Peter Mandl

Masterkurs Verteilte betriebliche Informationssysteme

Aus dem Programm Wirtschaftsinformatik

Weitere Titel des Autors:

Grundkurs Betriebssysteme
von P. Mandl

Grundkurs Datenkommunikation
von P. Mandl, A. Bakomenko und J. Weiß

Grundkurs Wirtschaftsinformatik
von D. Abts und W. Mülder

Anwendungsorientierte Wirtschaftsinformatik
von P. Alpar, H. L. Grob, P. Weimann und R. Winter

Grundkurs Verteilte Systeme
von G. Bengel

Masterkurs Parallele und Verteilte Systeme
von G. Bengel, C. Baun, M. Kunze und K.-U. Stucky

Java-Grundkurs für Wirtschaftsinformatiker
von K.-G. Deck und H. Neuendorf

Geschäftsprozesse realisieren
von H. Fischer, A. Fleischmann und S. Obermeier

Grundkurs Geschäftsprozess-Management
von A. Gadatsch

Grundkurs IT-Projektcontrolling
von A. Gadatsch

Masterkurs IT-Management
von J. Hofmann und W. Schmidt (Hrsg.)

Business Intelligence – Grundlagen und praktische Anwendungen
von H.-G. Kemper, W. Mehanna und C. Unger

Informationssysteme im Personalmanagement
von S. Strohmeier

www.viewegteubner.de

Peter Mandl

Masterkurs Verteilte betriebliche Informationssysteme

Prinzipien, Architekturen und Technologien

Mit 196 Abbildungen

STUDIUM

VIEWEG+
TEUBNER

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
<http://dnb.d-nb.de> abrufbar.

Höchste inhaltliche und technische Qualität unserer Produkte ist unser Ziel. Bei der Produktion und Auslieferung unserer Bücher wollen wir die Umwelt schonen: Dieses Buch ist auf säurefreiem und chlorfrei gebleichtem Papier gedruckt. Die Einschweißfolie besteht aus Polyäthylen und damit aus organischen Grundstoffen, die weder bei der Herstellung noch bei der Verbrennung Schadstoffe freisetzen.

1. Auflage 2009

Lektorat: Sybille Thelen | Andrea Broßler

Vieweg+Teubner ist Teil der Fachverlagsgruppe Springer Science+Business Media.
www.viewegteubner.de

Umschlaggestaltung: KünkelLopka Medienentwicklung, Heidelberg
Druck und buchbinderische Verarbeitung: MercedesDruck, Berlin
Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.
Printed in Germany

ISBN 978-3-8348-0518-8

Vorwort

In den 80er und 90er Jahren wurde die objektorientierte Softwareentwicklung vor allem durch das Marketing einiger Produkthersteller als *die* Lösung für alle Probleme der Softwareentwicklung dargestellt. Parallel dazu entwickelten sich die Konzepte und Technologien für verteilte Systeme. Anfangs musste man die Kommunikation verteilter Anwendungen noch auf Basis von Sockets oder einer ähnlichen Transportzugriffsschnittstelle programmieren. Etwas später kam dann der Remote Procedure Call (RPC) hinzu. Die meisten Probleme für Entwickler waren in der Kommunikationsschicht zu lösen. Die eigentliche Anwendungslogik blieb oft auf der Strecke.

Als schließlich die Object Management Group (OMG) Anfang der 90er Jahre CORBA standardisierte, prophezeite man, mit diesem Standard endlich vernünftige objektorientierte Software auch in einer verteilten Umgebung entwickeln und betreiben zu können. Mehrere Hersteller entwickelten CORBA-Plattformen und verkauften diese als die Lösung aller Probleme. Die Aufgabe von Marketing- und Vertriebsabteilungen ist es nun einmal, Produkte so darzustellen, dass sie gekauft werden.

Lange dauerte es nicht und da bemerkte man, dass die Netzwerke und auch die Serversysteme bei vielen kleinen Methodenaufrufen nicht mithalten konnten. Man brauchte - wie man zu sagen pflegte - etwas „grobkörnigere“ Methoden, um die Ressourcen zu schonen. Im Zuge der Verbreitung von Java kam daher auch eine neue Technologie auf den Markt, die als Komponententechnologie bezeichnet wurde und in Enterprise Java Beans (EJB) mündete. Nun waren aber alle technischen Schwierigkeiten gelöst, dachte man. Dem war natürlich wieder nicht so. Auch die Komponentenentwicklung brachte ihre Probleme mit sich. Der nächste Hype kam mit der XML-Technologie und in den letzten Jahren mit serviceorientierten Architekturen und als konkrete Ausprägung dieser mit Webservices. Viele Hersteller beteiligten sich an diesem Themenkomplex.

Wenn auch die meisten Konzepte und Technologien der letzten Jahrzehnte viele Fortschritte für die Softwareentwicklung brachten, so kommen doch immer noch ständig Verbesserungen und auch neue Technologien auf uns zu. Die stabilen und gewinnbringenden aus den im Versuchsstadion befindlichen Technologien herauszufiltern, ist eine komplizierte Aufgabe, der man sich heute zunehmend stellen

muss. Hier ist der Techniker, meist der Informatiker oder im betriebswirtschaftlichen Umfeld der Wirtschaftsinformatiker gefragt, die richtige Technologie- und Produktauswahl für seine Umgebung zu treffen und dies ist bei der kurzen Halbwertzeit keine einfache Aufgabe, die auch Praktiker immer wieder vor große Herausforderungen stellt.

Wirtschaftsinformatiker, Systemarchitekten und Softwareentwickler im Umfeld betrieblicher Steuerungs- und Informationssysteme müssen heutzutage eine Fülle von Regeln anwenden, um gute Anwendungssysteme konzipieren und entwickeln zu können. Das pure Wissen über Softwaretechnik und wie man intelligente Klassenmodelle erstellt, reicht für die Umsetzung eines größeren Projekts bei Weitem nicht aus. Man muss sich auch mit Betriebssystemen, Datenbanken, Netzwerken, Kommunikationsprotokollen, Middleware, Objektorientierung, Programmiersprachen, Nebenläufigkeitsproblemen usw. auskennen und das Zusammenspiel dieser komplexen Themenfelder beherrschen.

Dieses Buch soll dazu beitragen, dass der Themenkomplex etwas transparenter und greifbarer wird. Wichtige Prinzipien, Softwarekonzepte, Architekturen und Technologien zur Unterstützung verteilter Systeme werden erörtert, und zwar im Speziellen für betriebliche Informationssysteme (in Abgrenzung zu technischen Anwendungssystemen wie Anlagensteuerungen, Robotersystemen, Echtzeitsystemen usw.). Vor allem werden auch praktische Gesichtspunkte hinterfragt.

Viele Aspekte verteilter Systeme sind interessant und nicht alle können in diesem Buch behandelt werden. Es wird Wert auf die Erläuterung praktisch nutzbarer Konzepte und Technologien gelegt.

Das Buch behandelt die folgenden Themenkomplexe, denen jeweils ein eigenes Kapitel gewidmet ist:

1. Einführung in verteilte Systeme
2. Konzepte und Modelle verteilter Kommunikation
3. Verteilte Dienste und Webservices
4. Konzepte, Modelle und Standards verteilter Transaktionsverarbeitung
5. Architekturen verteilter betrieblicher Anwendungen

Kapitel 1 erklärt einige Grundbegriffe und führt in die Vor- und Nachteile verteilter Systeme ein. Kapitel 2 stellt die wichtigsten Konzepte und Modelle verteilter Anwendungsentwicklung wie das RPC-Modell, Message-Passing, verteilte Prozeduren, verteilte Objekte, verteilte Transaktionen, verteilte Komponenten usw. vor. Kapitel 3 befasst sich mit SOA-Konzepten und Webservices als spezieller SOA-

Implementierungsvariante im Detail. In Kapitel 4 werden die grundlegenden Konzepte, Modelle und Standards verteilter Transaktionsverarbeitung erläutert und deren Umsetzung in verschiedenen Technologien diskutiert. Kapitel 5 gibt schließlich einige wichtige Hinweise für die Gestaltung von verteilten Architekturen betrieblicher Informationssysteme

Die Einzelkapitel können aus Platzgründen oft keine allzu detaillierten technischen Erläuterungen, sondern nur die grundlegenden Aspekte und Prinzipien wiedergeben. Weiterführende Literatur ist zur Genüge im Anhang aufgeführt. Im Übrigen ist der Autor allerdings der Auffassung, dass man nur durch viel Übung eine Entwicklungstechnik oder Technologie voll beherrschen kann. Dies ist natürlich dem Leser überlassen.

Bei der ständigen Weiterentwicklung von Konzepten und Technologien in verteilten Systemen ist es für Softwarearchitekten und Entscheider schwer, das für die praktische Nutzung Wichtige vom weniger Wichtigen auszufiltern. Dieses Buch soll derartige Entscheidungsprozesse ein wenig unterstützen bzw. die Studierenden in die Lage versetzen, Entscheidungsprozesse in diesem Umfeld vorzubereiten und in Zukunft mitzutragen.

Der Inhalt des Buches entstand aus mehreren Vorlesungen über Verteilte Systeme an der Hochschule für angewandte Wissenschaften – FH München. Bedanken möchte ich mich sehr herzlich für das Feedback meiner StudentInnen und Studenten. Es hat mir sehr bei der Strukturierung des Stoffes geholfen. Dem Verlag danke ich für die Unterstützung im Projekt und für die allzeit konstruktive und angenehme Zusammenarbeit.

Fragen und Korrekturvorschläge richten Sie bitte an mandl@cs.hm.edu.

Für begleitende Informationen zur Vorlesung siehe www.prof-mandl.de.

München, im Juni 2008

Peter Mandl

Inhaltsverzeichnis

1 Einführung in verteilte Systeme

Was ist denn eigentlich ein verteiltes System? Diese Frage wird heute in der Literatur nicht eindeutig beantwortet. Es gibt so viele verschiedene Facetten verteilter Systeme bzw. verteilter Anwendungen, dass es schwer fällt, eine allgemeingültige Definition zu finden. Einige Definitionsversuche sollen in diesem Kapitel vorgestellt werden. Da verteilte Systeme sehr viel mit den Technologien, die verwendet werden können, zu tun haben, wird im Anschluss daran eine kurze Historie der Technologieentwicklung speziell für Technologien, die in verteilten Umgebungen eingesetzt werden, gegeben.

Damit Rechner bzw. Programme miteinander über ein Netzwerk kommunizieren können, ist ein gemeinsames Verständnis über die Regeln der Kommunikation erforderlich. Alle Partner müssen einheitliche Kommunikationsprotokolle einhalten. In Referenzmodellen wie dem ISO/OSI-Referenzmodell und dem TCP/IP-Referenzmodell werden die Grundlagen festgelegt. Die Kenntnis der grundlegenden Mechanismen und der entsprechenden Referenzmodelle, insbesondere des TCP/IP-Referenzmodells wird für die weiteren Betrachtungen weitgehend vorausgesetzt (siehe hierzu Tanenbaum 2003 und Mandl 2008b).

Nach der Einführung werden wichtige Designziele verteilter Systeme, wie Lastverteilung, Skalierbarkeit, Ausfallsicherheit und Flexibilität diskutiert. Es wird auch erläutert, dass Transparenz in verschiedenen Varianten für die Entwicklung und Nutzung verteilter Systeme sehr wichtig ist. Die Probleme der Heterogenität und weitere Aspekte, die in verteilten Umgebungen gelöst werden müssen, werden ebenfalls erörtert.

Zielsetzung des Kapitels

Ziel dieses Kapitels ist es, eine Einführung in die Problematik der verteilten Systeme zu geben. Der Studierende soll grundlegende Begriffe verteilter Systeme kennenlernen und Technologien für die Entwicklung verteilter Systeme einordnen können.

Wichtige Begriffe

Verteiltes System, verteilte Anwendung, Transparenz und Heterogenität, Persistenz, Transaktionssicherheit, Fehlersemantiken, Maybe, At-Most-Once, At-Least-Once, Exactly-Once

1.1 Definitionen und Festlegungen

Heute sind fast alle betrieblichen Informationssysteme in der einen oder anderen Form verteilt, wobei unterschiedlichste Technologien eingesetzt werden. Dieser Abschnitt soll nach einer kurzen Einführung einiger Begriffe und einer Klärung, was man unter einem verteilten System heute versteht, eine kurze Geschichte zu den Technologien für die Entwicklung verteilter Systeme geben.

1.1.1 Grundlegende Begriffe

Für unsere weiteren Betrachtungen sollen zunächst einige Begriffe erläutert werden, die im Umfeld der betrieblichen Informationssysteme häufig verwendet werden. Hierzu gehören die Begriffe *System*, *Informationssystem*, *Technik* und *Technologie*.

System und Informationssystem: In der Informatik wird oft von Systemen gesprochen. Ganz allgemein wird der Begriff *System* in unterschiedlicher Bedeutung verwendet. Im Prinzip ist allerdings immer die "Zusammenstellung" mehrerer Elemente, die untereinander in Wechselwirkung stehen, gemeint.

Ein *Informationssystem* dient dagegen prinzipiell der rechnergestützten Erfassung, Speicherung, Verarbeitung, Verwaltung, Pflege usw. von Information. Es besteht aus Hardware (Rechner oder ein ganzer Rechnerverbund), Datenbank(en), Software (System- und Anwendungssoftware), Daten und deren Anwendungen. Informationssysteme sind soziotechnische Systeme, die auch aus einzelnen Teilsystemen bestehen können, und für die optimale Bereitstellung von Information und (technischer) Kommunikation dienen. Diese Beschreibung lässt viel Spielraum für Interpretationen.

Ein *betriebliches Informationssystem* (Business Information System) ist ein Informationssystem, das sich in erster Linie mit betrieblichen Funktionen und Daten befasst, um diese effizient zu bearbeiten und bereitzustellen.

In der praktischen Informatik bezeichnet man auch ein Rechnersystem mit Hard- und Software einfach kurz als System und spricht zum Beispiel von einer Systemarchitektur, wenn Hardware und Software gleichermaßen in die Architekturbetrachtung eines Informationssystems mit einbezogen werden.

Technik und Technologie: Unter *Technik* versteht man nach (WWW-038) Verfahren und Fähigkeiten zur praktischen Anwendung der Naturwissenschaften und zur Produktion industrieller, handwerklicher oder auch künstlerischer Erzeugnisse. Technik kann auch als die Fähigkeit des Menschen verstanden werden, Naturgesetze, Kräfte oder Rohstoffe zur Sicherung seiner Existenzgrundlage sinnvoll einzusetzen oder umzuwandeln.

Technologie ist nach (WWW-038) dagegen die Wissenschaft vom Einsatz der Technik im engeren Sinne, in der es um die Umwandlung von Roh- und Werkstoffen in fertige Produkte und Gebrauchsartikel geht, im weiteren Sinne geht es aber auch

um Handfertigkeiten und Können. Eine andere Definition bezeichnet mit Technologie die Gesamtheit der Verfahren zur Produktion von Gütern und Dienstleistungen, die einer Gesellschaft zur Verfügung steht. Technologie beinhaltet die Komponenten der Technik (Werkzeuge, Geräte, Apparate), die materiellen und organisatorischen Voraussetzungen und deren Anwendung. Häufig wird Technologie in der Praxis auch einfach anstelle von Technik verwendet. Spricht man im Zusammenhang z.B. bei Fahrzeugen von neuester eingesetzter Technologie, ist nur die Technik gemeint.

Unter *High-Tech* versteht man hochentwickelte Technologien, die neueste wissenschaftliche Erkenntnisse umsetzen, so beispielsweise die Produktion von CPUs oder bakteriell hergestelltes Insulin. Im Gegensatz dazu bezeichnet Low-Tech absichtlich möglichst einfache, ausfallsichere Technologien (bei deren Entwicklung natürlich auch neueste wissenschaftliche Erkenntnisse zum Einsatz kommen können), die dadurch einfach in Herstellung, Anwendung oder Wartung sind.

1.1.2 Was sind verteilte Systeme?

Was genau sind „verteilte Systeme“? Auch dieser Begriff ist in der Informatik nicht einheitlich definiert, obwohl er - insbesondere in der Praxis - sehr häufig verwendet wird. Mehrere Definitionen bzw. Beschreibungen für verteilte Systeme sind in der Literatur zu finden. Einige davon sollen hier erwähnt werden:

Tanenbaum definiert ein verteiltes Systen wie folgt (Tanenbaum 2003): „Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen“.

Bengel’s Definition eines verteilten Systems lautet folgendermaßen (Bengel 2004): „Ein verteiltes System ist definiert durch eine Menge von Funktionen oder Komponenten, die in Beziehung zueinander stehen (Client-Server-Beziehung) und eine Funktion erbringen, die nicht erbracht werden kann durch die Komponenten alleine.“[1]

(Kurmann 2004) beschreibt ein verteiltes System in einer „Arbeitsdefinition“ wie folgt: „Ein verteiltes System ist ein Softwaresystem, welches die Verteilung einzelner Bausteine eines Anwendungssystems auf verschiedene, räumlich getrennte Knoten eines Netzwerks, unterstützt. Einzelne Teile des Systems können dezentral administriert werden und kommunizieren, um gemeinsam Aufgaben zu bewältigen. Ein verteiltes System unterstützt nebenläufige Verarbeitung von Geschäftsprozessen und ist sowohl für den Anwender als auch für den Entwickler transparent“.

---

[1] Eine Client-Server-Beziehung ist nach Meinung des Autors nicht zwingend erforderlich. Wie wir noch sehen werden, gibt es auch genügend Anwendungssysteme, in denen die einzelnen Komponenten auf unterschiedliche Weise miteinander kommunizieren.

Über die Definitionen lässt sich wie immer diskutieren. Der Begriff der Transparenz wird z.B. in diesem Zusammenhang recht unterschiedlich benutzt. Einem Entwickler muss z.B. durchaus bewusst sein, auf welchen Systemen die einzelnen Komponenten ablaufen. Auch ist die Unterstützung von Nebenläufigkeit keine zwingende Notwendigkeit, obwohl sie in heutigen verteilten Systemen meist üblich ist.

Uns soll im Weiteren die Definition von Tanenbaum für *verteilte Systeme* genügen, da sie am flexibelsten ist. Die Begriffe *verteiltes System* und *verteiltes Anwendungssystem* verwenden wir in diesem Dokument als Synonyme, da es aus unserer Sicht in erster Linie um Anwendungssysteme geht. Wir erweitern daher die Definition von Tanenbaum und nehmen diese als Basis für unsere Betrachtungen:

Verteiltes System: Ein *verteiltes System* basiert auf einer Menge voneinander unabhängiger Rechnersysteme und Softwarebausteine, die dem Benutzer wie ein einzelnes, kohärentes System bzw. Anwendungssystem erscheinen. Jeder Softwarebaustein einer verteilten Anwendung kann auf einem eigenen Rechner liegen. Es können aber auch mehrere Softwarebausteine auf dem gleichen Rechner installiert sein.

Typische verteilte Systeme sind z.B. Webanwendungen, das E-Mailsystem im Internet, das Domain Name System (DNS) für die Verwaltung der IP-Adressen, verteilte Datenbankanwendungen, verteilte Dateisysteme wie das Network File System (NFS) von Sun Microsystems und das WWW. Dies sind verteilte Anwendungssysteme, die im Wesentlichen auf einem intelligenten Protokoll der Anwendungsschicht basieren und von der Komplexität her klar umrissen sind.

Verteiltes Informationssystem: Wir beschäftigen uns vorwiegend mit *verteilten Anwendungssystemen* im betrieblichen Umfeld bzw. verteilten Informationssystemen, bei denen einzelne Funktionen auf mehrere Rechner (mindestens zwei) verteilt sind, also verteilte Systeme, die in betrieblichen Informationssystemen relevant sind. Verteilte Informationssysteme sind spezielle, meist komplizierte verteilte Systeme, die sich nach (Hammerschall 2005) durch folgende Eigenschaften auszeichnen:

- Verteilte Informationssysteme sind meist sehr groß, was den Codeumfang anbelangt (mehrere 100.000, teilweise mehrere Millionen Lines of Code).
- Verteilte Informationssysteme sind sehr datenorientiert, d.h. die Datenhaltung steht im Zentrum der Anwendung. Üblicherweise werden die Daten in einer Datenbank verwaltet. Die zugrundeliegenden Datenmodelle sind meist sehr umfangreich.
- Verteilte Informationssysteme sind extrem interaktiv und verfügen (neben Hintergrund- und Batch-Funktionalität) überwiegend über graphische Benutzerschnittstellen.
- Verteilte Informationssysteme sind meistens auch sehr nebenläufig, was sich durch eine große Anzahl an parallel arbeitenden Benutzern äußert.

Hinzu kommt, dass derartige Anwendungen oft auch sehr unternehmenskritisch sind. Fällt beispielsweise ein Warenwirtschaftssystem eines Versandhändlers oder ein Core-Banking-System über längere Zeit aus, ist dies durchaus existenzbedrohend für das Unternehmen. Verteilte betriebliche Informationssysteme stehen auch im Mittelpunkt unserer weiteren Betrachtung. Derartig komplexe Systeme benötigen eine entsprechend komfortable Infrastruktur mit Transaktions-, Sicherheits- und sonstigen Diensten.

In anderen, mehr technischeren Bereichen (z.B. in Automobilen) entstehen derzeit ebenfalls immer komplexere verteilte Anwendungssysteme. In manchen Fahrzeugen kommunizieren mehr als 50 Microcontroller bzw. die darauf laufenden Softwarebausteine über verschiedene Bussysteme miteinander. Hier handelt es sich auch um hochkomplexe verteilte Systeme, die wir aber in unseren Ausführungen weniger betrachten.

In Abgrenzung hierzu sind Netzwerkbetriebssysteme und verteilte Betriebssysteme zu nennen:

Verteilte Betriebssysteme verteilen Betriebssystemfunktionalität so, dass das gesamte Netz wie ein Rechnersystem erscheint. Für den Anwender ist es nicht mehr ersichtlich, wo welche Ressourcen verwaltet werden (siehe Abbildung 1-1).

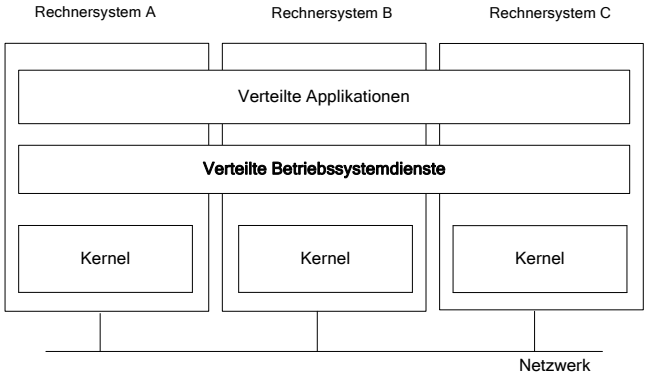


Abbildung 1-1: Verteiltes Betriebssystem nach Tanenbaum 2003

Netzwerkbetriebssysteme liegen über den lokalen Betriebssystemen und stellen Dienste wie den Zugriff auf entfernte Ressourcen (Dateien, Drucker) im Netzwerk zur Verfügung. Sie sind heute nicht mehr so verbreitet, als das Novell-System die PC-Netze eroberte. Diese Funktionalität ist heute schon in den Standardbetriebssystemen enthalten. In Abbildung 1-2 ist der Einsatz eines Netzwerkbetriebssystems skizziert.

Verteilte Betriebssysteme haben sich in der Praxis nie richtig durchgesetzt. Verteilte Systeme allerdings - und hier sind vor allem Anwendungssysteme gemeint -

sind heute fast überall im Einsatz. Sie sind im Internet genauso zu finden wie in Intranets und es spielt im Prinzip keine Rolle, welche Definition nun am besten passt.

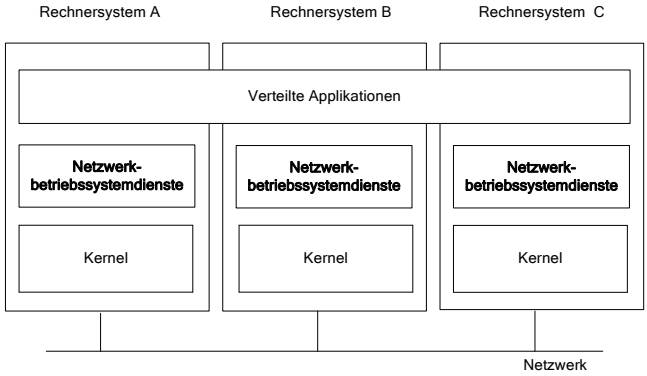


Abbildung 1-2: Netzwerkbetriebssystem

1.1.3 Klassifizierung verteilter Systeme

Eine Typisierung bzw. Klassifizierung verteilter Systeme wurde schon mehrfach versucht. Tanenbaum unterscheidet beispielsweise grob nach verteilten Computersystemen (Distributed Computing Systems), verteilten Informationssystemen (Distributed Information Systems) und verteilten pervasiven Systemen (Distributed pervasive Systems, pervasive = überall vorhanden, durchdringend) (Tanenbaum 2007):

– Unter *verteilten Computersystemen* versteht Tanenbaum verteilte Systeme für Hochleistungsaufgaben und ordnet hier *Cluster*[2]-Computersysteme und *Grid*-Computersysteme ein. Cluster-Computersysteme bestehen aus einer Menge ähnlich aufgebauter Rechnersysteme, die üblicherweise in einem LAN miteinander vernetzt sind und Komponenten einer verteilten Anwendung enthalten. Unter Grid-Computersystemen kann man auch eine Menge (evtl. sogar eine sehr große Menge) von Rechnersystemen verstehen, die weiter verteilt und zum Teil in sehr unterschiedlichen Adminstrationsdomänen stehen und gemeinsam eine verteilte Anwendung unterstützen. Typische Grid-basierte Anwendungen finden sich heute nur noch selten und dann vor allem für rechenintensive Aufgaben. Man versucht auch, die vielen Rechner im Internet, die ja die meiste Zeit ihre CPUs fast gar nicht nutzen, in

[2] Cluster = number of things of the same kind growing closely together (Bauke 2006).

Grids[3] zusammenzuschalten. Ein Beispiel für eine Grid-Anwendung ist das *Earth System Grid Project*, in dem Simulationen für die Klimaforschung durchgeführt werden. Cluster- und Grid-Computersysteme zeichnen sich durch eine sehr gute Skalierbarkeit, durch gute Möglichkeiten der Lastverteilung und durch hohe Fehlertoleranz aus.

- Unter *verteilten Informationssystemen* versteht Tanenbaum die bereits oben angesprochenen klassischen betrieblichen Anwendungen, die Transaktionen ausführen. Diese Anwendungssysteme stehen im Fokus unserer Betrachtung. Insbesondere Transaktionsanwendungen werden wir noch ausführlich besprechen.
- *Verteilte pervasive Systeme* sind kleine oder auch sehr kleine, oft batteriegetriebene Systeme und Systeme, die auch mobil sein können. Hierunter fallen Sensoren-Systeme für das Gesundheitswesen etwa zur Überwachung von Körperfunktionen (Herz-Kreislauf usw.). Diese Systeme erfreuen sich heute bereits zunehmender Verbreitung. Man spricht in diesem Zusammenhang auch von *Ubiquitous Computing* (kurz: Ubicomp). Ubicomp beschäftigt sich mit Computer-Systemen, die unauffällig und unsichtbar agieren, in Alltagsgegenständen vorhanden sind und intelligente Funktionen übernehmen. Für die Entwicklung derartiger Systeme gibt es heute noch keine Standards, da sie sich noch weitgehend im Forschungsstadium befinden.

Andere Klassifizierungsversuche ordnen verteilte Systeme anhand ihrer Architekturmerkmale. In (Dustdar 2003) wird beispielsweise eine Einordnung nach Architekturstilen vorgenommen. Es wird zwischen datenzentrierten Architekturen, datenflussorientierten Architekturen, Call-And-Return-Architekturen, unabhängigen Komponenten-Architekturen und auch virtuellen Maschinen-Architekturen unterschieden. Call-And-Return-Architekturen, unabhängige Komponenten-Architekturen sowie hierfür vorhandene Technologien werden wir im Rahmen unserer Betrachtung noch genauer untersuchen. Eine allgemein anerkannte Klassifizierung verteilter Systeme liegt aber nicht vor. Wie wir im Weiteren noch sehen werden, nutzen heutige verteilte Anwendungen meist mehrere Architekturstile und auch die Typen nach Tanenbaum sind in der Praxis nicht eindeutig abzugrenzen. Beispielsweise werden in heutigen verteilten Informationssystemen auch Cluster-Ansätze verwendet. Cluster findet man heute beispielsweise häufig in internetbasierten Verkaufssystemen vor, die sehr viele Anwender bedienen müssen.

1.1.4 Beispiel einer verteilten Systemlandschaft

Betrachten wir ein Beispiel aus dem Umfeld der betrieblichen Informationssysteme und zwar speziell eine (völlig unvollständige) Anwendungslandschaft eines Großhändlers, der seine Unternehmensdaten in einem ERP-System (Enterprise Resour-

[3] In Analogie zum Stromnetz kann man sich ein Computer-Grid auch so vorstellen, dass man Rechenleistung aus der Steckdose oder über den Internetanschluss bezieht.

ce Planning) verwaltet und der ein großes Lagersystem unterhält, das über IT-Systeme gesteuert wird. In Abbildung 1-3 ist die logische Sicht auf die Anwendungssysteme und deren Zusammenspiel dargestellt.

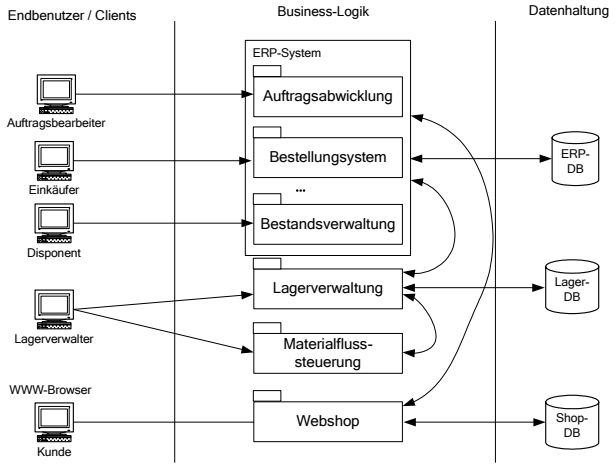


Abbildung 1-3: Beispiel einer verteilten Anwendung (logische Sicht)

In einem ERP-System werden u.a. die Teilsysteme zur Auftragsabwicklung, das Bestellwesen und die Bestandsverwaltung abgewickelt. Die Lagerverwaltung wird über ein eigenes Lagerverwaltungssystem ausgeführt, das wiederum über mehrere Subsysteme verfügt (im Bild nicht dargestellt). Die Materialflusssteuerung für die unterlagerten logistischen Systeme wie Fördertechnik und automatische Lager werden über ein Materialflusssystem bedient. Zudem wird den Kunden über einen Webshop eine direkte Bestellmöglichkeit über das Internet angeboten. Insgesamt werden drei disjunkte Datenbanken verwaltet. Die einzelnen Teilsysteme kommunizieren bei Bedarf miteinander. Die Clients bedienen verschiedene Benutzerrollen und verfügen entweder über eine klassische grafische Oberfläche oder über einen Web-Zugang.

