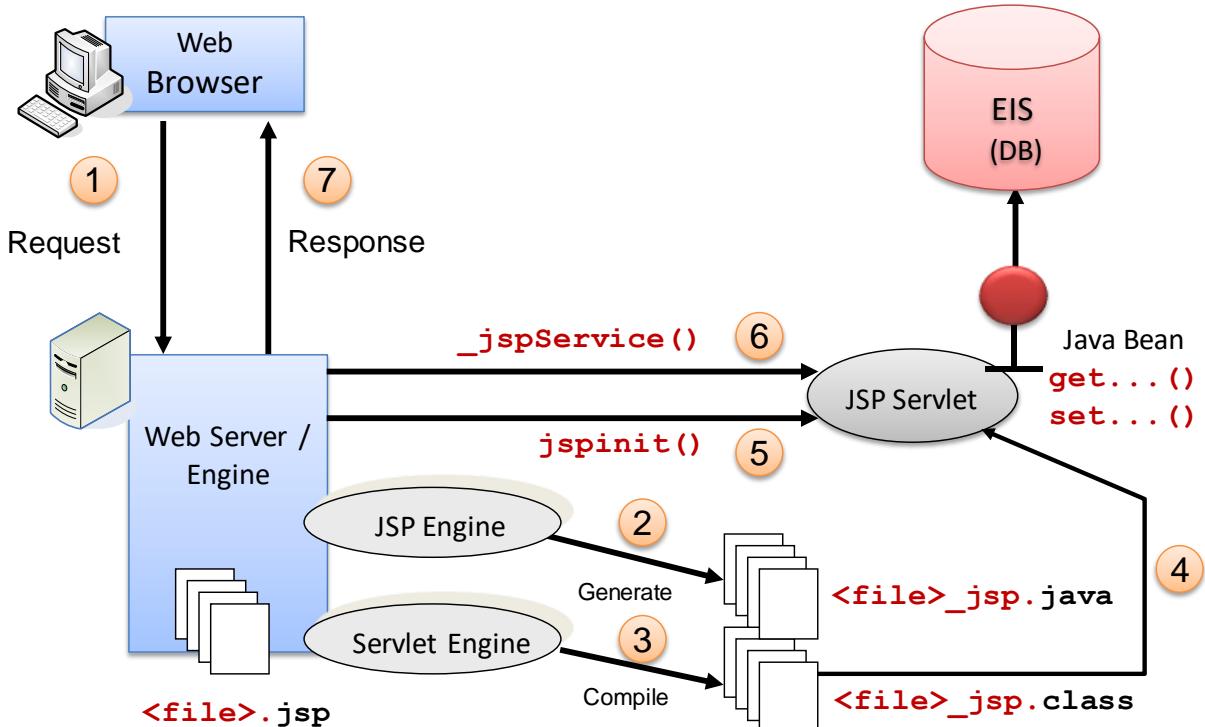


Abbildung 75: Lebenszyklus einer Java Server Page (JSP)

1. Ein Request trifft vom Web-Browser ein
2. die JSP-Engine des Webservers erzeugt ein entsprechendes Servlet, sofern dieses noch nicht vorhanden ist oder falls der JSP-Code zwischenzeitlich verändert wurde
3. Das generierte Servlet wird dynamisch (d.h. zur Laufzeit) übersetzt
4. Das Servlet wird von der Servlet-Engine geladen.
5. Die Initialisierung wird ausgeführt.
6. Ein Service-Aufruf wird verarbeitet und dabei ggf. auf eine Datenbank zurückgegriffen.
7. In die HTML-Seite mit der Response werden die Ergebnisse des Java-Service-Aufrufs eingebaut.

Beispielhaften Source-Code für eine einfache Additionsoperation als Funktion, die durch die Bean bereitgestellt wird, zeigt folgendes Listing:

```
<html>
<head>
<title>
    Zugriff auf Bean-Objekt
</title>
</head>
<jsp:useBean id="calc" class="de.hsos.vs.web.beans.CalcBean"
    scope="session"/>
<body><H1>Zugriff auf Bean-Objekt</H1>
<jsp:setProperty name="calc" property="*"/>
<FORM METHOD="POST" action=" CalculatorWithBean.jsp">
    <INPUT TYPE="text" size="2" name="op1">+
```

```

<INPUT TYPE="text" size="2" name="op2">
<INPUT TYPE="submit" VALUE="Eingabe">
</FORM>
<% if (request.getMethod().equals("POST")) {
    int z = calc.getSumme();
    out.println("<hr>Das Ergebnis: " +
        calc.getOp1() + " + " + calc.getOp2() + " = <b>" + z + "</b>");%
}
%>
</body>
</html>

```

Die Operanden werden als Attribute in der Bean gesetzt und anschließend wird die Operation auf der Bean aufgerufen. Die zugehörige CalcBean sieht folgendermaßen aus:

```

package de.hsoc.vs.web.beans;
public class CalcBean {
    protected int op1, op2;
    public int getOp1() { return this.op1; }
    public int getOp2() { return this.op2; }
    public void setOp1(int v1) {
        this.op1 = v1;
    }
    public void setOp2(int v2) {
        this.op2 = v2;
    }
    public int getSumme() {
        return op1 + op2;
    }
}

```

Die kompilierte Bean muss in dem WEB-INF/classes-Verzeichnis gespeichert werden, in deren übergeordnetem Verzeichnis die JSP-Datei gespeichert ist.

Auch wenn die Web-Programmierung ausführlich am Beispiel der Programmiersprache Java erläutert wurde, gibt es im Microsoft-Bereich vergleichbare Web-Technologien in .net.

Java Servlets sind vergleichbar zu ISAPI.

JDBC ist vergleichbar zu ADO.net.

JSP ist vergleichbar zu ASP.net

7.4.8 PHP

Eine weitere sehr beliebte Möglichkeit der web-basierten Programmierung im Server ist php. Die Entwicklung von PHP wurde 1994 von Rasmus Lerdorf begonnen. Die Abkürzung PHP wird sowohl als Personal Home Page als auch als PHP Hypertext Preprocessor ausgeschrieben. PHP-Skripte werden in HTML eingebettet. Ihre Interpretation erfolgt durch einen Interpreter, der in den meisten Web-Servern eingebaut ist. Beispiele sind libapache2-mod-php5 im Apache2 Web-Server. Es erfolgt keine Kompilierung, die Performance ist eher schwach, kann aber durch verschiedene Maßnahmen gesteigert werden.

Die Syntax von PHP ist an C/Java/Perl angelehnt, allerdings gibt es einige Vereinfachungen, die teilweise als problematisch betrachtet werden:

- Groß-/Kleinschreibung wird nur bei Variablenamen und Zeichenketten unterschieden
- Bei Funktionsnamen gibt es keine Groß-/Kleinschreibung.
- Die Speicherverwaltung erfolgt automatisch.
- Variablen werden durch erstmalige Benutzung deklariert und müssen nicht typisiert werden.

PHP unterstützt einfachen Zugriff auf Internet-Ressourcen, Datenbanken (SQL) und Systemfunktionen, wie folgender Code-Auszug zeigt:

```

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Einfache PHP Seite</title>
    </head>
    <body>

```

```
<?php  
$datum_uhrzeit=date("d.m.Y, H:i:s", time());  
echo "Heute ist der " . $datum_uhrzeit . " Uhr";  
?  
</body>  
</html>
```

Die Nutzung eines Formulars ist ebenfalls einfach möglich:

```
<html>  
    <head>  
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
        <title>Auslesen von Formularen</title>  
    </head>  
    <body>  
        <h2>Eingabeformular:</h2>  
        <form action="form_processing.php" method="post">  
            Vorname: <input type="text" name="firstname"><br>  
            Nachname: <input type="text" name="lastname"><br><br>  
            Alter: <input type="text" name="age"><br><br>  
            <input type="radio" name="gender" value="male">männlich<br>  
            <input type="radio" name="gender" value="female">weiblich<br><br>  
            <input type="submit" value="Absenden"> <input type="reset">  
        </form>  
    </body>  
</html>
```

Die Verarbeitung des Formulars erfolgt in `form_processing.php`:

```
<html>  
    <head>  
        <meta content="text/html; charset=UTF-8" http-equiv="content-type">  
        <title>Verarbeitung von Formularen</title>  
    </head>  
    <body>  
        <h1>Registration erfolgreich</h1>  
        Hallo  
        <?php  
        if ( $_POST['age'] < 16 ) {  
            echo $_POST['firstname'];  
        } elseif ( $_POST['gender'] == "male" ) {  
            echo " Herr " . $_POST['lastname'];  
        } else {  
            echo " Frau " . $_POST['lastname'];  
        }  
        ?,  
        sie sind nur registriert.  
    </body>  
</html>
```

PHP hat folgende Eigenschaften und Probleme:

- Implizit deklarierte Variablen, d.h. bei der Entwicklung gibt es keinen Deklarationszwang und keine Compiler / Interpreter-Prüfung
- Dynamische Typumwandlung
- Mangelnde Thread-Sicherheit
- Komplexe Fehlersuche

Die Sprache erfordert also sehr diszipliniertes Arbeiten, hat aber trotzdem große Bedeutung bei der Web-Programmierung erlangt. Weitere Informationen finden sich unter <http://de.php.net/manual/de/>.

7.4.9 Serverseitige Web-Programmierung und das MVC-Muster

Alle gezeigten Ansätze haben das Problem, dass sie ohne Erweiterungen die Präsentation (Benutzerschnittstelle) mit der Geschäftslogik vermischen. Abhilfe schaffen Weiterentwicklungen wie Template-Engines oder Frameworks, die es durch Erweiterungen ermöglichen, dem Model-View-Controller Muster (MVC Pattern) zu folgen.

7.5 Aktive Web-Seiten (*clientseitige Programmierung mit JavaScript etc.*)

Web-Seiten können Programme als Inhalte enthalten. Dadurch werden die Web-Seiten aktiv, d.h. ein auf der Client-Seite laufendes Programm kommuniziert mit dem Nutzer, ohne dass der Server benötigt wird.

7.5.1 JavaScript

JavaScript-Source-Code wird direkt im Klartext in die HTML-Web-Seite eingebettet und dann von einem Interpreter im Browser interpretiert. Anwendung findet JavaScript (standardisiert als ECMA-Script) z. B. bei Eingabemasken, Auswahldialogen, Wertebereichsprüfungen in Formularen etc.. Beliebt ist auch das Laden von Daten im Hintergrund (z.B. Suchbegriffe im Browser wie bei Google).

JavaScript ist objektorientiert, klassenlos und dynamisch typisiert. Aktuell ist die Version 1.9. Trotz des Namenspräfixes orientiert sich JavaScript eher an C als an Java. In allen modernen Browsern (auch in Smartphones) ist ein JavaScript-Interpreter eingebaut, der in einer Sandbox (also mit stark limitiertem Zugang zur Außenwelt) läuft.

Der prinzipielle Ablauf ist in folgendem Bild zu sehen:

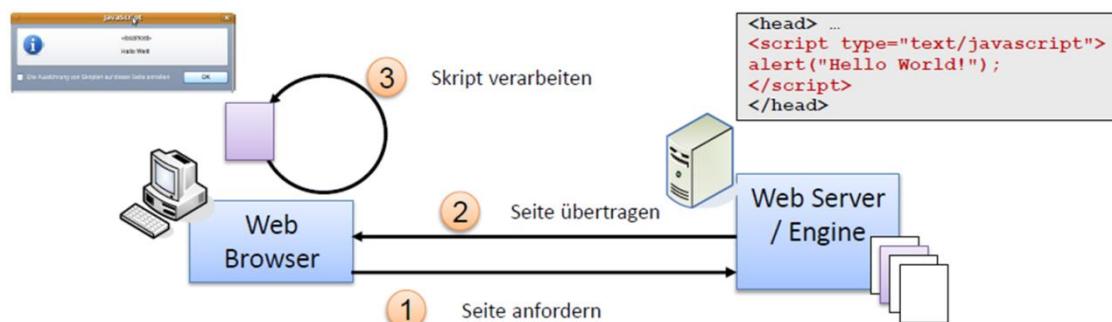


Abbildung 76: Anzeige eines Java-Applets mit InputArea im Web-Browser

1. Der Web-Browser fordert eine Seite mit eingebettetem JavaScript an.
2. Die Seite wird vom Web-Server zum Web-Browser übertragen.
3. Der HTML-Anteil wird angereichert durch Ausgaben aus den JavaScript-Abschnitten dargestellt.

Einige wichtige Spracheigenschaften sind:

- Dynamische Typisierung: Bsp.: `v = new String("shalali")`, Der Datentyp kann mit `typeof v` ermittelt werden.
- Funktionen: Bsp.:


```
function a (Param1, Parameter2, ...) { Anweisungen; return ausdruck; }
```
- Methoden: dynamische Zuweisung an Objekte Bsp.:


```
objekt.methodX = function (x) { ... return ...; };
```
- Fehlerbehandlung: `try {...} catch (...) {};`
- Kontrollstrukturen: `if(...){...} else..., while(...){...}, for(...){...}` analog zu C / Java
- Zusätzlich noch: `for(var...in...){} und for each(var...in...){}{...}`

Die Objektorientierung erfolgt in JavaScript nicht mittels Klassen, sondern mit Prototypen:

```
var Car = { speed: 100 };
var RaceCar = function () {
    this.speed = 320;};
RaceCar.prototype = Car;
// Anwendung:
var porsche911 = new RaceCar();
alert (porsche911.speed); /* 320 */
```

```
delete porsche911.speed;
alert (porsche911.speed); /* 100 */
```

Objekte, die fest in JavaScript eingebaut sind, werden auch implizite Objekte genannt. Dazu zählen `String`, `Math`, `Date`, usw.. Implizite Objekte gibt es auch zu Eigenschaften, die innerhalb eines HTML Dokumentes zur Verfügung stehen z.B. `document`, `window`, `location`, `history` usw..

Implizite Objekte haben Eigenschaften und Methoden, die folgendermaßen genutzt werden können:

```
<html>
<head>
    <title>Datum</title>
    <script type="text/javascript">
        var Jetzt = new Date();
        var Tag = Jetzt.getDate();
        var Monat = Jetzt.getMonth() + 1;
        var Jahr = Jetzt.getYear();
        var Stunden = Jetzt.getHours();
        var Minuten = Jetzt.getMinutes();
        var NachVoll = ((Minuten < 10) ? ":0" : ":");
        document.write("<h2>Guten Tag!</h2><b>Heute ist der "
            + Tag + "." + Monat + "." + Jahr + ". Es ist jetzt "
            + Stunden + NachVoll + Minuten + " Uhr.</b>");
    </script>
</head>
<body>
</body>
</html>
```

In JavaScript kann man gut auf die einzelnen Elemente einer HTML-Seite zugreifen:

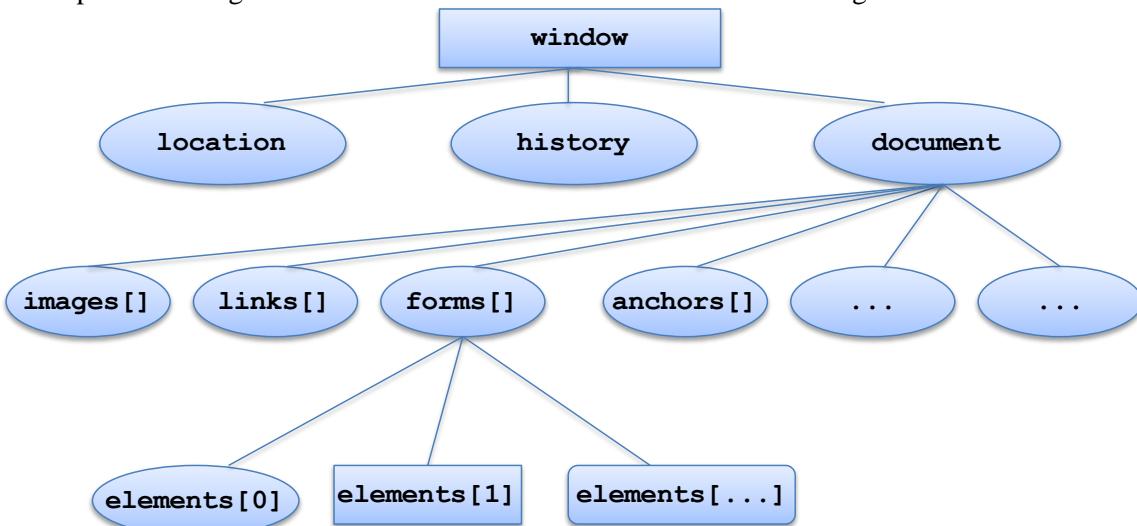


Abbildung 77: Zugriff auf HTML-Dokument

Ein Beispiel ist in folgendem Listing zu sehen:

```
<html>
<head>
    <title>Auslesen von Formularen</title>
    <script type="text/javascript">
        function chkFormular() {
            var res = true; var a = document.form1;
            if (a.firstname.value == "") { res = false; }
            if (a.lastname.value == "") { res = false; }
            if (a.age.value == "") { res = false; }
            if (a.gender[0].checked == false &&
                a.gender[1].checked == false) { res = false; }
            if (res == false) {
                alert ("Bitte füllen Sie Formular vollständig aus!")
                return res;
            }
        }
    </script>
</head>
<body> <h2>Eingabeformular:</h2>
```

```
<form name="form1" action="form_processing.php"
      method="post" onsubmit="return chkFormular()"> ...
</body>
</html>
```

EventHandler dienen zur Verarbeitung von Benutzereingaben:

```
... <form>
<input type="button" value="Abschicken"
       onClick="var val=confirm("Freuen Sie sich!");
       if (val == true) alert("Schönes Lächeln.");
       else alert("Gut geärgert.");>
</form> ...
```

Folgende Event Handler stehen bei den in Klammern angegebenen Eingabe-Elementen zur Verfügung:

- onAbort (bei Abbruch: image)
- onBlur (beim Verlassen: select, text, textarea, window)
- onChange (bei erfolgter Änderung: select, text, textarea)
- onClick (beim Anklicken: button, checkbox, ...)
- onError (im Fehlerfall: image, window)
- onFocus (beim Aktivieren: select, text, textarea, window)
- onKeyDown (bei gedrückter Taste)
- onKeyPress (bei gedrückt gehaltener Taste)
- onLoad (beim Laden: image, window)
- onMouseDown (bei gedrückter Maustaste) ...