In der logischen Sicht ist noch keine Aussage über die Verteilung der Teilsysteme genannt. In einer exemplarischen physikalischen Architektur wird diese Verteilung skizziert (siehe hierzu Abbildung 1-4). Die ERP-Teilsysteme und auch die Lagerverwaltung liegen auf einem doppelt ausgelegten und ausfallgesicherten ERP- bzw. einem LVS-Serverrechner. Die Datenhaltung für die beiden Datenbanken ERP-DB und Lager-DB wird in einem ebenfalls ausfallgesicherten Datenbankserver abgewickelt. Der Webshop wird von einem eigenen Shop-Server bedient, der auch die Datenhaltung übernimmt. Die Web-Zugriffe werden über einen Webserver an das Shopsystem weitergeleitet.

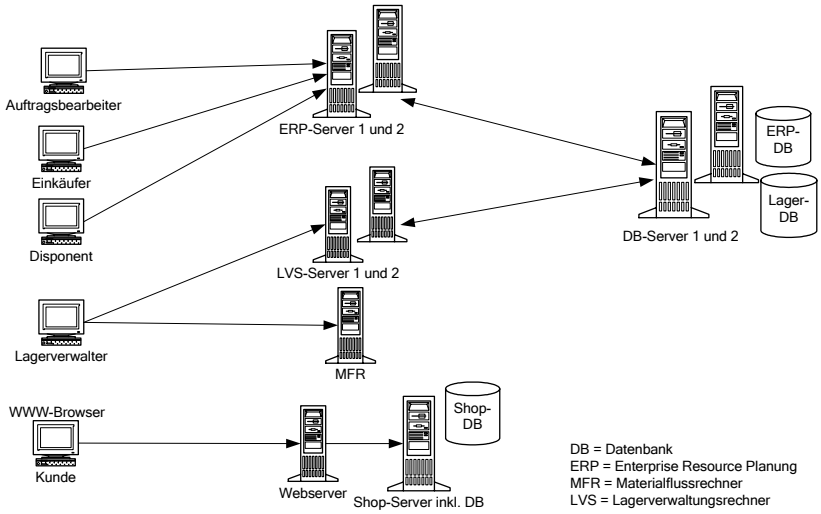


Abbildung 1-4: Beispiel einer verteilten Anwendung (physikalische Sicht)

Die Kommunikation der Rechner erfolgt über das interne Netzwerk des Unternehmens bzw. die WWW-Clients greifen über das Internet an den unternehmenseigenen Webserver. Wie die einzelnen Rechnersysteme miteinander kommunizieren und auf welche Basismechanismen und Technologien sie zurückgreifen, ist im Bild nicht dargestellt. Mehrere Varianten sind möglich und sollen auch noch diskutiert werden. Eines ist aber deutlich sichtbar: Die Systeme stehen teilweise in einer Client-Server-Beziehung. Dieses Grundmodell der Kommunikation wird noch ausführlich diskutiert.

In der Praxis wird die anwendungsübergreifende Kommunikation heute noch vielfach über den Austausch von Dateien in eigenen Formaten oder Standardformaten ausgeführt (EDIFACT, SWIFT, XML-Schemata, CSV-Dateien,...). Dieses Vorgehen hat den großen Vorteil, dass die kommunizierenden Anwendungssysteme keine technischen Abhängigkeiten haben und weitgehend unabhängig voneinander arbeiten können. Insbesondere bei der unternehmensübergreifenden Kommunikation ist eine Prozess-Prozess-Beziehung selten anzutreffen. Hier handelt es sich aber auch um eine verteilte Verarbeitung. Beispiele hierfür sind:

- Bestellaufträge werden über eine EDIFACT- oder XML-basierende Schnittstelle vom Besteller zum Auftraggeber gesendet.
- Banken tauschen für das Clearing ihre Umsatzdaten über sog. Datenträger aus.

1.2 Historische Technologie-Betrachtung

Aus unserer Sicht stehen die Softwarekonzepte für verteilte Systeme (Anwendungssysteme) insbesondere für die oben erwähnten Call-And-Return-Architekturen und für die unabhängigen Komponenten-Architekturen im Vordergrund. In den letzten Jahrzehnten ist hier sehr viel Pionierarbeit geleistet worden. Es wurde und wird auch in der Praxis viel Zeit und Geld in die Evaluation neuer Technologien gesteckt. Viele Projekte werden aber auch durch nicht ausgereifte Produkte für Basissysteme erschwert.

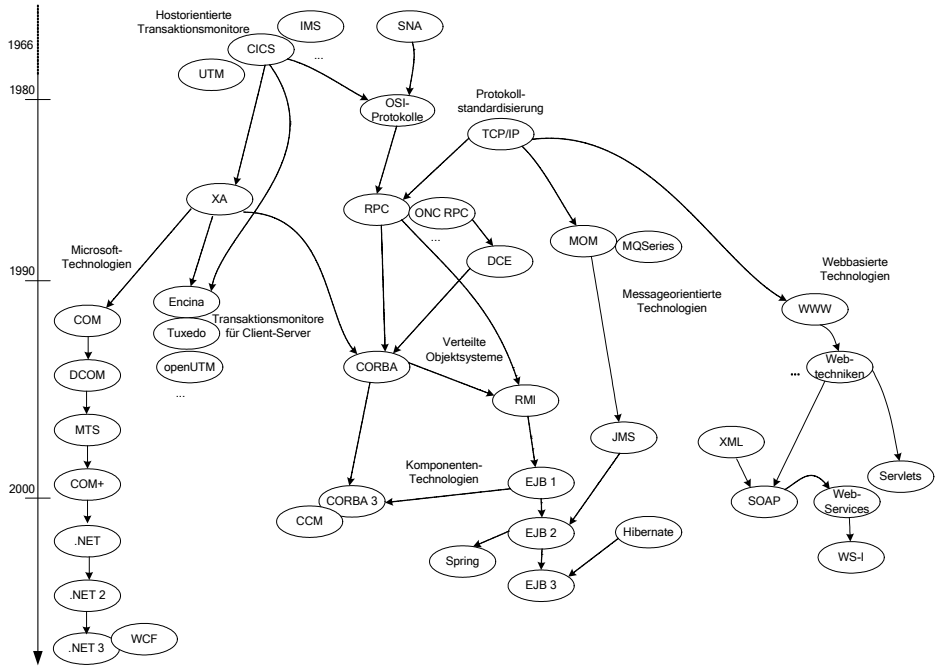


Abbildung 1-5: Entwicklung von Modellen/Technologien für verteilte Systeme

In Abbildung 1-5 ist der zeitliche Ablauf skizziert, wobei natürlich nicht alle, sondern nur die aus unserer Sicht wesentlichen „Erfindungen" eingezeichnet sind. Die Pfeile zwischen den Knoten deuten Einflüsse bzw. Weiterentwicklungen bestehender Technologien und Modelle an. Teilweise beeinflussen sich die Technologien gegenseitig, da die einzelnen Hersteller bzw. Organisationen voneinander profitieren. Wir beginnen mit den 80er Jahren.

Entwicklung in den 80er Jahren:

Ein wesentlicher Grund für die Dezentralisierungsbemühungen waren in den 80er Jahren des letzten Jahrtausends die hohen Kosten für Mainframes, ein anderer natürlich die hohe Erwartungshaltung bzgl. der Flexibilität neuer verteilter Systeme. Die Verbreitung von PCs brachte in diesem Jahrzehnt schließlich auch die ersten Netzwerke und Netzwerkbetriebssysteme (siehe Novell Netware) mit sich. Zunächst wurden Dienste wie File- und Druckersharing realisiert und nach und nach wollte man auch komplexere Anwendungssysteme auf Serverrechnern außerhalb der Hostsysteme platzieren. Datenbanksysteme auf eigenen Datenbankservern verbreiteten sich.

Die Standardisierung der Kommunikation zwischen Rechnersystemen wurde im Rahmen der ISO/OSI-Spezifikationen (OSI = Open Systems Interconnection) und der TCP/IP-Entwicklung in dieser Zeit stark vorangetrieben. Der Begriff des *offenen Systems* wurde durch die ISO/OSI-Spezifikationen geprägt. Ein offenes System ist nach OSI kein reales Rechnersystem, sondern allgemein ein System, dessen Komponenten sich dem OSI-Modell entsprechend verhalten. Ein System ist dann offen, wenn es nach außen ein nach OSI genormtes Verhalten zeigt. Diese Definition zeigt, dass hier der Grundstein für das Zusammenspiel von verteilten Systemen mehrerer Hersteller gelegt werden sollte.

Die Transportzugriffsschnittstelle *Sockets* und zeitweise *XTI* (Extendent Transport Interface) sowie *TLI* (Transport Layer Interface) haben sich als Standards für Transportzugriffssysteme entwickelt, wobei letztendlich Sockets für den Einsatz in der Praxis übrig blieb. Das Client-Server-Modell wurde Mitte der 80er Jahre herausgearbeitet. Remote-Procedure-Call-Implementierungen (RPC) entstanden. Schließlich wurde von der damaligen Open Software Foundation (OSF), einer Herstellervereinigung, die verschiedene Standards festzulegen versuchte, das Distributed Computing Environment (DCE) als Basis für die Entwicklung verteilter Anwendungen standardisiert. Das grundlegende Kommunikationsparadigma war auch hier der Remote Procedure Call (DCE RPC).

Entwicklung in den 90er Jahren:

Die 90er Jahre waren vor allem durch die Weiterentwicklung des Client-Server-Modells und der Remote-Procedure-Call-Mechanismen (RPC) geprägt (siehe vor allem Sun RPC). OSF DCE wurde um eine ganze Reihe von Standarddiensten (Time Service, Thread Service, Directory Service, Security Service,...) erweitert, die die Entwicklung verteilter Anwendungen erleichtern sollten.

Mit der Object Management Architecture (OMA) und der Common Object Request Broker Architecture (CORBA) wurden schließlich von der Object Management Group (OMG), einem Herstellerkonsortium, das in dieser Zeit die Entwicklung stark beeinflusste, Standards für objektbasierte verteilte Systeme entwickelt, die das „prozedurale“, C-basierte OSF DCE sehr schnell in Vergessenheit geraten lie-

sen. OMA (Object Management Architecture) und CORBA (Common Object Request Broker Architecture) spezifizierten eine Fülle von Diensten (Eventing Service, Transaction Service, Concurrrency Control Service, Persistency Service,...), die für verteilte Anwendungen nützlich sein sollten, und das auf Basis eines verteilten Objektmodells.

Sowohl DCE als auch CORBA trugen wesentlich dazu bei, dass sich die Softwarekonzepte für verteilte Anwendungen und deren Basistechnologien immer mehr durchsetzten. Aber auch CORBA wurde schnell wieder überboten, als man erkannte, dass die Entwicklung verteilter Anwendungen immer noch zu komplex und fehleranfällig war. Sun Microsystems brachte einen leichtgewichtigen Mechanismus zur objektbasierten Kommunikation in Java heraus, der als Java RMI (Remote Method Invocation) bezeichnet wurde. Parallel dazu entwickelte Microsoft verteilte Plattformdienste und komponentenbasierte Systeme wie COM (Component Object Model), DCOM (Distributed Component Object Model), COM+ und MTS (Microsoft Transaction Server).

Nebenbei verbreitete sich das World Wide Web mit seinen mittlerweile unzähligen Techniken zur Entwicklung von Webanwendungen von einer zunächst einfachen Informationsplattform immer weiter und ermöglichte komplexe Anwendungen. Webtechniken wie CGI, PHP, Java Servlets, Microsoft ASP usw. kamen auf den Markt und wurden vielfach eingesetzt.

Entwicklung seit dem Jahr 2000:

Ende der 90er und mit Beginn des neuen Jahrtausends wurden komponentenorientierte verteilte Systeme moderner, die die Objektorientierung nicht mehr so ganz in den Vordergrund rückten. Verteilte Komponenten, die man sich etwas größer als verteilte Objekte vorstellen kann, sollten möglichst einfach mit einer Schnittstelle versehen werden können und in einer komfortablen Umgebung zum Ablauf gebracht werden.

Aus der Java-Welt heraus wurde insbesondere von Sun Microsystems J2EE (Java 2 Extended Edition) mit ihren vielen Programmierschnittstellen und insbesondere Anfang 2000 die EJB-Spezifikation (Enterprise Java Beans) in ihrer ersten Version entwickelt, die heute als Basis für die Realisierung sog. EJB-Application-Server, eine Infrastruktur für serverseitige Komponenten, dient. EJB-Application-Server sind im Wesentlichen moderne Transaktionsmonitore für verteilte Systeme.

Parallel dazu wurden auch gesicherte Message-Queuing-Systeme weiterentwickelt. Ein typisches Produkt dieses Typs ist heute Websphere MQ von IBM. Derartige Produkte werden auch unter dem Begriff *MOM* (Message-orientierte Middleware) zusammengefasst.

Das WWW und die Webtechniken entwickelten sich ständig weiter. Komplexeste dynamische Webanwendungen wurden und werden realisiert und Webservices in Verbindung mit SOAP (Ursprünglich Kurzbezeichnung für *Simple Object Access*

Protocol) und XML sowie der Begriff der Service-orientierten Architektur (SOA) wurden erfunden. Teilweise entstand der Eindruck, dass man mit XML und Webservices alle Probleme lösen könnte. Aber das war natürlich nicht ganz korrekt.

Die Konzepte komponentenorientierter verteilter Systeme wurden weiterentwickelt. Eine „Standardisierung" von CORBA CCM (CORBA Component Model) mit CORBA V3.0 wurde durch die OMG durchgeführt. Microsoft brachte mit der .NET-Initiative ein entsprechendes Gegenstück heraus, in dem alle ihre gewachsenen Bausteine verteilter Systeme (COM, DCOM, COM+, …) vereint werden sollten. Im Mittelpunkt von .NET wurde vor allem auch die neue Sprache C# als Konkurrent zu Java platziert. Neue Ansätze der verteilten Kommunikation unter Windows sind im Windows Communication Framework (WCF) auf den Markt gekommen.

CORBA und vor allem J2EE/EJB sowie .NET mit WCF werden heute ständig weiterentwickelt. CORBA scheint allerdings nur noch für heterogene Lösungen und als Plattform-interne Technologie in Middlewareprodukten eingesetzt zu werden. Das Komponentenmodell CCM wurde bis heute kommerziell noch nicht implementiert. Die beiden praktisch relevanten, komponentenbasierten Basissysteme sind daher J2EE/EJB und .NET Enterprise Services. Aber auch Webservices scheinen sich für bestimmte Aufgabenstellungen durchzusetzen. MOM ist für bestimmte Anwendungen ebenfalls von großer Bedeutung und im Zuge der Vereinfachung wurde das Spring-Framework als leichtgewichtige Konkurrenz zu EJB entwickelt und erfreut sich zunehmender Beliebtheit (WWW-035).

Gerade in den letzten Jahren ist allerdings nach einer starken Euphorie eine leichte Ernüchterung eingetreten, da man erkannte, dass verteilte Systeme wesentlich aufwändiger in der Administration sind als zentralisierte Systeme. Die Software muss verteilt, die Serversysteme müssen überwacht und die vielen Clientsysteme müssen ständig aktuell gehalten werden. In der Praxis hat man viel über die Vor- und Nachteile der Verteilung gelernt. Man ist dazu übergegangen, auf eine gezielte und kontrollierbare („administrierbare") Verteilung zu setzen. Die im letzten Jahrzehnt entstandenen IT-Infrastrukturen brachten eine Inflation an Serversystemen mit sich, die heute wieder auf wenige, leistungsfähige (und damit Mainframeähnliche) Systeme konsolidiert werden. Dieser Trend der Serverkonsolidierung ist aktuell zu erkennen. Allerdings sind die Vorteile einer vernünftigen Verteilung so gravierend, dass eine Rückkehr zu reinen Mainframe-basierten Systemen sehr unwahrscheinlich ist.

1.3 Gründe für die Verteilung betrieblicher Informationssysteme

Dieser Abschnitt stellt einige Gründe für eine Verteilung von Softwarebausteinen vor. Nicht immer sind alle Gründe relevant, aber vor einer Entscheidung, ob ein System verteilt sein soll oder nicht, ist es nützlich, über die genannten Aspekte nachzudenken. Es werden heute mehrere Gründe für den Einsatz von verteilten

Systemen genannt. Manche gelten uneingeschränkt, andere nur unter bestimmten Randbedingungen. Die wesentlichen Gründe sollen zunächst im Überblick aufgezeigt werden:

- Lastverteilung
- Ausfallsicherheit
- Skalierbarkeit
- Flexibilität
- Transparenz

Lastverteilung: Lastverteilung bedeutet, dass man gewisse Systembausteine auf mehrere Rechnersysteme verteilen kann. Dadurch kann ein Leistungsgewinn erreicht werden. Beispielsweise ist es möglich, einen Serverbaustein mehr als einmal laufen zu lassen und die Anfragen der Clients (siehe auch Client-Server-Modell) auf die verschiedenen Instanzen zu verteilen. Damit kann auch die Netzbelastung beeinflusst werden. Man spricht in diesem Zusammenhang oft auch von Cluster-Lösungen, wobei ein Cluster aus mehreren Rechnersystemen besteht, die der Lastverteilung dienen.

Skalierbarkeit: Skalierbarkeit ist ein wichtiger Aspekt, der für verteilte Systeme spricht. Man kann - bei entsprechender Anwendungsarchitektur – ein System so konzipieren, dass es auf höhere Belastungen (z.B. durch eine zunehmende Anzahl an Benutzern) durch Hinzunahme weiterer Rechnersysteme reagieren kann.

Ausfallsicherheit: Ausfallsicherheit ist ebenfalls ein wichtiger Aspekt. Insbesondere durch die Verteilung bestimmter Services, die auf mehreren Serversystemen platziert werden, ist es möglich ein System redundant auszulegen und damit gegen Ausfälle zu schützen.

Flexibilität: Die Flexibilität ist einer der wichtigsten Vorteile verteilter Systeme. Durch eine geschickte Architektur kann man heute Systeme so gestalten, dass sie mit den Anforderungen wachsen können. Die Verteilung der verschiedenen Komponenten eines Anwendungssystems auf mehrere Rechner und bei entsprechender Architektur auf beliebige Rechnersysteme ermöglicht es, auf Änderungen zu reagieren. Verteilte Systeme können auch durch kooperierende Entwicklungsteams realisiert werden. Eine geeignete Architektur ist hier natürlich vorausgesetzt. Man kann auch fertige Bausteine hinzukaufen, die in die eigene Anwendung integriert werden. Dies setzt natürlich eine gemeinsame Ablaufumgebung voraus. Ein richtiger Markt für „Fertigbausteine“ wie bei Hardwarebausteinen ist heute noch nicht vorzufinden, ist jedoch seit langem ein Ziel. Es ist aber auch möglich, dass dieses nie oder zumindest in absehbarer Zeit nicht erreicht wird, weil Hersteller meist dagegen arbeiten.

Transparenz: Grundsätzlich wird der Begriff der Transparenz verwendet, um zu erläutern, wie die tatsächliche Verteilung der Bausteine eines verteilten Systems vor dem Anwender oder Entwickler verborgen werden kann. Es gibt verschiedene Facetten der Transparenz:

- Mit *Ortstransparenz* ist gemeint, dass man dem Anwender und wenn möglich sogar dem Entwickler verbergen möchte, auf welchen Rechnersystemen bestimmte Softwarebausteine ablaufen, wie man diese findet und wie sie dahin kommen.
- Der Begriff der *Migrationstransparenz* wird verwendet, um darzustellen, wie gut sich Softwarebausteine von einem Rechnersystem zum anderen verschieben lassen, ohne dass der Benutzer oder Entwickler es merkt.
- Von *Skalierungstransparenz* spricht man, wenn man ein System ausbauen kann, so dass es mehr leistet, ohne dass der Benutzer es merkt.
- Unter *Zugriffstransparenz* ist zu verstehen, dass man über ein System auf Ressourcen zugreifen kann (z.B. auf Objekte in einer Datenbank), ohne dass der Anwender wissen muss, wo diese liegen.
- Man spricht von *Fehlertransparenz*, wenn vor dem Anwender und/oder Programmierer gewisse Fehlersituationen des Systems verborgen werden.

Der Begriff der Transparenz wird darüber hinaus verwendet, um die Art der Datendarstellung in den verschiedenen Rechnersystemen zu verbergen (siehe weiter unten).

All diese Arten von Transparenz und auch noch weitere (*Nebenläufigkeitstransparenz*, *Persistenztransparenz*, *Replikationstransparenz*,...) können von verteilten Systemen ermöglicht werden. Einschränkend muss ergänzt werden, dass heutige Plattformen für verteilte Systeme weitestgehend für Transparenz aus Sicht des Benutzers sorgen, jedoch nur eingeschränkt aus der Sicht eines Anwendungsprogrammierers oder zumindest des Systemarchitekten, der meist genau darüber informiert sein sollte, wo die einzelnen Bausteine eines verteilten Systems platziert werden. Er trifft ja sogar die Entscheidungen darüber.

Die meisten der aufgeführten Gründe für eine Verteilung eines Systems sind sinnvoll. Es gibt aber auch gute Gründe gegen eine Verteilung, wie z.B. der erhöhte Aufwand an Administration und ggf. eine Beeinträchtigung der Leistungsfähigkeit eines Systems. Bei der Konzeption des Systems und bei der Gestaltung der Architektur sind jeweils die Vor- und Nachteile abzuwägen. Meist gibt es kein Patentrezept, wohl aber mittlerweile gute Erfahrungswerte.

Für die Unterstützung der genannten Aspekte ist es wichtig, dass das verteilte Anwendungssystem über eine geeignete Architektur verfügt, da ansonsten Skalierbarkeit, Lastverteilung usw. nicht oder nur erschwert möglich ist. Wir müssen uns also mit geeigneten Architekturen für verteilte Systeme befassen.

1.4 Das Problem der Heterogenität

Transparenz wird vor allem dadurch erschwert, dass kommunizierende Systeme heute oft nicht homogen konstruiert sind. In verteilten Systemen trifft man gewöhnlich auf verschiedene Rechnersysteme, auf denen Anwendungen ablaufen,

die möglicherweise auch noch in unterschiedlichen Programmiersprachen entwickelt wurden, aber miteinander kommunizieren.

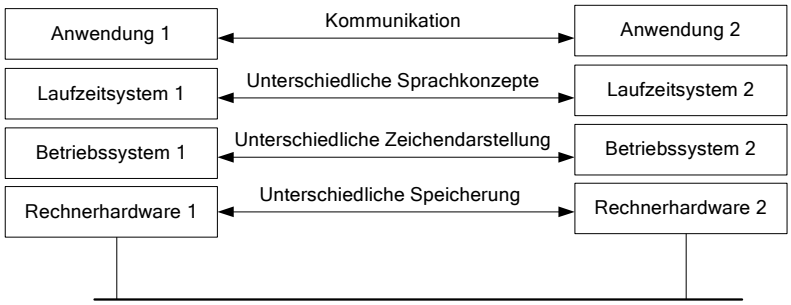


Abbildung 1-6: Heterogenität auf verschiedenen Ebenen

Die Heterogenität der beteiligten Systeme kann sich also über mehrere Ebenen erstrecken. Man muss in einem verteilten System möglicherweise verschiedene Rechnerarchitekturen, Betriebssysteme und Programmiersprachen (bzw. deren Laufzeitsysteme) so aufeinander abstimmen, dass sie auch kommunikationsfähig sind (siehe Abbildung 1-6).

Zur Verbesserung der Transparenz ist ein netzweites, oder noch besser, ein anwendungsweites Standardformat für Nachrichteninhalte anzustreben. Dies wird üblicherweise in höheren Kommunikationsschichten realisiert. Um die Problematik besser zu verstehen, sollen die verschiedenen Ebenen der Heterogenität in verteilten Systemen noch weiter erörtert werden:

Heterogenität in den Betriebssystemen: Betriebssysteme nutzen zum Teil recht unterschiedliche Zeichendarstellungen. Manche Betriebssysteme verwenden als internen Zeichensatz den EBCDIC-Code, andere den ASCII-Code und wieder andere nutzen eine Unicode-Variante.

Heterogenität in den Rechnerarchitekturen: Rechnerarchitekturen weisen Unterschiede bei der Speicherung der Daten auf. Ein klassisches Beispiel sind die Formate Big- und Little-Endian[4], die die interne Anordnung der Bytes eines Integerwerts festlegen. Beim Little-Endian-Format wird das höchstwertige Byte eines Integerwerts (short, long integer mit und ohne Vorzeichen) an der höheren Speicheradresse abgelegt. Beim Big-Endian-Format ist es gerade umgekehrt (siehe hierzu Abbildung 1-7). Das höchstwertige Byte wird beim Big-Endian-Format an der niedrigsten Speicheradresse abgelegt.

[4] Little Endian bedeutet wörtlich „Kleinender“, Big Endian bedeutet wörtlich „Großender“. Die Bezeichnung stammt von den Namen der Politiker in der Geschichte *Gulliver's Reisen*, die darüber Krieg führten, an welchem Ende ein Ei aufzumachen sei.

In Intel-Prozessoren nutzt man gewöhnlich das Little-Endian-Format. Motorola-Prozessoren vom Typ 68000, PowerPC- und SPARC-Prozessoren sowie z/Series-Prozessoren von IBM nutzen das Big-Endian-Format. Manche Prozessoren wie z.B. die Prozessoren der IA64-Architektur von Intel beherrschen beide Formate.

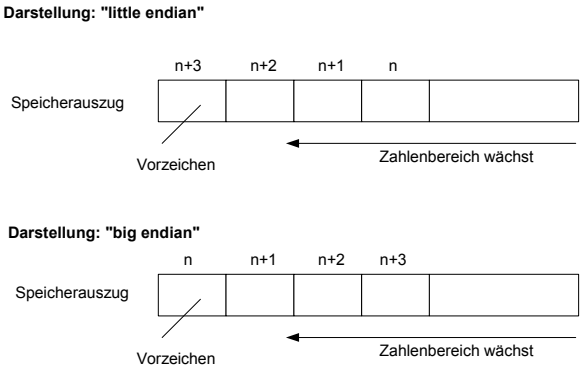


Abbildung 1-7: Little- und Big-Endian

Heterogenität in den Programmiersprachen und Laufzeitsystemen: Verschiedene Programmiersprachen haben zum Teil ganz unterschiedliche Daten- oder Objektmodelle. Wenn also z.B. eine Java-Anwendung ein Objekt vom Typ „Kunde“ an eine COBOL-Anwendung sendet, kann die COBOL-Anwendung nichts damit anfangen. Die interne Darstellung von Datentypen weicht zum Teil erheblich voneinander ab. Zeichenfolgen (Strings) können z.B. mit einer führenden Längenangabe ausgestattet sein oder nicht. Sie können mit einem speziellen Zeichen (z.B. „\0“ bei C/C++) abgeschlossen werden usw. Arrays können ebenfalls ganz unterschiedlich verwaltet werden. Schließlich können Referenzen bzw. Zeiger nicht sinnvoll an eine Partneranwendung übertragen werden, da es sich hier um Adressen im lokalen Adressraum handelt.

Es kommt sogar vor, dass Programmiersprachen auf verschiedenen Zielplattformen unterschiedlich implementiert sind und die Laufzeitsysteme damit nicht kompatibel sind. Dies war bei C-Implementierungen durchaus nichts Ungewöhnliches. Heute setzen sich für moderne Sprachen wie Java und C# zwar immer mehr sog. virtuelle Maschinen durch, die nach einer einheitlichen Spezifikation implementiert werden und unabhängig von der Plattform (Hardware+Betriebssystem) einheitliche Daten- und Objektformate unterstützen. Unterschiedliche virtuelle Maschinen können aber ebenfalls nicht ohne weiteres miteinander kommunizieren.

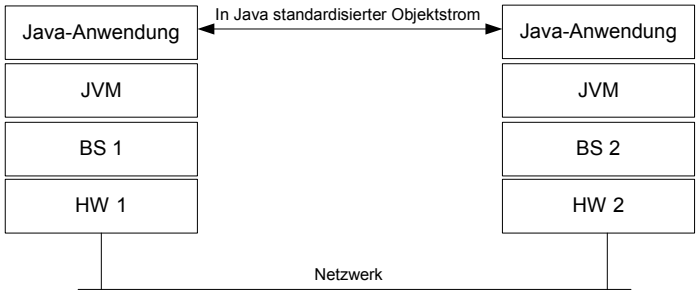


Abbildung 1-8: JVM-JVM-Kommunikation

Eine Kommunikationsverbindung könnte man auf der gemeinsamen Transportschicht TCP z.B. problemlos über eine Socket-Kommunikation herstellen, allerdings funktioniert damit die Kommunikation noch nicht. In den einzelnen Sprachen werden nämlich meist eigene Serialisierungsmechanismen, die in Java und C# als Objektströme bezeichnet werden, unterstützt. Diese sorgen für eine einheitliche Datenübertragung. Dies funktioniert innerhalb der gleichen Sprachumgebung sogar dann, wenn die Plattformen[5] unterschiedlich sind (siehe hierzu Abbildung 1-8 und Abbildung 1-9). Zwei Java-Anwendungen oder auch zwei C#-Anwendungen können also problemlos miteinander kommunizieren, wenn sie die vorhandenen Sprachmechanismen nutzen.

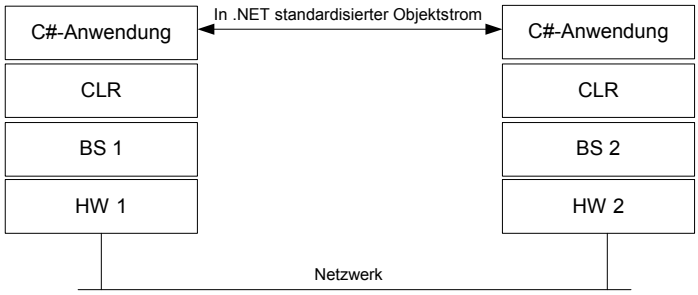


Abbildung 1-9: CLR-CLR-Kommunikation

Kommuniziert aber z.B. eine Java-Anwendung, die in einer Java Virtual Machine (JVM) abläuft mit einer C#-Anwendung, die in der .NET Common Language Runtime (CLR) abläuft, so muss man sich bewusst sein, dass diese beiden Laufzeitsysteme unterschiedliche Typsysteme aufweisen.

[5] Als Plattform bezeichnet man gewöhnlich die Hardware und das Betriebssystem.

Für alle .NET-Sprachen (C#, J#, C++,...) implementiert die CLR ein einheitliches Typsystem, so dass ein Datenaustausch sogar zwischen Anwendungen, die in unterschiedlichen .NET-Sprachen realisiert werden, möglich ist.

Die Übertragung von serialisierten Daten und Objekten sorgt aber noch nicht für allgemeine Kommunikationsfähigkeit. Beide Kommunikationspartner müssen die gleichen Serialisierungs- und Deserialisierungsregeln anwenden. Eine Java-Anwendung kann beispielsweise einen C#-Objektstrom nicht interpretieren und umgekehrt (siehe Abbildung 1-10). Wenn also eine Java-Anwendung mit einer C#-Anwendung kommunizieren muss, braucht man ein gemeinsames Verständnis der auszutauschenden Daten/Objekte. Dies könnte z.B. eine Einigung auf einen kleinsten gemeinsamen Nenner, wie z.B. dem ASCII-Zeichensatz sein.

Alternativ könnte man sich eine Zwischenschicht in Form eines Gateways vorstellen, das eine Transformation der Nachrichten vornimmt, ohne dass die Kommunikationspartner dies bemerken (Abbildung 1-11).

Die Transparenz kann durch eine Vereinheitlichung der Syntax erreicht werden, die sowohl der Sender als auch der Empfänger versteht. Der Vorgang des Umwandelns, also der Transformation in eine einheitliche Syntax nennt man *Kodierung*, *Serialisierung* oder auch *Marshalling*. Der umgekehrte Vorgang wird als *Dekodierung*, *Deserialisierung* oder *Unmarshalling* bezeichnet.

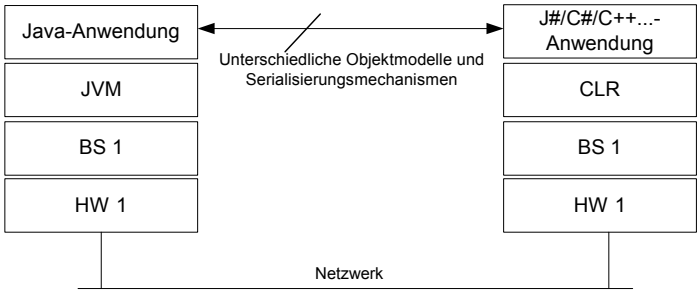


Abbildung 1-10: Problematische JVM-CLR-Kommunikation

Die ISO/OSI hat das Problem bereits in den 80er Jahren mit dem Standard für eine einheitliche Transportsyntax namens ASN.1 (Abstract Syntax Notation 1) und den dazugehörigen Kodierungsregeln BER (Basic Encoding Rules) adressiert. Dieser Standard ist funktional der Schicht 6 des ISO/OSI-Referenzmodells zugeordnet. ASN.1/BER sieht für jeden Datentypen und auch für komplexe Strukturen sog. Tags vor, die in der Nachricht jeweils vor den eigentlichen Werten übertragen werden. Aus ASN.1-Datenbeschreibungen kann man über von verschiedenen Herstellern angebotene ASN.1-Compiler Kodierungs- und Dekodierungsroutinen erzeugen, die in den Kommunikationsprogrammen eingesetzt werden, um einheitli-

che Datentypen zu übertragen. Heute nutzt z.B. SNMP (Simple Network Management Protokoll) noch ASN.1 zur Beschreibung der sog. Management Information Bases (MIBs). Dies sind die Daten, die im Rahmen des Netzwerkmanagements ausgetauscht werden.

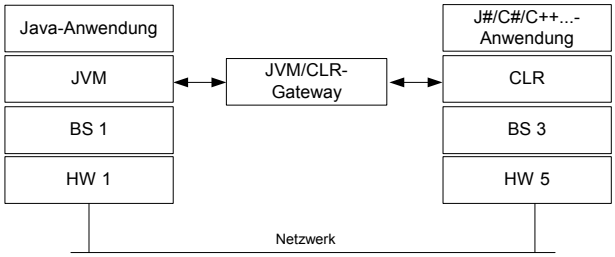


Abbildung 1-11: JVM-CLR-Kommunikation über Gateway

Wie wir in den folgenden Kapiteln noch sehen werden, bietet heute jede Middleware-Lösung ebenfalls eigene Mechanismen an. Da die Problematik recht einfach formalisierbar ist, setzt man auch das Mittel der Codegenerierung ein. In Implementierungen höherer Kommunikationsparadigmen wird die Erzeugung einer einheitlichen Transportsyntax meist über die automatisierte Erzeugung von sog. *Stubs* für die aktive Seite und *Skeletons* für die passive Seite unterstützt (Abbildung 1-12). Dies sind Programmteile, die meist automatisch aus einer Schnittstellenspezifikation ableitbar sind. Beispiele für diese Art der Codegenerierung werden noch gezeigt und sind u.a. in Sun ONC RPC, in CORBA, RMI, .NET Remoting und in den verschiedenen Webservice-Implementierungen auf recht ähnliche Weise integriert.

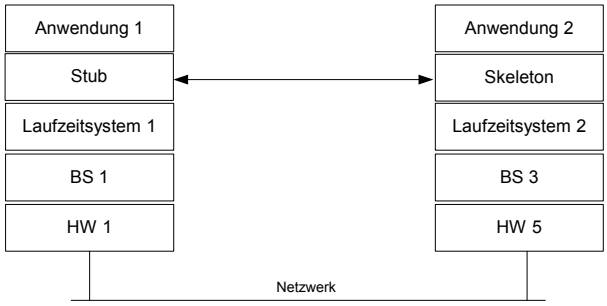


Abbildung 1-12: JVM-CLR-Kommunikation über Stub und Skeleton

Stub und Skeleton vereinheitlichen die transportierten Nachrichten und realisieren ein einheitliches Kommunikationsmodell. Die Transparenz wird dadurch unterstützt. Ob nun über diesen Mechanismus Anwendungen, die in verschiedenen

Sprachen entwickelt wurden, miteinander kommunizieren können, hängt von der Unabhängigkeit des zugrunde liegenden Konzepts ab. Sprachabhängige Mechanismen sind z.B. Java RMI, .NET Remoting, Sun ONC RPC, sprachunabhängige Mechanismen sind CORBA oder WSDL-basierte Webservices.

Für die Kommunikation ganz unterschiedlicher Anwendungen etwa über Webservices stellt sich immer mehr heraus, dass sich XML-basierte Sprachen sehr gut eignen. Wie wir noch sehen werden, gibt es für Webservices bereits einen anerkannten Standard zur Beschreibung von Kommunikationsdiensten, der mit WSDL (Web Service Description Language, siehe WWW-010) bezeichnet wird.

1.5 Spezielle Probleme verteilter Informationssysteme

In diesem Abschnitt gehen wir auf einige wichtige Problemstellungen speziell von verteilten betrieblichen Informationssystemen ein. Hierzu gehören die persistente Datenhaltung in verteilter Umgebung, die Fehleranfälligkeit, die sich aufgrund der Verteilung von Softwarekomponenten ergibt sowie die Gewährleistung der für betriebliche Systeme so wichtigen Transaktionsicherheit bei einer Verteilung der Systeme.

1.5.1 Persistente Datenhaltung in verteilter Umgebung

Unter dem Begriff „persistente Datenhaltung“ versteht man die Speicherung von Daten, die in einem Anwendungssystem über eine Sitzung hinweg benötigt werden. Persistente Daten sind also typischerweise alle Daten, die man heute in Datenbanken oder in sonstigen externen Medien ablegt. Klassisch verwendet man für größere Datenbestände relationale Datenbanken. Man kann sich heute keine größere Anwendung im betrieblichen Umfeld ohne Datenbanken vorstellen. Dies trifft auch auf verteilte Anwendungssysteme im betrieblichen Umfeld zu.

Wie wir noch sehen werden, gibt es für verteilte Systeme wie auch für lokal ablaufende Anwendungssysteme verschiedene Möglichkeiten, wie man auf persistente Daten zugreift. Größere Anwendungen kapseln üblicherweise den Zugriff auf die Datenbank(en) in einer Zugriffsschicht.

Seit es objektorientierte Sprachen gibt, besteht das Problem des *Impedance-Mismatch*. Objekte müssen auf Tabellen in relationalen Datenbanken abgebildet werden und umgekehrt. Für diese Aufgabe wurden sog. Object-Relational-Mapper (O/R-Mapper, ORM) entwickelt, die in eigenen Werkzeugen bzw. Produkten oder auch gemeinsam mit Basisprodukten für die verteilte Anwendungsentwicklung verfügbar sind. Diese O/R-Mapper ermöglichen es z.B. aus Objektklassen und deren Beziehungen automatisiert Tabellenstrukturen und Relationen zu erzeugen. Auch die Formulierung von Zugriffsoperationen wie Queries aus dem OO-Coding heraus ist möglich. Hier muss allerdings erwähnt werden, dass die Funktionalität von SQL

im Prinzip nur in eigenen Sprachen nachgebaut wird. Beispiele für derartige Ansätze sind:

- *JDO* (Java Data Objects) ist ein Ansatz aus dem Java Community Process (JSP) und gilt als ein Java-Standard für diese Aufgabe. JDO ist ein sehr breiter Ansatz, der nicht unbedingt relationale Datenbanken voraussetzt, sondern auch objektorientierte Datenbanken und XML-Speicherung unterstützt.
- Einen neueren Standardisierungsversuch der Java-Gemeinde stellt die *Java Persistence API* (*JPA*) dar, die als eigene API vorliegt und auch in neueren EJB-Versionen verwendet wird (WWW-003).
- *Hibernate* ist ebenfalls eine Lösung aus dem Java-Open-Source-Umfeld, die sich mehr und mehr zum Defakto-Standard entwickelt (Beeger 2006). Hibernate wird auch im Application-Server JBoss eingesetzt.
- *TopLin*k ist ein kommerzielles Produkt der Firma Oracle.

Die Wichtigkeit des Datenbankzugriffs wurde von den Standardisierungsgremien und Herstellern erkannt. Beispielsweise bietet auch CORBA eine Persistency-Spezifikation für diese Aufgabe. Allerdings wurde sie nie praktisch relevant.

Wir werden im Rahmen der Architekturdiskussion im Weiteren immer wieder auf den Datenbankzugriff zu sprechen kommen.

1.5.2 Fehleranfälligkeit verteilter Kommunikation

In lokalen Systemen wird ein Aufruf einer Methode oder Prozedur im lokalen Adressraum eines Prozesses durchgeführt. Sieht man von System- oder Programmabstürzen während der Ausführung ab, erfolgt die Ausführung genau einmal. Man spricht hier von einer Aufrufsemantik, die als Exactly-Once bezeichnet wird. Diese Eigenschaft möchte man auch gerne in einer verteilten Umgebung erreichen. Bei der Kommunikation in höheren Schichten gibt es aber, trotz gesicherter Transportprotokolle, viel mehr Fehlerursachen als in lokalen Umgebungen.

In manchen Situationen ist es schwierig, festzustellen, ob eine Anfrage schon ausgeführt wurde oder nicht. Dies geht weit über die Sicherstellung der Kommunikation hinaus. Es geht darum, ob aufgrund einer Nachricht (Auftrag, Request) schon eine Abarbeitungslogik durchlaufen wurde. Neben den klassischen Fehlern im Netzwerk kann z.B. ein Sender vor dem Empfang des Ergebnisses ausfallen, was zu *verwaisten Aufträgen* (sog. *Orphans*) führt. Der Empfänger kann während der Bearbeitung eines Requests ausfallen, wenn z.B. der Anwendungsprozess aufgrund eines Programmfehlers abstürzt.

Ausfälle sind zu jeder Zeit möglich und das Kommunikationssystem kann sich hier je nach Realisierung unterschiedlich verhalten. Wir gehen bei dieser Betrachtung davon aus, dass ein Dienstnehmer (oder auch Client) eine entfernte Operation aufruft, die ein Diensterbringer (auch Server genannt) bereitstellt. Je nachdem, wie viel Aufwand man in die Behandlung von Fehlern steckt, unterscheidet man bei

einem verteilt ausgeführten Request verschiedene Möglichkeiten der Fehlerbehandlung. Es gibt verschiedene Fehlerbearbeitungsmöglichkeiten (Coulouris 2002):

- Ein verteiltes System könnte z.B. überprüfen, ob beim Client eine Antwort ankommt und falls dies nicht der Fall ist, den Request erneut senden. Hier spricht man von Übertragungswiederholung. Dies ist ein bekannter Protokollmechanismus, der auch in niedrigeren Kommunikationsschichten eingesetzt wird.
- Wenn man vermeiden möchte, Requests erneut zu übertragen, kann der Server so realisiert werden, dass er Duplikate erkennt.
- Schließlich kann ein Server es ermöglichen, dass eine Antwort für einen Request gespeichert wird, damit sie noch einmal an den Client übertragen werden kann, wenn dieser denselben Request erneut sendet.
- Ein verteiltes System könnte auch Mechanismen bereitstellen, die bei Ausfall einer beliebigen Komponente zu einer beliebigen Zeit dafür sorgt, dass ein Request genau einmal ausgeführt wird und Duplikate ausgeschlossen sind. Dies ist sicherlich die komplizierteste Form der Fehlerbearbeitung.

Je nachdem, wie viel Aufwand man in die Behandlung von Fehlern steckt, spricht man bei der Fehlerbehandlung in verteilter Umgebung von den Fehlersemantiken *Maybe*, *At-Least-Once*, *At-Most-Once* und *Exactly-Once*:

Maybe ist die schwächste Form der Fehlersemantik, in der keine Fehlerbehandlung stattfindet. Im Fehlerfall kann man nicht feststellen, ob der Auftrag ausgeführt wurde oder nicht.

At-Least-Once stellt sicher, dass der Auftrag wenigstens einmal ausgeführt wird, garantiert aber nicht, dass er genau einmal abgearbeitet wird. Eine Mehrfachausführung ist möglich. Dies bedeutet aber, dass eine Anwendung, die *At-Least-Once* nutzt, es auch tolerieren muss, dass ein Request ggf. mehrmals ausgeführt wird. Die Ausführung einer erneuten Überweisungsoperation in einem Online-Banking-System ist sicherlich nicht tolerierbar.

At-Most-Once ist ähnlich wie *At-Least-Once*, filtert aber auch noch zusätzlich Duplikate aus. *At-Most-Once* wird heute von den meisten RPC-Mechanismen und deren Nachfolgern erfüllt. *At-Most-Once* hat gegenüber *At-Least-Once* Vorteile, ist aber auch aufwändiger in der Implementierung. Der Server muss eine Requestliste verwalten und bei jedem ankommenden Request prüfen, ob für ihn schon ein Eintrag in der Liste steht. Je nachdem sind dann die entsprechenden Aktionen auszuführen. Ist der Request z.B. schon in der Liste und schon ausgeführt, ist nur noch die Antwort zu übertragen. Weiterhin benötigt man eine zusätzliche ACK-Nachricht des Clients, die bestätigt, dass eine Antwort angekommen ist. Bei Systemausfällen gibt es auch bei *At-Most-Once* keine Garantie, also *At-Most-Once* kann dann ggf. nicht eingehalten werden.

Exactly-Once bedeutet, dass ein Request, komme was wolle (auch bei einem Systemausfall), genau einmal ausgeführt wird. Wenn man die Anforderung genau

betrachtet, ist sie sogar noch strenger als die Anforderung, die an lokale Aufrufe gestellt wird, da im lokalen Umfeld auch nicht ohne weiteres garantiert werden kann, dass die Ausführung einer Methode nicht durch einen Systemausfall unterbrochen werden kann. Will man *Exactly-Once* vollständig erfüllen, erfordert dies für die Behandlung von Systemausfällen ausgefeilte Recovery-Mechanismen, ein konsistentes Zurücksetzen und damit die Verwaltung von Wiederaufsetzinformationen über Logging-Mechanismen. Diese Mechanismen sind in verteilten Transaktionssystemen und Datenbankmanagementsystemen zu finden. *Exactly-Once* kann also nur durch komplexe Transaktionsmechanismen (Alles-oder-nichts-Regel) gewährleistet werden. Für die herkömmliche Kommunikation ist eine Umsetzung dieser Fehlersemantik zu aufwändig.

Verschiedene Beispielsituationen sollen die Problemstellung nochmals verdeutlichen. Es kann u.a. passieren, dass

- ein Request verloren geht
- das Ergebnis des Servers verloren geht
- der Server während der Ausführung des Requests abstürzt
- der Server für die Bearbeitung des Requests zu lange braucht.

In Abbildung 1-13 ist an dem Fehlerszenario „Ergebnis des Servers geht verloren“ dargestellt, was diese Fehlersituation nach sich zieht. Der Server hat die Auftragsbearbeitung schon ausgeführt und muss sich das merken, bei *Exactly-Once* sogar über einen Systemausfall hinweg. Der Client muss mit einer erneuten Anfrage reagieren, der Server darf aber die Anfrage bei *At-Most-Once* und *Exactly-Once* nicht noch einmal bearbeiten, sondern muss das temporär gespeicherte Ergebnis an den Client senden.

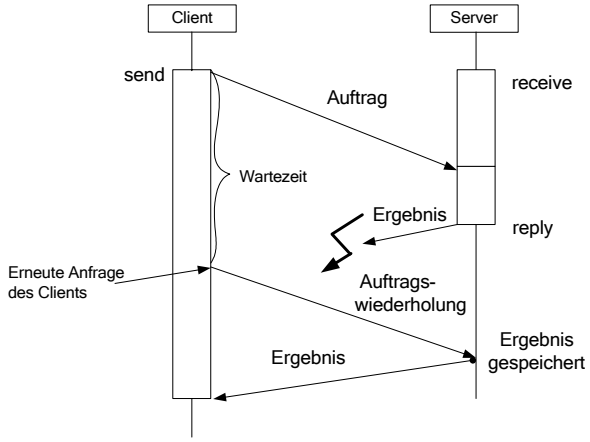


Abbildung 1-13: Beispielsituation: Ergebnis geht verloren nach Weber (1998)

Wie wir in den folgenden Kapiteln noch sehen werden, ist in Technologien wie CORBA und Java RMI maximal die Fehlersemantik *At-Most-Once* realisiert. Exact-

ly-Once erfordert spezielle Mechanismen, die einen Ausfall überleben. Nutzbar sind hier z.B. Transaktionsmechanismen für verteilte Systeme. Transaktionsprotokolle, die *Exactly-Once* garantieren sollen, verwenden für die Ergebniskoordination sog. Commit- oder Koordinationsprotokolle (1-Phase-Commit, 2-Phase-Commit, 3-Phase-Commit). Für einen Entwickler wird die Implementierung der Fehlerbehandlung in höheren Kommunikationsmechanismen wie RPC, Java RMI, CORBA usw. aber verborgen. Er merkt z.B. nichts, wenn eine Antwort nicht rechtzeitig eintrifft. Das System sorgt z.B. für eine erneute Anfrage bei einem Timerablauf. Hier geht es, wie oben bereits beschrieben, um Transparenz. Ein Request soll wie der Aufruf einer lokalen Prozedur behandelt werden.

1.5.3 Transaktionssicherheit in verteilten Informationssystemen

Aus der lokalen Verarbeitung in Großrechnern entwickelte sich das Konzept der Transaktionen als Fehlertoleranzkonzept zur Sicherstellung bzw. Verbesserung der semantischen Integrität der Daten eines Anwendungssystems. Die Grundüberlegung ist, dass es möglich sein muss, mehrere Operationen auf Datenbestände ganz oder gar nicht auszuführen. Man spricht hier von Atomarität. Eine Unterbrechung einer Operationsfolge, die zusammengehört, könnte zu einer Inkonsistenz in den Daten führen. Beispielhaft werden hier häufig eine Soll- und eine Habenbuchung in einem Buchhaltungssystem aufgeführt. Beide Ümsätze müssen entweder ganz oder gar nicht ausgeführt werden, sonst stimmen die Buchhaltungsdaten nicht mehr. In lokalen Umgebungen ist dieses Problem auch als Synchronisationsproblem bekannt und wir nutzen zur Lösung Semaphore, Monitore, Sperren und dergleichen.

Will man die Alles-oder-Nichts-Semantik vollständig implementieren, benötigt man ein System, das den Status der Bearbeitung auch im Fehlerfall rekonstruieren kann, um ein Wiederaufsetzen bei einem Neustart zu ermöglichen. Dieses System bezeichnet man auch als Transaktionssystem. Es muss Mechanismen bereitstellen, die es ermöglichen, jede Zustandsänderung auf einem ausfallsicheren Speicher zu protokollieren. Der Entwickler einer Transaktionsanwendung muss Möglichkeiten haben, in seinem Programm die Grenzen von Transaktionen anzugeben (Begin-Transaction, Commit) oder ggf. auch selbst ein Zurücksetzen einer Transaktion (Rollback) zu initiieren. Dies alles selbst zu programmieren wäre zu aufwändig. Transaktionssysteme sorgen für die notwendigen Basismechanismen.

Das Transaktionskonzept stammt aus der Welt der Datenbanken. Datenbanksysteme unterstützen dieses Konzept schon lange, dort ist es heute eine Selbstverständlichkeit. Keine etwas komplexere Anwendung, die eine Datenbank benutzt, kommt ohne Transaktionen aus.

Seit langem versucht man auch, das Transaktionskonzept in die Welt der verteilten Systeme zu transferieren. Wie bereits diskutiert, kann man z.B. für die Implementierung einer *Exactly-Once*-Fehlersemantik für einen Remote-Procedure-Call das Transaktionskonzept ebenfalls sehr gut nutzen, obwohl es hier nur um die Aus-

führung einer Einzeloperation geht. Wenn man aber ausfallsicher sein möchte, benötigt man Wiederanlaufmechanismen und diese sind in Transaktionssystemen implementiert.

Es stellt sich die Frage, ob das Transaktionskonzept für die Kommunikation in verteilten Anwendungen genau so sinnvoll ist wie in lokaler Umgebung und welche zusätzlichen Mechanismen man benötigt, um eine Transaktion über Rechnergrenzen hinweg zu realisieren. Eine andere Frage beschäftigt sich grundsätzlich mit dem Sinn von Transaktionen in verteilten Umgebungen und versucht, seine Grenzen für praktisch nutzbare Anwendungen zu ermitteln. Wir werden in diesem einführenden Kapitel nicht weiter auf diese Fragen eingehen, sondern sie in den späteren Kapiteln immer wieder aufgreifen.

1.6 Übungsaufgaben

1. Versuchen Sie eine Definition für ein verteiltes Informationssystem mit eigenen Worten!
2. Nennen Sie drei Gründe für die Verteilung eines betrieblichen Informationssystems und erläutern Sie diese!
3. Welche Arten der Heterogentät wurden identifiziert?
4. Welche Fehlersituationen muss eine At-Most-Once-Implementierung für einen entfernten Aufruf zusätzlich zu At-Least-Once implementieren und wie kann die Implementierung erfolgen?
5. Wird in lokalen Methodenaufrufen innerhalb eines Prozesses eine *Exactly-Once*-Fehlersemantik garantiert?
6. Ist es sinnvoll, dass eine Java-Anwendung und eine C#-Anwendung ohne weitere Maßnahmen über eine Kommunikationsschnittstelle Objekte austauschen? Begründen Sie Ihre Entscheidung!
7. Wie löst Java das Problem der Heterogenität in der reinen Java-Welt?
8. Was ist ein Big-Endian- im Unterschied zu einem Little-Endian-Format?
9. Wozu braucht man in verteilten Systemen eine symbolische Adressierung der verteilten Bausteine und welche Lösung gibt es hierfür?
10. Wozu braucht man Object-Relational-Mapper?
11. Erläutern Sie anhand eines Beispiels, wie eine Verteilung von Anwendungsbausteinen dazu beitragen kann, dass ein Anwendungssystem ausfallsicherer wird!

2 Konzepte und Modelle verteilter Kommunikation

Die Entwicklung von Kommunikationsanwendungen ist in den letzten Jahren aufgrund neuer Technologien immer komfortabler geworden. Vor nicht allzu langer Zeit musste man sich bei der Programmierung noch mit allen Aspekten (auch der niedrigen Schichten) der Kommunikation befassen. Heute gibt es Programmiermodelle, die dies wesentlich erleichtern und man setzt mindestens auf eine vernünftige Transportzugriffsschicht auf. Für die Entwicklung von verteilten Anwendungen kann man also meistens einen funktionierenden Transportdienst nutzen. Eine Methode der Programmierung basiert auf der Socket-Schnittstelle. Dies ist eine Transportzugriffsschnittstelle, die in mehreren Sprachen implementiert ist und aus der Unix-Welt kommt. Moderne Sprachen wie Java und C# stellen eine komfortable Nutzungsmöglichkeit über vordefinierte Packages (Java) oder Namespaces (C#) bereit. Sockets sind zwar die Basis für alle im Internet vorhandenen höheren Programmiermodelle, jedoch ist die Programmierung verteilter Anwendungen direkt über die Sockets-Schnittstelle relativ aufwändig. Wir beschäftigen uns in diesem Buch mit komfortableren Mechanismen. Weiter fortgeschritten sind Konzepte wie RPC (Remote Procedure Call) und noch moderner sind objektorientierte Kommunikationsmechanismen wie RMI (Java-Welt), .NET Remoting (Microsoft) oder CORBA (unabhängiger Standard). Ganz aktuell sind komponentenorientierte Kommunikationsparadigmen wie EJB (Enterprise Java Beans), aber auch Message-orientierte Systeme sind heute weit verbreitet. Moderne Kommunikationsmechanismen werden heute vorwiegend von Middleware-Systemen angeboten, die auch als Application-Server bezeichnet werden.

Die Art und Weise, wie man eine verteilte Anwendung programmiert, hängt von den Anforderungen ab. Die meisten verteilten Anwendungen stützen sich heute auf das Client-Server-Modell, in dem ein Serverprozess oder ggf. mehrere Serverprozesse auf Requests von Clients warten und diese beantworten. In diesem Fall geht die Kommunikation immer vom Client aus. Aber auch andere Kommunikationsparadigmen sind häufig anzutreffen. Ein anderes Modell ist die sog. Peer-to-Peer-Kommunikation, in der zwei Anwendungsprozesse gleichberechtigt miteinander kommunizieren. Beispielsweise benötigt man in Automatisierungsanwendungen meist eine gleichberechtigte Kommunikation, in der jeder Partner zu jeder Zeit eine Nachricht (oft Telegramm), senden kann, wenn ein bestimmtes Ereignis (wie z.B. „eine Palette ist an einem bestimmten Meldepunkt zum Einlagern in ein Hochregal bereit“) eintritt. Weiterführende Kommunikationsmechanismen befassen sich mit gesicherten Message-Queues, transaktionsgesicherter Kommunikation mehrerer Objekte usw.

In diesem etwas umfangreicheren Kapitel wird eine Einführung in typische Softwarekonzepte und Implementierungsmöglichkeiten verteilter Anwendungssysteme gegeben. Wir beginnen mit dem Client-Server-Modell und diskutieren anschließend Varianten, mit denen sich das Client-Server-Modell implementieren lässt. Im Blickwinkel stehen verteilte Prozeduraufrufe, verteilte Objekte, verteilte Komponenten und Message-Passing-Systeme. Die eingeführten Modelle werden anhand von konkreten Technologie-Beispielen vertieft.

Zielsetzung des Kapitels

Der Leser soll in diesem Kapitel einen Überblick über gängige Konzepte und Modelle für die verteilte Kommunikation erhalten und ein grundlegendes Verständnis darüber erwerben, wie man Technologien für verteilte, betriebliche Informationssysteme einsetzt.

Wichtige Begriffe

Synchrone und asynchrone Kommunikation, zustandslose und zustandsbehaftete Server, meldungs- und auftragsbezogene Kommunikation, Client-Server-Modell, Trader, Verteilte Prozeduren, Verteilte Objekte, Verteilte Komponenten, Message-Passing, Nachrichtenwarteschlangensystem, CORBA, EJB, RMI, .NET Remoting, RPC, Sun ONC.

2.1 Client-Server-Modell

Das Client-Server-Modell ist eines der wichtigsten Softwaremodelle für verteilte Anwendungen und soll etwas ausführlicher dargestellt werden. Implementierungsüberlegungen sollen ebenfalls betrachtet werden. Das Client-Server-Modell wird gelegentlich als Hardwaremodell dargestellt: Auf der Clientseite werden kleinere Rechner eingesetzt, auf der Serverseite meist größere Serverrechner. Trotzdem ist das Modell in erster Linie ein Softwaremodell.

Im Client-Server-Modell gibt es zwei verschiedene Bausteintypen, Server und Clients, wobei ein Server wiederum Client eines anderen Servers sein kann. Client und Server kommunizieren über ein Request-/Response-Verhalten. Der Server bietet Dienste an, die der Client nutzt. Hierzu sendet der Client eine Nachricht, einen sog. Request, an den Server. Dieser bearbeitet den Request und sendet das Ergebnis zurück. In Abbildung 2-1 ist diese Art der Kommunikation skizziert, wobei ein Client einen Dienst eines beliebigen Anwendungsservers aufruft. Dieser nutzt zur Bearbeitung des Requests wiederum einen Datenbankserver, um bestimmte Daten anzufordern und sendet deshalb einen Request an diesen. Der Applikationsserver ist also in einer Doppelrolle. Alle Requests werden in diesem Beispiel synchron und blockierend bearbeitet. Der Aufrufer wartet auf die Antwort. Wie wir noch sehen werden, gibt es auch andere Möglichkeiten.

Im Bild wird deutlich, dass es sich um eine dreischichtige Architektur handelt. Hier sind, wie wir in Kapitel 5 noch sehen werden, verschiedene Ansätze denkbar (2-, 3-, n-schichtige Architekturen). In größeren Applikationen sind drei oder mehr Schichten üblich.

Die vom Server bereitgestellten Funktionen werden auch als *Services* oder Dienste bezeichnet. Auch das ISO/OSI-Modell verwendet den Dienstbegriff. Die Beschreibung der Dienstschnittstelle repräsentiert einen Vertrag zwischen Diensterbringer (= Server) und Dienstnutzer (= Client).