JavaScript verringert den notwendigen Datenverkehr zwischen Client und Server. und ist damit Grundlage für AJAX (Asynchronous JavaScript and XML wie in 7.5.5. näher beschrieben). Dabei wird eine asynchrone Client-/Server-Kommunikation ermöglicht, die zu Web-Applikationen führt, die sich wie Desktop-Applikationen verhalten. Da dieser Ansatz auch auf Web-Browsern moderner Smartphones nutzbar ist, werden je nach Anforderung nativ programmierte mobile Applikationen und WebApps eingesetzt. Ein Beispiel ist die erste Version der OsnaApp (der Stadt Osnabrück), die an der Hochschule Osnabrück von Fünftsemestern der Informatik im Software-Engineering-Projekt entwickelt wurde.

7.5.2 Java-Applets

Mit Java-Applets gibt es in Java seit langem eine Möglichkeit, Java-Programme in Web-Browsern auszuführen. Zu Testzwecken (ohne Web-Browser) gibt es im SDK einen Applet-Viewer. Applets werden mit einem Applet-Tag in die HTML-Seite eingebunden. Auf dem Web-Server werden sie im Byte-Code-Format (.class) aufbewahrt und bei Anforderung zum Client übertragen. Im Web-Browser gibt es einen Interpreter, der den Byte-Code des Programms innerhalb einer eigenen JVM ausführt. Die JVM in Web-Browsern zeigen teilweise anderes Verhalten als die Standard-JVM von Sun, daher ist jeder Browser einzeln zu prüfen und Sun hat sich entschlossen, für alle wichtigen Browser eigene Plugins zur Verfügung zu stellen.

Als Beispiel Calc soll ein Applet zur Anzeige des Signalverlaufs einer mathematischen Funktion dienen. Es wird in folgender Web-Seite CalcApplet.html aufgerufen:

```
<html>
  <head>
    <title>Java-Applet mit einfachem Diagramm</title>
  </head>
  <body>
    <h1>Java-Applet mit einfachem Diagramm</h1>
    <hr>
    <applet code=Calc.class width=300 height=120>
      alt="Browser versteht &lt;APPLET&gt;, ignoriert es aber"
    </applet>
    <hr>
    <a href="Calc.java">Zum Quellcode</a>.
  </body>
</html>
```

Man sieht, dass dem Applet ein Rahmen definierter Größe zugewiesen wird. Der Source-Code in `Calc.java` ist für ein Anwendungsprogramm genauso wie für ein Applet geeignet:

```
/* Calc.java
 * Zeichnen eines Sinussignals mit zwei Frequenzkomponenten
 * Copyright (c) 2003 Sun Microsystems, Inc. All Rights Reserved.
 * Ausführlicher Copyright-Text im Java-Demo-Verzeichnis von JSE */
import java.awt.Graphics;
public class Calc extends java.applet.Applet {
    double f(double x) {
        return (Math.cos(x/5)+Math.sin(x/7)+2)*getSize().height/4;
    }
    public void paint(Graphics g) {
        for (int x = 0 ; x < getSize().width ; x++) {
            g.drawLine(x, (int)f(x), x + 1, (int)f(x + 1));
        }
    }
    public String getAppletInfo() {
        return "Zeichnet ein Sinussignal mit zwei Frequenzkomponenten.";
    }
}
```

Nach der Kompilierung steht in der Datei `Calc.class` der Byte-Code zur Verfügung. Beim Aufruf der Web-Seite wird die Datei geladen und in der Laufzeitumgebung des Web-Browsers ausgeführt:

Java-Applet mit einfachem Diagramm

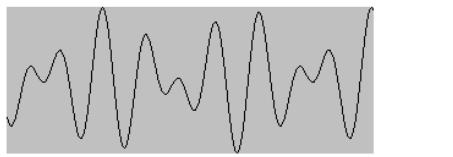


Abbildung 78: Anzeige eines Java-Applets im Web-Browser

7.5.3 Beispiel Fibo – Applets mit Java-RMI

Im folgenden Beispiel [Haro01] wird gezeigt, dass ein Java-Applet über Remote Method Invocation (wie in Kapitel 6 beschrieben), auf ein Server-Programm zugreifen kann. Das eigentliche Programm wird also auf dem Server ausgeführt. Dadurch ist sowohl eine dynamische als auch eine aktive Komponente der web-basierenden Programmierung verteilter Systeme enthalten.

Im Beispiel wird die Fibonacci-Zahl berechnet. Die Fibonacci-Zahl zu einer Ganzzahl ergibt sich aus Summe der Ganzzahlen (ab Null), die kleiner als die Ganzzahl sind. Die zu 5 zugehörige Fibonacci-Zahl ist 10 und ergibt sich also aus der Addition der Zahlen 1 bis 4.

Das Interface zum Server ist in der Datei `Fibo.java` beschrieben:

```
import java.rmi.*;
import java.math.BigInteger;

public interface Fibo extends Remote {
    public BigInteger getFibonacci(int n) throws RemoteException;
    public BigInteger getFibonacci(BigInteger n) throws RemoteException;
}
```

Der Server `FiboServer.java` ist analog zu den Erläuterungen im Kapitel 6 geschrieben:

```
// FiboServer.java berechnet Fibonacci-Zahlen
import java.net.MalformedURLException;
import java.rmi.*;
```

```

public class FiboServer {
    public static void main(String[] args) {
        try {
            Fibo_impl f = new Fibo_impl();
            Naming.rebind("Fibo", f);
            System.out.print("Fibo_server bereit. Rufen mit Fibo_client ");
            System.out.println("rmi://Hostname:Fibo <Zahl(en)>");
        }
        catch (RemoteException re) {
            System.out.println("Exception in Fibo_server: " + re);
        }
        catch (MalformedURLException e) {
            System.out.println("MalformedURLException " + e);
        }
    }
}

```

Die Berechnung der Fibonacci-Zahl befindet sich vom eigentlichen Server getrennt in der Datei `FiboImpl.java`:

```

// FiboImpl.java Implementierung der Berechnung der Fibonacci-Zahlen
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.math.BigInteger;

public class FiboImpl implements Fibo {
    public FiboImpl() throws RemoteException {
        UnicastRemoteObject.exportObject(this);
        // setLog(System.out);
    }
    public BigInteger getFibonacci(int n) throws RemoteException {
        return this.getFibonacci(new BigInteger(Long.toString(n)));
    }

    public BigInteger getFibonacci(BigInteger n) throws RemoteException {
        System.out.println("Berechne die " + n + ". Fibonacci-Zahl ... ");
        BigInteger zero = new BigInteger("0");
        BigInteger one = new BigInteger("1");

        if (n.equals(zero)) return zero;
        if (n.equals(one)) return one;

        BigInteger i = one;
        BigInteger a = zero;
        BigInteger b = one;
        while (i.compareTo(n) == -1) {
            BigInteger temp = b;
            b = b.add(a);
            a = temp;
            i = i.add(one);
        }
        return b;
    }
}

```

Der Server könnte auch aus einem Java-Standalone-Programm aufgerufen werden. In unserem Fall ist der Client aber als Applet realisiert und in `Fibo_applet.java` zu finden:

```
// Fibo_applet zum Aufruf des Fibonacci-Zahl-Servers
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.math.BigInteger;
public class Fibo_applet extends Applet {
    private TextArea resultArea
        = new TextArea("", 20, 72, TextArea.SCROLLBARS_BOTH);
    private TextField inputArea = new TextField(24);
    private Button calculate = new Button("Berechnung");
    private String server;
    public void init() {
        this.setLayout(new BorderLayout());
        Panel north = new Panel();
        north.add(new Label("Geben Sie eine ganze Zahl ein:"));
        north.add(inputArea);
        north.add(calculate);
        this.add(resultArea, BorderLayout.CENTER);
        this.add(north, BorderLayout.NORTH);
        Calculator c = new Calculator();
        inputArea.addActionListener(c);
        calculate.addActionListener(c);
        resultArea.setEditable(false);
        // server = "rmi://" + this.getCodeBase().getHost() + "/Fibo";
        server = "rmi://localhost/Fibo";
    }
    class Calculator implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            try {
                String input = inputArea.getText();
                if (input != null) {
                    BigInteger index = new BigInteger(input);
                    Fibo f = (Fibo) Naming.lookup(server);
                    BigInteger result = f.getFibonacci(index);
                    resultArea.setText(result.toString());
                }
            } catch (Exception ex) {
                resultArea.setText(ex.getMessage());
            }
        }
    }
}
```

Er wird aus Fibo_applet.html aufgerufen:

```
<html>
<head>
    <title>Fibo-Applet für Fibonacci-Zahlen über RMI</title>
</head>
<body bgcolor="#ffffff text="#000000>
    <h1>Fibo_applet für Fibonacci-Zahlen über RMI</h1>
    <P>
        <applet align="center"
            code="Fibo_applet" width="500" height="300">
```

```
</applet>
<hr>
</P>
</body>
</html>
```

Die Erzeugung der notwendigen Klassen geschieht durch

```
npca-1:~> javac Fibo_server.java
npca-1:~> java Fibo_server
Fibo_server bereit. Rufen mit java Fibo_client rmi://Hostname:Fibo
<Zahl(en)>
CLIENT> javac Fibo_applet.java
```

Beim Aufruf der Web-Seite `Fibo_applet.html` wird folgendes angezeigt:

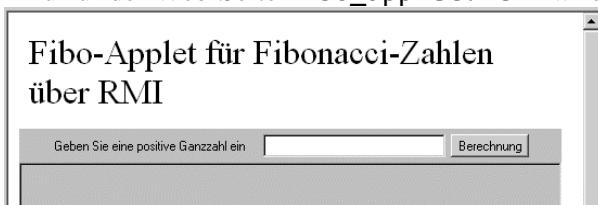


Abbildung 79: Anzeige eines Java-Applets mit InputArea im Web-Browser

Nach Eingabe einer Zahl wird diese über RMI an den Server übermittelt. Dort geschehen Berechnung und Rückgabe des Ergebnisses (und eine kleine Infomeldung in der Server-Ausgabe). Das Ergebnis wird im Browser ausgegeben. Auf diese Weise kann ein zentrales Server-Objekt Daten aus verschiedenen Quellen holen, Berechnungen durchführen usw. und die Ergebnisse an das Applet übermitteln. Java-Applets sind gut geeignet, um das Prinzip der client-seitigen Programmierung web-basierender Anwendungen zu erläutern. Sie haben sich aber aufgrund von Inkompabilitäten in den Web-Browsern nicht überall durchgesetzt.

7.6 Zusammenfassung Web-Applikationen

Zusammenfassend lassen sich folgende Grundlagen feststellen: Web-Applikationen können mit wenigen Basistechniken umgesetzt werden:

- Request / Response, Kapselung des Socket-Interfaces durch HTTP
- Personalisierung durch Sessions / Cookies
- Basis-Sicherheit (Authentifizierung, Verschlüsselung)
- Server: Java Servlets / JSPs, PHP
- Client: Java Applets, JavaScript / Ajax, ...

Meist ist eine Kombination von Techniken erforderlich.

7.7 Web Services

Für die losere Koppelung verteilter Systeme hat sich Ende der Neunziger Jahre XML-RPC etabliert. Als Weiterentwicklung zu einer vollständig dienstorientierten Architektur für verteilte Systeme (Service-Oriented Architecture SOA) sind heute Web Services üblich. In unserer Middleware-Betrachtung werden sie folgendermaßen eingeordnet:

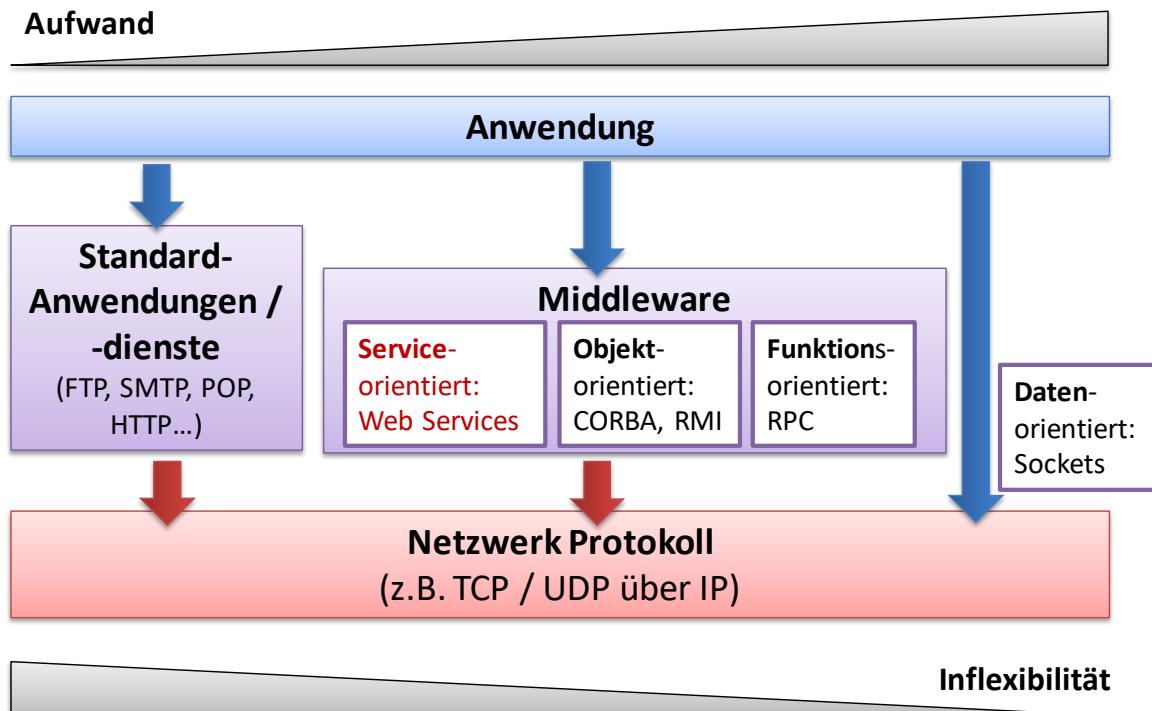


Abbildung 80: Web Services repräsentieren den service-orientierten Ansatz von Middleware

Sie verbinden Anwendungen, die in unterschiedlichen Middleware-Systemen entwickelt wurden und bieten eine einfache Lösung für Schnittstellen-, Konvertierungs- und Zugangsprobleme, wie folgendes Bild zeigt:

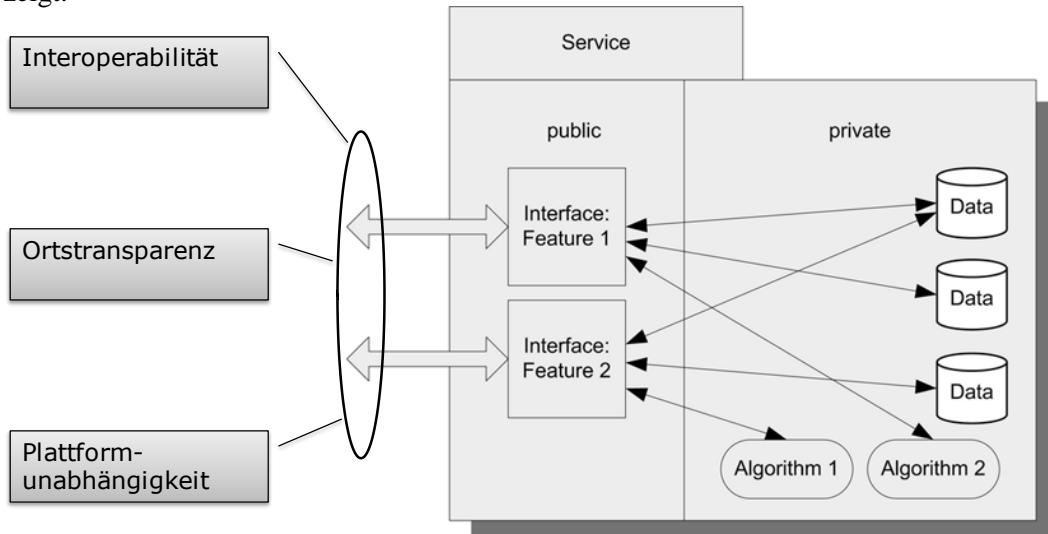


Abbildung 81: Service-basierter Ansatz für Web Services⁵

⁵ Quelle: Dirk Krafzig, Karl Banke, Dirk Slama: *Enterprise SOA*

Es gibt hier eine beachtliche Kompatibilität der Definition des Begriffs Service mit dem Konzept der Klasse: Auch diese kapseln durch eine Schnittstelle Funktionen (Algorithmen) und Daten. Bei einem Service werden

- die eine Menge von Schnittstellen zur Verfügung gestellt,
- diese Schnittstellen durch ein dediziertes Protokoll angesprochen;
- die Funktionen des Service durch eine Menge von Klassen realisiert.