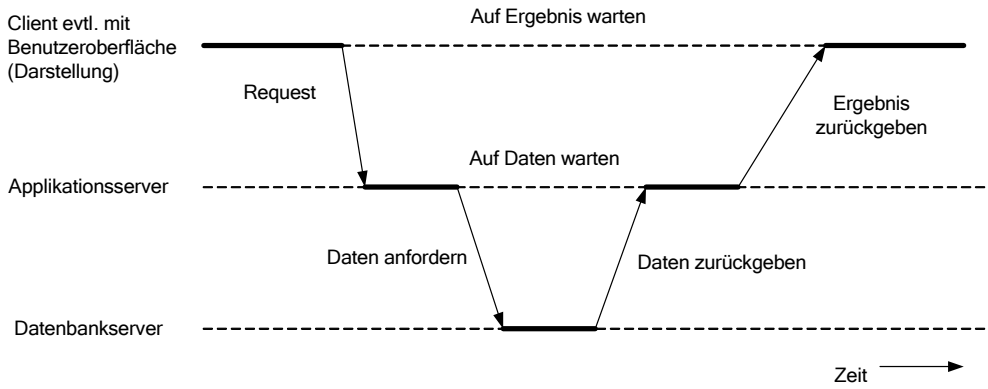


Abbildung 2-1: Beispiel einer Client-Server-Kommunikation nach (Tanenbaum 2003)

Der Dienstbegriff spielt in vielen verteilten Anwendungen eine zentrale Rolle und ist neuerdings über den Begriff *SOA* (Service Oriented Architecture) im Rahmen von Webservices wieder sehr aktuell geworden.

2.1.1 Kommunikations- und Interaktionsformen

Beim Nachrichtenaustausch unterscheidet man nach (Weber 1998) grundsätzlich zwischen *synchroner* und *asynchroner* Kommunikation. *Synchron* bedeutet im Zusammenhang mit der Programmierung von verteilten Anwendungen eine blockierende Kommunikation, d.h. der Sender wartet solange, bis eine Methode, die oft den Namen *send* (oder so ähnlich) hat, mit einem Ergebnis zurückkehrt und kann in dieser Wartezeit nichts tun. Asynchron bedeutet, dass der Sender nach dem Absenden der Nachricht mit Hilfe einer Methode *send* nicht blockiert und bis zum Eintreffen eines Ergebnisses andere Aufgaben erledigen kann. Die Antworten müssen dann irgendwie eingesammelt werden. Bei der asynchronen Kommunikation, hat man den Vorteil der zeitlichen Entkoppelung von Sender und Empfänger. Damit sind auch eine bessere Parallelarbeit sowie eine ereignisgesteuerte Kommunikation möglich. Von Nachteil ist, dass eine Zwischenpufferung der Nachrichten

notwendig ist. Wenn der Puffer voll wird, dann führt dies trotzdem zum Blockieren, um eine gesicherte Übertragung zu gewährleisten.

Weiterhin unterscheidet man nach (Weber 1998) grundsätzlich zwischen *meldungsorientierter* und *auftragsorientierter* Kommunikation. Während die meldungsorientierte Kommunikation eine Einwegnachricht ohne Antwort ist, spricht man bei der auftragsorientierten Kommunikation von einen Request/Response-Mechanismus, bei dem der Sender ein Ergebnis vom Empfänger erhält (entfernter Dienstaufruf). Synchrone und asynchrone Kommunikation lässt sich mit meldungs- und auftragsorientierter Kommunikation jeweils kombinieren.

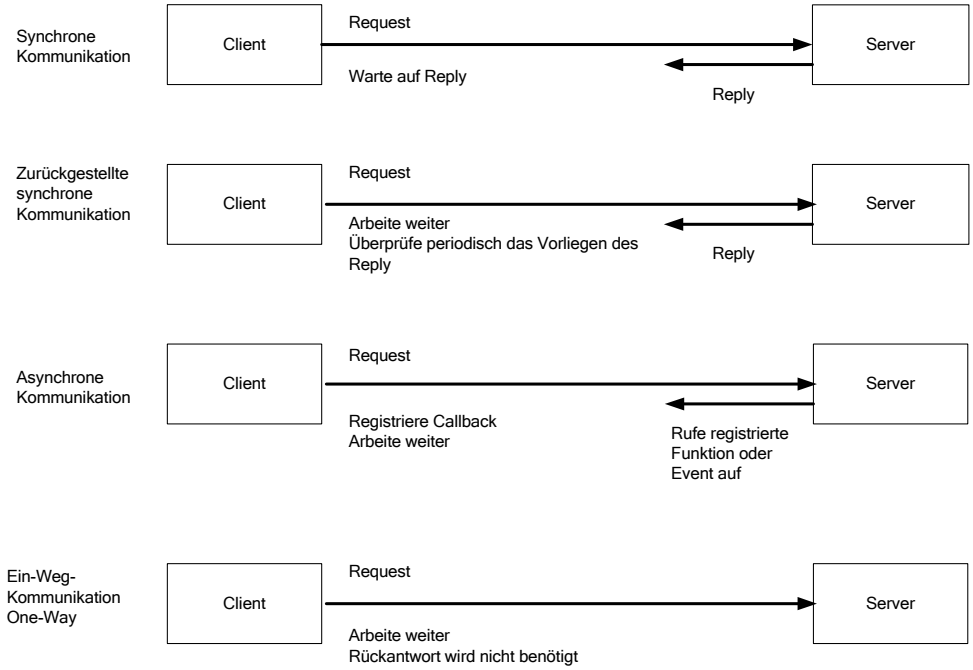


Abbildung 2-2: Interaktionsformen nach (Bengel 2004)

Auch die Art der Interaktion kann variieren. Standardmäßig wird in Implementierungen des Client-Server-Modells die synchrone Kommunikation verwendet. Es ist aber auch möglich, dass ein Request unidirektional (One-Way, Ein-Weg-Kommunikation) bearbeitet wird, wenn keine Antwort erforderlich ist. Es kann auch sein, dass der Client nicht synchron, sondern asynchron auf das Ergebnis wartet und der Server den Client informiert, wenn er das Ergebnis bearbeitet hat.

Bei der asynchronen Interaktion gibt es wiederum die Möglichkeit, dass der Client von Zeit zu Zeit nachfragt (polling), ob die Antwort schon da ist, oder der Client stellt eine Callback-Routine bereit, die (z.B. von einer Middleware) dann aufgeru-

fen wird, wenn die Antwort auf der Clientseite angekommen ist. Die verschiedenen Arten der Interaktion sind in Abbildung 2-2 dargestellt (siehe auch Bengel 2004).

2.1.2 Implementierungskonzepte für das Client-Server-Computing

Client-Server-Systeme bzw. die zugrundeliegenden Basissysteme, die meist als Middleware realisiert werden, können anhand verschiedener konzeptioneller Fragestellungen näher beleuchtet und miteinander verglichen werden. Die folgenden gundlegenden Konzeptfragen sollen herangezogen werden:

1. Was sind die wesentlichen Architekturkonzepte?
2. Welche Konzepte der Zustandsverwaltung werden unterstützt?
3. Wie werden Dienstschnittstellen von Servern beschrieben?
4. Wie werden Parameter und Returnwerte bei Dienstaufrufen übergeben?
5. Wie wird das Marshalling und Unmarshalling durchgeführt?
6. Welche Kommunikationsmechanismen (synchron, asynchron, Fehlersemantik) und welche Mechanismen zur Adressierung von Diensten (Naming-Service) werden bereitgestellt?
7. Wie wird das verteilte Garbage-Collection ausgeführt?
8. Welche Nebenläufigkeitsunterstützung wird im Server zur Verfügung gestellt?
9. Welchen Lebenszyklus durchlaufen Server?
10. Welche Basisdienste werden von einer Client-Server-Basislösung noch bereitgestellt?
11. Welche sonstigen konzeptionellen Unterscheidungsmerkmale gibt es (spezielle Mechanismen, Möglichkeiten der Skalierbarkeit, Erhöhung der Verfügbarkeit,…)?

Wir ziehen diese Fragestellungen für die weitere Betrachtung der verschiedenen Lösungsansätze insbesonders bei den Fallbeispielen zu Vergleichszwecken heran.

Da die Terminologie in der Literatur uneinheitlich ist, sollen zunächst für die weitere Betrachtung einige Definitionen festgelegt werden. Wir unterscheiden in unserer Modellbetrachtung zwischen einem Server, einem Serverbaustein, einem Service (Dienst) und einer Servermethode- bzw. -prozedur (manchmal auch als Dienstprimitive bezeichnet):

- Ein Server stellt eine Ablaufumgebung für einen oder mehrere Serverbausteine bereit.
- Ein Serverbaustein ist ein Objekt oder ein Modul (je nach verwendeter Programmiertechnik), das zum Ablaufzeitpunkt instanziiert und bei Bedarf einem Client für die Abarbeitung einer Anforderung (eines Requests) zugeordnet wird.

- Ein Service oder Dienst wird durch einen Serverbaustein bereitgestellt und enthält ein oder mehrere Servermethoden oder Serverprozeduren.
- Eine Servermethode oder eine Serverprozedur ist Bestandteil eines Service, den ein Client durch das Senden eines entsprechenden Requests nutzen kann. Ein Serverbaustein stellt eine oder mehrere Methoden bzw. Prozeduren in einem oder mehreren Diensten bereit. Vereinfachend sprechen wir auch schon von einem Dienst, wenn es sich auch nur um eine Servermethode oder Prozedur, die ein Serverbaustein bereitstellt, handelt.

Die Beziehungen zwischen den Modellkomponenten sind in Abbildung 2-3 dargestellt. Wir gehen etwas vereinfacht davon aus, dass ein Serverbaustein nur in einem Server abläuft und ein Service nur von einem Serverbaustein bereitgestellt wird. In der Praxis sind, insbesondere aus Skalierungsgründen auch andere Zuordnungen üblich. Ein Client kann beliebig viele Servermethoden nutzen und diese stehen – im allgemeinen Fall – auch beliebig vielen Clients zur Verfügung.

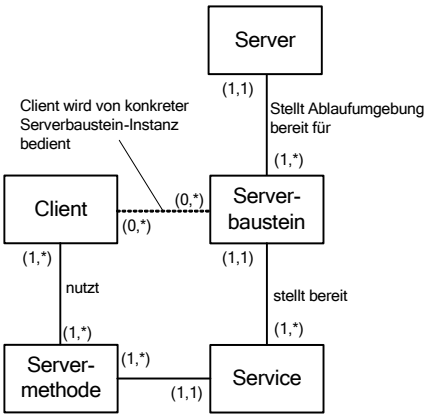


Abbildung 2-3: Beziehungen zwischen den Modellkomponenten

Man kann auch noch zwischen einem Server und seiner konkreten Instanzierung, sowie zwischen einem Serverbaustein und seiner konkreten Instanzierung unterscheiden. Zur Laufzeit sind natürlich nur konkrete Ausprägungen im Spiel. Zur Laufzeit wird auch eine Beziehung zwischen einer konkreten Instanz eines Serverbausteins und einer konkreten Instanz eines Clients hergestellt. Dies ist in unserem statischen Modell nur angedeutet, wobei ein Client beliebig viele Serverbaustein-Instanzen nutzen und eine Serverbausteininstanz auch mehrere Clients bedienen kann.

Wir unterscheiden bei Diensten zwischen zustandsinvarianten und zustandsändernden Diensten. Zustandsinvariante Dienste liefern nur Daten, wie beispielsweise ein Webserver mit rein statischen HTML-Seiten. Diese Arten von Diensten sind relativ unkritisch und leicht zu pflegen. Man kann die Dienstaufrufe beliebig oft

wiederholen ohne dass sich etwas ändert. Man nennt diese Dienste auch idempotent. Bei zustandsändernden Diensten kann ein Request auch zu einer Veränderung eines Zustands führen. Beispiele hierfür sind die meisten Server von komplexeren Anwendungen mit Datenbanken. Bei einem Request kann hier der Inhalt der Datenbank geändert oder erweitert werden. Auch die Reihenfolge der Abarbeitung spielt bei zustandsändernden Servern eine Rolle. Ein Serverbaustein kann sowohl zustandsinvariante als auch zustandsändernde Dienste anbieten.

Im Weiteren verwenden wir den Begriff Server vereinfachend auch für einen Serverbaustein und auch für eine Serverbaustein-Instanz, wenn der Kontext eindeutig ist.

Ein Server empfängt die Client-Requests und leitet diese zur Verarbeitung an den entsprechenden Serverbaustein. Ein Server ist seinerseits in eine Ablaufumgebung innerhalb eines Betriebssystems eingebettet. Eine konkrete Realisierung und Einbettung in eine Betriebssystemumgebung sieht beispielsweise so wie in Abbildung 2-4 skizziert aus.

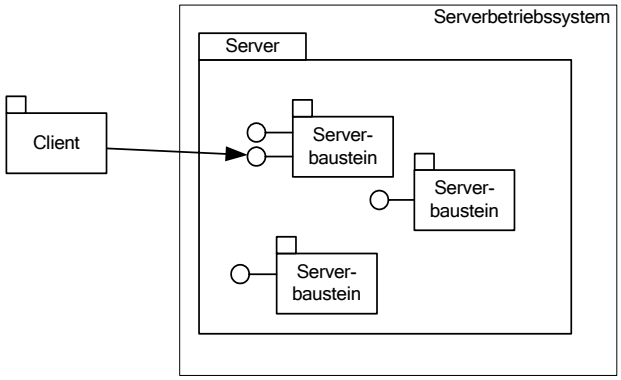


Abbildung 2-4: Server, Serverbausteine und Servermethoden

Architektur

Die Architekturkonzepte der verschiedenen Lösungen sind an manchen Stellen sehr ähnlich. So wird das Konzept des Client-Stubs und des Server-Skeletons praktisch überall verwendet. Aber an einigen Stellen gibt es auch größere Unterschiede in der Architektur. Die wesentlichen Merkmale der Architekturkonzepte einiger Client-Server-Lösungen sollen bei dieser Fragestellung anhand von konkreten Fallbeispielen erörtert werden.

Zustandsverwaltung

Bei *stateful* Servern (also Serverbausteinen) merkt sich die Serverseite in einem Gedächtnis, also einer speziellen Datenstruktur oder einem Objekt einen Konversations- oder Sessionkontext über einen Request hinweg. Ein *stateless* Server vergisst

nach einem Request die gesamte Bearbeitung und kann beim nächsten Request nicht darauf aufsetzen.

Unter einem Singleton versteht man im Software Engineering ein spezielles Software-Pattern. Es verhindert, dass mehr als eine Objektinstanz einer Klasse erzeugt wird. Wenn ein Server entwickelt werden muss, der für mehrere Clients Daten im Hauptspeicher (z.B. ein Software-Cache) verwaltet, dann eignet sich ein Singleton. Der Begriff sagt noch nichts über die Nebenläufigkeit der Zugriffe aus. Greifen mehrere Clients nebenläufig auf ein Singleton, muss dieses thread-safe implementiert sein.

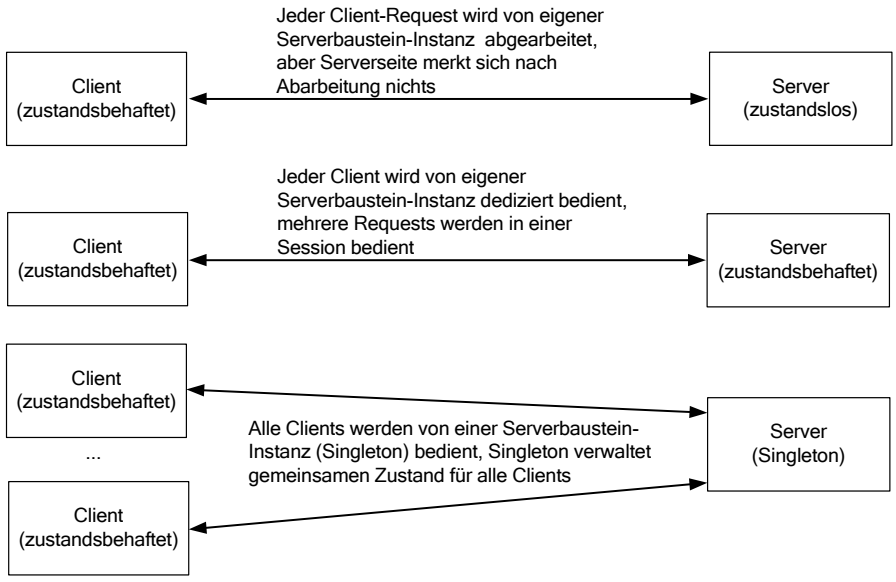


Abbildung 2-5: Client-Server-Beziehungen unter Berücksichtigung des Zustands

In Abbildung 2-5 sind die drei Varianten dargestellt. Zustandslose Server sind im Vergleich zu zustansbehafteten Servern von Natur aus robuster und können nach einem Ausfall einfach wiederaufsetzen. Zustandsbehaftete Server sind wesentlich komplizierter und auch bei einem Fehlerfall schwerer wieder aufzusetzen. Ein Recovery des Gedächtnisses funktioniert nur, wenn man alle Zustandsveränderungen und den aktuellen Zustand persistent speichert. Bei einem Serverausfall sind daher meist die Verbindungen zu den Clients weg und die Clients kennen ihren Zustand im Server nicht. Bei einem Client-Ausfall weiß der Server nicht, wie lange er den Zustand des abgestürzten Clients aufheben soll und ob er jemals wieder aktiv wird. Wann kann er die Zustandsinformationen vernichten, also Garbage-Collection betreiben? Netzwerkprobleme verursachen ähnliche Situationen. Der

Server kann sich in diesem Fall mit einer Timerüberwachung behelfen. Ein Sessionzustand wird dann wie z.B. in Webanwendungen (siehe WWW-007) nach einer vorgegebenen Zeit verworfen.

Im Übrigen muss bei zustandsbehafteten Servern auch der Client über den aktuellen Zustand der Verarbeitung Bescheid wissen. Meist wird vom Server bei Beginn einer Konversation eine Session-Id erzeugt und dem Client mit der Antwort übergeben. Diese sendet der Client bei jeder erneuten Anfrage innerhalb derselben Konversation im Request mit. Über einen solchen Mechanismus kann sich der Client auch Session-Informationen dauerhaft gewissermaßen als Profil speichern (vgl. hierzu Cookies in Web-Browsern) und diese bei einer späteren Konversation gleich wieder im Request mitsenden.

Als Empfehlung gilt, dass man als Architekt einer verteilten Anwendung so lange wie möglich mit zustandslosen Servern auskommen sollte. Dies geht in vielen Fällen, was ein klassisches Beispiel verdeutlichen soll:

Beispiel: Wenn man eine große Anzahl von Objekten bzw. Datensätzen (z.B > 1000) des selben Typs über einen Request lesen möchte, weil der Client diese zur Bearbeitung benötigt, ist es meist nicht sehr effizient, dies in einem Aufruf zu erledigen. Auf der Serverseite wird bei Einsatz einer Datenbank in der zugehörigen Requestbearbeitung eine Selektion abgesetzt (*select * from ... where ...*), die diese Daten besorgen soll. Nehmen wir an, es wären 5000 Sätze (z.B. alle Aufträge einer Auftragsbearbeitung), die in der Ergebnismenge sind. Eine SQL-Datenbank würde hierzu einen *Cursor* anlegen, der die aktuelle Leseposition in der Ergebnismenge markiert. Man könnte im Design nun entscheiden, dem Client folgende zwei Operationen bereitzustellen:

```
findData(...)
findNextData(...)
```

Mit *findData* wird die Suche durchgeführt bzw. neu initiiert und die ersten n Sätze werden als Ergebnis zurückgegeben. Mit *findNextData* werden die nächsten n Sätze an den Client geliefert. Die Bereitstellung dieser Operationen könnte im Server nur implementiert werden, wenn der aktuelle Lesezeiger (Cursor) für jeden Client in einem Sessionkontext verwaltet wird. In diesem Fall haben wir einen *stateful* Server. Garbage-Collection-Probleme zur Beseitigung der gespeicherten Daten, für den Fall, dass der Client sich nicht mehr meldet, müssten zusätzlich behandelt werden. Man könnte das Problem aber auch zustandslos lösen, indem man eine Operation *findData* mit einem zusätzlichen Parameter versieht, der den Schlüssel des zuletzt gelesenen Satzes enthält. In diesem Fall hätte der Server bei Ankunft eines Requests immer die vollständige Information, um eine neue Selektion anzustoßen und müsste sich keine Zustandsinformation speichern. Sollte die Leistung aufgrund der mehrfachen Zugriffe auf die Datenhaltung nicht ausreichen, könnte man sich noch zusätzlich den Einsatz von Caching-Mechanismen überlegen.

Dies geht natürlich nicht immer. Ein Web-basiertes Shopsystem benötigt zum Beispiel den Zustand eines Warenkorbs und die wenigsten Webshops senden immer die gesamte Zustandsinformation über die entprechenden Requests. Die meisten führen ein Gedächtnis (Konversationszustand) für jeden Client. Ähnliches gilt für eine Remote-Login-Session (siehe Telnet) und für sicherheitsrelevante Zustandsinformationen. Generell kann man festhalten, dass alle Anwendungen, die eine kontinuierliche Konversation benötigen, auch Client-bezogene Sessioninformation im Server verwalten müssen. Oftmals ist ein Abbruch einer Session, wie z.B. bei einer Telnet-Session, aber auch nicht so problematisch, da sie einfach wieder aufgesetzt werden kann.

Dienstschnittstellen

Ein verteilter Softwarebaustein wird über seine *Schnittstellen* angesprochen. Es muss also einen Mechanismus geben, mit dem eine Dienstschnittstelle eines Serverbausteins beschrieben werden kann.

Generell unterscheidet man zwei Varianten der Schnittstellenbeschreibung:

- Die Schnittstellenbeschreibung ist eingebettet in die Hostsprache. Konkrete Beispiele hierfür sind die Implementierungen von Java-RMI und .NET Remoting.
- Die Definition der Schnittstellen erfolgt über eine eigene Definitionssprache. Beispiel hierfür ist die CORBA IDL (Interface Definition Language).

Jede Methode bzw. Prozedur ist mit ihren Ein- und Ausgabeparametern, ihrem Rückgabewert und evtl. den Exceptions (Ausnahmen), die beim Aufruf auftreten können, zu beschreiben. Wir werden die einzelnen Lösungsalternativen noch anhand von Beispielen näher betrachten.

Parameterübergabe

Bei verteilten Aufrufen werden, ähnlich wie bei lokalen Aufrufen, vom Client Parameter an den Server übergeben und der Server kann Rückgabewerte zurückliefern. Aus Sicht des Anwendungsprogrammierers sieht dies wie bei einem lokalen Prozeduraufruf aus, allerdings kann eine Parameterübergabe nicht über den lokalen Stack erfolgen. Client und Server laufen schließlich meistens in eigenen Adressräumen und sogar auf unterschiedlichen Rechnersystemen, möglicherweise sogar in unterschiedlichen Betriebssystemtypen ab.

Bei der Übergabe von Parametern unterscheidet man mehrere Call-Semantiken, die in der Literatur teilweise auch unterschiedlich benannt werden:

- *Call-by-value*: In diesem Fall wird ein Parameter als Wert übermittelt. Dies ist die einfachste Parameterübergabe-Variante und wird bei verteilten Aufrufen auch als *Call-by-copy* bezeichnet. Es wird eine Kopie des Parameters übertragen. Änderungen im Server haben keine Auswirkungen auf den Client.

- *Call-by-reference*: Bei dieser Variante wird eine Referenz auf eine Variable übertragen. Alle Änderungen des Servers sind dann sofort im Client sichtbar. Dies ist bei verteilten Aufrufen etwas komplizierter als bei lokalen Aufrufen. Während bei lokalen Aufrufen einfach eine Adresse der Speicherzelle, in welcher der Wert der Variablen liegt, übergeben wird, kann ein Server, der in einem anderen Adressraum und evtl. sogar in einer anderen Prozessorarchitektur abläuft, nichts mit einer lokalen Referenz eines entfernten Clients anfangen. Referenzen sind im verteilten Umfeld also etwas komplexere Verweise auf entfernte Variablen, die als Adressbestandteile auch Informationen über das Rechnersystem und den Prozess, in dem die Variable zu finden ist, enthalten. Client und Server brauchen also eine Vereinbarung über das genaue Format einer Remote-Referenz. Wie wir noch sehen werden, wird bei Java-RMI Call-by-reference für Remote-Objekte und bei CORBA für IORs (Interoperable Object References) eingesetzt.
- *Call-by-copy/copy-back* (Synonyme: *Call-by-copy-restore*, *Call-by-value-result)*: Im Unterschied zu call-by-reference wird hier nicht direkt auf der Originalvariable gearbeitet. Der Server arbeitet mit einer Kopie des Parameters, der nach der Bearbeitung an die Adresse, auf welche die Referenz verweist, zurückkopiert wird. Die Parameter werden vor dem Versenden also kopiert. Damit werden Änderungen beim Kommunikationspartner erst nach Beendigung des Aufrufs sichtbar.

In verteilten Umgebungen spricht man meist von *Call-by-copy* und meint damit semantisch *Call-by-value*. Die Realisierung von Call-by-reference ist in verteilten Systemen auch schwierig und wird daher durch von *Call-by-copy/copy-back* simuliert. Wir verwenden aber vereinfachend auch hier die Bezeichnung *Call-by-reference.* Wir werden bei der weiteren Betrachtung der verschiedenen Modelle der Kommunikation noch auf die jeweilig unterstützten Call-Semantiken eingehen.

Marshalling und Unmarshalling

Wie bereits in den einführenden Kapiteln erläutert, müssen Daten bzw. Objekte, die zwischen zwei verteilten Bausteinen ausgetauscht werden, in eine unabhängige Transfersyntax gebracht werden. Gemäß ISO/OSI-Referenzmodell ist dies eine Aufgabe der Schicht 6. Man spricht im Client-Server-Umfeld auch von Marshalling (Serialisierung) und Unmarshalling (auch Demarshalling oder Deserialisierung). Es gibt grundsätzlich zwei Varianten von Transfersyntaxen:

- Tag-basierte Transfersyntax wie z.B. ASN.1. Hier wird jedes Objekt bzw. Datum mit einem vorangestellten Tag beschrieben. Die Information wird in Form von (Tag, Value)-Paaren übermittelt. In der Regel wird auch die Länge des Datums übermittelt. Man spricht dann von TLV-Kodierung (Type, Length, Value).
- Tag-freie Transfersyntax wie bei Sun ONC (XDR) oder CORBA NDR (Network Data Representation): Hier wird die Beschreibung der Daten aufgrund

der Stellung in der Nachricht festgelegt. Der Aufbau der Datenstrukturen ist dem Sender und dem Empfänger bekannt.

Auf Basis der Beschreibung in der jeweiligen Transfersyntax kann eine, meist automatisierte, Erzeugung von Marshalling- und Unmarshalling-Routinen durchgeführt werden. Diese Routinen werden für die Laufzeit zur Abbildung der lokalen Syntax in die Transfersyntax und umgekehrt benutzt. Bei Nutzung der Transfersyntax ASN.1 sind die entsprechenden Kodierungsregeln in den sog. Basic Encoding Rules (BER) des entsprechenden ISO-Standards ITU-T[1] X.690 spezifiziert.

Adressierung und Kommunikation

Für die Kommunikation zwischen Client und Server ist eine geeignete *Adressierungsmöglichkeit* und ein *Kommunikationsprotokoll* erforderlich. Eine komfortable Adressierung benötigt einen Adressauflösungsmechanismus, der als *Naming- oder Directory-Service* bezeichnet wird. Nach der Bestimmung der Serveradresse über einen Naming-Service muss ein sog. *Binding* stattfinden, in dem der Client eine *Verbindungskontext* zu einem Server aufbaut. Diese Aspekte sollen hier allgemein erläutert werden:

Namensauflösung und Naming-/Directory-Service: Clients müssen die Dienstimplementierungen im Netzwerk finden können. Es muss also einen Mechanismus geben, der das Auffinden von entfernten Serverbausteinen bzw. Diensten unterstützt, ohne komplizierte Adressierungsinformationen über den gesamten Protokollstack kennen zu müssen. Betrachtet man z.B. die TCP/IP-Protokollfamilie, so ist ein Anwendungsprozess letztendlich über eine Transportadresse bestehend aus einer IP-Adresse und einem TCP- oder UDP-Port adressierbar. Möchte man ein Objekt oder eine Prozedur adressieren, benötigt man neben der IP-Adresse und dem Port noch weitere Informationen wie z.B. eine Objektidentifikation. Hinzu kommt, dass Adressen sich auch bei einer Neukonfiguration verändern können. Damit man diese eher kryptischen Adressen nicht im Programmcode verankern muss, gibt es in den jeweiligen Modellen Basisdienste, die eine symbolische Adressierung erleichtern. Diese Dienste werden auch als Naming- oder Directory-Services bezeichnet. Wir wollen hier nur die Grundlagen von Naming- und Directory-Services vorstellen. Konkrete Implementierungen sind z.B. der Portmapper in Sun RPC, der CORBA-Naming-Service, JNDI-Implementierungen im Java-JEE-Umfeld oder *rmiregistry* bei Java RMI. Sie werden weiter unten in den Fallbeispielen betrachtet.

In Abgrenzung zu Datenbanken sind Directory-Services (Verzeichnisdienste) für einfache, hierarchisch geordnete Informationen prädestiniert. Directory-Services dienen heute dazu, symbolische, einfach verständliche Namen für bestimmte Enti-

[1] ITU-T ist der Telekom-Sektor der International Telekommunication Union (ITU) und löste 1993 die CCITT ab. Die ITU-T dient der Standardisierung im Telekom-Umfeld.

tätstypen wie Rechneradressen, Adressen verteilter Objekte, Benutzer und Rollen, Telefonnummern usw. bereitzustellen. Der Zugriff ist überwiegend lesend mit einfachen Abfragen und ohne Transaktionen gedacht. Genau genommen gibt es einen Unterschied zwischen Naming- und Directory-Service. Letzterer bietet insbesondere bei der Suche nach Einträgen mehr Möglichkeiten.

In einem Verzeichnisdienst gibt es eine definierte Regelung, welche Namen die Objekte im Verzeichnis erhalten. Es gibt sog. atomare Namen und zusammengesetzte bzw. vollqualifizierende Namen. Jedem Objekt können Attribute (Name-Value-Pairs) zugeordnet werden, die auch ausgelesen werden können und man kann die Suche auch über Attributinhalte filtern.

Folgende Naming- und Directory-Services sind heute u.a. im Einsatz:

- In DNS werden z.B. Rechneradressen, E-Mailadressen sowie weitere Entitäten global, also organisationsübergreifend verwaltet. In DNS ist z.B. der Hostname ein atomarer Name. Innerhalb der DNS-Hierarchy hat ein Host einen zusammengesetzten Namen, der aus allen Bestandteilen des Pfads von der Wurzel des DNS-Baumes bis zum eigentlichen Hostnamen besteht (Mandl 2007b).
- NIS bzw. NIS+ ist ein weiterer Namensdienst, über den man Benutzerkennungen und auch Rechneradressen innerhalb von Organisationen verwaltet. Beides sind Standards der Internet-Gemeinde.
- Filesysteme in Betriebssystemen wie Unix oder Windows verfügen auch meistens über eine hierarchische Anordnung der Namen und über einen integrierten Naming-Service.
- Ein international seit 1988 genormter Standard bzw. eine Empfehlung für Verzeichnisdienste ist *X.500* der ITU-T. X.500 wurde als ISO Standard 9594 übernommen und legt fest, dass für die Kommunikation zwischen dem Verzeichnis-Client und dem Verzeichnis-Server das Directory Access Protocol (DAP) verwendet wird. Als ein Protokoll der Anwendungsschicht benötigt DAP den gesamten OSI-Protokoll-Stack, um zu funktionieren. Da dies sehr aufwändig in der Implementierung ist, wurde ein „leichtgewichtigeres" Protokoll namens LDAP (Lightweight Directory Access Protocol) für den Zugriff auf einen X.500-Service-Provider entwickelt.

Initialer Kontext und Bootstrapping: Um den Verzeichnisdienst im Netz zu finden, benötigt man einen sog. *initialen Kontext* (initial context), der die Adresse des Verzeichnisdienstes darstellt. Dieser wird typischerweise über einen speziellen Bootstrapping-Mechanismus besorgt. Der initiale Kontext ist der Einstiegspunkt für die Ausführung von Operationen in einem Verzeichnisdienst. Anzumerken ist hier allerdings, dass derzeit viele Lösungen entweder technologie- oder sogar herstellerabhängig sind. Dies macht den Anwendungscode abhängig. Der sog. Lookup-Code zum Auflösen von symbolischen Adressen in tatsächliche Adressen

von Objekten bzw. Prozeduren ist damit nicht so einfach auf andere Technologien oder Produkte portierbar.

Binding und Kontext: Eine wichtige Frage ist, wie ein Client einen Server finden und mit diesem eine Verbindung eingehen kann. Dieser Vorgang wird auch als *Binding* (Binden) bezeichnet. Üblicherweise registriert sich ein Server beim Start bei einem Verzeichnisdienst (Directory-Service) und ein Client findet die Adresse des Servers über eine Anfrage an das Directory.

Während des Bindings wird ein *Kontext* festgelegt. Ein Kontext ist definiert durch eine Menge von Zuordnungen von Namen zu Verzeichnis-Objekten. Ein Kontext ermöglicht die Auflösung von Namen zu Adressinformationen. Ein Serverbaustein, der ein Objekt bzw. eine Prozedur im Netzwerk bekannt machen möchte, kann dies über eine *bind*-Operation erledigen. Dabei wird ein Kontext erzeugt. Dieser Kontext kann dann von einem Client zur Adressauflösung über eine *lookup*-Operation genutzt werden. Man unterscheidet zwischen statischem und dynamischem Binding:

- Beim statischen Binding gibt der Client die Serveradresse direkt an und das Binden erfolgt zum Übersetzungszeitpunkt des Clients.
- Beim dynamischen Binding wird die Adresse des Servers zur Laufzeit z.B. über einen Broadcast oder eine sog. Broker-Instanz, die die Serveradressen verwaltet, ermittelt.

Kommunikationsprotokoll: Was die Kommunikation zwischen Client und Server anbelangt, so ist ein entsprechendes Kommunikationsprotokoll erforderlich, das in der Regel über TCP oder UDP (Mandl 2008a) liegt. Der Verbindungsaufbau muss im Client und im Server explizit programmiert werden. Vor dem Senden einer Nachricht muss die Nachricht zusammengestellt und so aufbereitet (serialisiert) werden, dass der Server sie interpretieren kann. Der Server empfängt die Nachricht und deserialisiert sie. Nach der Bearbeitung des Requests sendet er eine Antwort (Response-Nachricht), die der Client empfängt und ebenfalls deserialisiert. Wenn ein Server viele Clients bedienen muss, muss der Serverprogrammierer dies ebenfalls „von Hand“ ausprogrammieren.

Wir betrachten in diesem Kapitel höherwertigere Kommunikationsmechanismen, die dem Anwendungsprogrammierer die Entwicklung eigener Kommunikationsprotokolle abnehmen. Grundidee in allen betrachteten Basissystemen ist, dass ein Client den Dienst eines Servers so anfordert, als ob es sich um einen lokalen Prozedur- oder Methodenaufruf handeln würde. Für den Client bleibt die Kommunikation über das Netz weitgehend transparent, der Anwendungsprogrammierer schreibt also in sein Programm einen Prozedur- oder Methodenaufruf und dieser wird auf einen verteilten Aufruf umgesetzt.

Die erforderlichen Kommunikationsmechanismen werden durch eine Softwareschicht verborgen, die auch als *Kommunikations-Middleware* bezeichnet wird. Die folgenden Protokollmechanismen sind bereitzustellen:

- Der Funktionsaufruf muss in eine Nachricht serialisiert (Marshalling) werden.
- Die Nachricht muss sicher übertragen werden. Hierzu gehört das Aussortieren von Duplikaten (Duplikatfilterung), das Wiederholen von Nachrichten bei Nachrichtenverlust und die Fehlerbehandlung anhand einer vorgegebenen Fehlersemantik (*at-least-once*,...).
- Im entfernten Rechner muss die Nachricht entpackt und deserialisiert werden (Unmarshalling) und die adressierte Prozedur ist aufzurufen.
- Die Ergebnisse des Prozeduraufrufs müssen zum Aufrufer zurückgesendet werden, wobei ebenfalls eine Nachricht erzeugt werden muss.
- Auf der aufrufenden Seite wird das Ergebnis aus der Nachricht genommen und schließlich an die Aufrufstelle transportiert, als ob das ganze lokal ausgeführt worden wäre.
- Ein verteilter Garbage-Collection-Mechanismus und ein verteiltes Exception-Handling sind bereitzustellen.

Die Kommunikationsprotokolle für die Client-Server-Kommunikation sind meist synchron, d.h der Client wartet auf das Ergebnis des Servers. Aber auch andere Varianten sind möglich. Das Kommunikationsprotokoll muss aus Sicht des Anwendungsprogrammieres einige Details verbergen und eine gesicherte Kommunikation ermöglichen. Der typische Inhalt eines Requests enthält z.B einen Nachrichtentypen (z.B. 0=Request, 1=Reply), eine Request-Id als Folgenummer (aufsteigende Zahl), eine Adresse des Serverbausteins, die Identifikation der zu rufenden Methode bzw. Prozedur sowie Argumente für den Aufruf.

Garbage-Collection

In objektorientierten Sprachen wie C# und Java ist bekanntlich ein automatischer Garbage-Collection-Mechanismus bereits in der Sprache verfügbar. Das Laufzeitsystem erkennt, ob ein Objekt noch benötigt wird oder nicht. Im letzteren Fall wird es aus dem Heap entfernt. In der lokalen Laufzeitumgebung einer Sprache ist Garbage-Collection zwar performance-kritisch, aber relativ einfach zu implementieren. Es muss von Zeit zu Zeit untersucht werden, ob es Objekte gibt, auf die keine Objektreferenz im lokalen Adressraum mehr verweist. Ist dies der Fall, kann das Objekt eliminiert werden.

Bei einem entfernten Objekt, auf das mehrere andere Objekte referenzieren, ist dies nun nicht so ohne weiteres möglich. Eigentlich kann nur der Server, der es verwaltet, entscheiden, ob es noch benötigt wird oder nicht. Ein verteilter Garbage-Collection-Mechanismus soll erkennen, wenn ein verteiltes Objekt nicht mehr benötigt wird und es dann freigeben. Solange das Objekt aber noch von mindestens einem anderen Objekt benutzt wird, darf es nicht gelöscht werden. Das verteilte Garbage-Collection arbeitet sinnvollerweise mit dem lokalen Garbage-Collection-Mechanismus der jeweiligen Sprache (sofern vorhanden) zusammen.

Eine Variante zur Lösung des Problems basiert auf einem einfachen Zählermechanismus, der auch als verteilter *Reference-Counting-Algorithmus* bezeichnet wird. Jeder Serverprozess verwaltet hierzu eine Liste aller Clientprozesse bzw. Proxies, die entfernte Objektreferenzen auf seine Objekte nutzen. Generell ist es zweckmäßig, beim *verteilten Reference-Counting* zum Zeitpunkt der Erstellung einer neuen Referenz auf ein Objekt und zum Zeitpunkt des Löschens eines Verweises auf ein Objekt vom Nutzer (Client) eine Nachricht an den entfernten Objektserver zu versenden. Der Server kann dann den zugehörigen Referenzzähler entsprechend inkrementieren bzw. dekrementieren. Es ist ein Protokoll notwendig, das zum Austausch dieser Informationen dient. Das Protokoll ist in (Coulouris 2003) wie folgt skizziert:

- Wenn ein Client eine neue Objektreferenz eines entfernten Objekts erhält, ruft es eine bereitgestellte Methode *AddRef(Object)* im Server auf, damit der Server diesen Client in der Liste der Referenzierenden vermerken kann. Im Client wird ein Proxy angelegt.
- Wenn der Client über das lokale Garbage-Collection den Proxy freigibt, wird die serverseitige Methode *RemoveRef(Object)* aufgerufen. Wenn in der Referenzliste zum Objekt keine weiteren entfernten Proxies mehr enthalten sind, kann der Serverprozess bzw. der lokale Garbage-Collector des Serversystems das Objekt freigegeben.

Wie man sieht, sind zur Realisierung eines verteilten Garbage-Collection weitere Methodenaufrufe erforderlich, die am besten über Nachrichten angestoßen werden müssen. Ist kein eigener Garbage-Collector im lokalen System, so kann das Freigeben eines Proxy-Objekts (in C++ z.B. über die *delete*-Methode) das Ereignis zum Aufruf der Methode *RemoveRef(Object)* sein.

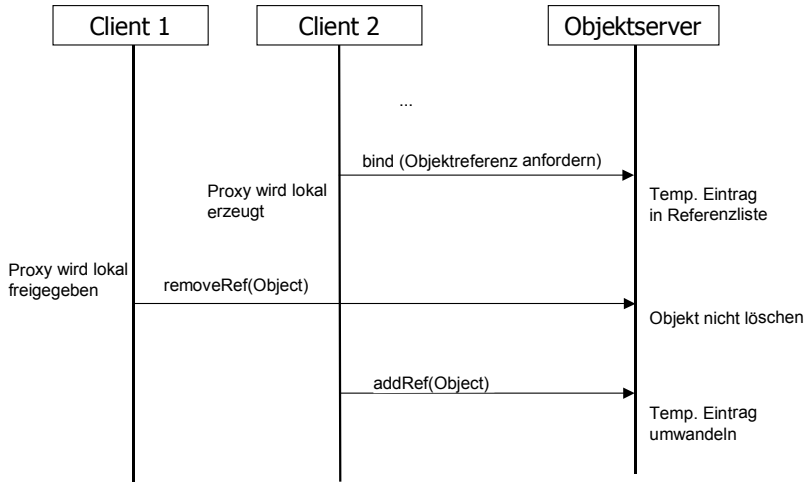


Abbildung 2-6: Mögliches Fehlerszenario beim verteilten Garbage-Collection

Netzwerkfehler können dazu führen, dass Nachrichten verloren gehen oder die Reihenfolge der Add- bzw. Remove-Nachrichten vertauscht wird. Es kann vorkommen, dass eine Remove-Nachricht eines Clients fälschlicherweise vor einer Add-Anweisung eines anderen Clients ankommt.

Probleme kann es geben, wenn ein Client gerade das letzte *RemoveRef(Object)* für ein Objekt abgesetzt hat und zur selben Zeit ein neu referenzierender Client einen *AddRef(Object)* aufruft. Der Server könnte nun das Objekt löschen. Da aber gerade eine Anforderung der Objektreferenz (*bind*-Aufruf) durch den zweiten Client läuft und der Server diese kennt, kann er den Aufruf des neuen Clients abwarten und das Objekt nicht löschen. Das Sequenzdiagramm in Abbildung 2-6 soll diese Situation verdeutlichen.

Ein weiterer Mechanismus, der neben *Reference-Counting* in verteilten Systemen für das Garbage-Collection eingesetzt wird, sind *Leases*. Server geben hier die Referenz nur eine begrenzte Zeit an einen Client frei. Meldet sich der Client längere Zeit nicht beim Server, kann dieser die Referenz löschen. Der Client muss dann in der Regel dafür sorgen, dass sein *Lease* zyklisch verlängert wird, indem er dem Server eine Nachricht sendet. Damit erreicht man, dass Objekte im Server auch dann freigegeben werden, wenn sich ein Client nicht mehr meldet.

Nebenläufigkeit

Man unterscheidet hier zwischen Servern, welche die nebenläufige Bearbeitung mehrerer Client-Requests erlauben, und solchen, die die Requests seriell abarbeiten. Erstere nennt man auch *parallele* Server, letztere *iterative* Server. In heutigen Implementierungen findet man aus Leistungsgründen kaum mehr iterative Server für Client-Server-Anwendungen.

Bei parallelen Servern ist implementierungstechnisch ein sog. *Dispatcher* erforderlich. Ein Dispatcher ist ein Softwarebaustein, der im Server für die Verteilung der Requests an sog. Worker-Threads oder -Prozesse zuständig ist. Threads oder Prozesse können im Vorfeld der Bearbeitung, etwa beim Systemstart, erzeugt und in einem Pool verwaltet werden.

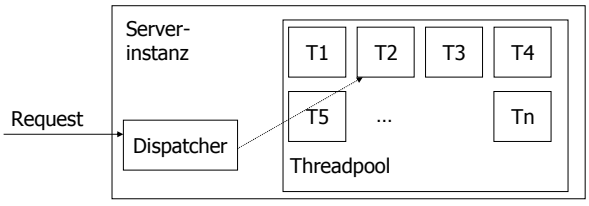


Abbildung 2-7: Request-Dispatching

Der Dispatcher analysiert die Anfrage und leitet diese dann an einen freien Worker-Thread weiter. Der Worker-Thread führt die Anfrage unabhängig vom Dispat-

cher aus, während letzterer bereits auf den nächsten Request wartet. Wenn der Worker-Thread fertig ist, beendet er sich oder legt sich selbstständig wieder in den Threadpool (siehe Abbildung 2-7).

Durch Einsatz von Threads ist es möglich, dass jeder Request für sich sequentiell abgearbeitet werden kann. Die Programmierung ist relativ überschaubar und man kann blockierende *send*- und *receive*-Operationen des jeweiligen Transportsystems bzw. der Middleware verwenden. Falls es keine Thread-Implementierung im Betriebssystem gibt, muss die Parallelität simuliert werden. Dies kann über die Implementierung eines Zustandsautomaten in einem Prozess erfolgen. Da dann viele Netzwerkereignisse innerhalb eines Prozesses auftreten können, darf der Prozess an keiner Stelle blockieren und muss so programmiert werden, dass er nach getaner Arbeit sofort wieder an einem zentralen Ereigniswartepunkt auf ankommende Requests wartet. Es dürfen auch keine blockierenden *send*- und *receive*-Aufrufe benutzt werden. Der *inetd*-Prozess in Unix-Systemen ist ein Beispiel für diese Vorgehensweise.

Eine andere Variante, die in Unix häufig vorzufinden war, ist das Starten eines eigenen Prozesses für jeden Request. Ein Prozess arbeitet in diesem Modell als Dispatcher und wartet auf ankommende Requests. Sobald ein Request eintrifft erzeugt er einen neuen Arbeitsprozess und übergibt diesem die Bearbeitung. Er geht schnellstmöglich wieder in einen Wartezustand, um den nächsten Request annehmen zu können. Bei hoher Last ist die Prozessgenerierung allerdings problematisch für die Leistung des Systems.

In moderneren Middleware-Implementierungen wird das Threadhandling meist durch die Middleware versteckt, was bedeutet, dass der Programmierer damit gar nichts zu tun hat. Er muss also serverseitig „nur“ seine Request-Bearbeitung sequentiell programmieren.

Lebenszyklus von Serverbausteinen

Ein Serverbaustein wird zur Laufzeit von einem Server instanziiert und durchläuft, je nach Bausteintyp und Implementierung verschiedene Zustände. Auch hier ist die Terminologie bei den einzelnen Lösungsansätzen sehr uneinheitlich. Daher werden wir für unsere Zwecke ein vereinfachtes, allgemeines Modell als Grundlage für die weitere Diskussion verwenden. Ein Serverbaustein kann in unserem Modell im Einzelnen folgende Zustände einnehmen:

- *Instanziiert*: Die Instanz eines Serverbausteins wurde erzeugt, ist aber noch keinem Client zugeordnet.
- *Deinstanziiert*: Die Instanz wurde zerstört (Endzustand) und ist daher nicht mehr verfügbar.
- *Aktiviert*: Eine Instanz des Serverbausteins ist einem Client zugeordnet.

– *Passiviert*: Die Instanz ist dem Client entzogen und je nach Implementierung ausgelagert (auf einen Sekundärspeicher), etwa um Ressourcen frei zu machen.

In Abbildung 2-8 sind die Zustandsübergänge dargestellt.

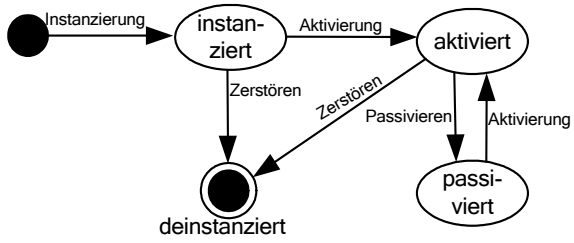


Abbildung 2-8: Zustandsdiagramm für eine Serverbaustein-Instanz

Der Lebenszyklus einer Serverbaustein-Instanz wird durch eine Instanzierung begonnen. Die Instanzierung kann durch den Server bzw. dessen Infrastruktur selbst gesteuert oder aber durch den Client angestoßen werden. Serverbausteine werden dann meist in sog. Pools verwaltet. Der umgekehrte Vorgang ist die Deinstanzierung oder Zerstörung der Instanz, die meist der Server einleitet.

Unter einer *Aktivierung* verstehen wir die Zuordnung eines Serverbausteins zu einem konkreten Client zum Zwecke der Ausführung eines oder mehrerer Requests. Zudem werden Serverbausteine ggf. passiviert (Passivierung) wenn der Server knapp an Ressourcen ist oder auch aus anderen Implementierungsgründen. Bei der Aktivierung unterscheidet man Serveraktivierung und Clientaktivierung. Weiterhin kann ein Serverbaustein einem Client nur für einen Request (*Singlecall* oder *Per-Request-Server*) oder für viele Requests zugeordnet werden. Beim *Per-Request-Server* erfolgt für die Bearbeitung eines Requests eine Aktivierung und nach Abarbeitung des Requests eine Passivierung.

Eine erneute Aktivierung nach einer Passivierung ist dann durchzuführen, wenn die Instanz wieder benötigt wird. In konkreten Implementierungen (siehe z.B. EJB stateful Session Beans) kann die Instanz bei einer erneuten Aktivierung einem anderen Client zugeordnet werden. Wichtig ist, dass die neue Instanz den aktuellen Konversationszustand der Kommunikation zwischen dem konkreten Client und dem Server bereitstellt.

Serverbausteine, die bereits zum Startzeitpunkt des Servers erzeugt werden und solange leben, bis dieser wieder beendet wird, werden auch als *Persistent Server* bezeichnet.

Aktivierung und Passivierung hängen vor allem von der Art der Zustandsverwaltung ab. Stateless Server werden in der Regel nur für einen Request (Singlecall) aktiviert, stateful Server dagegen für die Dauer der Konversation. Wird während

der Konversation längere Zeit nicht kommuniziert, kann eine Passivierung erfolgen.

Basisdienste

Neben den genannten Kriterien unterscheidet man Client-Server-Systeme auch anhand der bereitgestellten Basisdienste. Alle Basissysteme (auch Middlewaresysteme genannt) verfügen über den bereits erwähnten Namens- und Verzeichnisdienst. Weitere wichtige Dienste, die insbesondere bei betrieblichen Informationssystemen eine große Rolle spielen, sind Persistenzdienste für den Zugriff auf externe Daten, Sicherheitsdienste und Transaktionsdienste. Hier unterscheiden sich die verschiedenen Lösungsalternativen, die wir noch betrachten werden, erheblich.

Sonstige Unterscheidungsmerkmale

Schließlich sollen noch ganz spezielle Eigenheiten von Basissystemen betrachtet werden. Interessant sind auch Mechanismen für die Unterstützung der Skalierbarkeit von Systemen sowie Möglichkeiten, die Verfügbarkeit von Systemen zu erhöhen. Weiterhin sind die Komplexität der Nutzung und auch eine Einschätzung der heutigen Nutzung für die Auswahl von Basissystemen interessant.

2.1.3 Serveroptimierung durch Caching

Bei umfangreichen Client-Server-Anwendungen ist es durchaus möglich, dass mehr als Hundert oder sogar ein paar Tausend Clients auf einen oder mehrere Server zugreifen. Meist sind serverseitig Datenbanken im Einsatz, bei denen ein ständiger Zugriff durch nebenläufige Requests zu einer enormen Belastung führen kann.

Der Architekt einer verteilten Anwendung muss hier auf die Leistungsfähigkeit achten und möglichst Zugriffe sowohl vom Client zu den Anwendungsservern als auch von diesen zu den Datenhaltungsservern vermeiden. Eine Optimierungsvariante ist Caching. Caching kann im Server erfolgen und zwar für Daten, die sonst vom Datenhaltungsserver besorgt werden müssten. In diesem Fall können Client-Requests evtl. ohne Zugriff auf den Datenhaltungsserver befriedigt werden, wenn die gewünschten Daten bereits im Cache liegen. Dies bringt bei lesendem Zugriff Leistungsverbesserungen. Genauso kann man im Client Kopien von bereits gelesenen Daten ebenfalls in einem Cache verwalten. Benötigt ein Client Daten, die schon im Client-Cache gelagert sind, spart er sich den Zugriff über das Netz auf den Server. Sinnvollerweise wird ein Cache im Hauptspeicher verwaltet, im Client wäre auch noch eine verfügbare Festplatte nutzbar, was bei sog. *diskless* Clients nicht funktioniert. Web-Browser verfügen z.B. über einen konfigurierbaren Cache.

Zwei Probleme sind beim Caching zu bedenken:

- Zum einen ist der Hauptspeicher begrenzt, man muss sich also beim Design überlegen, welche Daten man im Cache hält. Ein Ersetzungsalgorithmus wie z.B. *LRU* (Least Recently Used) muss implementiert werden.
- Zum anderen kann es zu Inkonsistenzen kommen, wenn die im Cache befindlichen Daten an Aktualität verlieren. Dies kommt vor, wenn andere Clients oder Systembausteine der Anwendung die Daten im Server bzw. in der Datenhaltung verändern. Im Server tritt dieses Problem nicht bei zustandsinvarianten, wohl aber bei zustandsbehafteten Servern auf.

Eine komplexe Lösung für das Problem ist, dass man z.B. einen *Write-Through*-Algorithmus implementiert, bei dem Änderungen „gecachter" Daten sofort an den Anwendungsserver bzw. den Datenhaltungsserver weitergereicht und dann an alle Caches propagiert werden. Eine vollständige Konsistenz ist allerdings auch hier schwer zu erreichen, weshalb der Aufwand nicht gerechtfertigt erscheint.

Als praktische Empfehlung für die Realisierung von Client-Server-Anwendungen mit Caches kann gelten, dass man vorwiegend Daten in den Cache legt, die sich nie oder sehr selten ändern. Sollten Sie sich aber doch ändern, so dürfen die Änderungen keine ernst zu nehmenden Konsistenzprobleme verursachen.

2.1.4 Erweiterungen des Client-Server-Modells

Nach (Bengel 2004) unterscheidet man einige Varianten und Erweiterungen des Client-Server-Modells, die vor allem Implementierungsaspekte beleuchten. Hierzu gehören der Proxy-Server, der Broker, der Trader, der Balancer und der Agent:

Proxy-Server: *Proxy-Server* sind Stellvertreter, die zwischen den Clients und den Servern liegen. Ein Proxy kann Stellvertreter für viele Server sein und kann auch viele Clients bedienen. Clients senden ihre Requests an den Proxy, der diese evtl. direkt über zusätzliche Logik bearbeiten kann oder die Anfrage an einen oder mehrere Server weiterleitet. Mit diesem Modell können mehrere Vorteile erreicht werden. So sind z.B. Client-Anwendungen einfacher zu programmieren und haben nur eine Verbindung zu einem Proxy-Server zu unterhalten. Der Proxy-Server im lokalen Netz kann mit zusätzlicher Funktionalität ausgestattet sein. Beispielsweise kann er für alle Clients einen gemeinsamen Cache verwalten und man spart sich den möglicherweise aufwändigen Zugriff auf entfernte Server. Proxy-Server dienen also der Leistungsoptimierung. Weiterhin kann im Proxy auch Funktionalität etwa für Logging und Sicherheit hinterlegt werden. Typischer Einsatzfall eines Proxy-Servers ist das WWW. Proxy-Webserver speichern lokal Kopien von bereits gelesenen HTML-Seiten in ihrem Cache und stellen sie allen Web-Clients für weitere Zugriffe zur Verfügung. Bei diesen Proxy-Servern gibt es natürlich auch Schwierigkeiten. Beispielsweise dürfen dynamisch von Webanwendungen aufgebaute Webseiten nicht „gecachet" werden, da sie beim nächsten Zugriff evtl. anders aussehen. Sinnvoll ist Caching hier für statische Seiten. Dynamische Webanwendungen schalten das Caching für ihre HTML-Seiten aus, was über eine Para-

metrisierungsmöglicheit im verwendeten HTTP-Protokoll möglich ist. Verantwortlich sind dafür die Anwendungsprogrammierer.

Broker: Ein *Broker* verwaltet die Informationen über die richtigen Serveradressen und kann als Vermittler (daher der Name Broker) auftreten. Hierfür muss er eine eigene Datenbasis pflegen. Bei Zwischenschalten eines Brokers müssen die Clients nicht mehr wissen, wo die benötigten Server laufen. Broker tragen damit zur Ortstransparenz bei. Aber auch zur Migrationstransparenz kann ein Broker beitragen. Sollte ein Server aus irgendeinem Grund das Rechnersystem wechseln müssen, kann dies in der Datenbasis des Brokers verändert werden und die Clients merken nichts davon.[2] Zusätzlich ist auch ein mehrfaches Starten eines Servers möglich, sofern der Broker eine gewisse Logik zur Verteilung der Requests unterstützt. Ein Client muss dann gar nicht wissen, wie viele Serverinstanzen aktiv sind (Replikationstransparenz).

Trader: Ein *Trader* dient der Auswahl eines Servers nach einem bestimmten Regelwerk aus einer Menge von Servern, die jeweils den gleichen Dienst anbieten, aber mit unterschiedlichen Eigenschaften (z.B. Qualitätsunterschiede). Die Logik der Auswahl muss dann nicht im Client programmiert werden, sondern kann einer dedizierten Instanz, dem Trader, überlassen werden. Das Trader-Modell entspricht im Wesentlichen dem Broker-Modell mit einem Unterschied. Der Trader hat zusätzlich die Aufgabe, einen geeigneten Service auszuwählen. Hierzu benötigt er vom Client Informationen über den gewünschten Service und dessen Charakteristik. Es kann natürlich auf Grund des zentralen Ansatzes auch hier zu einem Flaschenhals kommen, der entsprechend berücksichtigt werden muss.

Balancer: Leistungsaspekte sprechen in Client-Server-Anwendungen oft dafür, eine weitere serverseitige Technik, den sog. *Balancer* zu verwenden. In diesem Fall werden Server repliziert und der Balancer verteilt gemäß einer geeigneten Strategie die Requests auf die Serverinstanzen. Eine einfache Strategie ist z.B. das aus dem Prozessscheduling in Betriebssystemen bekannte Round-Robin-Verfahren, in dem die Server der Reihe nach zugeordnet werden. Aber auch die Nutzung bestimmter Metriken wie die aktuelle Netzbelastung, die Bandbreite des Netzwerkzugangs, die Rechnerausstattung der Serverrechner usw. sind möglich. Der Balancer soll für eine möglichst gute Auslastung im Sinne der vorgegebenen Strategie sorgen.

Eine Mischung von Broker-, Trader- und Balancer-Funktionalität ist auch möglich und natürlich kann ein Balancer auch zum Engpass werden. Balancer findet man in vielen Middleware-Implementierungen und heute auch zunehmend im Web zur Unterstützung stark frequentierter Websites.

---

[2] Die Clients bemerken es natürlich, wenn durch eine Migration eines Servers ein Request unterbrochen wird.

Große Webanwendungen werden meist durch mehrere Webserver bedient, denen ein Balancer (manchmal auch Dispatcher genannt) vorgeschaltet ist. Dies kann z.B. ein spezieller Hardware-Balancer sein, also ein Webserver mit dedizierter Hardware (siehe Abbildung 2-9), der auf die zugeordneten Webserver z.B. über eine Round-Robin-Strategie verteilt. Intelligentere Verfahren beziehen bestimmte Metriken wie die aktuelle Serverauslastung, die Größe der Serverrechner, die Antwortzeit des Servers usw. mit ein.

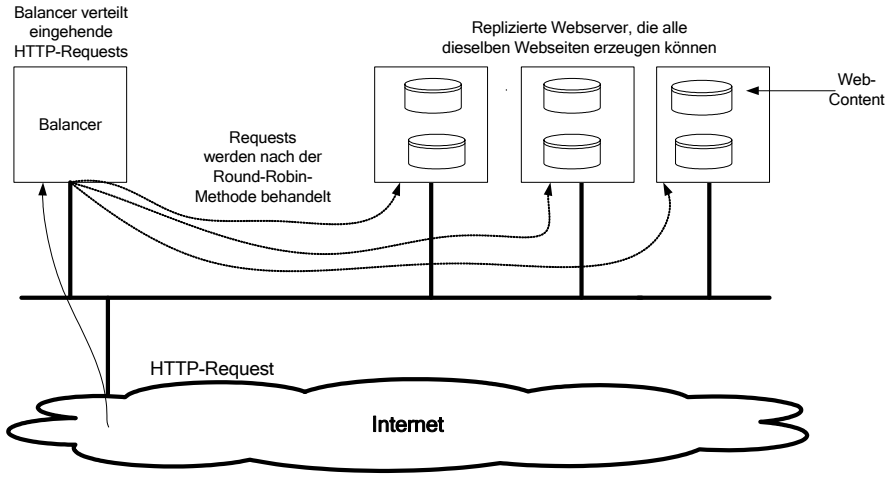


Abbildung 2-9: Balancer-Einsatz im Web

Eine andere Variante, die in diesem Zusammenhang verwendet wird, ist das DNS-basierte Round-Robin-Verfahren, das Mehrfach-DNS-Einträge für dieselbe IP-Adresse nutzt. Mehrere Webserverrechner bekommen die gleiche IP-Adresse und die Zuteilung eines Requests erfolgt dynamisch durch den Domain Name Service. DNS übergibt bei jeder Anfrage zur Auflösung eines Hostnamens die nächste IP-Adresse in der konfigurierten Adressliste zurück und fängt am Ende der Liste wieder am Anfang an. Nachteilig an diesem Verfahren ist, dass die DNS-Server nichts über den Ausfall eines Servers erfahren und diesem trotzdem einen Request zuordnen. Dieses Verfahren wird eher für statische Webseiten verwendet und wird auch als Load Sharing auf Basis von DNS bezeichnet.