Web Services sind eine spezielle Ausprägung des Begriffs Service. Folgende drei Rollen sind beim Einsatz von Web Services geläufig:

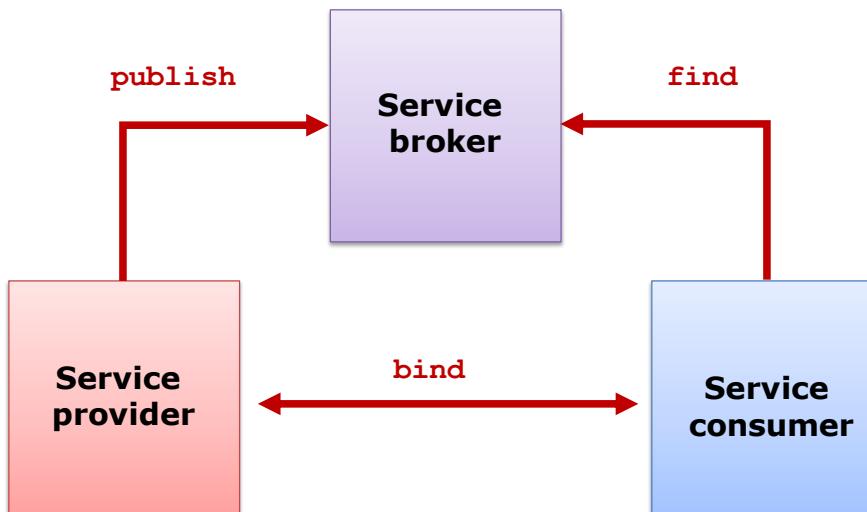


Abbildung 82: Rollen bei Web Services

- Ein Service Provider bietet einen Dienst an und macht dies über einen Service Broker bekannt (publish).
- Ein Service Consumer sucht beim Service Broker einen Dienst (find).
- Der Service Broker informiert den Service Consumer über die Nutzung des Dienstes des Service Providers.
- Der Service Consumer bindet den Service an und kann ihn dann nutzen (bind).

Um das Prinzip der Web Services zu verstehen, muss zunächst eine Einführung von XML erfolgen.⁶

7.7.1 XML (Extended Markup Language)

Die bisherigen Betrachtungen wendeten sich an Web-Entwickler, die Server und Clients entwickeln wollten. Für Informatiker stellt sich aber zunehmend die Aufgabe, verschiedene IT-Systeme so zu verbinden, dass die transportierten Daten sowohl für Menschen als auch für Maschinen, nämlich mehrere beteiligte IT-Systeme, interpretierbar sind. HTML und der Nachfolger xHTML sind dafür aus verschiedenen Gründen nicht geeignet.

Daher wurde XML (wie HTML) mit dem Ziel einer plattform- und systemübergreifenden Darstellung von Daten entwickelt. Wichtigstes Entwicklungsziel war bei XML aber anders als bei HTML, Inhalt und Darstellung voneinander zu trennen und eine leichte automatisierte Verarbeitung von Daten zu ermöglichen. Trotzdem sind XML-Dokumente durch ihre ASCII-Darstellung und den logischen Aufbau für Menschen leicht lesbar. Näheres zur Standardisierung findet sich unter <http://www.w3.org/XML/>, weitere Infos auch unter <http://de.selfhtml.org/xml/intro.htm>. Wie bei HTML werden in XML Daten in Tags eingefasst. XML-Tags machen jedoch Aussagen über die Bedeutung des eingeschlossenen Textes und nicht über dessen Darstellung wie bei HTML. Zu jedem XML-Dokument gehört ein root-Element (Wurzel), das durch Start- und End-Element (mit / als Slash gekennzeichnet) Anfang und Ende des

⁶ Web Services sind inzwischen ein Oberbegriff, der über XML-basierende Ansätze (auch verkürzt SOAP-Web Services genannt) hinaus geht. Das Prinzip der REST-Web Services wird im AJAX-Kapitel näher erläutert.

Dokumente markieren. In diese Elemente sind in einer Baumstruktur weitere Elemente mit eigenen Start- und End-Tags eingebettet. Es können weitere innere Verzweigungen enthalten.

Das folgende XML-Beispiel zeigt Adressinformation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE address SYSTEM "/var/www/dtd/address.dtd">
<address>
    <name>Dieter Krause</name>
    <street>Albrechtstr. 8</street>
    <city> D-49076 Osnabrück</city>
</address>
```

<address> ist das Wurzelement, das die übrigen Daten bis zum abschließenden </address>-Tag einfasst. Eine Hierarchiestufe tiefer sind in dieser XML-Darstellung Elemente <name>, <street> und <city> zu finden.

Die inhaltsbezogene hierarchische Darstellung erleichtert das Filtern, Suchen und Weiterverarbeiten von Informationen erheblich. XML ist durch Entwickler erweiterbar, wovon intensiv Gebrauch gemacht wird. Um daraus resultierende Namenskonflikte zu vermeiden, wurden Namespaces eingeführt. Um erweiterte XML-Darstellungen vernünftig zu strukturieren, besteht die Möglichkeit, den Aufbau in Dokumenttypdefinitionen zu beschreiben. Der gezeigte Aufbau von XML-Dokumenten ermöglicht es, Informationen gut strukturiert abzulegen. Um eine einheitliche und automatisierte Verarbeitung der Daten zu ermöglichen, sind Vorgaben über den Aufbau von XML-Dokumenten e.

7.7.2 Validierung von XML-Daten

Bisher wurden die Daten nur inhaltlich betrachtet. Man möchte aber häufig auch eine Grammatik angeben, die beschreibt, wie die erwarteten XML-Daten aufgebaut sind. Damit sind zwei wichtige Anforderungen an XML-Beschreibungen zu unterscheiden:

- XML-Dokumente sind **wohlgeformt**, wenn sie allen allgemeinen XML-Regeln genügen (→ Validierung der **Syntax**). Wohlgeformte Dokumente sind syntaktisch korrekt, aber nicht unbedingt sinnvoll für einen bestimmten Anwendungsbereich.
- XML-Dokumente sind darüber hinaus **gültig**, wenn ihre Struktur gemäß bestimmter Regeln interpretierbar ist (→ Validierung der **Semantik**). Sie müssen die in den Regeln beschriebene Grammatik einhalten.

Die beiden folgenden Dokumente zeigen den Unterschied anhand eines Beispiels auf. Die beiden folgenden Dokumente sind wohlgeformt, unterscheiden sich aber in der zusätzlichen Angabe einer E-Mail-Adresse, was je nach semantischen Regeln gültig ist oder nicht.

<pre><address> <name>Dieter Krause</name> <street>Albrechtstr. 8</street> <city> D-49076 Osnabrück</city> </address></pre>	<pre><address> <name>Dieter Krause</name> <street>Albrechtstr. 8</street> <city> D-49076 Osnabrück</city> <email>d.krause@gmx.net</email> </address></pre>
--	--

Soll nun die Gültigkeit untersucht werden, muss diese in einer Semantik-Beschreibung dargestellt werden. Eine nicht mehr gebräuchliche Form sind die DTDs.

7.7.3 XML Schema

DTDs sind nicht XML-konform und wurden daher durch XML Schemata ersetzt. Deren Spezifikation enthält Basisdatentypen und viele Ausdrucksmöglichkeiten zur Definition komplexer Datentypen, wie man sie z. B. aus Programmiersprachen kennt. Dies illustriert folgende Tabelle:

Einfache Datentypen / Elemente	Komplexe Elemente	Kompositionen
<pre><xsd:element name="city" type="xsd:string"/></pre>	<pre><xsd:element name="address"> <xsd:complexType> ... </xsd:complexType> </xsd:element></pre>	<pre><xsd:sequence> <xsd:element name="name" type="xsd:string" /> ... </xsd:sequence></pre>

Die Datentypdefinitionen werden in einer Datei mit der Endung .xsd beschrieben. Jede XML-Datei enthält einen Link auf die zugehörige XML-Schema-Definition z. B.:

```
<?xml version="1.0"
<offer xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="offer.xsd">
```

Ein Beispiel-XML-Schema für die im vorigen Abschnitt dargestellt Adressinformation ist

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Die in XML Schemata vorgesehenen Datentypen sind in folgendem Bild dargestellt:

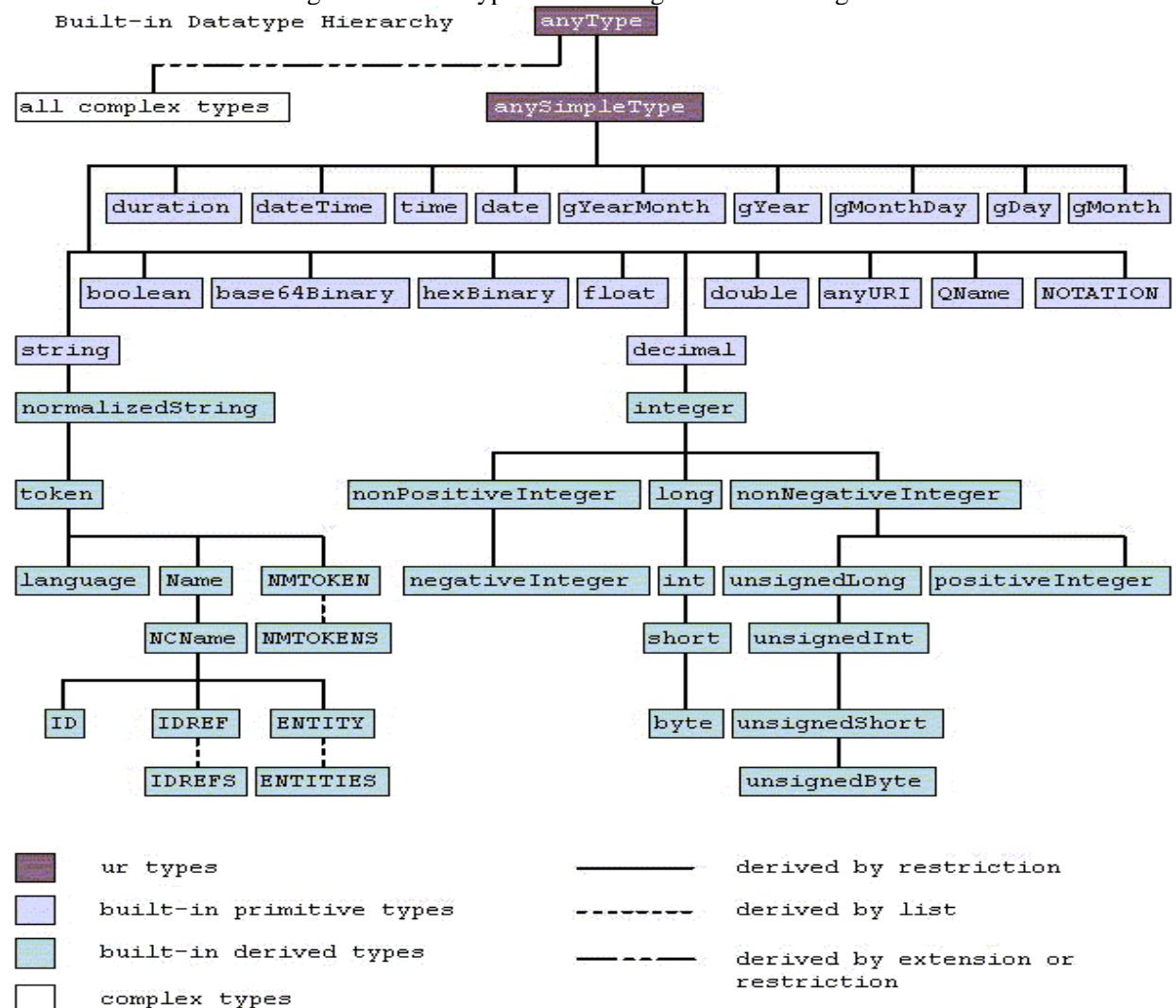


Abbildung 83: Datentypen in XML Schemata

In XML-Schemata sind viele Einschränkungsmöglichkeiten (Wertebereiche, Aufzählungen) für die Daten möglich.

Die folgende Tabelle zeigt eine Übersicht mit Beispielen:

Eigenschaft	Bedeutung
Anzahl der Elemente (hier 1 - ∞ Autoren)	<xsd:elementname="author" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
Einschränkung des Wertebereichs basierend auf Standarddatentyp	<xsd:simpleTypename="Schulnote"> <xsd:restrictionbase="xsd:short"> <xsd:minInclusivevalue="1"/> <xsd:maxInclusivevalue="6"/> </xsd:restriction> </xsd:simpleType>
Enumeration	<xsd:simpleTypename="Farben"> <xsd:restrictionbase="xsd:string"> <xsd:enumerationvalue="rot"/> <xsd:enumerationvalue="blau"/> <xsd:enumerationvalue="weiss"/> </xsd:restriction> </xsd:simpleType>
Zusammenstellung von Listen	<xsd:simpleType name="Hobbys"> <xsd:listitemType="xsd:string"/> </xsd:simpleType>

Die Interpretation der XML-Dateien und der zugehörigen Schema geschieht mittels entsprechender Spracherweiterungen (z.B. in JAXB) oder eigener Hilfswerkzeuge (XML-Parser/-Editoren etc.). In der Vorlesung wird die Nutzung eines kommerziellen XML-Editors am Beispiel der Erstellung multimedialer Flugzeugdokumentation für den Airbus A310 aus einem EU-Projekt unter Beteiligung der Hochschule Osnabrück, Airbus Bremen und anderer Projektpartner gezeigt.

Wenn z. B. eine Anwendung eine Preisliste als XML-Daten und zugehöriges XML-Schema hat, sind alle Informationen für die Verarbeitung vorhanden, wie folgende Bild zeigt:

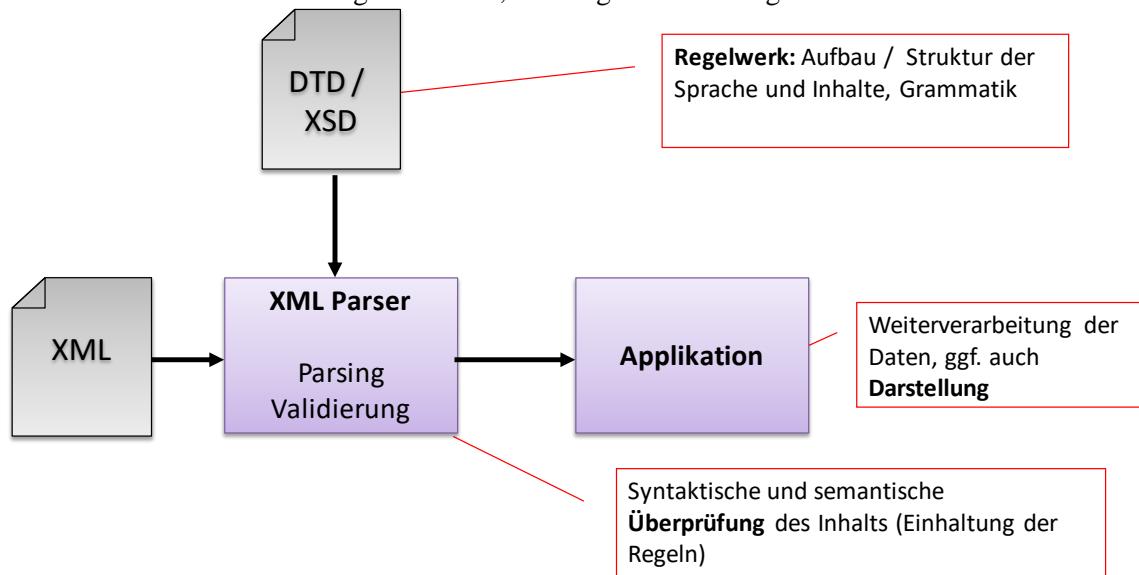


Abbildung 84: Nutzung eines XML-Parsers zur Überprüfung der Einhaltung von Schemata

Der XML-Parser dient dazu, die Einhaltung der Festlegung in XML Schema zu überprüfen. Die Applikation erhält das Ergebnis des XML-Parsers und kann nun selbst entscheiden, wie sie die Daten filtern und anzeigen möchte.

7.7.4 XSL - Automatisierte Verarbeitung von XML-Daten in XML-Parsern

Viele Programmiersprachen bieten gute Möglichkeiten zur Verarbeitung von XML-Dokumenten an. Von Interesse sind die folgenden Standardschnittstellen für die Verarbeitung von XML:

- Document Object Model (DOM)
- Simple API for XML (SAX)
- Streaming API for XML, StAX

Bei DOM wird die XML-Datei vollständig eingelesen und die Daten in eine speicherinterne Baumstruktur überführt. Nach dem Einlesen finden sich alle Daten im Speicher und es kann mit den normalen Mitteln der Programmiersprache direkt (und damit schnell) auf sie zugegriffen werden. Nachteilig ist, dass bei großen Datenmengen der Speicherbedarf sehr groß werden kann.

Bei SAX hingegen wird die Datei sequenziell gelesen und jedes neue Element als ein Ereignis betrachtet, auf dass man geeignet reagieren kann. Damit folgt SAX dem sequenziellen Aufbau des XML-Dokuments. SAX ist sehr effizient, da beliebig große XML-Dokumente schnell erarbeitet werden können. Da die Ereignisse sequenziell eintreffen, kann allerdings nicht wahlfrei (direkt) auf einzelne Teile des XML-Dokuments zugegriffen werden. Hier kann also ein Nachteil entstehen, wenn sich die relevanten Informationen hinten in der XML-Datei befinden und zunächst der vordere Teil geparsert werden muss. StAX ist sowohl Ereignis-orientiert (Events werden per Pull erzeugt) als auch Baum-basierend. Der Abruf (Pull) von einzelnen Elementen ist ebenso möglich.

Java bietet eine komfortable API zur XML-Verarbeitung, die Java API for XML Processing (JAXP). Sie bietet Möglichkeiten zum Validieren, Parsen, Generieren und Transformieren von XML-Dokumenten mit Java. Unterstützt werden alle drei Standardschnittstellen, also DOM, SAX und StAX. Es gibt darüber hinaus Stylesheet-Verarbeitung für Extensible Stylesheet Language Transformations (XSLT). Da SAX ereignisorientiert arbeitet, muss der Benutzer eine Event-Handler-Klasse mit `startElement`, `endElement` und `characters`-Callback-Methoden definieren.

Das folgende Bild zeigt die prinzipielle Nutzung des SAX-Teils der JAXP:

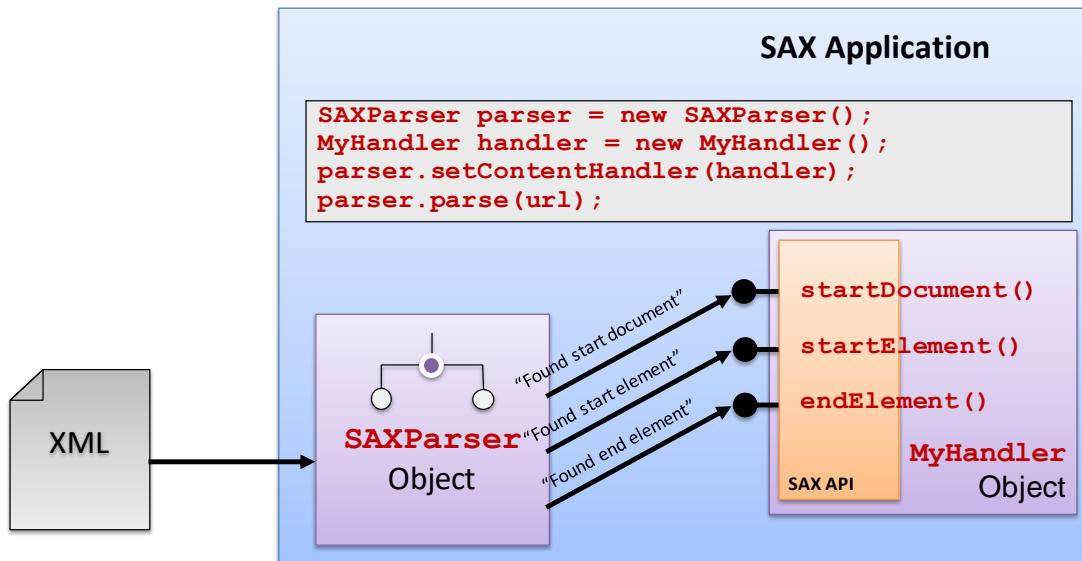


Abbildung 85: XML-Verarbeitung mittels SAX-Parser und Event-Handler

Bei der DOM-Nutzung wird das XML-Dokument durch den DOM-Parser geparsert und vollständig im Speicher aufgebaut. Das folgende Bild zeigt das zugehörige Prinzip:

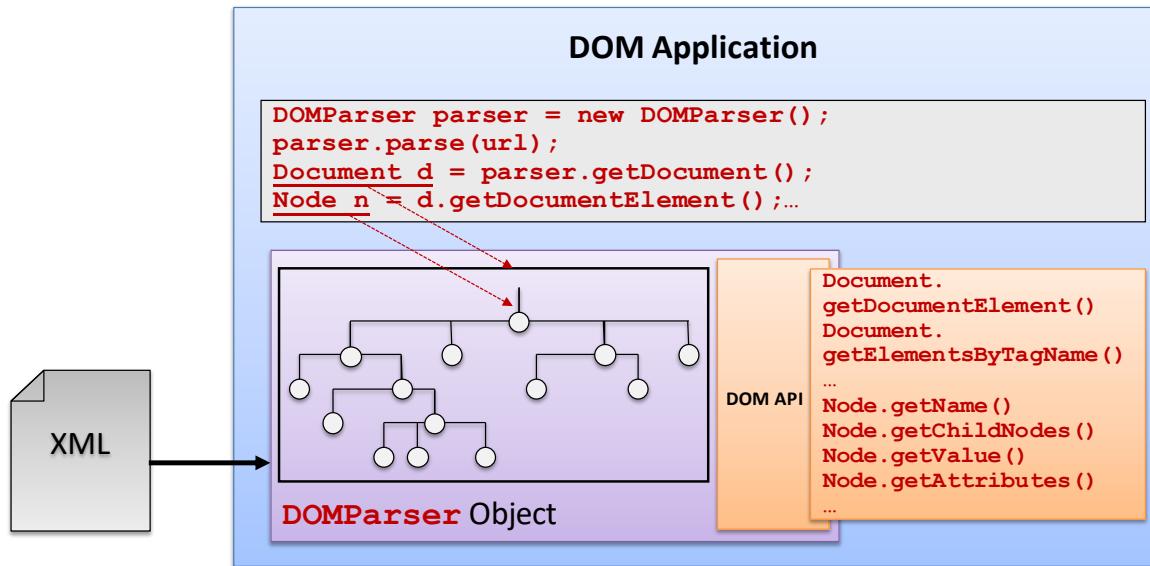


Abbildung 86: XML-Verarbeitung mit DOM-Parser und vollständiger Speicherung

Die Verwendung von StAX schließlich kombiniert die Vorteile der beiden Ansätze und funktioniert prinzipiell so:

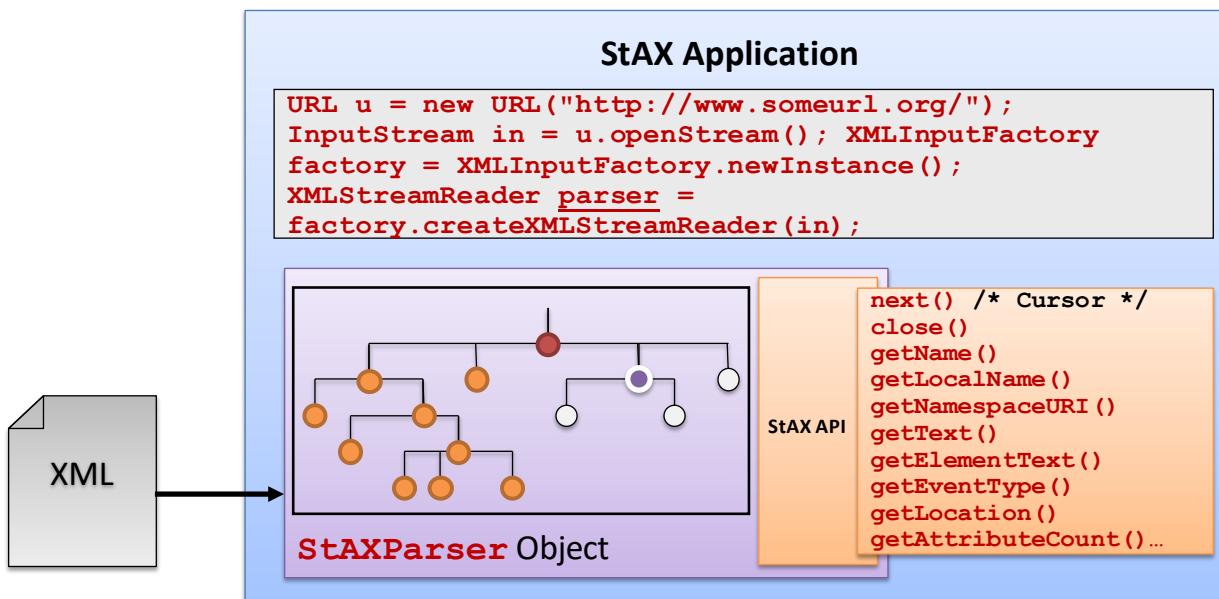


Abbildung 87: XML-Verarbeitung mit StAX in JAXP

Entscheidender Unterschied ist: StAX arbeitet mit einem Cursor: über diesen kann in dem Dokument navigiert (kann nur vorwärts geschoben werden, nicht zurück) werden.
Siehe <http://www.xml.com/pub/a/2003/09/17/stax.html>

Die folgende Tabelle aus [Ryan12]⁷ zeigt die wichtigsten Eigenschaften der drei Ansätze auf:

	StAX	SAX	DOM
API Stil	Streaming; Pull-API	Streaming; Push-API	Baumstruktur im Speicher
Aufwand Nutzung	Gut	Mittel	Hoch
CPU / Speicher-bedarf	Gering	Gering	Abhängig von Anwendung
Lesen	Ja	Ja	Ja
Schreiben	Ja	Nein	Ja
Create, Read, Update, Delete (CRUD)	Nein	Nein	Ja

CRUD bezieht sich auf die Operationen Create, Read, Update, Delete, die als atomare Operationen realisiert sein sollten.

7.7.5 Extensible Stylesheet Language

Die XSL dient allgemein dazu, XML-Dateien zur Weiterverarbeitung aufzubereiten. Bei der Aufbereitung in ein neues XML-basierendes Zwischenformat spricht man von Transformation und nutzt dazu XSLT (XSL Transformation). Damit kann eine Schnittstellenanpassung und eine Auswahl der zu übergebenden Informationen durchgeführt werden.

Bei der Aufbereitung für die Endausgabe hingegen spricht man von Formatierung und nutzt XSL-FO XSL Formating Objects. Gängige Ausgabeformate sind z. B. PDF, ASCII-Text und RTF (Rich Text Format), aber auch die Aufbereitung auf Geräte mit beschränkten Ressourcen (Smartphones) ist auf diese Art möglich.

Die folgende Grafik illustriert die Nutzung von XSL in einem XSL-Prozessor:

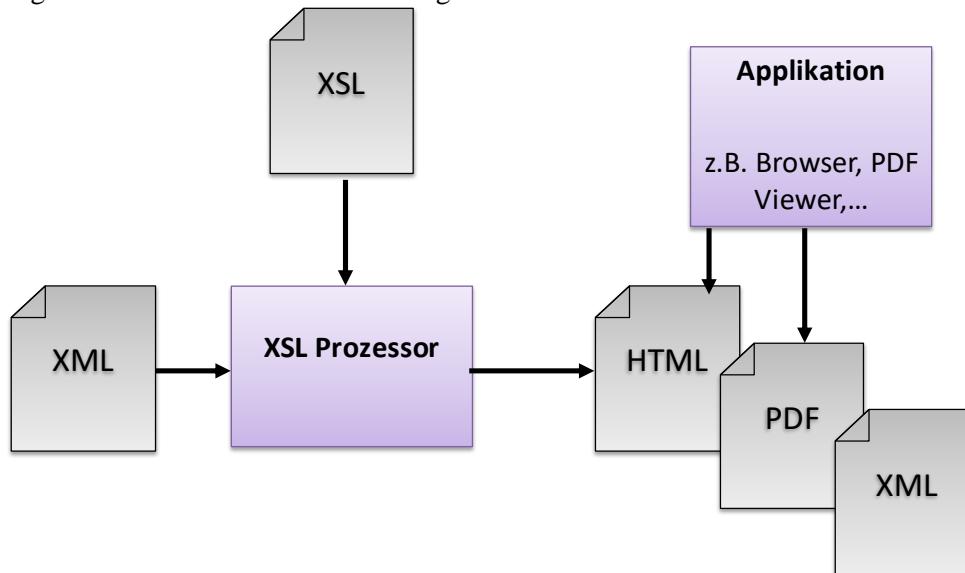


Abbildung 88: Transformation/Formatierung von XML mit XSL (Extensible Stylesheet Language)

⁷ "Does StAX Belong in Your XML Toolbox?" (<http://www.developer.com/xml/article.php/3397691>) by Jeff Ryan)

Die Aufbereitung für die Ausgabe in einem Web-Browser kann je nach Sichtweise als ein Sonderfall der Transformation oder der Formatierung gesehen werden. Einerseits ist das Ziel wieder ein XML-Dokument (mit XHTML). Andererseits sind dabei natürlich viele Formatierungsaufgaben zu lösen. Ein Adressbeispiel soll die Nutzung von XSLT illustrieren:

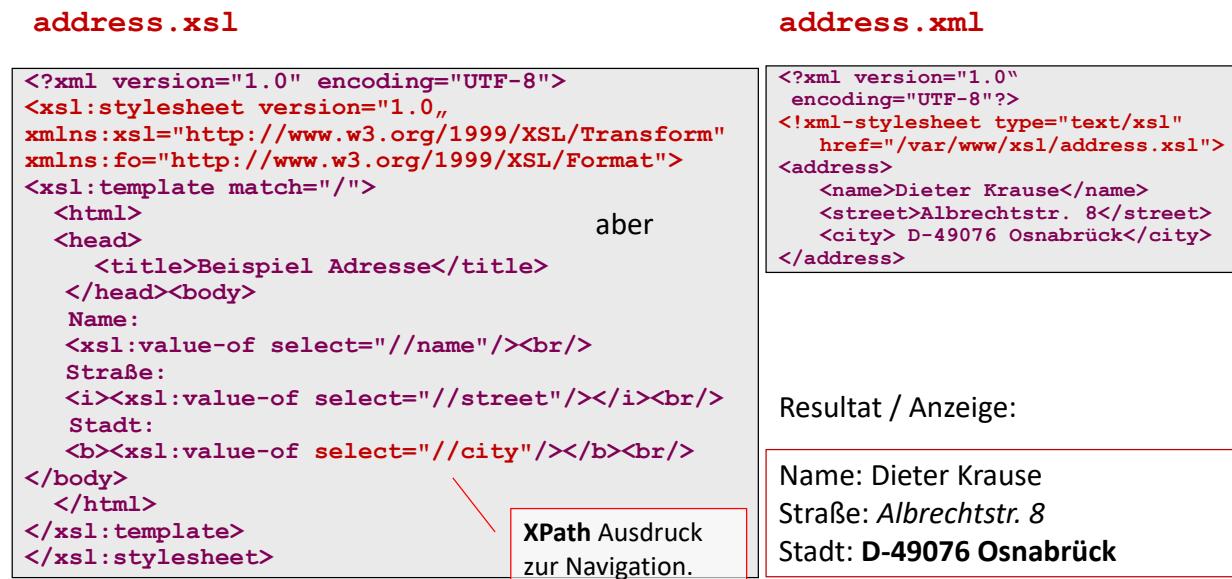


Abbildung 89: Beispiel der Transformation mittels XSLT

In der ersten Zeile beider Dateien sehen wir, dass ein XML-Dokument eine Deklaration besitzen kann. Beispiel: `<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>`

Hier gibt Version gibt die XML Version an, Encoding legt die Kodierung der Zeichen im Dokument fest. und Standalone gibt an, ob eine Anwendung ein externes XML-Schema lesen muss, um die korrekten Werte zu den Elementen zu verifizieren.

Bei Nutzung mehrerer XML-Dokumente sind Namenskonflikte bei Element- und Attributnamen möglich. Daher wurden (wie bei C++ etc.): Namensräume eingeführt. XML-Namensräume werden durch URI angegeben `xmlns="URI"`. Sie können durch Angabe eines Präfixes identifiziert werden. Präfix und Namen sind dabei durch : getrennt. Beispiel:

```

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:m="http://www.w3.org/1999/Math/MathML">
    ...
    ... XHTML-Elemente
    <m:math>
        <span>Hier steht XHTML. Variable x: <m:i>x</m:i></span>
    </m:math>
    ...
    ... XHTML-Elemente
</html>

```

7.7.6 AJAX - Ajax: Erstellung von Rich Internet Clients mit JavaScript & XML

XML wird auch als Teil eines Ansatzes genutzt, Clients intelligenter zu machen und dadurch die Server zu entlasten. Der Ansatz wurde unter dem Namen AJAX - Asynchronous JavaScript & XML erstmals durch Jesse James Garrett beschrieben [Garr05].⁸

Die Idee ist eine asynchrone Signalisierung: Bei Änderungen der aktuellen Web-Seite werden nur die geänderten Teile neu geladen. Es entstehen sog. Rich Internet Clients (siehe Web 2.0). Gelegentlich wird hierfür auch der Begriff Rich Internet Applications (RIAs) verwendet. Anwendung findet AJAX z.B. bei Google Suggest, dem Ergänzen von Vorschlägen für Suchworte bei Google.

AJAX nutzt (exzessiv) JavaScript im Client. Über registrierte Callbacks werden Elemente der aktuellen Seite aktualisiert. Die Daten können als XML-Dokumente ausgetauscht werden (dynamisch erzeugt).

⁸ Zu finden unter <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>

Das Formular im folgenden Bild nutzt AJAX:

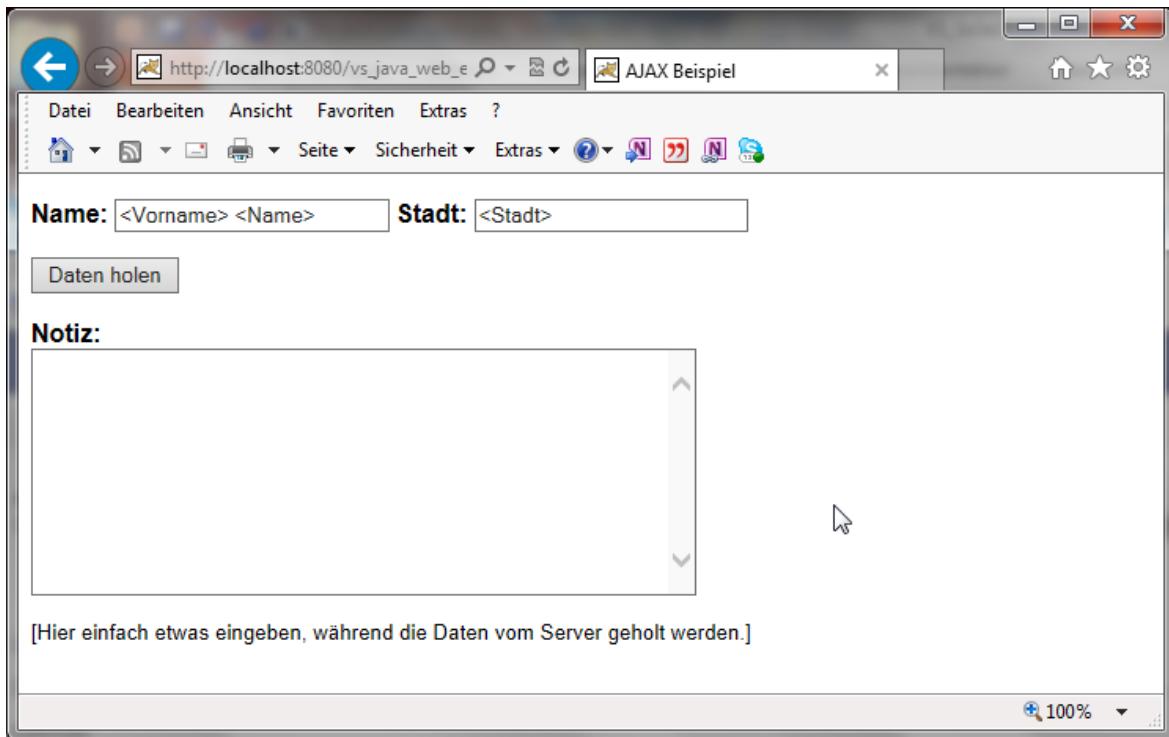


Abbildung 90: Formular mit AJAX-Nutzung

Die Formular-Daten werden vorgeladen. Restliche Elemente können weiter editiert werden. Dabei müssen nur Teile der Seite müssen neu geladen werden.

Die entsprechende HTML-Seite ist in folgendem Listing beschrieben:

```
<!DOCTYPE html>
<html>
  <head>
    <title>AJAX Beispiel</title>
    <script language="JavaScript" src="js/datapicker.js"
      type="text/javascript">
    </script>
  </head>
  <body style="font-family:arial;">
    <p><b>Name: </b>
    <input type="text" id="namefield" name=nameFieldText value="<Vorname>
    <Name>" size=25>
    <b>Stadt: </b>
    <input type="text" id="cityfield" name=cityFieldText value="<Stadt>"
    size=25>
    </p>
    <p><b>Daten holen</b></p>
    <p><b>Notiz:</b><br>
    <textarea id="notefield" name=noteFieldText cols="50" rows="10">
    </textarea>
    <br><p style="font-size:smaller;">
      [Hier einfach etwas eingeben, während die Daten vom Server geholt
      werden.]
    </p>
  </body>
</html>
```

Die zugehörigen Daten finden sich in folgender XML-Datei:

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <address>
        <name>Dieter Krause</name>
        <street>Kalker Weg</street>
        <no>5</no>
        <zip>51103</zip>
        <city>Koeln-Kalk</city>
    </address>
    <address>
        <name>Liesbeth Krause</name>
        <street>Kalker Weg</street>
        <no>5</no>
        <zip>51103</zip>
        <city>Koeln-Kalk</city>
    </address>
</data>
```

Sie könnten auch dynamisch z.B. in einem Servlet erzeugt werden.