Agent: Schließlich soll noch das Konzept des Agenten erwähnt werden. Agenten sind z.B. sinnvoll, wenn man eine komplexe Serviceleistung abwickeln möchte, die mehrere Server beansprucht. Um den Client zu entlasten kann man einen Agenten vorschalten, der den komplexen Request entgegennimmt und ihn über die Kontaktierung mehrerer Server beantwortet. Die Ausführung der Requests durch den Agenten kann iterativ (nacheinander) oder parallel erfolgen. Der Client wird durch einen Agenten stark entlastet und muss keine Kenntnisse über die Server haben.

Das DNS ist z.B. eine typische Client-Server-Anwendung mit Agenten-Funktionalität, da der zuständige DNS-Server weitere DNS-Server kontaktiert, um die entsprechende Adressumsetzung durchzuführen (Mandl 2008b). Eine andere Anwendung von Agenten ist in SNMP (Simple Network Management Protocol) zu finden. Hier werden SNMP-Requests vom SNMP-Client abgesetzt und an einen Agenten in einem Rechnersystem oder ein sonstiges Gerät gesendet. Der Agent wickelt den Request im Zielsystem ab und meldet das Ergebnis zurück. Auch im Internet werden intelligente Agenten z.B. für die Ermittlung einer bestimmten Information (Reiseplanung) immer interessanter.

2.2 Verteilte Prozeduraufrufe

2.2.1 Grundlegendes Modell

Verteilte Prozeduraufrufe, üblicherweise auch Remote Procedure Call (RPC) genannt, sind eine der ältesten Implementierungen des Client-Server-Modells. Der Ablauf eines RPC-Aufrufs ist in Abbildung 2-10 schematisch dargestellt. Im Clientrechner sind das eigentliche Anwendungsprogramm und einige Middleware-Komponenten vorhanden. Zur Middleware gehören ein sog. *Client-Stub* und eine *Kommunikationskomponente*. Im Server sind ebenfalls als Bestandteil der Middleware eine entsprechende Kommunikationskomponente sowie ein *Server-Stub* vorhanden. Schließlich ist auf der Serverseite das Coding der Anwendungslogik für die entfernte Prozedur implementiert.

In der Abbildung ist zu erkennen, dass das clientseitige Anwendungsprogramm einen Prozeduraufruf zunächst lokal an den Client-Stub übergibt und dieser über die Kommunikationskomponente, die ein internes Protokoll benutzt, mit dem Server kommuniziert. Der Client ist blockiert bis die Antwort des Servers ankommt. Es handelt sich von der Kommunikationsform her um einen synchronen, entfernten Dienstaufruf (auftragsbezogen). Bei einigen RPC-Implementierungen ist auch ein asynchroner Dienstaufruf möglich. In diesem Fall muss der Client nachfragen können, ob der Dienstaufruf schon ausgeführt ist.

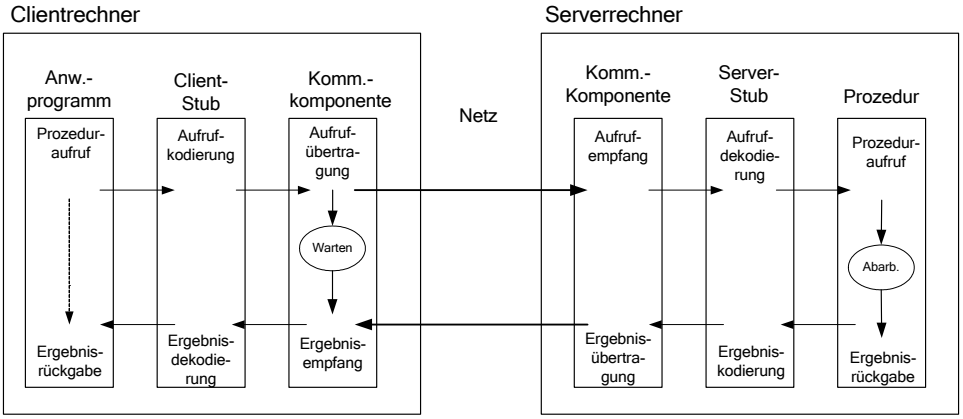


Abbildung 2-10: Schematischer Ablauf eines Remote Procedure Calls nach (Schill 2007)

Client- und Server-Stub übernehmen die Aufbereitung der zu übermittelnden Parameter und Returnwerte. Wie bereits besprochen, nennt man diesen Vorgang auch *Marshalling bzw. Unmarshalling* oder etwas verwirrend *Kodierung* bzw. *Dekodierung* oder auch *Serialisierung* bzw. *Deserialisierung*. Alle Objekte, die mit dem Request bzw. dem Ergebnis als Parameter oder als Returnwert übertragen werden, werden auf eine unabhängige Transfersyntax abgebildet, die dann auf der Gegenseite wieder in eine dort verständliche lokale Syntax gebracht wird.

In Abbildung 2-11 ist dargestellt, was in den einzelnen Rechnersystemen alles ablaufen muss, während der RPC-Aufruf bearbeitet wird. Vieles läuft im Kernel ab, da ja auch das Transportsystem meist ein Bestandteil des Betriebssystems ist. Interessant ist auch, dass der Aufruf in der Regel über einen Systemcall zu einem Kontextwechsel führt (Mandl 2008a). Dazu muss vom Usermodus in den Kernelmodus gewechselt werden. Im Kernel wird die Nachricht über das Netzwerk gesendet und auf die Antwort gewartet. Damit wird dann auch der aktuelle Prozess suspendiert. Er muss warten, bis das Ergebnis vom Server empfangen wird. Bis dahin kann der Dispatcher des Betriebssystems einen anderen Prozess oder Thread aktivieren. Wenn das Ergebnis ankommt, wird der anfordernde Prozess erst wieder in den Zustand „ready“ gesetzt.

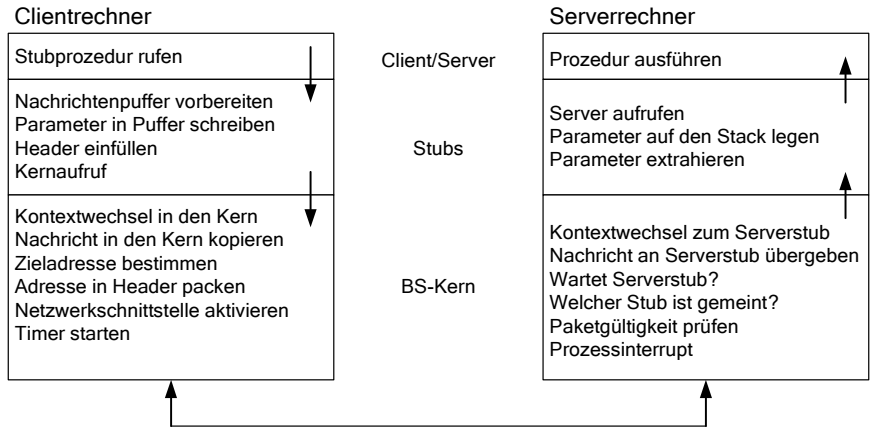


Abbildung 2-11: Ablauf eines RPC-Aufrufs nach (Weber 1998)

Bei RPC wird der für das *Binden* verantwortliche Softwarebaustein ebenfalls als *Binder* bezeichnet. Der Server muss sich beim Start bei diesem registrieren und vor dem Beenden abmelden. Hierzu sind spezielle Nachrichten bzw. Funktionsaufrufe definiert.

Wir werden nun das Konzept des RPC anhand der RPC-Implementierung von Sun (ONC RPC) noch näher beleuten.

2.2.2 Fallbeispiel ONC RPC von Sun

Es gibt einige praxisrelevante RPC-Implementierungen. Die bekannteste ist Sun RPC bzw. ONC RPC. ONC steht für Open Network Computing, eine verteilte Architektur von Sun Microsystems. Auch das heute nicht mehr so bedeutende DCE RPC[3] trug zur Weiterentwicklung von RPC bei. Wir wollen nun eine Einordnung von ONC RPC anhand unserer Kriterienliste vornehmen:

Architektur

Abbildung 2-12 zeigt die Architektur einer Client-Server-Anwendung, die ONC RPC nutzt. Aufsetzend auf dem Transportsystem (TCP oder UDP) ist ein RPC-Protokoll definiert, das für die Nachrichtenübertragung zuständig ist. Sowohl der Client als auch der Server müssen Stub-Prozeduren und Marshalling-/Unmarshalling-Routinen einbinden. Ein Server ist ein Betriebssystemprozess, der die Lauf-

[3] DCE = Distributed Computing Environment wurde von der Open Software Foundation (heute Open Group) als herstellerunabhängige Lösung konzipiert, bevor CORBA eingeführt wurde.

zeitumgebung für einen oder mehrere Serverbausteine, die Services repräsentieren, bereitstellt. Die über RPC zugänglichen Services sind in Programme und diese wiederum in Versionen organisiert. Jeder Version sind die aufrufbaren entfernten Prozeduren zugeordnet. Ein Service wird also in einem Serverbaustein bereitgestellt und wird durch ein Bündel von Programmen, eventuell in mehreren Versionen, bestehend aus jeweils mehreren Prozeduren (siehe Abbildung 2-13) implementiert.

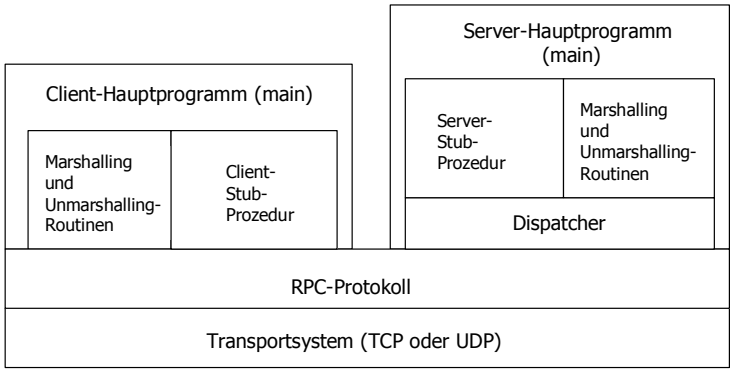


Abbildung 2-12: Sun-ONC-RPC-Architektur

Bei Sun RPC verwendet man für die Beschreibung der Schnittstelle (des Dienstes) eine Schnittstellensprache, kurz XDR- oder RPC-Sprache genannt. *XDR* (eXternal Data Representation) nutzt das sog. *implizite Typing*, d.h. nur die Werte werden versendet, nicht der Variablentyp. Parameter werden auch ohne weitere Zusatzinformation mit dem RPC-Request gesendet. Das Typensystem der XDR-Sprache ist dem C-Typensystem relativ ähnlich.

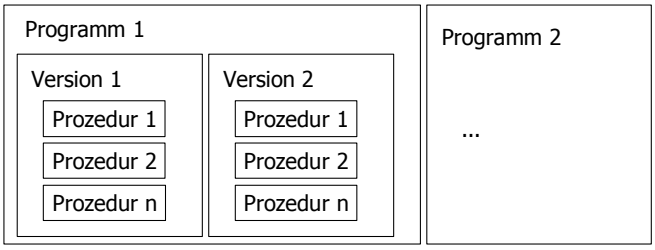


Abbildung 2-13: Organisation der Services in Sun ONC RPC

ONC RPC unterscheidet zwei verschiedene API-Komplexitätsstufen, die in der ONC-Library als C-Routinen bereitgestellt werden. Die erste Stufe, auch als High-Level-API bezeichnet, ermöglicht eine weitgehend automatisierte Erzeugung von

Codes für das Marshalling/Unmarshalling und für die Client-Server-Kommunikation. Man kommt bei der Entwicklung sowohl client- als auch serverseitig mit nur wenigen Funktionsaufrufen aus. Ein Server registriert seine Dienste mit der Funktion *registerrpc* und springt dann mit *svc_run* in das RPC-Laufzeitsystem, das serverseitig die Kontrolle übernimmt.

Wenn man spezielle Lösungen entwickeln möchte, so kann man dies durch Eigenentwicklung unter Nutzung der Low-Level-API erreichen. Beispiele für Aufgabenstellungen, die man mit der Low-Level-API realisieren kann, sind asynchron auszuführende Remote-Procedure-Calls sowie nebenläufige Serverbausteine.

Wir wollen nun eine Einordnung von ONC RPC anhand unserer Kriterienliste vornehmen:

Zustandsverwaltung

Für die Zustandsverwaltung gibt es keinerlei Unterstützung bei ONC RPC. Die Zustandsverwaltung und damit die Programmierung von zustandslosen oder zustandsbehafteten Serverbausteinen obliegen allein dem Programmierer.

Dienstschnittstellen

Ausgehend von einer Schnittstellenbeschreibung werden der Client- und der Server-Stub sowie ein Headerfile und XDR-Filter erzeugt. Ebenso werden XDR-Filter für das Marshalling-/Unmarshalling der eigenen Datentypen generiert. Die Generierung erfolgt mit dem Basistool *rpcgen* in die Zielsprache C.

Beispiel „Kundenobjekt als XDR-Schnittstelle“: Ein Kundenobjekt könnte in XDR-Sprache wie folgt beschrieben werden, wobei die Definitionen z.B. in einer Datei kunde.x abgelegt werden können:

```
const MAX = 30;
struct Kunde {
   string name<MAX>; /* String mit maximaler Länge */
        string vorname<MAX>;
        int customerId;
};

program KUNDENDIENSTE { /* Frei zu vergebender Programmname */
   version VERSION {
        void createCustomer(Kunde customer);
        int getCustomerId(string name)=1;
        string getName(int customerId)=2;
        string getVorname(int customerId)=3;
   }=20000001;
}=5000;
```

Wie man sieht, gibt es in der XDR-Sprache keine Objektklassen. Die Datenstrukturen und die Prozeduren sind daher getrennt definiert. Die Struktur *Kunde* enthält die Daten eines Kunden und in dem Abschnitt, der mit *program* beginnt, sind die entfernten Dienste (remote procedures) definiert. Wie man sieht, wurde ein Dienst *createCustomer* als „Konstruktorersatz" sowie ein weiterer Dienst *getCustomerId* definiert. Letzterer dient für den Zugriff auf die Variable *customerId*.

Bei der Beschreibung der Dienste werden eine Programm- und eine Versionsnummer angegeben. Die Notation hierfür ist etwas gewöhnungsbedürftig. Im Beispiel werden die Programmnummer mit 5000 und die Versionsnummer mit 2 belegt. Die Prozeduren sind mit 1 bis 3 durchnummeriert. Diese Informationen werden mit entsprechenden RPC-PDUs gesendet, so dass der jeweilige Empfänger die Nachrichten prüfen kann. Jede Prozedur erhält zudem eine Prozedursignatur bestehend aus dem Ergebnistyp, dem Prozedurnamen und dem Typ des Eingabeparameters.

ONC RPC erlaubt dem Anwendungsprogrammierer Programmnummern zwischen 0x20000000 und 0x3FFFFFFF, die anderen sind reserviert. Über die Nummern registriert der Server seine Services und darüber kann ein Service auch adressiert werden. Beispiele für reservierte Programmnummern sind 0x100000 für den Portmapper (siehe unten), 0x10003 für NFS (Network File System), sowie 0x10004 für NIS (Network Information Service).

Man kann sich nun auf Basis der XDR-Sprache eine entfernte Schnittstelle fast beliebiger Komplexität definieren. Je Prozedur werden allerdings nur ein einziger Eingabe- und ein Returnwert erlaubt. Man muss ggf. Strukturen definieren, um mehrere Parameter unterzubringen. Die Sprache unterstützt u.a. Konstanten, Typendefinitionen (typedefs wie in C), Strukturen, Aufzählungstypen (wie in C) und Unions (wie in C).

Parameterübergabe

Die Übergabe der Parameter und Returnwerte erfolgt bei ONC RPC ausschließlich über Call-by-value, also als Kopien der lokalen Parameter.

Marshalling und Unmarshalling

Mit dem Tool *rpcgen* (RPC-Compiler) wird zum einen der gesamte Marshalling-/Unmarshalling-Code für den Client- und für den Server sowie der Client- und der Server-Stub generiert. Weiterhin werden ein Grundgerüst für eine serverseitige *main*-Prozedur und ein Request-Dispatcher erzeugt. Die Codegenerierung erfolgt in der Sprache C. Die Anwendungslogik für die serverseitigen Prozeduren muss noch ausprogrammiert werden. Die generierten Codeteile sind dann mit einem C-Compiler zu übersetzen. Der generierte Marshalling-/Unmarshalling-Code wird auf die Standarddatentypen zurückgeführt. Für jeden Standarddatentypen wird eine Basisroutine zur Kodierung- und Dekodierung bereitgestellt. Dies soll am Beispiel des Typs *int* dargestellt werden:

```
bool_t xdr_int(xdrs, objp)
   XDR *xdrs;
   int *objp;
{ … }
```
[4]

Die Prozedursignaturen der XDR-Filter sind immer gleich aufgebaut. *xdrs* ist ein Zeiger auf einen offenen XDR-Stream, in den die serialisierten Daten geschrieben werden bzw. aus dem Daten deserialisiert werden. *objp* ist ebenfalls ein Zeiger, der auf einen Speicherbereich zeigt, in dem ein Integer-Wert Platz hat. Die Routine wird sowohl für die Kodierung als auch für die Dekodierung verwendet. Im ersten Fall schreibt die Routine den kodierten Wert an den Speicherbereich, auf den *objp* zeigt. Im zweiten Fall wird der Wert an dieser Speicherstelle erwartet. Weitere XDR-Filterfunktionen sind z.B. *xdr_long*, *xdr_bool*, *xdr_double*, *xdr_enum* usw. Als Rückgabewert wird *true* oder *false* zurückgegeben, je nachdem ob der Aufruf erfolgreich war oder nicht.

Beispiel 1: Der durch *rpcgen* erzeugte Code für die Kodierung/Dekodierung der Struktur *Kunde* sieht wie folgt aus:

```
#include <rpc/rpc.h>   /* Basistypen und Definitionen der Basisfunktionen */
#include "xdr_kunde.h"
bool_t xdr_Kunde(xdrs, objp)
   XDR *xdrs;
   int *objp;
{
   if (!xdr_string(xdrs, &objp->name,30)) /* Länge ! */{
        return (FALSE);
   }
   if (!xdr_string(xdrs, &objp->vorname,30)) /* Länge ! */{
        return (FALSE);
   }
   if (!xdr_int(xdrs, &objp->customerId)) {
        return (FALSE);
   }
   return (TRUE) /* Alle Prüfungen in Ordnung! */
}
```

Man sieht, dass eine Codegenerierung für Filter hier über klar definierte Regeln ermöglicht wird und daher auch sehr sinnvoll ist. Ein weiteres, einfaches, aber umfassendes Beispiel soll die Problematik noch weiter veranschaulichen.

Beispiel 2: Addieren von zwei Integerzahlen über eine entfernte Prozedur

[4] XDR enthält einen eigenen Typen bool_t.

Der Aufruf eines Dienstes zum Addieren zweier Integerzahlen über RPC könnte wie folgt aussehen:

```
result = add (request);
```

Eine geeignete Schnittstellenbeschreibung in der XDR-Sprache hätte den Dateinamen *add.x* und folgenden Inhalt:

```
struct result {int x;};
struct request {int a; int b;};
program ADD_PROG {
          version ADD_VERS {
               result ADD (request) = 1;
          } = 1;
} = 20;
```

Der RPC-Compiler *rpcgen* generiert aus der Schnittstellenbeschreibungsdatei (*add.x*) die erforderlichen Dateien (Headerdatei, Stub, Skeleton, usw.), die hier nicht weiter erläutert werden sollen.

Adressierung und Kommunikation

Namensauflösung und Naming-/Directory-Service: Sun RPC unterstützt sowohl statisches als auch dynamisches Binden. Bei Sun RPC übernimmt die Aufgabe des Verzeichnisdienstes bzw. Binders ein sog. *Portmapper*. Unter Unix heißt der Dämon-Prozess *portmap* oder *rpcbind*.

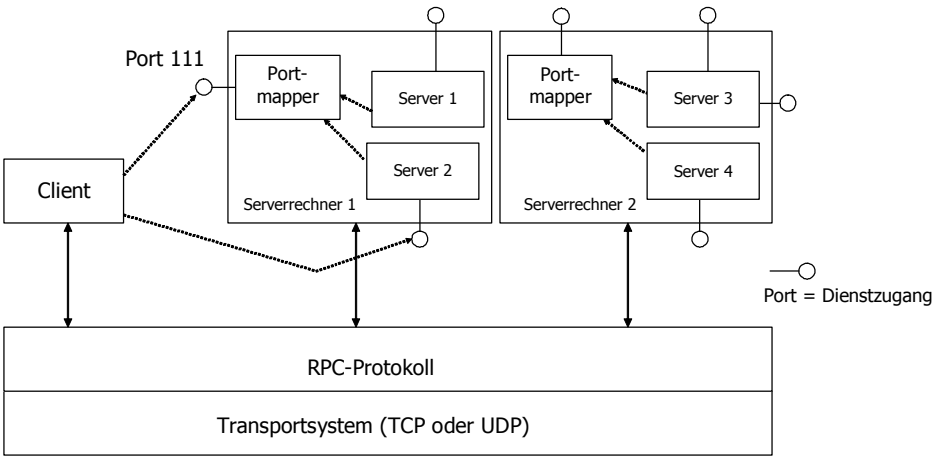


Abbildung 2-14: Portmappper bei Sun ONC RPC

Jeder Serverrechner verfügt über einen Portmapper-Prozess, bei dem sich jeder lokal ablaufende Serverprozess registriert. Der Portmapper ist ein Hintergrund-

prozess, der an einer bekannten (well-known) Portnummer (TCP und UDP als Transportprotokoll, Standardport 111) horcht. Die Serverprozesse registrieren ihre Serverprogramme (Serverbausteine) während des Startvorgangs (über generierten Code) beim lokalen Portmapper. Abbildung 2-14 zeigt zwei Serverrechner, auf denen jeweils ein Portmapper-Prozess und zwei Service-Prozesse ablaufen.

Kommunikation: Der in ONC RPC realisierte Remote Procedure Call unterstützt bei Einsatz von TCP eine *At-Most-Once*-Semantik, bei Einsatz von UDP dagegen eine *At-Least-Once*-Semantik. Im Fehlerfall sind also insbesondere bei Nutzung von UDP Probleme wie eine doppelte Prozedurausführung nicht auszuschließen, was in der Spezifikation der verteilten Anwendung betrachtet werden muss. Eine „Alles-oder-Nichts-Semantik" bzw. eine *Exactly-Once*-Semantik kann mit diesem Mechanismus nicht realisiert werden. Hierzu benötigt man das Konzept der Transaktionen (siehe Kapitel 4). Weitere ONC-RPC-Konzepte wie der sog. Broadcast-RPC bzw. Callback-RPC sind in der angegebenen Literatur beschrieben.

Beim Starten eines Clients ermittelt dieser den Port des Serverprozesses über eine Anfrage an den Portmapper des Hosts, auf dem der Serverprozess läuft. Den Host bzw. die IP-Adresse des Serverrechners muss der Client-Programmierer also kennen. Sun RPC unterstützt also (wie die meisten anderen Implementierungen) keine vollständige Transparenz über den Server-Aufenthalt. Der Serverprozess stellt die Ablaufumgebung für die Serverbausteine dar.

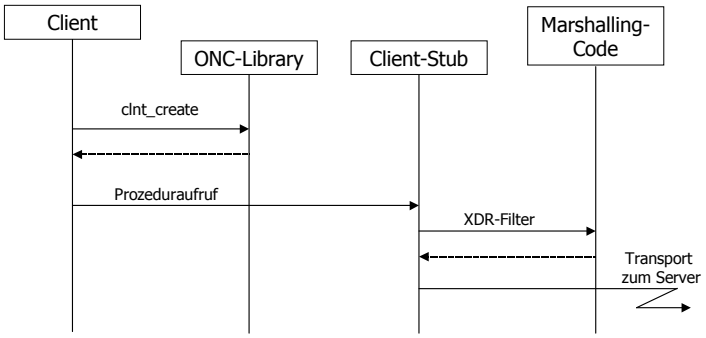


Abbildung 2-15: Ablauf der Kommunikation bei Sun ONC RPC im Client

Der Client wendet sich zunächst über einen Aufruf der Prozedur *callrpc* oder über den Low-Level-Call *clnt_create* an den auf dem entsprechenden Serverrechner laufenden Portmapper, wobei als Inputparameter die Programmnummer, die Versionsnummer des Programms und das gewünschte Transportprotokoll (TCP, UDP) übertragen werden. Über die High-Level-API kann allerdings nur auf Basis von UDP kommuniziert werden. Der Portmapper liefert dem Client eine Referenz (Client-Handle) auf das Serverprogramm zurück. Der Hostname des Serverrech-

ners und die Portnummer des Service werden in das Client-Handle eingetragen. Bei allen folgenden Aufrufen gibt der Client das Client-Handle an.

Ein Aufruf einer Prozedur hat dann prinzipiell den in Abbildung 2-15 und Abbildung 2-16 dargestellten Ablauf, wobei die Bearbeitung des Low-Level-Aufrufs *clnt_create* sehr vereinfacht dargestellt wird. Auch hier erfolgt nämlich, implizit aus Sicht des Clients, eine Kommunikation mit dem Server.

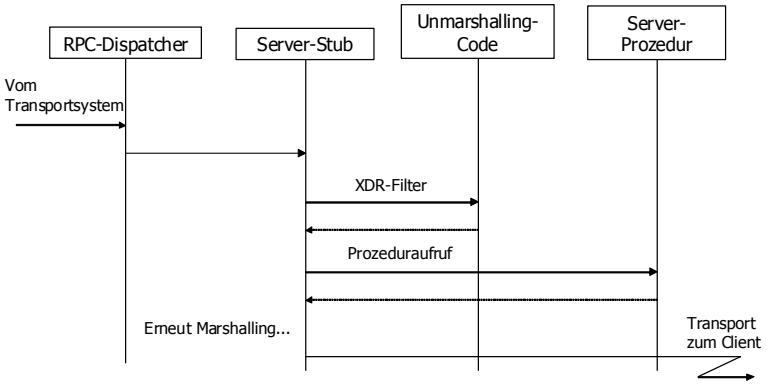


Abbildung 2-16: Ablauf der Kommunikation bei Sun ONC RPC im Client

Garbage-Collection

In ONC RPC sind keine eigenen Garbage-Collection-Mechanismen verfügbar, was aufgrund der genutzten Programmiersprache C nicht verwunderlich ist. Der Programmierer muss in Eigenregie für ein Garbage-Collection sorgen. Dazu nutzt er die C-Funktion *free* zum Freigeben von Heap-Speicher.

Nebenläufigkeit

Der Standardmechanismus aus dem höheren API-Level sieht nur iterative Server vor. Durch den Aufruf der ONC-RPC-Funktion *svc_run* im Server wird nach dem Anmelden (Registrieren) der Serverbausteine in das RPC-Laufzeitsystem gesprungen. Dort erfolgt die Abarbeitung der Requests iterativ in einem Prozess in der sog. Serverschleife. Die Registrierung der Server und die Übergabe der Kontrolle an das Laufzeitsystem soll anhand des folgenden Beispiels verdeutlicht werden.

Beispiel „Serverschleife für einen iterativen Server“: Im Beispiel wird der oben skizzierte Additions-Service sowohl über UDP als auch über TCP zugänglich gemacht. Das Programm besorgt sich jeweils ein Service-Handle durch Aufruf der Low-Level-Funktion *svcudp_create* bzw. *svctcp_create* und übergibt diese zum Registrieren beim Funktionsaufruf *svc_register*. Danach erfolgt der Eintritt in die Serverschleife und damit zum iterativen Abarbeiten der ankommenden Requests über den Aufruf der Funktion *svc_run*.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "add.h"
static void add_prog_1 ( );

main ( )
{
    register SVCXPRT *transp;
    (void) pmap_unset (ADD_PROG, ADD_VERS);
    transp = svcudp_create (RPC_ANYSOCK);
    if (transp == NULL)
    {
        fprintf (stderr, "UDP-basierter Service kann nicht erzeugt werden!");
        exit (1);
    }
    if (!svc_register (transp, ADD_PROG, ADD_VERS, add_prog_1, IPPROTO_UDP))
    {
        fprintf (stderr, "Registrierung fehlgeschlagen!");
        exit (1);
    }   ...
    transp = svctcp_create (RPC_ANYSOCK, 0, 0);
    if (transp == NULL)
    {
        fprintf (stderr, "TCP-basierter Service kann nicht erzeugt
         werden!.");
        exit (1);
    }
    if (!svc_register (transp, ADD_PROG, ADD_VERS, add_prog_1, IPPROTO_TCP))
    {
        fprintf (stderr, "Registrierung fehlgeschlagen!");
               exit (1);
    }
    svc_run ( ); /* Serverschleife */

    fprintf (stderr, "Zurück aus der Serverschleife");
    exit (1);
    /* NOTREACHED */
}
/* Service-Prozedur zum Addieren zweier Zahlen, nur angedeutet */
static void add_prog_1 (rqstp, transp)
    struct svc_req *rqstp;
    register SVCXPRT *transp;
{
```

```
    union {
        request add_1_arg;
    } argument;
    char *result;
    bool_t (*xdr_argument) ( ), (*xdr_result) ( );
    char * (*local) ( );
    switch(rqstp -> rq_proc ) { … }
    return;
}
```

Möchte man nun eine Parallelisierung der Abarbeitung erreichen, muss man über das Low-Level-API einen eigenen RPC-Dispatcher erzeugen. Dieser Dispatcher kann dann beispielsweise für jeden Request einen Prozess oder Thread starten und auch entsprechende Prozess-/Threadpools verwalten. Der Dispatcher kann auch bei Aufruf der Funktion *svc_register* angegeben werden. Weiterführende Erläuterungen findet man in (Bloomer 1993).

Lebenszyklus von Serverbausteinen

Standardmäßig erfolgt bei ONC RPC die Instanzierung eines Serverbausteins zum Startzeitpunkt des Servers. Die Aktivierung, also die Zuordnung zu einem Client erfolgt jeweils bei Ankunft eines Requests. Der Client baut seinerseits vor dem Aufruf einer Serverprozedur eine Transportverbindung zum Server auf. Der Aufbau der Transportverbindung hat aber noch keine Aktivierung zur Folge. Erst der konkrete Prozeduraufruf führt zur Aktivierung.

Beispiel-Client für Add-Prozedur: Im folgenden Beispielcode erzeugt ein Client über die Low-Level-API *clnt_create* einen Client-Handle. Dabei wird implizit der Portmapper zum Auffinden des Servers kontaktiert und eine Transportverbindung (hier über UDP) mit dem Server hergestellt.