In einem XMLHttpRequest-Objekt können die Daten browser-abhängig verarbeitet werden:

```
var requestobj = getXMLHttpRequest();
function getXMLHttpRequest() {
    // XMLHttpRequest for Firefox, Opera, Safari
    if (window.XMLHttpRequest) {
        return new XMLHttpRequest();
    }
    if (window.ActiveXObject) { // Internet Explorer
        try { // for IE new
            return new ActiveXObject("Msxml2.XMLHTTP")
        }
        catch (e) { // for IE old
            try {
                return new ActiveXObject("Microsoft.XMLHTTP")
            }
            catch (e) {
                alert("Your browser does not support AJAX!");
                return null;
            }
        }
    }
} return null;
}
```

Zur Verarbeitung der Requests stehen folgende Methoden zur Verfügung:

Eigenschaft	Bedeutung
getResponseHeader	Liefert Header der Antwort auf die die Server-Anfrage.
getAllResponseHeaders	Liefert Header der Antwort auf die die Server-Anfrage.
abort	Abbruch der aktuellen Anfrage.
open	Initialisierung der Anfrage unter Angabe der Abfrageart (GET/POST).
send	Anfrage verschickt verschickt.

Der Ajax-Request wird durch folgenden Code-Ausschnitt verschickt:

```
function GetData(url) {
```

```

if (requestobj) {
    requestobj.open('GET', url, true);
    requestobj.onreadystatechange = AsyncResult;
    requestobj.send(null);
}
}

```

Das zuvor erzeugte XMLHttpRequest-Objekt kann also eine Anfrage erzeugen. Dabei wird mit `onreadystatechange` eine Callback-Funktion registriert. Diese wird aufgerufen, wenn sich Zustand des Objektes ändert (z.B. Daten stehen bereit). Die Funktion wird asynchron aufgerufen.

Einen Teil der XMLHttpRequest-Eigenschaften zeigt folgende Tabelle:

Eigenschaft	Bedeutung
<code>onreadystatechange</code>	Zeiger auf die Funktion welche aufgerufen wird wenn sich der <code>readyState</code> ändert. (rw)
<code>readyState</code>	Status des Objektes. (r) 0 : Objekt nicht initialisiert, 1 : Daten werden geladen, 2: Daten geladen, Header aber nicht 3 : Daten sind teilweise geladen (interaktiv) 4 : Daten sind komplett verfügbar.
<code>status</code>	HTTP Statuscode, den der Webserver zurück geliefert hat. (r)
<code>responseXML</code>	Response als XML-Dokument. (r)
<code>responseText</code>	Darstellung der XML-Antwort als String (d.h. Entity Body als Zeichenkette). (r)

Die Callback-Funktion `AsyncResult` ist in folgendem Code-Ausschnitt zu sehen:

```

function AsyncResult() {
    if (requestobj.readyState == 4) {
        if (requestobj.status == 200) {
            var data=requestobj.responseXML.getElementsByTagName('address')[0];
            var cField = document.getElementById('namefield');
            var browserName = navigator.appName;
            if (browserName == "Microsoft Internet Explorer")
                cField.defaultValue = data.childNodes[1].testContent
            else
                cField.defaultValue = data.childNodes[1].firstChild.wholeText;
            cField = document.getElementById('cityfield');
            if (browserName == "Microsoft Internet Explorer")
                cField.defaultValue = data.childNodes[9].testContent
            else
                cField.defaultValue = data.childNodes[9].firstChild.wholeText;
        }
    }
}

```

Das Auslesen von XML-Dokumenten ist Browser-spezifisch. Im Beispiel sollen die Felder Name und Stadt über einen Ajax-Request befüllt werden.⁹

Das Ergebnis zeigt folgendes Bild:

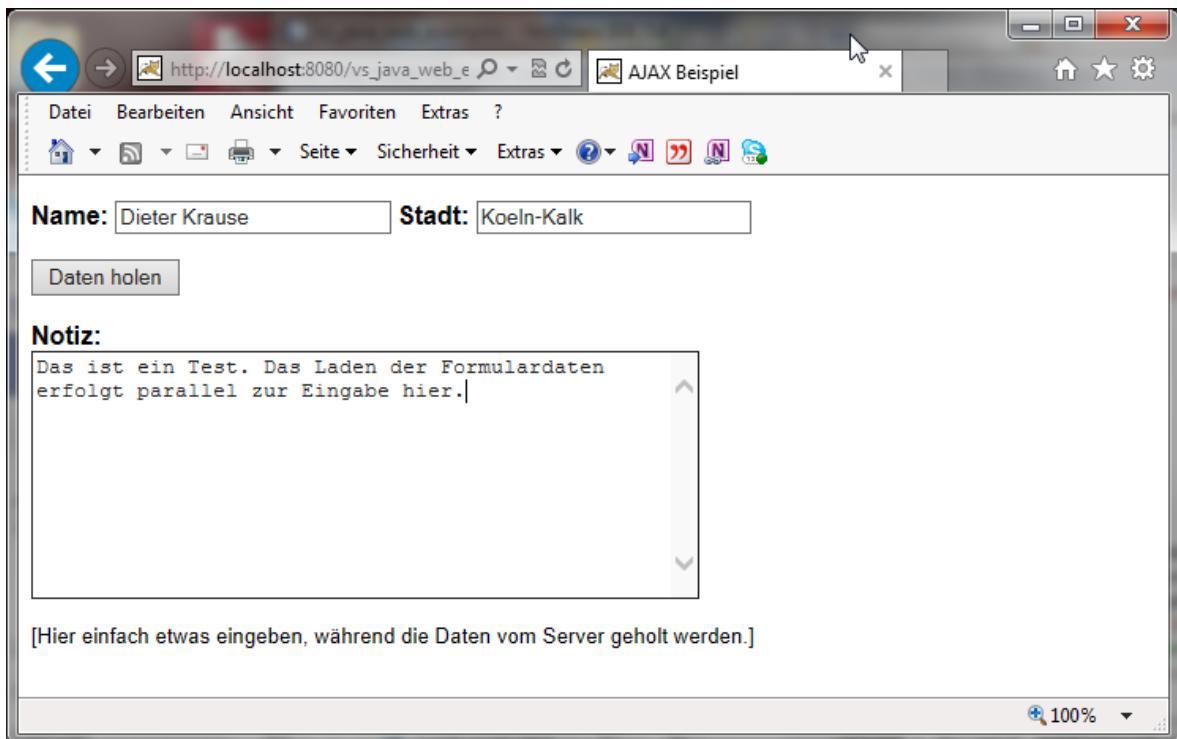


Abbildung 91: Formular mit AJAX-Nutzung (ausgefüllt)

Um die asynchrone Verarbeitung besser sehen zu können, verzögert man den Zugriff auf `data/data.xml`. Dies wird über ein `FileLoader`-Servlet (hier mit 4 Sekunden) implementiert:

```
<p>
<button onClick = "GetData('FileLoader?filename=
data/data.xml&delay=4000')>Daten holen</button>
</p>
```

Der Aufruf muss in der HTML-Datei geändert werden. Dabei ist der Parameter `filename` die auszulesende Datei und der Parameter `delay` die Verzögerung. Das Debugging von AJAX-Anwendungen ist nicht ganz einfach.

⁹ Achtung: Beim Internet Explorer (IE) muss der Browser-Cache jeweils geleert werden. <STRG>-Reload reicht nicht aus! Bei FF wird jeweils ein neues Tab bei einem Reload erzeugt und `datat.xml` aus dem Cache entfernt. Das ist bei IE nicht so.

Es kann z.B. über Extras-F12 Entwicklertools (bei Internet Explorer) oder Firebug (Firefox) erfolgen, wie folgendes Bild zeigt:

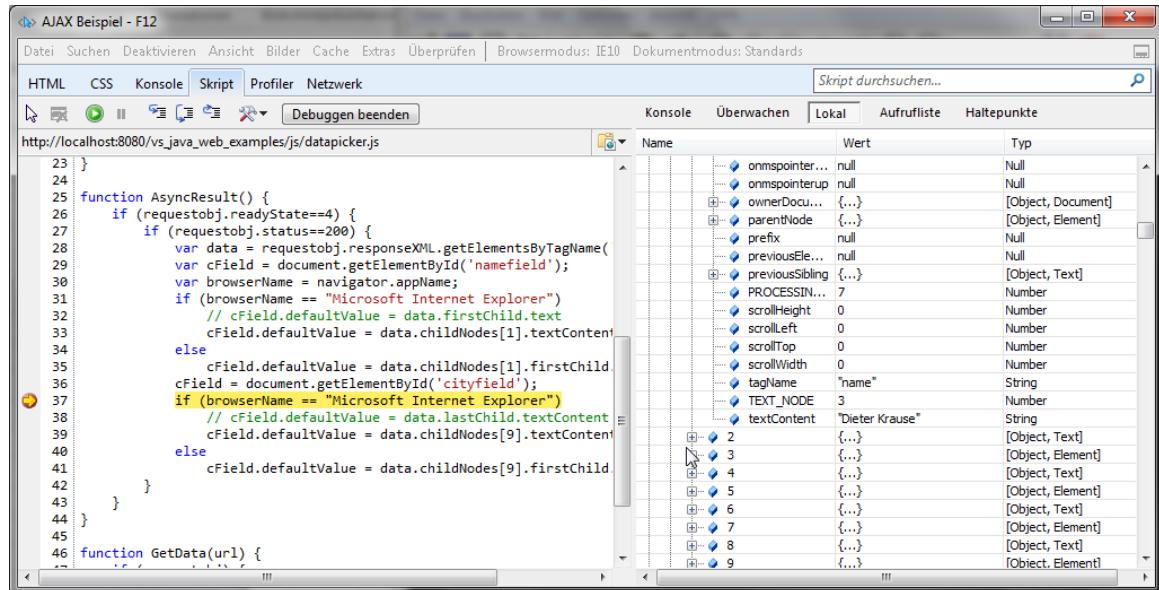


Abbildung 92: Debugging von AJAX im Internet Explorer

Das folgende Bild zeigt zusammengefasst den Ablauf bei AJAX-Anwendungen:

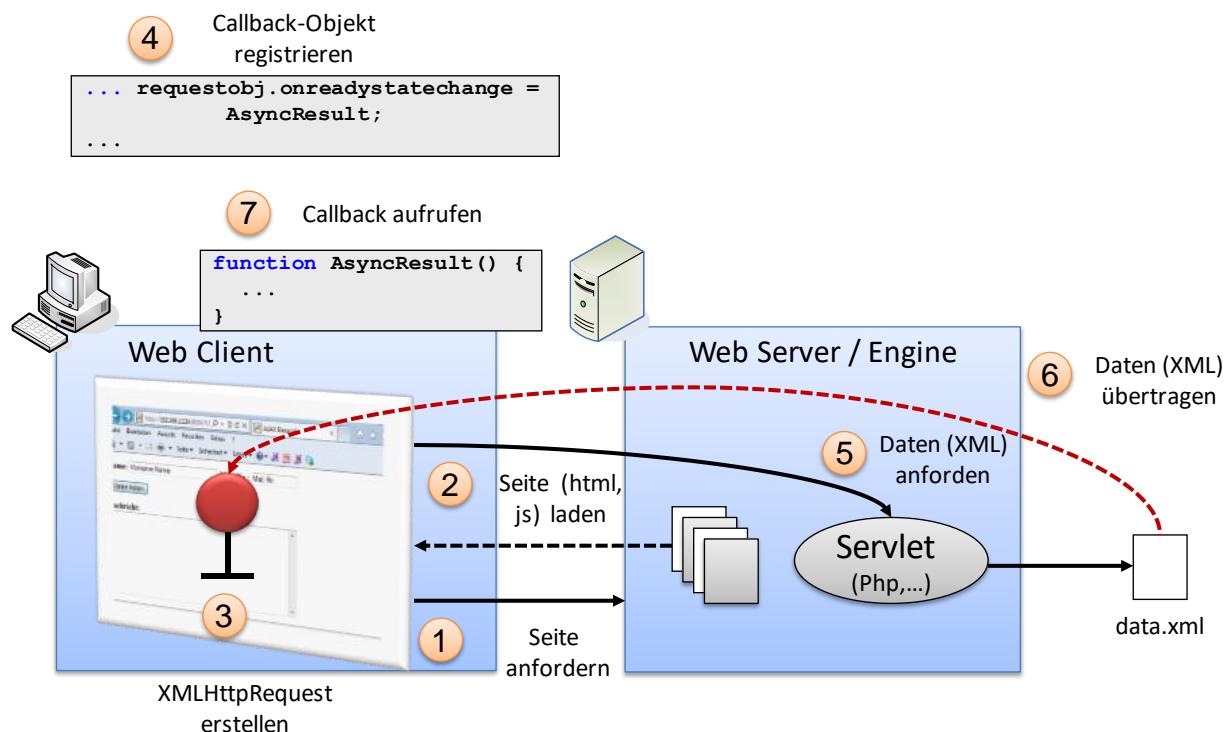


Abbildung 93: Genereller Ablauf bei AJAX-Anwendungen

- Der Web Browser fordert beim Web-Server die Seite an.
- Der Web-Server liefert eine HTML-Seite mit eingebettetem JavaScript.
- Darin wird der XMLHttpRequest erzeugt.
- Dabei wird das Callback-Objekt registriert.
- Die XML-Daten werden angefordert und bereitgestellt (im Beispiel aus einer Datei).
- Die XML-Daten werden an den Web-Browser ausgeliefert.
- Im Web-Browser nimmt die registrierte Callback-Methode die Daten entgegen und verarbeitet sie.

AJAX ist heute zu einer wichtigen Technologie geworden. Der Browser (Web-Client) bekommt mehr Autonomie. Viele Web-Seiten werden als Rich Clients gerendert (Google, Facebook, Twitter, ...). Die asynchrone Verarbeitung von XML-Daten verursacht einen gewissen Aufwand und wird teilweise durch schlankere Darstellungen z.B. *JavaScript Object Notation* (JSON) ersetzt. Die Daten werden per HTTP-Methode (GET / POST) abgerufen. Dies entspricht dem Representational State Transfer (REST). Dieses Prinzip hat sich auch beim automatisierten Austausch von Daten (unabhängig von AJAX) eingebürgert und wird dort auch als REST Web Services bezeichnet. Als Architektur-Stil werden sog. „*Pipes & Filters*“ betrachtet. Der Client ruft Daten von Server ab, der wiederum Daten von weiterem Server bezieht usw..

7.7.7 JavaScript Object Notation - JSON

Ressourcen können häufig als Objekte angesehen werden. Ihr Inhalt (Zustand) lässt sich mit dem später beschriebenen SOAP als XML übertragen. Alternativ kann aber auch das vereinfachte JSON (JavaScript Object Notation) eingesetzt werden. Es ist kompakter und lesbarer als XML, aber weniger mächtig in Bezug auf überprüfbare Typsysteme, Linking, Queries, Transformationen etc.

Syntax an JavaScript angelehnt

Realisiert Attribute per Namen : Werte – Paare

Schachtelung / Aggregation möglich

In JSON wird unser zuvor in XML ausgedrücktes Beispiel folgendermaßen ausgedrückt:

```
{  
    "addresses": [  
        {"name": "Dieter Krause",  
         "street": "Kalker Weg", "no": 5,  
         "zip": 51103, "city": "Koeln-Kalk"  
        },  
        {"name": "Liesbeth Krause",  
         "street": "Kalker Weg", "no": 5,  
         "zip": 51103, "city": "Koeln-Kalk"  
        }  
    ]  
}
```

In Javascript kann JSON mittels `JSON.stringify()` bzw. `JSON.parse()` serialisiert bzw. deserialisiert werden, wie folgender Code-Abschnitt zeigt:

```
var obj = JSON.parse (address_str);  
output = '<table>\n';  
for (var i = 0; i<= obj.addresses.length-1; i++) {  
{  
    output += '<tr>';  
    output += '<td>' + obj.addresses[i]['name'] + '</td>';  
    output += '<td>' + obj.addresses[i]['city'] + '</td>';  
    output += '</tr>\n';  
}  
output += '</table>\n';
```

Auf die Attribute wird dabei mittels Index-Operator zugegriffen.

7.7.8 Websockets

Bei AJAX kann die Verbindung mit Long-Polling aufrecht erhalten werden. Dadurch ist eine Emulation bidirekionaler, strom-basierter Kommunikation möglich. Dies erinnert an die bidirektionale Kommunikation die in Kapitel 4 anhand der Sockets erläutert wurde. Wenn der Client den Server über Ereignisse informieren will, ist das Request / Response-Prinzip ungünstig. Daher wurden 2011 für die

bidirektionale Kommunikation von Web-Applikationen in Echtzeit WebSockets¹⁰ eingeführt. Sie gehen davon aus, dass es eine bestehende TCP-Verbindung zwischen Client- und Server gibt, die für einen bidirektionalen Datenaustausch erweitert (upgrade) wird. Als Adressierung werden ws://... oder wss://... (sichere Variante) verwendet. Das folgende Bild aus [Abts15]¹¹ veranschaulicht den Ablauf:

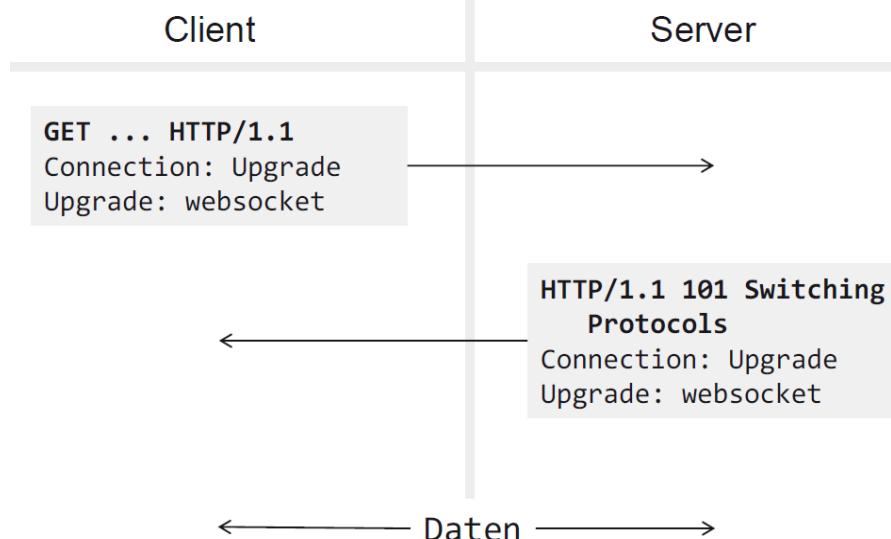


Abbildung 94: Erweiterung einer HTTP-Verbindung für Websocket-Kommunikation

Die Verarbeitung im Web-Browser geschieht mittels Javascript, wie folgendes Bild zeigt:

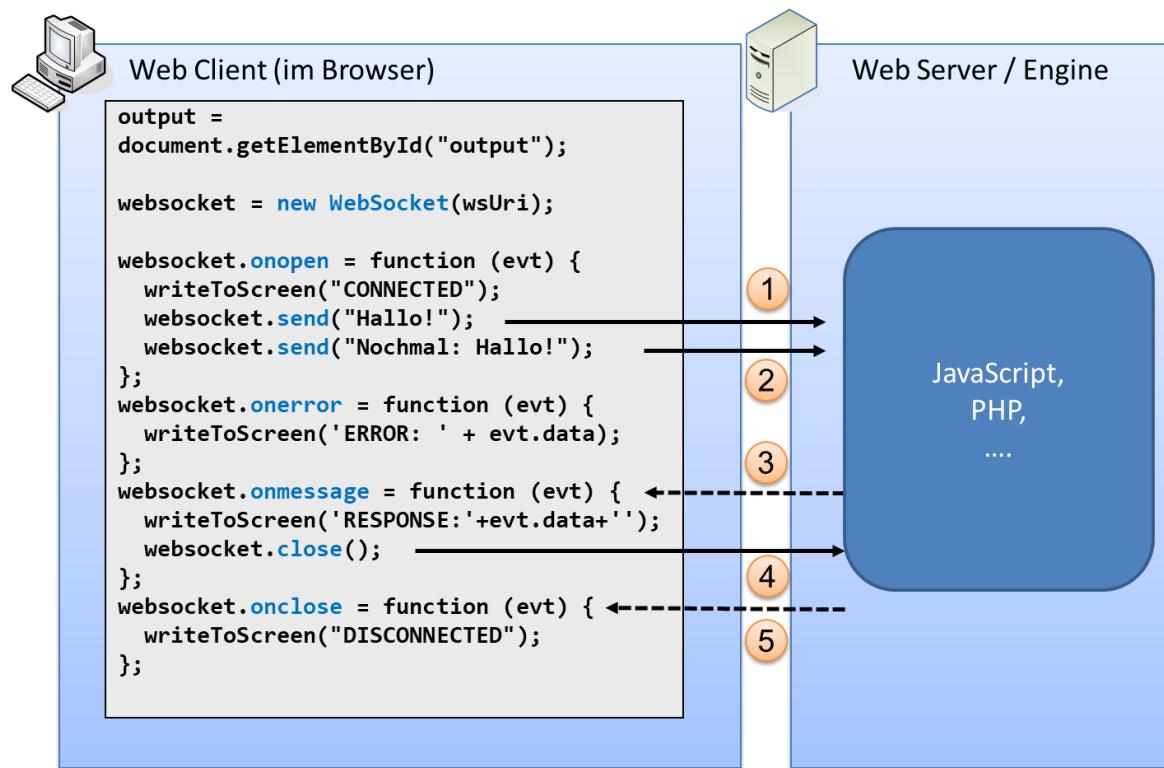


Abbildung 95: Nutzung von Websockets in Javascript

¹⁰ <http://www.ietf.org/rfc/rfc6455.txt>:

¹¹ Aus: Dietmar Abts: Masterkurs Client/Server-Programmierung mit Java: Anwendungen entwickeln mit Standard-Technologien, Springer-Vieweg 2015, 4. Auflage, Kapitel 7 (in Springer-Link erhältlich)

7.8 Web Services

Web Services dienen als Integrationstechnologie dem Datenaustausch zwischen Servern. Sie basieren damit auf bekannten Middleware-Modellen wie DCOM, CORBA, JEE, DCE. Grundlage ist wieder ein Client / Server-Modell, das aber auf unterschiedlichen Implementierungsplattformen realisiert werden kann. Web Services besitzen:

- eine standardisierte Schnittstelle und
- ein standardisiertes Format, in dem Daten über Schnittstelle ausgetauscht werden

und stehen in zwei Ausprägungen zur Verfügung:

- SOAP-basiert: Verwendung eines Austauschprotokolls (analog IIOP, RMI, RPC) zur Kommunikation (Marshaling per XML) (früher Akronym für „Simple Object Access Protocol“)
- RESTful: Kommunikation via HTTP & HTTP-Methoden (GET, POST, PUT, ...)

SOAP Web Services basieren auf drei Protokollen:

Spezifikation	Anwendung
SOAP (W3C)	XML-basiertes Protokoll (von Microsoft / IBM, basiert auf XML-RPC von Microsoft) für den Informationsaustausch in dezentralen, verteilten Umgebungen (Nachrichtenformat). W3C Standard, aktuelle Version: 1.2
WSDL (W3C)	Beschreibung der Schnittstellen und Datenformate des Webservice (Nachrichten, Datentypen, Kommunikationsschema, URL) aktuelle Version: 2.0
UDDI (OASIS)	Spezifiziert den Web Service „Broker“ Dienst („Telefonbuch“), der selbst wieder ein Web Service ist.

Die Web Services Spezifikation ist also zusammenfassend durch:

- ein Protokoll
- eine Schnittstellenbeschreibungssprache
- eine Menge von Basisdiensten

charakterisiert.

Die Grundidee der Web Services zeigt folgendes Bild:

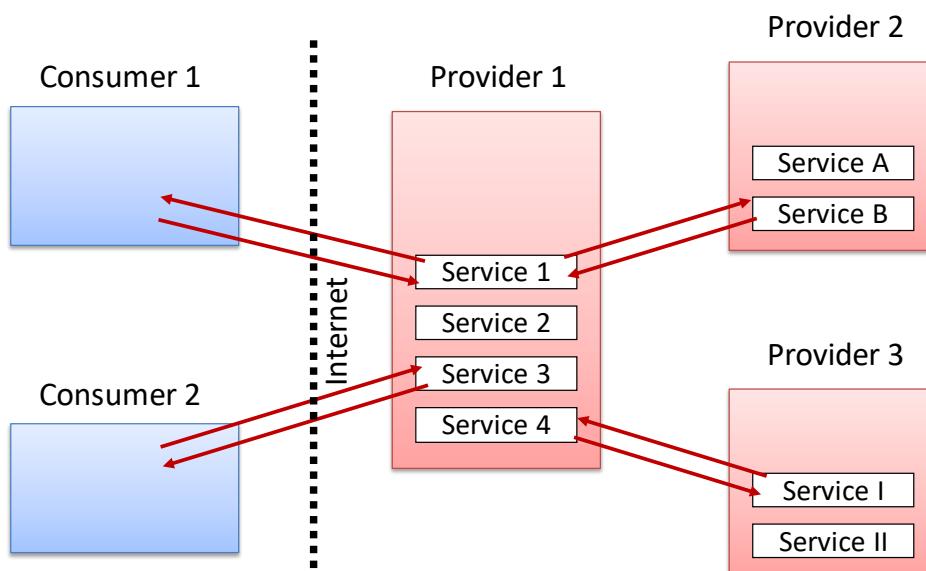


Abbildung 96: Verallgemeinerte Grundidee der Web Services

Die Grundidee der Web Services ist recht einfach: ein **Provider** stellt Funktionalitäten bereit, die von einem **Konsumenten** (als Client) über das Internet (via Applikation) aufgerufen werden können. Zur Realisierung der Funktionen kann der Provider auf Funktionen anderer Provider zurückgreifen. Man spricht in diesem Zusammenhang auch von **Service-Komposition**. Diese im Hintergrund stattfindenden Interaktionen sind für den Konsumenten allerdings nicht sichtbar.

Als praktisches Beispiel für eine solche Komposition sei an dieser Stelle die Zusammenfassung von Internet-Diensten von Southwest Airlines und der Dollar-Autovermietung erwähnt, wodurch die Transaktionskosten um 80% gesenkt werden konnten.

Web Services werden heute, wie aus der Grundidee ersichtlich, vielfach im Zusammenhang mit **lose gekoppelten** verteilten Systemen und Organisationen gesehen, was eine Betrachtung im Zusammenhang mit aktuellen Themen wie **Business Process Outsourcing** (BPO) nahe legt.

Ein Anwendungsbeispiel soll den Einsatz von Web Services erläutern:

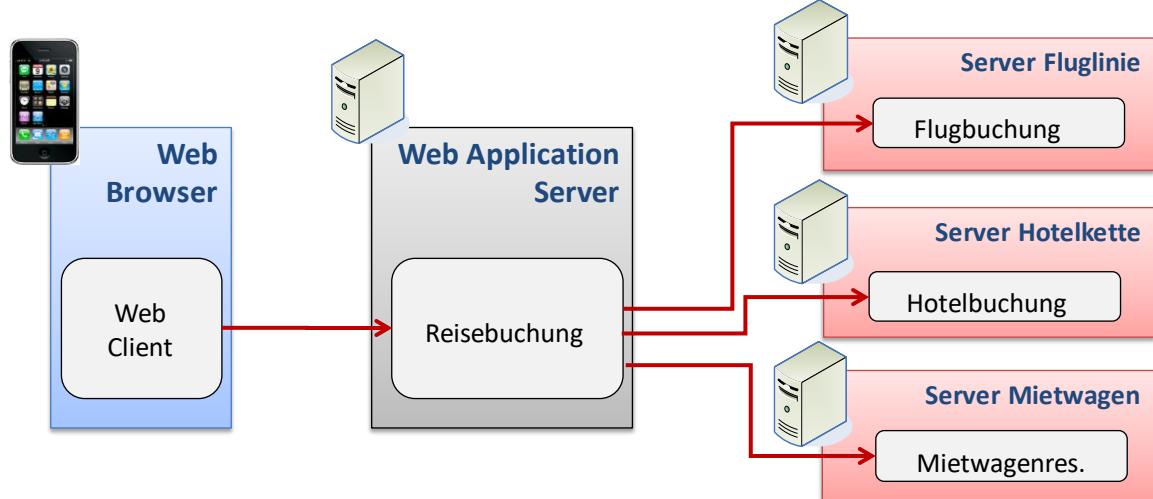


Abbildung 97: Anwendungsbeispiel für mehrere kombinierte Web Services

In der Regel sind die Buchungssysteme von Fluglinien (Lufthansa,...), Hotelketten (Holiday Inn,...) und Mietwagenfirmen (Europcar,...) online. Im Kundenportal geschieht der Datenaustausch zwischen Web-Client und Server per HTML. Dies ist aber ungeeignet für Integration in andere Anwendungen und verhindert eine Automation. Die weitere Verarbeitung muss also einen Datenaustausch in maschinenlesbarer Form ermöglichen. Dies wird durch XML gewährleistet. Damit ist eine einfache, verlässliche und schnelle Anwendungsintegration mittels Web Services ermöglicht. Diese Problemstellung wird auch Enterprise Application Integration (EAI) genannt. Sie wäre auch durch andere Technologien (z.B. CORBA) möglich, Web Services haben sich aber speziell im Bereich Business to Business (B2B) durchgesetzt, zumal die Nutzer und Anbieter leicht und flexibel austauschbar sind.

Der wesentliche Unterschied gegenüber anderen Ansätzen ist im Hinblick auf das sog. Binding zu sehen, also der Anbindung von Clients an Web Services oder die Komposition von solchen Services untereinander. Dieses Binding basiert bei Web Services nicht auf einem gemeinsamen Typsystem, sondern auf Schemata und sog. Contracts für den Nachrichtenaustausch. Die einzelnen Web Services existieren dabei autonom voneinander und es werden keinerlei Anforderungen hinsichtlich einer gemeinsamen Ausführungsumgebung (Betriebssystem, Plattform,...) gestellt. Das Binding ist somit hochgradig dynamisch, d.h. Contracts zwischen Web Services werden dynamisch eingegangen und wieder freigegeben. Web Services sind auf XML zur Schnittstellen und Datenbeschreibung festgelegt. Sie definieren damit:

- das Nachrichtenformat, mit dem Webservices kommunizieren in SOAP
- das Beschreibungsformat, welches Informationen der Webservices beschreibt in WSDL

Webservices besitzen also sowohl eine standardisierte Schnittstelle als auch ein standardisiertes Format, in dem Daten über Schnittstelle ausgetauscht werden.

7.8.1 SOAP für den Nachrichtenaustausch

Der Nachrichtenaustausch von Web Services wird auf Basis des SOAP-Protokolls realisiert. Früher wurde SOAP als Akronym für **S**imple **O**bject **A**ccess **P**rotocol oder **S**ervice **O**riented **A**rchitecture **P**rotocol genutzt. SOAP ermöglicht Aufrufe ähnlich RPCs, also basierend auf einem Request/Response-Prinzip. SOAP-Nachrichten werden in XML beschrieben und sind unabhängig vom Transportprotokoll. Es gibt kein komplexes Objekt- oder Komponentensystem wie z.B. bei CORBA.

Folgende Eigenschaften sind für SOAP wichtig:

- **Portabilität** durch XML-Kommunikation:
 - Plattform-übergreifend, implementierungsunabhängig
 - Besonders geeignet für die lose Kopplung von Applikationen über das Internet
 - Marshalling von Nachrichten via XML
 - W3C Standard, aktuelle Version: 1.2
- **Modularität:**
 - Realisierung **eigener** Webservices
 - Anknüpfung / Integration **externer / existierender** Services

Das folgende Bild zeigt die Gesamtarchitektur von Web Services mit den Transportprotokollen unten:

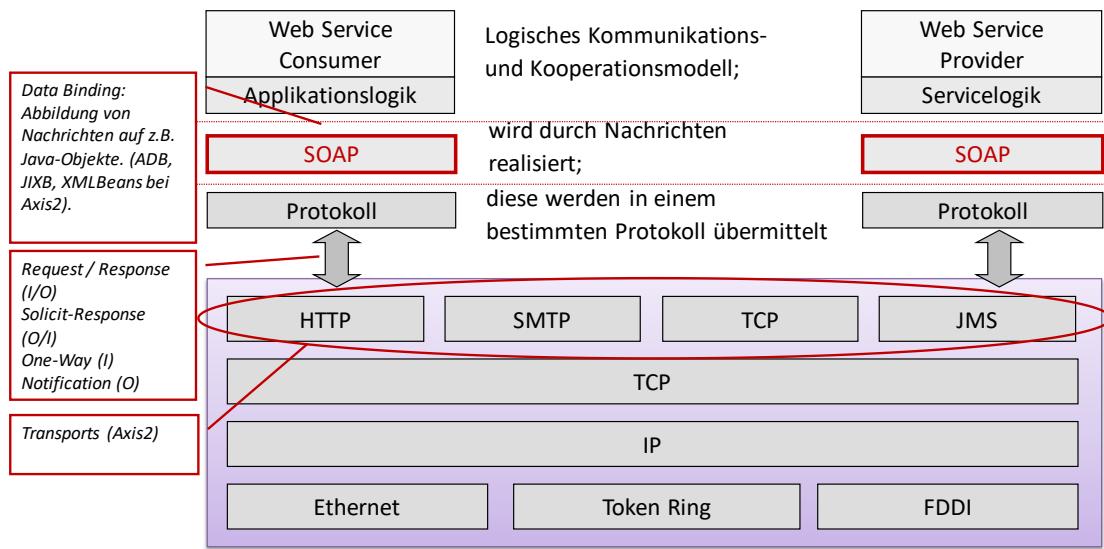


Abbildung 98: SOAP-Kommunikationsmodell

Der prinzipielle Aufbau einer SOAP-Nachricht ist in folgendem Bild zu sehen:

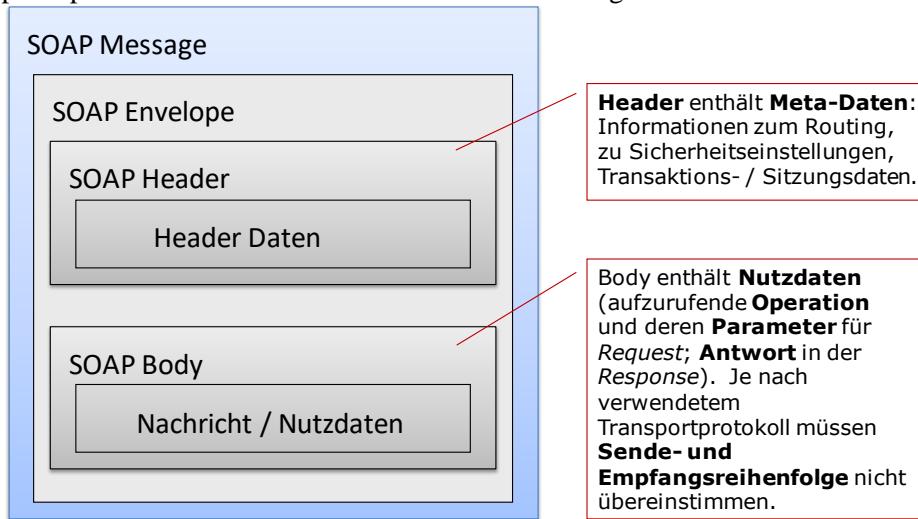


Abbildung 99: Aufbau einer SOAP-Nachricht ohne Anhang

Der Header muss nicht vorhanden sein. Er legt fest, welcher Namespace verwendet wird und charakterisiert die Verarbeitung der Nachricht. Häufig werden Daten innerhalb einer Nachricht übertragen, ohne dass eine zugehörige Server-Funktion sichtbar ist. Man kann aber auch (ähnlich RPC und RMI) einen Dienst beim Server anbieten und diesen als Web Service mit explizitem Funktionsaufruf nutzen. Für die zugehörigen SOAP-Nachrichten sind folgende Eigenschaften vereinbart:

- Im Body gibt es nur ein Element, das als Kindelemente die Parameter der Operation enthält (Vergl. Funktions- bzw. Methodenargumenten bei RPC bzw. RMI)
- Die Kindelemente enthalten jeweils den Namen des zugehörigen Parameters der Operation

Es gibt nicht *die* SOAP Programmierung, sondern eine Reihe von Toolkits, die SOAP unterstützen. SOAP Implementierungen existieren für viele Programmiersprachen z.B. JAVA, Perl, Python, C++, PHP, C#. Web Service-Unterstützung ist auch für C/C++ z. B. in Apache Axis C++ verfügbar. Es gibt Bindungen an unterschiedliche Transportprotokolle wie HTTP(S), SMTP, MIME (E-Mail), TCP (d.h. SOAP direkt), JMS (Java Message Service). Die Nutzung von E-Mail als Transport ist manchmal der einzige Weg, Firewall- und damit Unternehmensgrenzen zu überwinden. Zwei Interaktionsmuster gibt es zur Verarbeitung des Bodies:

- RPC: Methodenaufruf mit typisierten Daten
- Document: untypisierte Nachrichten

Die folgende Übersicht zeigt dazu Details:

	Encoded	Literal
	Regeln zur Serialisierung der Daten werden über ein Attribut encodingStyle definiert.	Serialisierung der Daten wird über Regeln in einem XML-Schema definiert.
RPC Service-Implementierung interpretiert Body als Darstellung eines Methoden-Aufrufs.	RPC encoded <pre><soap:envelope> <soap:body> <method> <xsi:type ="xs:int">5</x> </method> </soap:body> </soap:envelope></pre>	RPC literal <pre><soap:envelope> <soap:body> <method> <x>5</x> </method> </soap:body> </soap:envelope></pre>
Document Der Body kann beliebiges XML enthalten.	Document encoded ...	Document literal <pre><soap:envelope> <soap:body> <xElement>5</xElement> </soap:body> </soap:envelope></pre>

Folgende Grenzen und Einschränkungen sind bei SOAP zu beachten:

- SOAP enthält **keine** Definition einer bidirektionalen Kommunikation, dies muss, wenn benötigt, muss von den Applikationen oder dem Toolkit realisiert werden
- SOAP erzwingt **keine Objektorientierung** der Applikation; Ob eine Nachricht an ein „echtes“ Objekt weitergeleitet wird, hängt von der Implementierung ab
- SOAP definiert **keine „höherwertigen“ Dienste**, wie Lebensdauerverwaltung oder Aktivierung von Objekten (s. CORBA); Auch die Service-Suche über UDDI = Universal Description Discovery and Integration hat sich nicht durchgesetzt.
- Es gibt **kein einheitliches SOAP-API**

SOAP wird in sogenannten SOAP Engines verarbeitet, die die Interpretation und den endgültigen Aufruf im Server erledigen. Dies zeigt folgendes Bild:

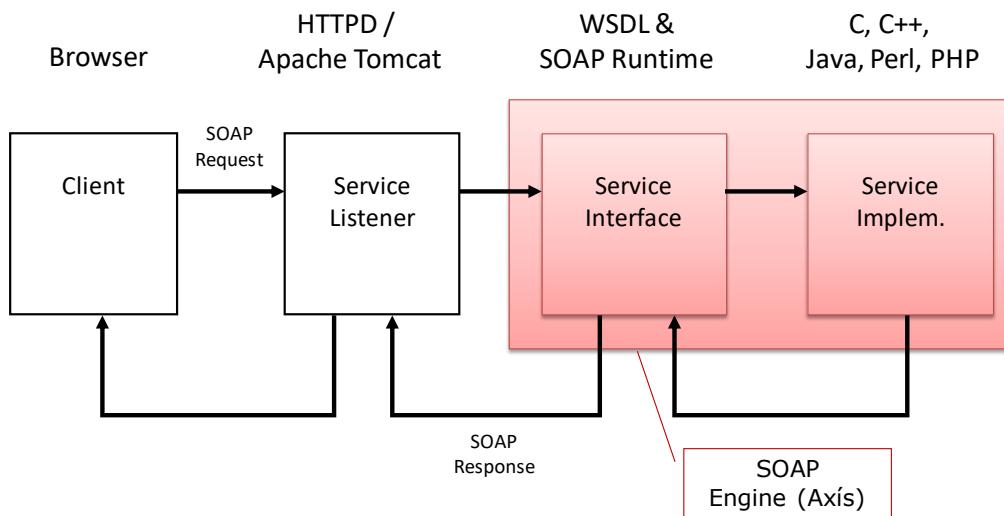


Abbildung 100: SOAP-Nachrichten mittels SOAP Engine Axis

Die wichtigste SOAP Engine Apache Axis2 hat folgende Eigenschaften:

- Apache EXtensible Interaction System
- Open Source als Teil des Apache-Projekts
- Nachfolgeprojekt von Apache SOAP,
- basiert auf IBM-Entwicklung
- Zentrale Idee: konfigurierbare Ketten von sog. *Message Handlern*
- Sehr stabil, aber nicht besonders performant
- Einbindung in IDEs: z.B. Netbeans, Eclipse WTP, z.B. für WSDL Unterstützung. Hier hat man meist die Optionen:
Generierung der WSDL-Schnittstellen Bottom-Up (aus dem Java-Code)
Generierung von Java-Code aus der WSDL (Top-Down)

Weitere Engines sind Apache CXF / XFire, cSOAP / gSOAP, GLUE, Microsoft .net. Das Apache-Handler-Konzept funktioniert folgendermaßen:

- Jede SOAP-Nachricht wird in einen *Message Context* eingebettet.
- Dieser Kontext kann von einem Handler bearbeitet werden.
- Mehrere Handler können zu einer Chain zusammengefasst werden.
- In Axis gibt es drei vordefinierte Chains:
 - Transport Chain: Übertragung und Empfang von Daten zwischen Sender und Empfänger.
 - Global Chain:
 - Aktionen, die jeden Service betreffen
 - z.B. Sicherheit (Authentifizierung, Autorisierung), Sessions, ...
 - Service Chain:
 - Aktionen, die jeden Service betreffen
 - z.B. Fehlerkorrektur von Eingaben, Umwandlung von Daten, ...
- Axis Runtime kann sowohl auf der Caller- als auch auf der Server-Seite laufen

Chains können sowohl im Request- als auch im Response-Flow instanziert werden:

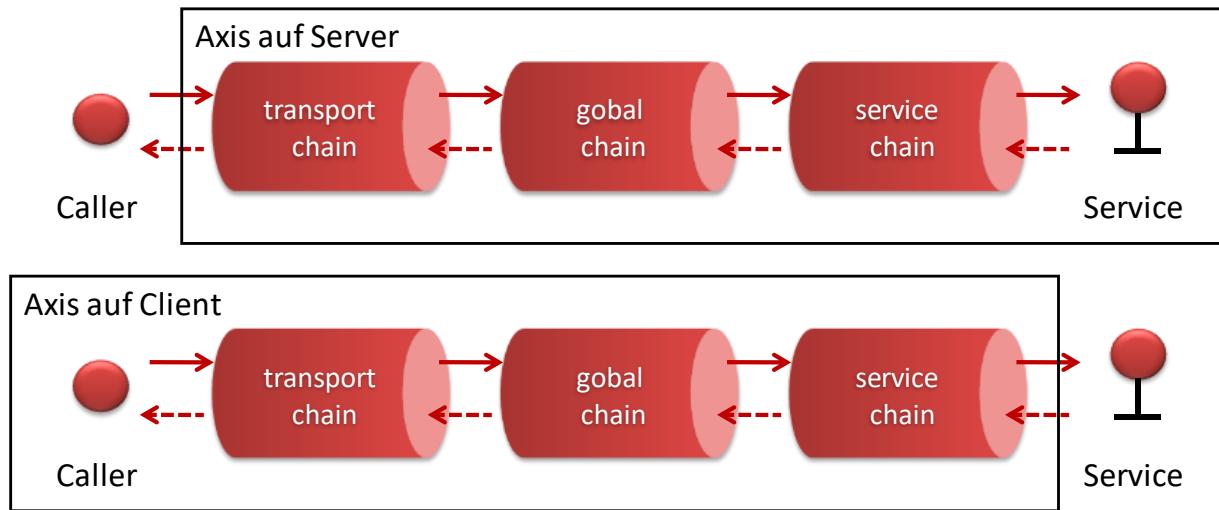


Abbildung 101: Handler-Konzept der SOAP Engine Axis

Die Entwicklung mit Axis2 geschieht folgendermaßen:

- Konfiguration:
 - Integration der Runtime als **Servlet** in Tomcat:
„Applikationsserver im Applikationsserver“
- Entwicklung:
 - Implementierung der Services in **Java**,
alternativ C++ (dann aber mit Axis C++)
 - Deployment von Web-Services:
 - Spezielles Archive-Format (.aar):
 - hot deployment,
 - hot update,
 - undeploy
 - über Service-Deskriptor (services.xml)

Axis2 wird meist in IDEs mit Webservice Toolkits eingesetzt. Diese erweitern eine vorhandene Entwicklungsumgebung / Programmiersprache zur Entwicklung von Web Service Anwendungen (Server und/oder Client). Für nahezu jede Programmiersprache bzw. Umgebung existiert ein Web Service-Toolkit

- Java erlaubt die Verwendung von Annotations im Java Code z.B.
@WebService und
@SOAPBinding(style=Style.RPC)
- Microsoft .NET - Web Services sind fest in die .net Plattform integriert
Steuerung der Service-Generierung ebenfalls via Annotations
- Netbeans oder Eclipse: Erstellung von Web Services via Axis2 Plugin / Feature
Einbindung von Axis / Axis2 & Build via Ant Task¹²

Die prinzipielle Spezifikation der Dienstnutzung und der entsprechenden Schnittstellen erfolgt über WSDL.

7.8.2 WSDL – Web Service Description Language

Um zu wissen, wie ein Service benutzt werden kann, muss der Aufrufer irgendeine Art von Dienstbeschreibung kennen. Dazu werden formale, maschinen-verarbeitbare Beschreibungen mittels

¹² Axis2 kann in aktuellen Versionen von Netbeans nicht verwendet werden, da Oracle die Nutzung des Glassfish-Servers bevorzugt.

einer Dienstbeschreibungssprache (allgemein: Interface Definition Language IDL) eingesetzt. Darin wird die Dienstnutzung aus Sicht der Schnittstellen beschrieben. Folgende IDLs sind bereits bekannt:

- RPC (Sun-Version und OSF DCE)
- CORBA IDL
- Java RMI (mit Java als IDL)

Bei der Nutzung von Web Services wird WSDL (Web Service Description Language) = **Vertrag**¹³(Schnittstellenbeschreibung) zwischen Client und Server genutzt.

WSDL ist eine XML-Sprache und beschreibt Dienste mit folgenden Details:

- **Operationen** des Dienstes ⇒ **Port Types**
- **Parameter** der Operationen:
 - Struktur des Requests
 - Struktur der Response
 - Verweis auf XML Schema bzgl. Typdefinitionen
- **Protokoll** des Dienstes ⇒ **Bindings**:
 - HTTP, SMTP etc.
 - **Lokation** des Dienstes: Adresse (URI / URL)

Es erfolgt keine semantische Dienstbeschreibung. Dienste können also nicht automatisch aufgelöst werden (Suche nach einem Reisebuchungsservice). Es erfolgt keine automatische Verschaltung von Diensten. Der interne Aufbau einer WSDL-Beschreibung ist in folgendem Bild zu sehen:

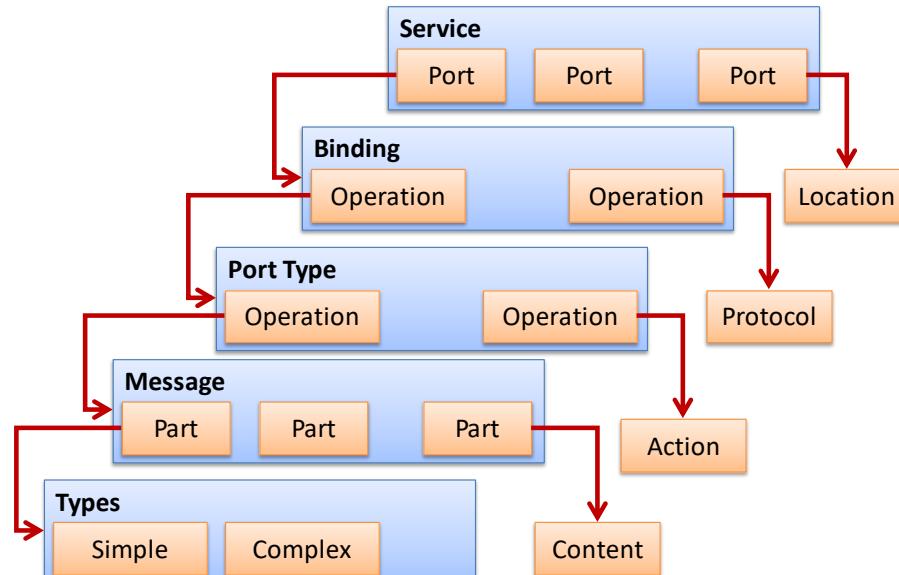


Abbildung 102: Struktur eines WSDL-Dokuments

Zur Beschreibung der abstrakten Fähigkeiten dient das Interface (früher portType), zur Beschreibung der aktuellen Implementierung das binding-Element. Die Bindung wird mit einer Adresse kombiniert. Man erhält man einen konkreten Endpunkt, über den die Implementierung angesprochen werden kann.

WSDL-Sprachelemente sind:

- interface: die abstrakte Definition einer Schnittstelle (entspricht einem Java Interface)
- message: eine Menge von Parametern, die von den Operationen verwendet werden
- Typen: alle Datentypen, die in den Nachrichten verwendet werden
- binding: beschreibt, wie die Elemente eines abstrakten Interfaces in eine konkrete Repräsentation umgewandelt werden (Kodierungsschema, SOAP über HTTP)

¹³ Eines der vielen Paradigmen der Informatik lautet Design by Contract und hat seinen Ursprung im Prinzip der Kapselung, bei der Module über ihre Schnittstelle angesprochen werden, wobei ihre Implementierung verborgen bleibt (vergleiche h/cpp-Datei bei C/C++).

Ein Service ist eine Sammlung von Endpunkten <endpoint>. Ein <endpoint> hat eine abstrakte Definition (<interface>) und eine konkrete Realisierung (<binding>). Interfaces sind Sammlungen von abstrakten Operationen, etc..

Das folgende Bild zeigt die anschauliche Darstellung eines einfachen Temperaturkonvertier-WebService (Celsius ↔ Fahrenheit) in Netbeans (siehe Anhang)¹⁴:

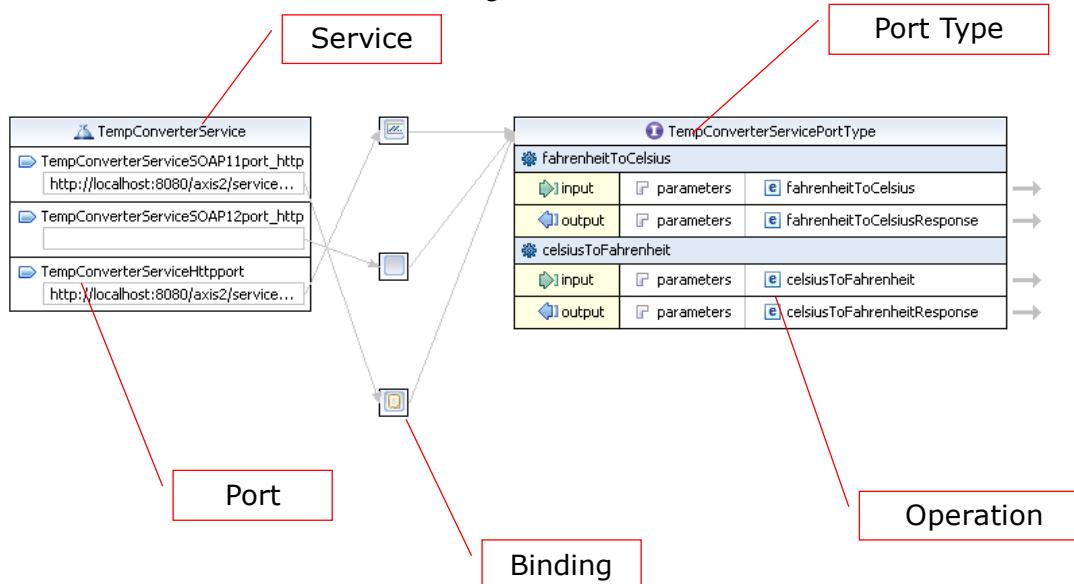


Abbildung 103: Temperaturkonvertier-Web-Service-WSDL in Netbeans

Die Schnittstellen-Beschreibung ist ein XML-Dokument, das aus verschiedenen Teilen besteht. Zunächst werden die notwendigen Datentypen (Types) und die Nachrichten (Messages) an den Service definiert. Die WSDL in unserem Beispiel hat folgenden Aufbau:

```
<wsdl:definitions xmlns:axis2=" http://ws.vs.hsos.de"
    ... targetNamespace="http://ws.vs.hsos.de">
    <wsdl:documentation> ... </wsdl:documentation>
    <wsdl:types>
        <xss:schema xmlns:ns="http://ws.vs.hsos.de" ...>
            <xss:element name="celsiusToFahrenheit">
                <xss:complexType>
                    <xss:sequence>
                        <xss:element minOccurs="0" name="celsius"
                            type="xss:double"/>
                    </xss:sequence>
                </xss:complexType>
            </xss:element>
            <xss:element name="celsiusToFahrenheitResponse">... </xss:element>
        </xss:schema>...
    </wsdl:types>
    <wsdl:message name="fahrenheitToCelsiusMessage">
        <wsdl:part name="part1" element="ns0:fahrenheitToCelsius" />
    </wsdl:message>
    <wsdl:message name="fahrenheitToCelsiusResponse">
        <wsdl:part name="part1" element="ns0:fahrenheitToCelsiusResponse" />
```

¹⁴ Die graphische Darstellung wird durch Anwahl des Tabs Design im Eclipse WTP sichtbar. Sie ist übersichtlicher als die Anzeige des XML-Dokumentes und zeigt die Struktur des Service:

- Ein **Service** kann über verschiedene Ports verfügen.
- Jeder Port kann mit einem **Binding** versehen werden.
- Von einem **Port Type** werden verschiedene Operationen unterstützt.
- In einer Operation werden **Input** und **Output** unterschieden.

```

</wsdl:message>
...
<wsdl:definitions xmlns:axis2="http://ws.vs.hsos.de"
... targetNamespace="http://ws.vs.hsos.de">
...

<wsdl:portType name="TempConverterServicePortType">
    <wsdl:operation name="fahrenheitToCelsius">
        <wsdl:input ... message="axis2:fahrenheitToCelsiusMessage"
        wsaw:Action="urn:fahrenheitToCelsius" />
        <wsdl:output message="axis2:fahrenheitToCelsiusResponse"/>
        <wsdl:fault message="axis2:fahrenheitToCelsiusFault"
        name="fahrenheitToCelsiusFault" />
    </wsdl:operation>
    <wsdl:operation name="celsiusToFahrenheit">
        <wsdl:input ...
        message="axis2:celsiusToFahrenheitMessage"
        wsaw:Action="urn:celsiusToFahrenheit" />
        <wsdl:output message="axis2:celsiusToFahrenheitResponse"
        />
        <wsdl:fault message="axis2:celsiusToFahrenheitFault"
        name="celsiusToFahrenheitFault" />
    </wsdl:operation>
</wsdl:portType>
<wsdl:definitions xmlns:axis2="http://...targetNamespace="http://...">
    ...
    <wsdl:binding name="TempConverterServiceSOAP12Binding"
    type="axis2:TempConverterServicePortType">
        <soap:binding transport="..." style="document" />
        <wsdl:operation name="fahrenheitToCelsius">
            <soap:operation soapAction="urn:fahrenheitToCelsius"
            style="document" />
            <wsdl:input><soap12:body use="literal"
            /></wsdl:input>
            <wsdl:output><soap12:body use="literal"
            /></wsdl:output>
            <wsdl:fault name="fahrenheitToCelsiusFault">
                <soap12:fault use="literal"
                name="fahrenheitToCelsiusFault" />
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="TempConverterService">
        <wsdl:port name="..." binding="axis2:TempConverterServiceSOAP11Binding">
            <soap:address
            location="http://...:8080/.../services/TempConverterService" />
        </wsdl:port>...
    </wsdl:service>
</wsdl:definitions>

```

Der Service kann im Web-Browser getestet werden, indem man das sogenannte REST-Interface anspricht (z.B. in Netbeans mit Test Operation in Browser im Unterordner Axis2 Web Services). Die aufzurufende URL ist:

`http://localhost:8084/axis2/services/TempConverterService/celsiusToFahrenheit?celsius=0.0`

Das Ergebnis lautet:

```
<ns:celsiusToFahrenheitResponse>
    <ns:return>32.0</ns:return>
</ns:celsiusToFahrenheitResponse>
```

Ein Java-Client, der den Web Service nutzt, könnte z.B. folgendermaßen aussehen:

```
package de.hsos.vs.ws;
public class TempConverterClient {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        try {
            de.hsos.vs.ws.TempConverterService service;
            service = new de.hsos.vs.ws.TempConverterService();
            de.hsos.vs.ws.TempConverterServicePortType port;
            port = service.getTempConverterServiceHttpSoap12Endpoint();

            double d = 0;
            double result = port.celsiusToFahrenheit(d);
            System.out.println(d + " Celsius = " + result + " Fahrenheit");
            result = port.fahrenheitToCelsius(d);
        } catch (Exception ex) {
            System.out.println("exception" + ex);
            ex.printStackTrace();
        }
    }
}
```

In WSDL werden zur Spezifikation der Struktur der ausgetauschten Daten meist XML Schemata eingesetzt. Um bei der Definition möglichst flexibel beliebige Schemabeschreibungen verwenden zu können, werden Namespaces eingesetzt. Damit werden Namen von Datentypen in einen lokalen Kontext gesetzt.

7.8.3 RESTful Web Services

Ein schlankerer Zugriff auf Informationen wird durch REST (Representational State Transfer) (ähnlich CGI) ermöglicht. Einzelne Funktionen werden direkt über eine URL mit direkter Parameterübergabe ohne WSDL aufgerufen. Die Mächtigkeit und Flexibilität der Web Services werden damit natürlich nicht erreicht, aber es ist für statische Schnittstellen eine einfache Zugriffsmöglichkeit gegeben.

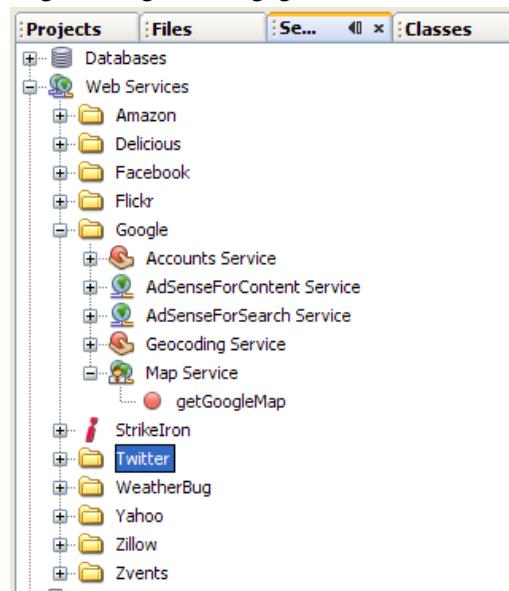
Öffentlich nutzbare Web Services werden heute zumeist als *RESTful* Web Services eingebunden. REST ist ein leicht-gewichtiger Ansatz zur Implementierung von Web-Services: kein XML → weniger Overhead.

Netbeans beinhaltet Referenzen zu Web-Services von Google, Amazon, Twitter,... zum Einbinden der Services per Drag & Drop.

Mobile Anwendungen (Apps) beherrschen normalerweise kein XML und werden daher nur mittels REST an externe Dienste angebunden.

Die Kommunikation erfolgt durch Austausch des Zustands der Ressourcen. Die Ressourcen werden per URI adressiert. Der Zugriff auf Ressourcen erfolgt über Standard-Schnittstellen (z.B. http-Methoden: GET, POST,...)

Im Server werden diese auf CRUD-Operationen abgebildet:



CRUD-Operation	http-Methode	Funktion
Create	POST	Erzeugen einer Ressource
Read	GET	Lesen einer Ressource
Update	PUT	Aktualisierung einer Ressource
Delete	DELETE	Löschen einer Ressource

Zum Schluss lohnt sich noch einmal eine Einordnung anhand der W3C Web Service-Architektur:

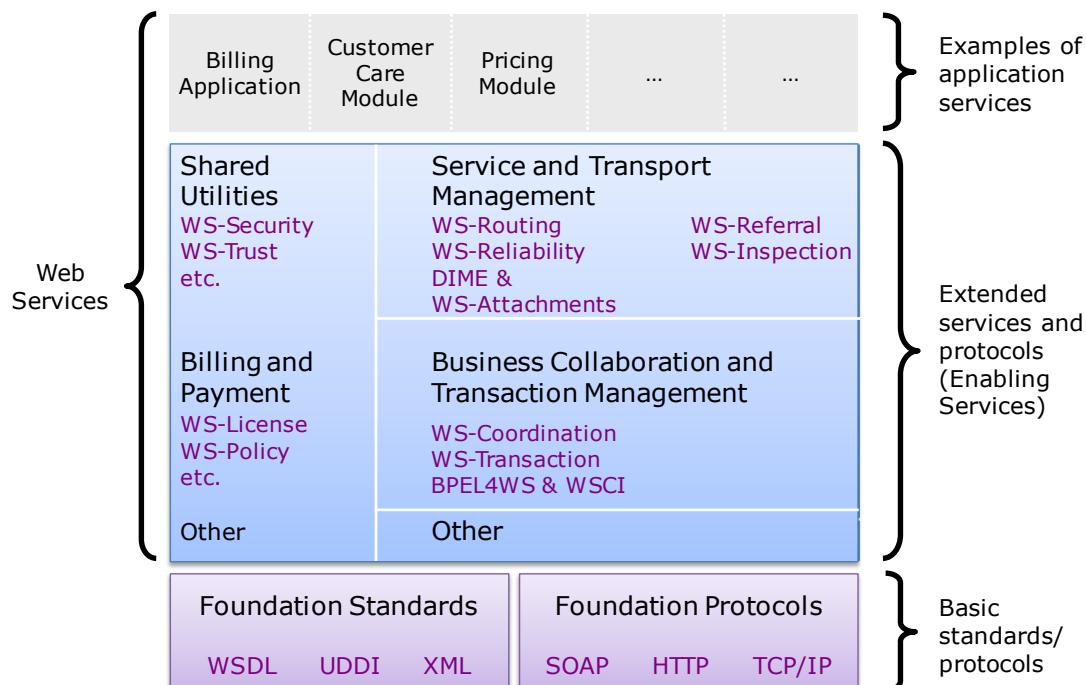


Abbildung 104: Web Services: Web Service Architecture (W3C)

	WS-* / SOAP	REST
Arbeitsweise	Funktions-orientiert (‘XML-RPC’)	Ressourcen-orientiert (Ressourcen = Web Objekte)
Java-Toolkits	JAX-WS	JAX-RS
Trägerprotokolle	HTTP (SMTP, JMS, ...)	HTTP
Frameworks	Security, Transactions, Prozesse (BPEL),...	%
Meta-Daten	WSDL	%
Anwendungsbereich	Selbstbeschreibende, lose gekoppelte verteilte Systeme	Skalierbarer Zugriff auf große Datenmengen

7.8.4 UDDI

Um Web Services anhand von WSDL-Beschreibungen weltweit und organisationsunabhängig nutzen zu können, wurde ein Registrierungsdienst auf Basis von UDDI entwickelt. UDDI hat sich aber aufgrund von Vertrauensproblemen trotz Beteiligung unterschiedlicher Organisationen und großer Initiatoren nicht durchgesetzt und öffentliche UDDI-Knoten wurden 2006 abgeschaltet. UDDI wird nur im kleineren / kontrollierten Rahmen eingesetzt z.B. für:

- das Management von Services innerhalb einer großen Firma Vertrauen ist innerhalb der Firma, also im Intranet, gegeben
- B2B Internet Marktplätze, bei denen ein Zutritt nur nach vertraglicher Bindung und Prüfung erfolgt.

7.8.5 Nachteile von Web Services

Web Services haben sich wegen ihrer Vorteile rasch verbreitet, aber es gibt auch Nachteile:

- Der Adressraum, Referenzen auf Objekte und Sitzungskontexte müssen selbst verwaltet werden, da die Kommunikation zustandslos erfolgt. Nach jedem Request/Response wird die Verbindung abgebaut. Teilweise werden zum Nachstellen der Sitzungs-IDs Cookies verwendet.
- XML bläht den Datenumfang auf. Dies kann bei Netzen mit reduzierter Dienstgüte (Quality of Service) oder volumenabhängiger Abrechnung (Mobilfunk) störend sein.
- Der Entwickler muss sich selbst um Sicherheit und Verschlüsselung kümmern. Firewall-Hersteller ermöglichen daher inzwischen eine Filterung und Interpretierung des Verkehrs über Port 80 und 8080. Abhilfe schafft das Tunneln in https-Verbindungen.

7.8.6 Vergleich zwischen CORBA und Web Services

Die folgenden Tabellen ermöglichen einen Vergleich der Web Services mit den in Kapitel 6 vorgestellten Ansätzen von CORBA:

	CORBA	Web Services
Objekte-Ebene	IDL	WSDL
Service-Ebene	CORBA Common Object Services	UDDI
Nachrichten-Ebene	CORBA stubs / skeletons & DII/DSI	SOAP messages
Daten-Ebene	CDR binary encoding	XML UTF encoding
Träger-Ebene	IIOP/GIOP	HTTP (SMTP and others)
Netzwerk-Ebene	TCP/IP (and others)	TCP/IP
Typsystem	IDL - static and runtime checks	XML Schemas - runtime checks only
Marshalling Syntax	CDR - binary	XML - UTF
Zustandsbehaftete Verbindungen	Stateful or Stateless	Stateless
Registratur	Interface Repository and Implementation Repository	UDDI/WSDL
Service Discovery	Naming and Trading	UDDI
Sicherheit	Signature CORBA security service & IIOP over SSL	HTTPS, XML
Firewall Tunneling	IIOP Proxies, Bi-directional, IIOP, Spec. being revised	Layered over HTTP, works HTTP proxies too

7.9 Ausblick

Web Services können als konsequente Erweiterung der traditionellen Webtechnologien gesehen werden. Ihre Programmierung ist vielfach als Projekttyp integriert in IDEs (meist als RESTful Web Services). Perspektivisch erfolgt der Ausbau in Richtung folgender Ergänzungen:

- Sicherheit: WS-Security
- Anwendungen: B2B, B2C
- Web Ontologien: Formalisierung von Diensten und deren Beziehungen untereinander
- Cloud / Grid Computing: Rechnerleistung wie im Strom- oder Wassernetz zur Verfügung stellen

Anhang

Anhang A.1 Index

- AJAX - Asynchronous JavaScript & XML 147
- Anforderungen an die Programmierung verteilter Systeme 23
- Asynchrone Kommunikation* 11
- Begriffsdefinitionen 10
- Client-Server 12
- CORBA 73
- Datenorientierte Programmierung verteilter Systeme 24
- Ein/Ausgabe-Multiplexing 42
- Funktionen zur Socketprogrammierung 25
- HTTP 110
- Java RMI 98
- Java Server Pages 126
- Java Servlets 114
- Java-Applets 133
- JavaScript 131
- JSON 153
- JSP 126
- Multi-Tier-Architekturen* 12
- OSI-Referenzmodell 14
- Peer-to-Peer 13
- php 129
- Port-Nummern 21
- Remote Method Invocation 98
- Remote Procedure Calls 48
- RPC 48
- `select()`-Funktion 42
- SOAP 156
- Socketadressen 24
- Socketfunktionen 27
- Socketprogrammierung 25
- Sockets 24
- Synchrone Kommunikation* 11
- TCP/IP-Protokollfamilie 17
- URI 112
- URL 112
- Web Service Description Language 160
- Web Services 138
- Websockets 153
- WSDL 160
- XML 139
- XML Schema 141
- XML-Parser 143
- XSL - Extensible Stylesheet Language 146