```
main (argc, argv )
          int argc; char *argv [ ];
{
    CLIENT *cl;
    char *server;
    request parameter;
    result *ergebnis;
    /* Serveradresse in 1. Parameter */
    server = argv [1];
    cl = clnt_create (server, ADD_PROG, ADD_VERS, "udp");
    /* Falls fehlgeschlagen, entsprechend reagieren */
    parameter.a = 27;
    parameter.b = 53;
    ergebnis = add_1 (&parameter, cl);
    printf ("Summe ist %d\n", ergebnis -> x);
}
```

Möchte man hierfür eine andere Vorgehensweise, kann man über die Low-Level-API eine eigene Implementierung über einen individuellen RPC-Dispatcher einhängen.

Basisdienste

Außer dem bereits erwähnten Naming-Service werden von ONC RPC noch Authentifizierungdienste über die sog. *Advanced Programming Features* zur Verfügung gestellt. Ansonsten werden keine weiteren Dienste bereitgestellt.

Andere RPC-Varianten bieten hier mehr. Beispielsweise stellt DCE eine komplette Plattform mit mehreren wichtigen Diensten (Thread-Service, Security-Service, Verteiltes Filesystem) zur Verfügung. DCE konnte sich aber nur sehr begrenzt durchsetzen und spielt heute nur noch eine untergeordnete Rolle.

Sonstige Unterscheidungsmerkmale

Möglichkeiten zur Skalierung und zur Erhöhung der Verfügbarkeit der Serverdienste sind bei ONC RPC standardmäßig nicht vorhanden. Auch hier muss man eigene Lösungen, wie etwa einen Dispatcher, der die Requests auf mehrere Server verteilt, selbst implementieren.

Die höheren API-Levels sind zwar recht einfach, will man aber mehr Leistung, Skalierbarkeit und Ausfallsicherheit, so muss man eigene Lösungen ergänzen.

2.3 Verteilte Objekte

2.3.1 Grundlegendes Modell

In der Literatur wird gelegentlich das Attribut „objektorientiert“ auch direkt auf verteilte Objekte übertragen. Wir ziehen den ebenfalls verwendeten Begriff „objektbasiert“ vor, da verteilte Objekte doch einige Einschränkungen in Kauf nehmen müssen und nicht alle Mechanismen der Objektorientierung unterstützen.

Ein objektorientiertes Programm kann man sich als eine Menge von miteinander kommunizierenden Objekten vorstellen. Objekte senden sich Botschaften zu, die die Ausführung von Methoden bewirken. Beim Methodenaufruf werden (wie beim Aufruf einer Prozedur) Parameter übergeben und es kann ein Rückgabewert zurückgeliefert werden. Die Kommunikation wird dabei durch das Laufzeitsystem der jeweiligen Programmiersprache wie z.B. bei C++ oder durch eine virtuelle Maschine wie bei Java oder den .NET-Sprachen (C#,...) unterstützt. Der Aufruf einer Methode wird gewöhnlich über einen Stackmechanismus unterstützt. In lokalen Objektimplementierungen kann man auch auf Attribute eines Objekts zugreifen, sofern sie als öffentlich deklariert sind. In der Regel erfolgen die Zugriffe aber über dedizierte Methoden.

Die Erzeugung eines lokalen Objekts, also einer Objektinstanz, ist relativ einfach. Man deklariert eine Variable von dem entspechenden Objekttyp und erzeugt sich

ein Objekt, das gewöhnlich im Heap abgelegt wird. Je nach Programmiersprache muss man sich entweder selbst als Programmierer um die Freigabe von Objekten kümmern, wenn man diese nicht mehr benötigt (siehe z.B. in C++), oder aber das Laufzeitsystem wickelt diese Aufgabe automatisch über einen Garbage-Collection-Mechanismus ab (siehe z.B. Java, C#). Der Zugriff auf lokale Objekte erfolgt über Objektreferenzen.

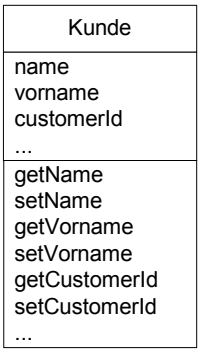


Abbildung 2-17: Einfache Objektklasse

In Java sieht die Nutzung eines Objekts einer beliebigen Klasse (hier *Kunde*, siehe Abbildung 2-17) wie folgt aus:

```
class Kunde {
   private String name;
   private String vorname;
   public int customerId;[5]
        …
   public Kunde(String name, String vorname, int id)
   {
        /* Konstruktorcode */
   }
   public String getName() {…}
   public String getVorname() {…}
   …
}
public static void main() {
   Kunde customer = new Kunde();
   String name = Customer.getName();
   int id = Customer.customerId;
}
```

[5] Nur zur Demonstration von Zugriffen auf öffentliche Attribute eines Objekts, nicht unbedingt sinnvoll im „richtigen Leben“.

Das einfache Beispiel zeigt eine Klasse *Kunde* und die Nutzung eines Objekts dieser Klasse innerhalb eines Programms. Mit der Anweisung *new* wird eine neue Objektinstanz im Heap erzeugt, im Beispiel wird ein Aufruf der in Kunde definierten Methode *getName*, aber auch der Zugriff auf ein öffentliches Attribut namens *customerId* gezeigt. Zur Laufzeit des Programms spielt sich alles im Adressraum eines Prozesses ab. Sowohl das nutzende, als auch das verwendete Objekt sind im gleichen Adressraum und das Betriebssystem muss bei Aufruf der Methode lediglich dafür sorgen, dass der Programmzähler auf die Anfangsadresse der gerufenen Methode gesetzt wird.

Nun ist es in lokal ablaufenden, objektorientierten Programmen relativ klar, wie ein Objekt erzeugt und benutzt wird. Der Aufruf einer Objektmethode wird immer genau einmal ausgeführt und das ist so möglich, weil die Ausführung im lokalen Prozesskontext einfach durch den Ansprung des Methodencodes, der irgendwo im Prozess-Adressraum liegt, erfolgen kann. Es ist keine prozessübergreifende Kommunikation und schon gar keine Kommunikation in einem Netzwerk erforderlich.

In verteilten Objektsystemen wird nun dieses lokale Objektparadigma auf die Nutzung in verteilten Anwendungen erweitert. Die Anforderung lautet: Ein Objekt soll auf einem beliebigen Rechnersystem instanziiert werden können und von einem Objekt, das auf einem anderen Rechnersystem liegt, genutzt werden können, und dies möglichst so wie im lokalen System. Der Aufruf von Methoden eines entfernten Objekts wird auch als *RMI (Remote Method Invocation*), bezeichnet.

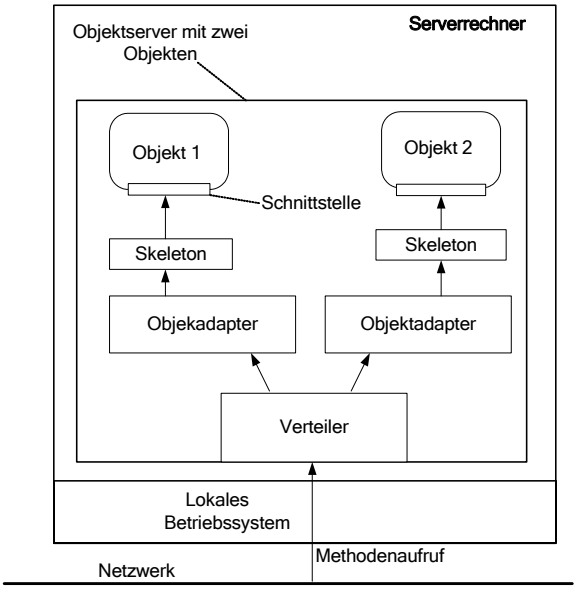


Abbildung 2-18: Prinzipieller Aufbau eines Objektservers nach Tanenbaum 2003

Viele der zu lösenden Probleme wurden bereits im prozeduralen *Remote Procedure Call (RPC)* gelöst, also einem Mechanismus, der es ermöglicht, eine entfernte Prozedur aufzurufen. Hier gibt es zwar keine Objekte, aber die grundlegenden Probleme treten auch schon auf. Es liegt also nahe, das Konzept des Remote Procedure Calls auf verteilte Objekte anzuwenden. Das Konzept der verteilten Objekte ist eine Erweiterung des lokalen Objektmodells kombiniert mit dem Remote Procedure Call. Entfernte Objekte bieten ihre Methoden für Clients an. Clients rufen die Methoden entfernter Objekte ortstransparent auf, so als ob sie lokal ablaufen würden.

Aus Implementierungssicht muss man es schaffen, Objekte, die üblicherweise in Adressräumen von Betriebssystemprozessen in Serverrechnern ablaufen, aus anderen Prozessen heraus nutzen zu können. Dies ist aber auch vom Grundprinzip her nicht viel anders als der Zugang auf einzelne Funktionen. Es wird lediglich ein anderes Programmierparadigma unterstützt.

Im klassischen Client-Server-Modell haben wir bisher von Serverbausteinen gesprochen, in denen die Services bereitgestellt wurden. Die verteilten Objekte werden in sog. *Objektservern* verwaltet und stellen über diese ihre Dienste in Form von Methoden bereit. Objektserver sind also (softwaretechnisch) Server, die Clientanwendungen (oder Client-Objekte) die Methoden einzelner Objekte (in der Regel) über ein Netzwerk zugänglich machen. Sie stellen die Infrastruktur für die verteilten Objekte (Remote Objects) bereit.

Anstelle eines Servers ist bei verteilten Objekten ein Objektserver für die Verwaltung und die Nutzung der entfernten Objekte zuständig. Dies ist im Prinzip auch mindestens ein Betriebssystemprozess.

In einem Serverrechner ankommende Methodenaufrufe werden über ein entsprechendes Kommunikationsprotokoll zunächst an den Objektserver weitergeleitet.

Die Anatomie eines Objektservers ist in Abbildung 2-18 skizziert. Im Folgenden soll das Zusammenspiel der Komponenten kurz beschrieben werden:

- Ein Methodenaufruf wird zunächst von einem *Demultiplexer (Dispatcher)* entgegengenommen. Dieser ermittelt den für das Objekt zuständigen Objektadapter.
- Der *Objektadapter* ist für die Aktivierungsstrategie verantwortlich. Er entscheidet, wann das Objekt konkret erzeugt wird und wie lange es am Leben bleibt. Ein Objektadapter kann mehrere Objekte bedienen und stellt einen Objekt-Wrapper dar. Es können auch mehrere *Objektadapter* mit unterschiedlichen Aktivierungsstrategien in einem Objektserver ablaufen.
- Der Objektadapter ermittelt adressierte Objekte und das verantwortliche *Skeleton* anhand einer Objektreferenz und leitet die Anfrage an den verantwortlichen *Skeleton* (auch Serverstub) weiter.
- Das *Skeleton* entpackt den Methodenaufruf, deserialisiert die Parameter und ruft schließlich die Methode des adressierten Objekts auf.

– Das Objekt bearbeitet den Methodenaufruf und sendet die Antwort über das *Skeleton* und den *Objektadapter* an den Client zurück.

Objektserver-Implementierungen sind meist leistungsfähige, Thread-basierte Softwaresysteme, die auch Threadpooling vewenden. Da ein entferntes Objekt für den Client nicht direkt im Zugriff ist, wird lokal beim Client ein Stellvertreter-Objekt, ein sog. *Proxy-Objekt (*Stellvertreter-Objekt) erzeugt. Dieses Objekt stellt gegenüber dem Nutzerobjekt die Objektschnittstelle bereit und zwar so, als ob das entfernte Objekt lokal verfügbar wäre. Es wickelt die Kommunikation mit dem Serverobjekt ab und liefert das Ergebnis des Methodenaufrufs an das Client-Objekt zurück.

Bevor aber ein Proxy-Objekt im Client erzeugt werden kann, muss ein sog. *Binding* erfolgen. Hierfür gelten im Wesentlichen die gleichen Mechanismen wie bei RPC. Der Client wendet sich an einen Naming- oder Directory-Service und dieser muss das entfernte Objekt aufgrund einer Registrierung kennen und die Transportadresse bzw. die sog. *Objektreferenz* des Serverobjekts zur Verfügung stellen. Die Objektreferenz ist ein Bezeichner für ein entferntes Objekt bzw. ein entfernter Objektverweis, der innerhalb des gesamten verteilten Systems Gültigkeit hat. Genauer gesagt handelt es sich um die Adresse eines Objekts innerhalb eines Serverprozesses, der wiederum auf einem Rechnersystem abläuft. Diese Zusammenhänge sind in Abbildung 2-19 dargestellt.

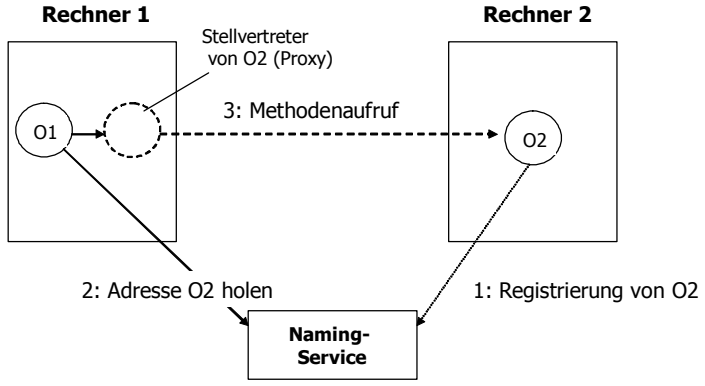


Abbildung 2-19: Kommunikation zwischen Objekten

2.3.2 Fallbeispiel CORBA

Ziel der Object Management Group (OMG), einem Industriekonsortium, ist es u.a., verteilte objektbasierte Systeme zu standardisieren und deren Weiterentwicklung zu fördern. Die OMG hat in den 90er Jahren eine Architektur (OMA, Object Management Architecture) für verteilte, objektorientierte Anwendungen entwickelt. Wir betrachten CORBA nun anhand unserer Unterscheidungskriterien.

Architektur

Die OMA-Architektur ist ein sehr umfassender Architekturansatz und enthält mehrere Grundkomponenten, die in Abbildung 2-20 skizziert sind:

- *Application Objects*: Dies sind die speziellen verteilten Objekte der eigenen Anwendung und nicht Bestandteil der Standardisierung.
- *CORBAdomains*: In diese Kategorie von Services fallen sog. *vertikale* Dienste, die sich auf eine bestimmte Anwendungsdomäne beziehen. Sie sind speziell für diese konzipiert, aber innerhalb der Domäne als Basisdienste zu betrachten. Die Beispiele hierfür sind vielfältig (Gesundheitswesen, Financial, Transport, Telekommunikation,...).
- *CORBAservices*: Dies sind allgemeine Services auf Systemebene, die für die Anwendungsentwickler bereitgestellt werden.
- *CORBAfacilities*: Hier handelt es sich um spezielle, *horizontale* Services, die von Anwendungsdomänen unabhängig sind, die aber für Anwendungen meist benötigt werden. In diese Kategorie von Diensten gehören u.a. Systemmanagement-Dienste, Internationalisierungsmechanismen und Workflow-Dienste. Die Trennung zwischen horizontalen und vertikalen Diensten scheint etwas willkürlich.
- Der Object Request Broker (ORB) ist das Kernstück der Architektur (OMG 2004). Er ermöglicht, dass Objekte in einer verteilten heterogenen Umgebung miteinander kommunizieren können. Da der ORB als wesentlicher Bestandteil gilt, hat sich die Bezeichnung CORBA (Common Object Request Broker Architecture) in der Praxis als geläufigerer Name der Architektur durchgesetzt.

Ein ORB ermöglicht die Kommunikation aller beteiligten Komponenten und deren Zusammenspiel. CORBA-Implementierungen sind auf nahezu allen Betriebssystem-Plattformen erhältlich. Sie sind auch häufig in Middleware-Produkten integriert.

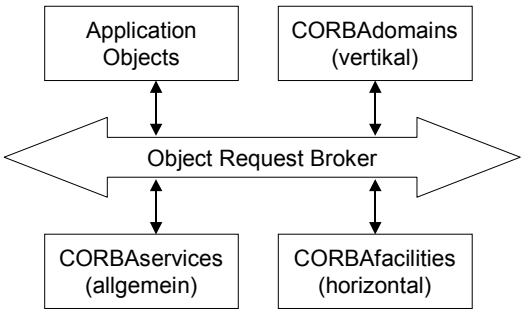


Abbildung 2-20: Object Management Architecture (OMA)

Kommunizierende Objekte können sich auf beliebigen Rechnern im Netzwerk befinden. Ein CORBA-Client muss nicht wissen, wo sich ein CORBA-Server befindet, wie ein Methodenaufruf zu ihm transportiert wird und in welcher Programmiersprache der CORBA-Server implementiert ist.

In praktischen Implementierungen gibt es einen ORB meist im Anwendungsprozess oder als eigenständigen Prozess auf jedem Rechner (Client- und Serverrechner). In Abbildung 2-21 sind die verschiedenen Varianten skizziert.

Die Bausteine der CORBA-Architektur im Client und im Server sind in Abbildung 2-22 skizziert. Dies sind im Einzelnen:

- Client
- Stub (Client) oder DII (Dynamic Invocation Interface)
- Objektadapter (BOA bzw. POA)
- Skeleton (Server) oder DSI (Dynamic Skeleton Interface)
- Servant

ORBs übernehmen den Transport eines Methodenaufrufs vom Client zum Server sowie den Transport der Ergebnisse auf dem umgekehrten Weg und verwenden hierzu ein definiertes Protokoll, das allgemein als GIOP (General Inter-ORB-Protocol) bezeichnet ist.

Der individuelle Clientcode greift entweder über einen statisch generierten Stub oder über eine dynamische Aufrufvariante (DII) auf die entfernten CORBA-Objekte zu. Auf der Serverseite ist ein Objektadapter, der die Requests entgegennimmt und entweder an ein statisch generiertes Skeleton oder dynamisch über DSI an das CORBA-Objekt weiterleitet. Skeleton und Stub fungieren als Stellvertreter-Objekte (Proxy-Objekte) der jeweiligen Gegenseite.

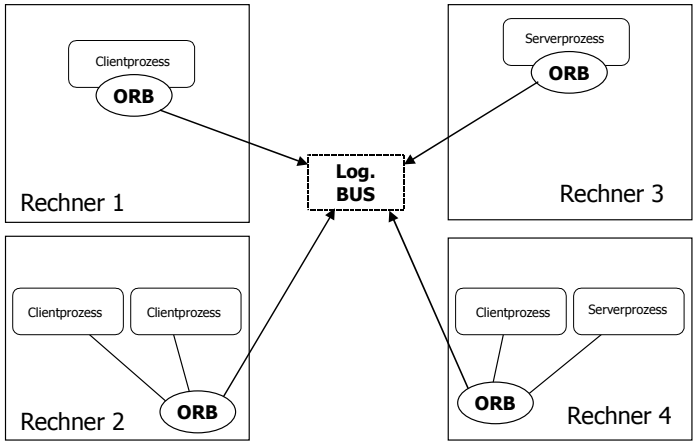


Abbildung 2-21: Komponenten von CORBA

Der Unterschied zwischen statischen Stubs bzw. Skeletons und DII/DSI ist also, dass mit ersteren die Operationen zur Übersetzungszeit definiert werden, während man mit DII/DSI die Informationen für eine Operation zur Laufzeit ermitteln kann. Damit stellt DII/DSI einen generischen Aufrufmechanismus bereit, mit dem ein Client die Schnittstelle eines CORBA-Objekts zur Laufzeit besorgen und Methoden davon aufrufen kann.

Der *Objektadapter* ist Mittler zwischen dem ORB und der Objekt-Implementierung. Er stellt eine Schicht zwischen ORB und Skeleton dar, nutzt das Implementation-Repository für Konfigurationsdaten, führt die Generierung von Objektreferenzen aus und erledigt den Aufruf von Operationen durch das Skeleton sowie die Aktivierung von Objekten. Ein Implementation Repository enthält Informationen darüber, wo die Serverobjekte ablaufen. Es ist Bestandteil einer ORB-Implementierung, aber nicht standardisiert.

Ein CORBA-Objekt ist eine abstrakte Einheit und besitzt ein Interface. Es kann vom ORB lokalisiert werden. Das Interface stellt Operationen bereit. Ein CORBA-Objekt ist durch eine Objektreferenz identifiziert (eindeutige Identifikation). Ein *Servant* ist eine Implementierung eines CORBA-Objekts in einer von CORBA unterstützten Programmiersprache. Ein Servant kann mehrere CORBA-Objekte realisieren. Requests des Clients an ein CORBA-Objekt werden im Servant bearbeitet.

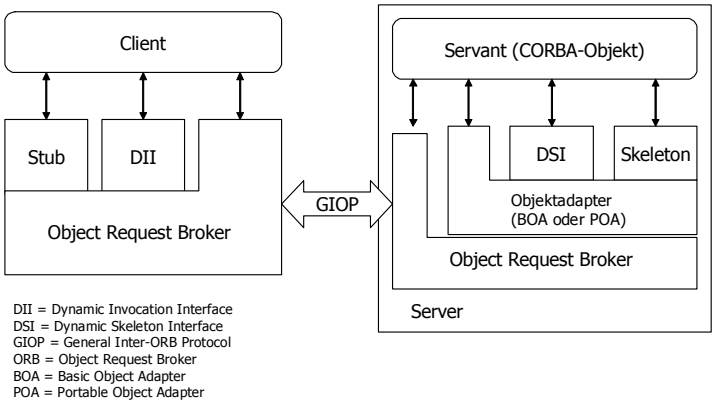


Abbildung 2-22: CORBA-Komponenten nach (Bengel 2004)

Als Objektadapter sind in CORBA der *Basic Object Adapter* (BOA) und der *Portable Object Adapter* (BOA) spezifiziert. Der *Portable Object Adapter (POA)* stellt seit der CORBA-Version 2.2 den Standard (OMG 2002a) dar und ist eine Weiterentwicklung des einfacheren BOA-Konzepts. Ein POA ist eine Verwaltungseinheit innerhalb des Servers. Er dient der logischen Trennung von CORBA-Objekt und Servant und verwaltet eine Gruppe von CORBA-Objekten, wobei er auch deren Objektreferenzen erzeugt. Ein POA setzt einen Request an ein CORBA-Objekt auf einen

Aufruf im Servant um. Innerhalb eines Servers (Ausführungsumgebung für CORBA-Objekte) sind mehrere POA-Instanzen möglich. Ein Objektserver enthält mindestens einen *RootPOA* und kann eine ganze POA-Hierarchie unterstützen (siehe Abbildung 2-23).

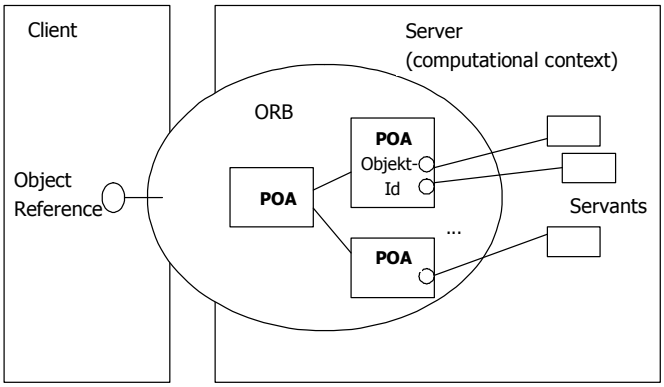


Abbildung 2-23: Der Portable Object Adapter im ORB-Kontext

Zustandsverwaltung

Die Zustandsverwaltung kann bei CORBA nur über den Programmierer beeinflusst werden. Es wird nicht zwischen stateful, stateless und Singleton-Serverbausteinen unterschieden.

Parameterübergabe

Die Parameter und Returnwerte werden in CORBA über *Call-by-value* zwischen Client und Server ausgetauscht, wenn sie auch lokal im Client beim Aufruf einer entfernten Objektmethode *per-reference (Call-by-reference)* übergeben werden. Die Parameter bzw. Returnwerte werden vor dem Versenden kopiert und serialisiert. Man spricht hier – wie bereits weiter oben dargestellt - auch von *Call-by-copy*.

Eine Ausnahme bilden Objektreferenzen, die als Parameter bzw. als Returnwerte übergeben werden. Für diese Fälle wird *Call-by-reference* verwendet.

Dienstschnittstellen

CORBA ist sprachunabhängig konzipiert, d.h. CORBA-Objekte können in beliebigen Sprachen implementiert sein. Die Implementierungssprache eines Serverobjektes ist für den Client transparent, d.h. er sieht nicht, in welcher Sprache ein Serverobjekt implementiert ist. Die Server-Schnittstelle (Dienste eines CORBA-basierten Servers) wird sprachneutral in der *Interface Definition Language* (*IDL*) beschrieben.

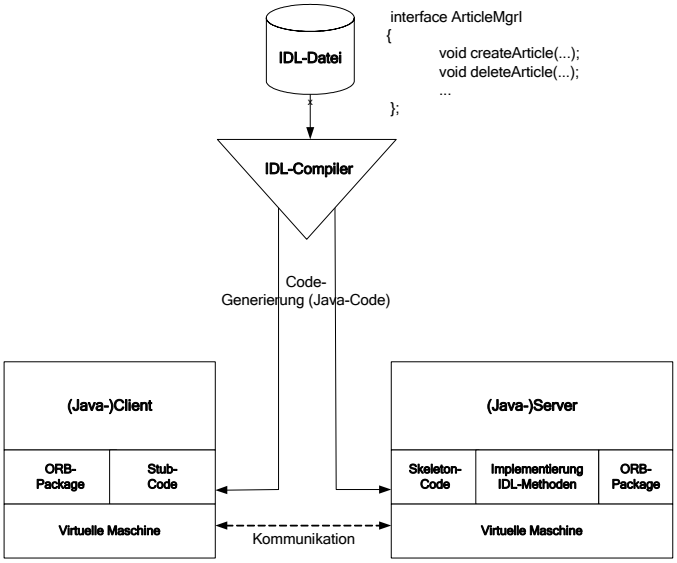


Abbildung 2-24: Codegenerierung aus einer IDL-Schnittstellenbeschreibung

Alle CORBA-Services verfügen auch über eine oder mehrere in IDL spezifizierte Schnittstellen. CORBA IDL ist rein deskriptiv (beschreibend). Man kann in der IDL explizit die Übergabemechanismen der Parameter spezifizieren, d.h. CORBA IDL unterscheidet Eingabe-, Ausgabe- und Ein/Ausgabe-Parameter. IDL ist eine stark an C angelehnte Sprache und dient nur zur Spezifikation der Schnittstelle. Die Implementierung eines Objektes erfolgt in einer Programmiersprache wie beispielsweise C, C++, Java, ADA oder COBOL.

Als Entwickler einer CORBA-Anwendung definiert man in einem frühen Entwicklungsstadium die Server-Schnittstelle mit IDL und erzeugt mit Hilfe eines IDL-Compilers den Stub- und Skeleton-Code. Der Compiler erzeugt je nach Sprach-Binding entsprechenden Code für Java, C++ usw. In Abbildung 2-24 ist die Generierung am Beispiel einer einfachen IDL-Datei skizziert. Ein *Modul* bildet die oberste Ebene einer IDL-Spezifikation und fasst inhaltlich zusammengehörende IDL-Definitionen zusammen. Module sind Namensräume und dienen der Vermeidung von Kollisionen mit Bezeichnern aus anderen Modulen. In Modulen können Interfaces, Typdeklarationen, Konstanten, Ausnahmen, Operationen, Attribute und auch andere Module enthalten sein.

Der IDL-Compiler erzeugt üblicherweise einen herstellerspezifischen Code. Für jede neue CORBA-Entwicklungsplattform ist die Generierung daher erneut auszuführen. Die Kommunikationsschnittstellen sind aber standardisiert. Ein Modul sieht vom Aufbau her grob wie folgt aus:

```
module <Modulname> {
     const ...
     typedef ...
     enum...
     struct ...
     exception ...
     interface ...
     module ...
}
```

Interfaces beschreiben eine Menge von Operationen und sind vererbbar, wobei Mehrfachvererbung möglich ist (siehe <*BaseInterface*> unten). Interfaces sind ebenfalls einem separaten Namensraum zuordenbar und können Typdeklarationen, Konstanten, Ausnahmen, Operationen und Attribute enthalten. Der Aufbau einer Interface-Spezifikation lässt sich wie folgt skizzieren:

```
interface <Interfacename> [: <BaseInterface> ] {
     const ...
     typedef ...
     enum...
     struct ...
     exception ...
}
```

Konstanten werden wie in C++ angegeben:

```
const short magic_number = 42;
```

IDL kennt mehrere Basis-Datentypen, die zum Teil der Sprache C/C++ sehr ähnlich sind: Hierzu gehören (*unsigned*) *short*, *char*, *wchar*, (*unsigned*) *long*, *float*, *double*, *boolean*, *octet* und *long long*. Der IDL-Typ *char* ist z.B. ein 8-Bit-Zeichen nach ISO 8859.1, bei *wchar* handelt es sich um ein 16-Bit-UNICODE-Zeichen. Schließlich gibt es in der CORBA-IDL auch noch den Datentyp *any*, mit dem man den Typen eines Datenelements zunächst nicht festlegt. Die Typbestimmung wird zur Laufzeit ausgeführt.

CORBA unterstützt auch, ähnlich wie bei den Sprachen C und C++ Aufzählungstypen. Darüber hinaus werden komplexe Strukturen (*struct*), Unions (*union*), Arrays und Sequenzen (*sequence*) unterstützt. Strukturen können Basis-Datentypen und wiederum Arrays, Unions und andere Strukturen enthalten.

Operationen werden in IDL ebenfalls ähnlich wie Methoden in C++ notiert. Zusätzlich zu C++ muss für jeden Parameter angegeben werden, ob es sich um einen Eingabeparameter (*in*), einen Ausgabeparameter (*out*) oder einen Ein-/Ausgabeparameter (*inout*) handelt. Parameter und Ergebnisse können einen beliebigen Typ haben. Die Definition einer Operation sieht z.B. wie folgt aus:

```
void tell(in string name, in string message);
```

Neben den genannten Datentypen und Operationen können IDL-Schnittstellen auch beliebig viele *Attributdeklarationen* enthalten. Standardmäßig können Attribute gelesen und geschrieben werden. Das Schlüsselwort *readonly* schränkt den Zugriff auf ein Attribut ein. Beispiele sind:

```
attribute short customerName;
readonly attribute long KontoNr;
```

Der IDL-Generator erzeugt für diese Attribute entsprechende Methoden zum Lesen und Schreiben.

Für die Programmierung verteilter Anwendungen mit CORBA ist das Mapping der IDL-Typen und Operationen auf die entsprechende Programmiersprache wichtig. Hier spricht man von *Language-Binding*. Die Formate der IDL-Basistypen sind in der CORBA-Spezifikation festgelegt (OMG 2004). Ein *Language-Binding* regelt die Abbildung auf spezielle Programmiersprachen wie Java oder C++. Das Mapping von IDL auf eine Sprache wie Java erfolgt also nach definierten Regeln. Beispielsweise wird in Java ein IDL-Modul auf ein gleichnamiges Java-Package abgebildet. Meistens gibt es bei konkreten Bindings für Programmiersprachen keine 1:1-Abbildung der Typensysteme. Ein IDL-Typ *unsigned long* kann z.B. größere Zahlen darstellen als ein 4-Byte-Integer in Java.

Marshalling und Unmarshalling

Für die Übertragung der Methodenaufrufe, Parameter und Returnwerte wird in CORBA eine einheitliche Transportsyntax mit der Bezeichnung *CDR* (Common Data Representation) verwendet. CDR stellt eine ähnliche Lösung wie ONC XDR bereit. Ein IDL-Compiler erzeugt die Routinen für die Serialisierung und Deserialisierung automatisch aus der IDL-Schnittstellenbeschreibung. Zur Laufzeit wird während des Serialisierungs-/Deserialisierungsvorgangs eine *CORBA::DATA_CONVERSION*-Ausnahme geworfen, wenn die Abbildung eines Werts der Transportsyntax auf eine konkrete, lokale Syntax nicht möglich ist.

Adressierung und Kommunikation

Namensauflösung und Naming-/Directory-Service: Die Namen der Serverobjekte werden auch bei CORBA über einen Naming-Service verwaltet, der in CORBA als eigener Dienst spezifiziert ist. Die Referenzierung von verteilten Objekten geschieht in CORBA über eine Objektreferenz. Man muss aber zwischen der lokal über den Stub verwendeten Objektreferenz im Client und der Objektreferenz, die im ORB verwaltet wird, unterscheiden.

Der Client ruft Methoden des verteilten Objekts über seine lokale Objektreferenz auf. In der Implementierung handelt es sich hier typischerweise um einen Pointer auf ein lokales Objekt, der nicht zwischen Prozessen ausgetauscht werden kann.

CORBA-Objekte müssen eindeutig identifizierbar sein. Dies geschieht über eine eindeutige, sprachunabhängige Objektreferenz. Diese sprachunabhängige Darstel-

lung wird als IOR (Interoperable Object Reference) bezeichnet. In der IOR müssen auch die Rechner- und die Prozessadresse des Objekts enthalten sein. Damit ist auch die Kommunikation zwischen Objekten, die in verschiedenen CORBA-Implementierungen ablaufen, möglich. Der Aufbau einer IOR ist in Abbildung 2-25 aufgezeichnet. Sie enthält alle Adressierungsinformationen zur Adressierung eines CORBA-Objekts aus Sicht des Clients und ist entsprechend Abbildung 2-25 aufgebaut:

- Die *Repository-Id* dient dem Auffinden der IOR in einem Interface-Repository. Dies ist ein einfaches Verzeichnis mit Metainformationen über die Schnittstellen von CORBA-Objekten. Das Interface-Repository ist Bestandteil der ORB-Implementierung.
- Die *Tagged Profiles* enthalten die komplette Information für die Nutzung des Objekts, wobei für jedes unterstützte Protokoll ein Profil angelegt wird (derzeit nur für IIOP). Im Bild ist ein Profil für eine IIOP-basierte IOR skizziert (IIOP siehe unten).
- Das Feld *Hostadresse* enthält entweder einen DNS-Hostnamen oder die IP-Adresse.
- Das *Port*-Feld gibt die TCP-Portnummer an, unter der der Server auf Requests für dieses Objekt horcht. Über die Hostadresse und den Port wird zunächst der Objektserver gefunden.
- Im Feld *Objektschlüssel* sind Informationen zur Adressierung innerhalb des Objektservers kodiert. Hierzu gehören eine POA-Id zur Adressierung des POA, eine *Objekt-Id* zur Adressierung des konkreten CORBA-Objekts innerhalb des POA und weitere Informationen. Die Zuordnung von abstrakten CORBA-Objekten zu konkreten Servants erfolgt mit Hilfe der *Objekt-Id*.
- Im Komponenten-Feld sind weitere Informationen zum Aufruf von Methoden eines Objekts, die hier nicht weiter erläutert werden sollen.

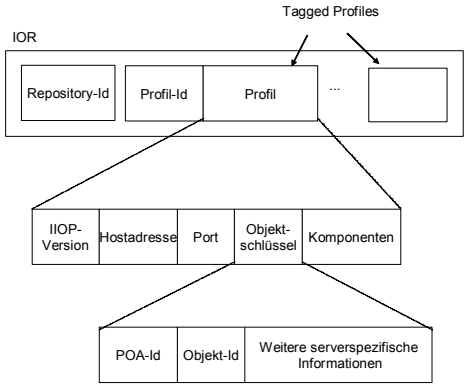


Abbildung 2-25: CORBA-IOR mit IIOP-Informationen nach (Tanenbaum 2003)

Der CORBA-Naming-Service unterstützt den Client beim Auffinden einer Objektadresse, also der entfernten Objektreferenz. Namen, an die man Objekte binden kann, sind frei wählbar und ortsunabhängig. Der Implementierungsort eines CORBA-Objektes kann für Clients transparent geändert werden. Der CORBA Naming-Service ist selbst ein CORBA-Serverobjekt und muss als solcher implementiert sein. In der Anwendung kann man bei einer geeigneten Implementierung für Ortstransparenz sorgen.

Das Bootstrapping-Problem ist auch in CORBA vorhanden. Der Client muss zuerst die Adresse eines Naming-Service herausfinden, bevor er von diesem die Adressen der verteilten Objekte ermitteln kann. Für das initiale Bootstrapping bietet CORBA zwei spezielle Operationen:

- *list_initial_references*: Gibt die Namen aller initial verfügbaren Dienste zurück - dazu gehört auch der Naming Service.
- *resolve_initial_references*: Liefert zu einem gegebenen Dienstnamen eine IOR (ORB-unabhängige Referenz) auf das entsprechende CORBA-Objekt. Für den Naming-Service liefert die Methode einen globalen Namenskontext als Referenz.

Auf der Clientseite kann der Namenskontext nach einer Referenz auf den gesuchten Dienst abgefragt werden.

Kommunikation: Der ORB als zentrale Komponente wickelt die Kommunikation ab und nutzt dabei einen definierten Protokollstack, der in Abbildung 2-26 zu sehen ist. In diesem Bild sind auch der logische und der tatsächliche Nachrichtenfluss skizziert.

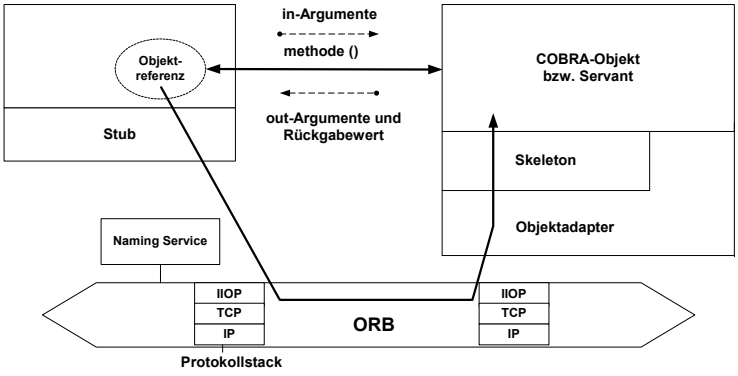


Abbildung 2-26: Zusammenspiel im Überblick

Der Client-Stub übernimmt das Marshalling und gibt Requests an den ORB weiter. Der ORB ist dafür zuständig, das Objekt, also den zugehörigen Servant, zu lokalisieren und den Aufruf serverseitig an den entsprechenden Objektadapter weiterzugeben. Der Objektadapter ist u.a. für die Aktivierung des CORBA-Objekts zu-

ständig, falls es im Server noch nicht läuft. Der Aufruf wird vom Objektadapter an das Skeleton weitergegeben. Die Parameter werden dabei entpackt und die eigentliche Objekt-Implementierung mit der Ausführung der Methode beauftragt. Die Rückgabe der Ergebnisse (Response) geschieht analog.

ORBs kommunizieren über ein von der OMG definiertes Protokoll, das als GIOP (General Inter-ORB Protocol) bezeichnet wird. GIOP ist die allgemeine, transportunabhängige Spezifikation des Protokolls und beschreibt minimale Anforderungen an das Transportprotokoll. Das zugrundeliegende Transportprotokoll sollte zuverlässig und verbindungsorientiert sein und auch ein Byte-Streaming-Konzept wie beispielsweise TCP unterstützen. Es stellt eine Familie von Protokollen, keine konkrete Implementierung dar. Als Fehlersemantik wird *At-Most-Once* gefordert. In Abbildung 2-27 ist der Ablauf eines synchronen CORBA-Requests mit statischen Stub und Skeleton skizziert. Die Nachrichtenübertragung erfolgt hier über die bisher einzige praxisrelevante GIOP-Implementierung mit der Bezeichnung *IIOP* (Internet IOP). IIOP nutzt TCP als Transportprotokoll.

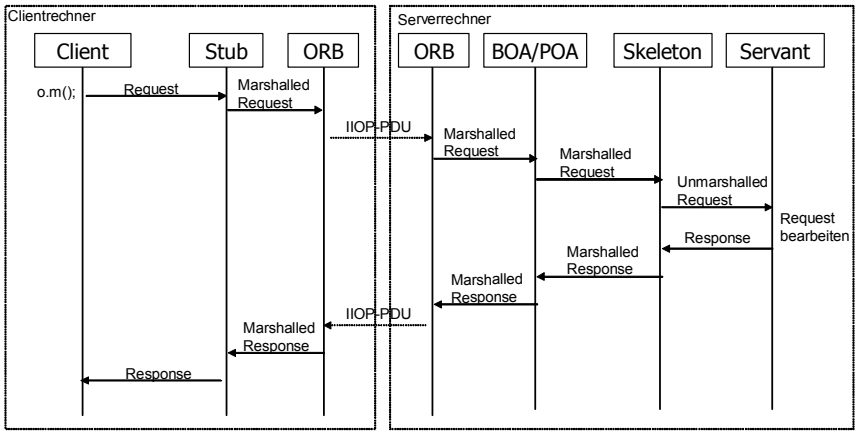


Abbildung 2-27: Zeitlicher Ablauf eines synchronen CORBA-Requests

GIOP/IIOP ist ein einfaches Request-/Response-Protokoll. Als Datenrepräsentation wird *CDR (Common Data Representation)* verwendet, das dem XDR aus Sun ONC ähnelt. GIOP/IIOP ist ursprünglich unidirektional (halbduplexfähig) spezifiziert, aber ab CORBA V3.0 gibt es auch eine bidirektionale (vollduplexfähige) GIOP-Variante. Ein Client sendet einen Request an den Server und der Server antwortet mit einer Response-Nachricht. Der Client ist auch grundsätzlich für den Verbindungsaufbau über das verwendete Transportprotokoll verantwortlich. Der Server darf dies nicht.

Es gibt acht GIOP/IIOP-PDUs (OMG 2004):

- *Request*: Wird vom Client gesendet und enthält einen Methodenaufruf (Request) an ein verteiltes Objekt.
- *Reply*: Wird vom Server als Antwort auf einen Request gesendet.
- *CancelRequest*: Wird vom Client gesendet, um einen *Request* oder einen *LocateRequest* abzubrechen. Der Client wartet nach dem Senden dieses Nachrichtentyps nicht mehr auf einen *Reply*. Die Nutzung von *CancelRequest* bedeutet allerdings nicht, dass der Request im Server nicht mehr ausgeführt wird.
- *LocateRequest*: Der Client sendet diesen Nachrichtentyp an das Implementierungs-Repository (also an den entfernten ORB), um herauszufinden, wo ein bestimmtes Objekt erreicht werden kann.
- *LocateReply*: Der Server sendet diesen Nachrichtentyp als Antwort auf LocateRequest. Das Implementierungs-Repository (der entfernte ORB) antwortet mit diesem Nachrichtentyp und sendet damit Informationen zum Ort des Objekts, welches gesucht wurde.
- *CloseConnection*: Wird vom Server gesendet, um eine Verbindung zu schließen.
- *MessageError*: Wird von beiden Partnern bei einem Fehler gesendet.
- *Fragment*: Über diesen Nachrichtentyp ist es möglich, eine große IIOP-Nachricht in einzelne Fragmente zu zerlegen, die beim Empfänger vor der Zustellung zum Anwendungsbaustein zusammengebaut werden.

Jeder Request ist durch eine Request-Id gekennzeichnet, die im Aufrufer eindeutig ist und vom ORB erzeugt wird. Eine Reply-PDU muss sich auf den Request über die Rückgabe der Request-Id beziehen.

In der GIOP/IIOP-Spezifikation werden Protokollregeln festgelegt. Beispielsweise darf nur der Client den Verbindungsaufbau zu einem Server initiieren. Nur der Server initiiert dagegen den Verbindungsabbau mit einer *CloseConnection-PDU*, und zwar nur dann, wenn er alle Requests bearbeitet hat. Clients müssen nach Absetzen eines Requests nicht auf einen Reply warten (nicht blockierend). Requests dürfen sich überholen, die GIOP/IIOP-Instanzen verwaltet die Request-Reihenfolge. Ein offener Request kann vom Client mit einer *CancelRequest-PDU* abgebrochen werden. Zudem können Verbindungen mehrfach zu einer Zeit verwendet und damit multiplexiert werden (multiplexed Connections).

Abbildung 2-28 zeigt den Nachrichtenfluss für mehrere Requests eines Clients an einen CORBA-Server. Hier werden nacheinander drei Requests ausgeführt, wobei Request 2 und 3 asynchron gesendet werden. Die asynchrone Ausführung eines Requests ist vor allem im hier nicht weiter ausgeführten *Dynamic Invocation Interface (DII)* notwendig.

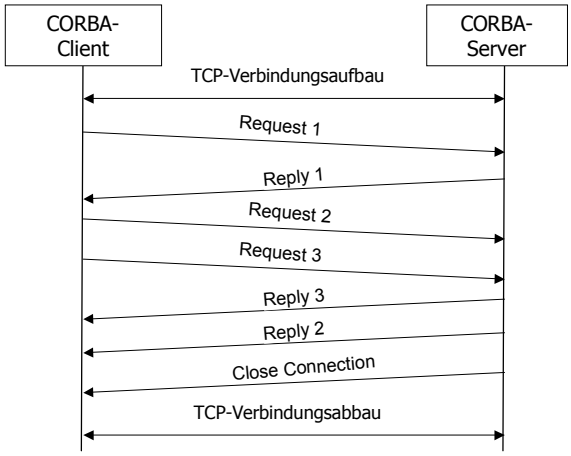


Abbildung 2-28: Nachrichtenfluss bei einer GIOP/IIOP-Verbindung

GIOP-PDUs unterscheiden sich je nach GIOP-Version. Im Folgenden werden einige vereinfachte Request-PDU gezeigt. Im Request/Reply-Body werden die Parameter (in/inout) bzw. die Ergebnisse (inout/out) übertragen. Der grundsätzliche Nachrichtenaufbau ist in Abbildung 2-29 dargestellt.

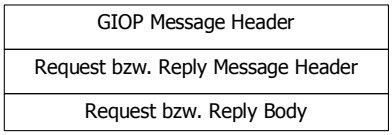


Abbildung 2-29: GIOP-Nachrichtenaufbau

Die Spezifikation des GIOP-Headers sowie des Request- bzw. Reply-Headers sieht in der IDL-Notation wie folgt aus:

```
module GIOP { // GIOP-Header
     struct Version {
     octet major;
     octet minor;
     };
     enum MsgType {
        Request, Reply, CancelRequest, LocateRequest, LocateReply,
        CloseConnection, MessageError, Fragment
     };
     struct MessageHeader {
        char magic[4]; // enthält immer „GIOP"
```

```
        Version GIOP_version; // Version von GIOP
        boolean byte_order; // big oder low endian
        octet flags; // Kennzeichen, ob Fragmentierung zulässig,...
        MsgType message_type; // Typ der Nachricht
        unsigned long message_size; // Länge der folgenden Nachricht
    };
};

module GIOP { // GIOP-Request-/Reply-Header
     struct RequestHeader_1_1 {
        IOP::ServiceContextList; // Service-Daten des Clients
        unsigned long request_id;  //Eindeutige Request-Identifikation
        boolean response_expected; // Antwort erwartet oder nicht
        octet reserved[3];
        sequence<octet> object_key; // Adressiertes Zielobjekt
        string operation; // Aufzurufende Operation im Server
        CORBA::OctetSeq requesting_principal; // Requestor-Id
     };
     // Nach dem Request-Header werden die Parameter in der in
     // IDL festgelegten Reihenfolge übertragen.
};
```

Im Feld *object_key* ist der oben beschriebene *Objektschlüssel* mit *POA-Id*, *Objekt-Id* usw. enthalten. Jede IIOP-Nachricht enthält eine eindeutige Request-Id im Feld *request_id*.

Garbage-Collection

Aufgrund der Sprachunabhängigkeit ist es in CORBA nicht möglich, einen verteilten Garbage-Collection-Mechanismus zu unterstützen, der für alle Bindings funktioniert. Auch das in CORBA definierte Protokoll GIOP bzw. IIOP sieht keine Garbage-Collection-Unterstützung vor. Wenn also der Client eine Referenz auf ein entferntes Objekt erhält, kann er dies nicht an den Server kommunizieren. Das Garbage Collection muss also die CORBA-Implementierung eigenständig und ohne weitere Vorgaben durch die Spezifikation lösen.

Nebenläufigkeit

Die Nebenläufigkeit von CORBA-Implementierungen kann durch die Einstellung von sog. Policies im POA beeinflusst werden. Mit der Angabe der *Thread Policy* kann man z.B. festlegen, dass für jede Transaktion oder für jede Session ein eigener Thread erzeugt wird. Folgende Konfigurationsmöglichkeiten sind u.a. denkbar:

- SINGLE_THREAD_MODEL: Ein Thread bedient alle Requests nacheinander (iterativer Server). Dies ist nur bei Servercode, der nicht thread-safe ist, sinnvoll.

– ORB_CTRL_MODEL: Der ORB erzeugt für jeden eingehenden Request einen eigenen Thread. Dies entspricht einem parallelen Server.

Die Zuordnung von CORBA-Objekten zu Servants wird ebenfalls über POA-Policies kontrolliert. Über die *Object Id Uniqueness Policy* können z.B. folgende Einstellungen vorgenommen werden:

– UNIQUE_ID: Servant verwaltet nur ein CORBA-Objekt
– MULTIPLE_ID: Servant kann mehrere CORBA-Objekte verwalten

Beispiel: POA-Konfigurierung: Die POA-Konfigurierung soll anhand eines kleinen Serverprogramms gezeigt werden. In dem Programm werden zunächst ein ORB und anschließend ein Root-POA erzeugt und aktiviert. Danach wird ein eigener POA mit der *Thread Policy* ORB_CTRL_MODEL ergänzt. Schließlich wird die Kontrolle an den ORB übergeben.

```
public class MyCORBAServer {

    public static void main (String [] Args) throws Exception {
        ORB orb = ORB.init (Args, null);
        POA rootPOA = POAHelper.narrow
          (orb.resolve_initial_references ("RootPOA"));
        rootPOA.the_POAManager ().activate ();
        // Policies fuer neuen POA festlegen
        POA myPOA = rootPOA.create_POA ("myPOA", rootPOA.the_POAManager ());
        myPOA.create_thread_policy(ThreadPolicyValue.ORB_CTRL_MODEL);
        myPOA.the_POAManager().activate();
        orb.run ();
        …
}
```

Lebenszyklus von Serverbausteinen

Über die *POA Lifespan Policy* kann ein Objekt als transient (lebt solange der Server lebt) oder persistent (lebt länger als der Server) definiert werden. Damit werden also Persistenzeigenschaften festgelegt.

Über die POA-Policy *Implicit Activation* kann die Aktivierung von CORBA-Objekten gesteuert werden. Folgende Einstellungen sind möglich:

– IMPLICIT_ACTIVATION: Implizite Erzeugung bzw. Aktivierung durch den POA.
– NO_IMPLICIT_ACTIVATION: Keine implizite Erzeugung, d.h. der Server muss Objekte explizit aktivieren. Dies muss also programmiert werden.

Darüber hinaus verfügt CORBA über einen speziellen Lifecycle Service, mit dem man CORBA-Objekte erzeugen, löschen, kopieren und verschieben kann. Über diesen Service kann der Lebenszyklus von Serverbausteinen über eine Factory

gesteuert werden. Allerdings hat dieser Service nie praktische Relevanz erlangt und soll daher auch nicht weiter erläutert werden.

Basisdienste

CORBA stellt mit den *CORBAservices* eine Fülle von Dienstspezifikationen zur Verfügung, die von Herstellern zum Teil implementiert werden müssen. Insgesamt sind mehr als 15 dieser Services spezifiziert. Sie stellen Basismechanismen bereit, die für die Entwicklung verteilter Anwendungen notwendig sind. Als Beispiele sind hier der Naming-Service, ein Event-Service, ein Security-Service, ein Externalization-Service, ein Transaction-Service, ein Security-Service, ein Concurrency-Control-Service, der Lifecycle-Service usw. zu nennen. Jeder Service stellt Schnittstellen bereit, die über IDL spezifiziert sind. Auch über die *CORBAfacilities* werden die bereits genannten Standarddienste angeboten. Allerdings haben die meisten Dienste keine praktische Relevanz erlangt. Eine Ausnahme bildet der CORBA-Transaktionsdienst, der in Kapitel 4 noch erläutert wird.

Sonstige Unterscheidungsmerkmale

Aufgrund der vielfältigen POA-Einstellmöglichkeiten (Policies) kann für CORBA-basierte Anwendungen eine gute Skalierbarkeit erreicht werden. Der Naming-Service kann von einem CORBA-Hersteller so umgesetzt werden, dass er auch einen Beitrag zur Erhöhung der Verfügbarkeit eines CORBA-basierten Systems leisten kann, was man etwa durch Clustering-Unterstützung erreichen kann. Lösungen hierfür sind allerdings nicht standardisiert, sondern den CORBA-Implementierungen überlassen.

2.3.3 Fallbeispiel Java-RMI

Java-RMI (Remote Method Invocation, im Weiteren kurz als RMI bezeichnet) kann auch als Weiterentwicklung des Remote Procedure Call (RPC) aufgefasst werden. Während RPC-Mechanismen eine prozedurale Notation vorweisen, ist Java-RMI eine Weiterentwicklung in Richtung einer objektorientierten RPC-Variante. Bei Java-RMI werden im Gegensatz zu RPC keine Einzelfunktionen aufgerufen, sondern entfernte Methoden von Remote-Objekten. RMI (Remote Method Invocation) ist nur in der Sprache Java und zwar im Package *java.rmi* verfügbar.

Architektur

Die RMI-Architektur (Abbildung 2-30) sieht vor, dass sowohl der RMI-Client, als auch der RMI-Server in einer eigenen Java Virtual Machine (JVM) ablaufen. Damit ist eine einheitliche Ablaufumgebung gewährleistet. Ein RMI-Client verwendet für den Zugriff auf ein Serverobjekt einen Proxy, der die eigentliche Kommunikation und auch das Marshalling/Unmarshalling ausführt. Ein RMI-Client sendet den Methodenaufruf an einen RMI-Server, der in einer eigenen JVM einen oder mehrere Serverbausteine als Java-Objekte beherbergt.

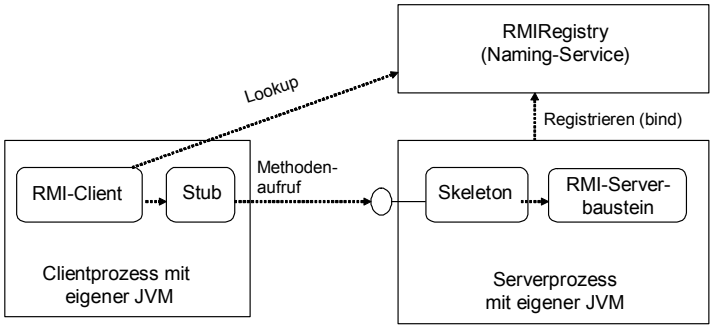


Abbildung 2-30: Java-RMI Architekturüberblick

Ein Methodenaufruf wendet sich an den lokalen Stub. Der Stub versendet dann den serialisierten Methodenaufruf samt den zugehörigen Parametern an das Serverobjekt, erwartet das Ergebnis und gibt es an den Client weiter. Die Nachrichtenübermittlung kann aufgrund der einheitlichen Umgebung im Client und im Server in einem Java-Objektstrom erfolgen. Das Marshalling/Unmarshalling ist damit bereits durch die Sprachumgebung vorgegeben.

Auf der Server-Seite wird der Aufruf von einem Skeleton-Objekt empfangen, das eine Deserialisierung der Parameter vornimmt und den Aufruf an das adressierte Serverobjekt (instanziierter Serverbaustein) weitergibt. Nach der Bearbeitung durch den eigentlichen Methodencode wird das Ergebnis über den gleichen Weg ebenfalls serialisiert zurück an den Client gesendet.

RMI stellt für die Programmierung von Serverbausteinen spezielle Klassen im Package *java.rmi.server* zur Verfügung. Diese sind vereinfacht im Klassenmodell aus Abbildung 2-31 dargestellt. Die Klasse *RemoteObject* wird von der Standard-Java-Objektklasse *Object* abgeleitet und enthält u.a. eine spezielle Methode *equals()* zum Vergleich von entfernten Objekten. Die Klasse *Remote-Stub* dient als Basis für die Client-Stubs. Der Anwendungsprogrammierer muss sich um diese Klasse nicht kümmern. Die abstrakte Klasse *RemoteServer* erweitert die Klasse *RemoteObject* und dient als Basis für die Implementierung eines Remote-Servers. Ein spezieller Remote-Server ist in der Klasse *UnicastRemoteObject* definiert. Diese Klasse wird üblicherweise zur Programmierung eigener Server verwendet, indem man den eigenen Server von *UnicastRemoteObject* ableitet.

Der Anwendungsentwickler sieht also im Wesentlichen nur die Klasse *UnicastRemoteObject*, weil er seine Serverobjektklasse davon ableiten muss, und das Interface *Remote*, das seine Serverobjektklasse implementieren muss. Die Klasse *UnicastRemoteObject* stellt ein Rahmenwerk für entfernte Objekte bereit, dessen Transportmechanismen auf TCP basieren. RMI-Server und -Client bauen zur Kommunikati-

on also auf TCP-Verbindungen auf, was jedoch für den Programmierer transparent bleibt.

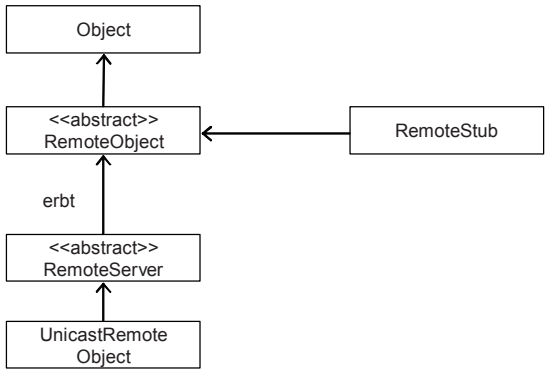


Abbildung 2-31: RMI-Klasse *UnicastRemoteObject* mit Vererbungshierarchie

Erwähnenswert ist noch, dass die Objektklasse *UnicastRemoteObject* ein *nicht replizierbares* entferntes Objekt definiert, dessen Objektreferenz auch nur innerhalb des Serverprozesses gültig ist, in dem es erstmalig konstruiert wurde. „Nicht replizierbar" bedeutet also, dass ein RMI-Serverobjekt nicht zur Laufzeit auf einen anderen Serverprozess und schon gar nicht auf einen anderen Serverrechner verlegt (migriert) werden oder mehrmals im Netz laufen kann.

Bei Fehlern in der Ausführung einer entfernten Methode wird vom Server eine *RemoteException* geworfen. Es können wie bei lokalen Methoden auch eigene Exceptions definiert werden. Der Client erhält eine Exception mit der Response-Nachricht.

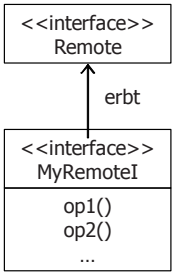


Abbildung 2-32: RMI-Remote-Interface definieren

Auf Basis dieser Klassen und Interfaces kann man nun einen eigenen Dienst bzw. dessen Interface definieren. Ein Remote-Interface für ein entferntes Serverobjekt wird vom Interface *Remote* abgeleitet (siehe Abbildung 2-32). Im Interface werden

die Methoden, so wie es in Java üblich ist, definiert. Die Implementierung des Remote-Interface entspricht der Implementierung eines lokalen Interface mit dem einen Unterschied, dass die Implementierungsklasse von der Klasse *UnicastRemoteObject* erbt.

Die Zusammenhänge zwischen Klassen und Interfaces zur Implementierung eines eigenen RMI-Objekts sind nochmals in Abbildung 2-33 zusammengefasst. Dabei gilt:

- *MyRemoteI*: Beliebiger Name für eigenes Remote Interface
- *MyRemote*: Eigene Implementierungsklasse
- *UnicastRemoteObject*: Standard-RMI-Klasse, von dem alle eigenen Remote-Objekte erben müssen
- *Remote*: Standard-RMI-Interface, von dem alle Remote-Interfaces erben müssen

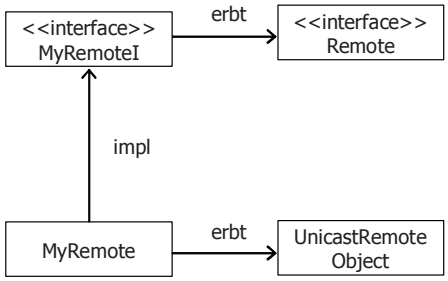


Abbildung 2-33: Entwicklung eines eigenen RMI-Objekts

Zustandsverwaltung

In Java-RMI wird nicht zwischen zustandslosen und zustandsbehafteten Serverbausteinen unterschieden. Auch Singletons werden nicht explizit unterstützt. Das bedeutet, dass der Programmierer selbst für die Zustandsverwaltung zuständig ist.

Parameterübergabe

Einfache Datentypen (Elementartypen wie *int* und *long*) werden bei Java-RMI an den RMI-Proxy als Werte übergeben (Call-by-value).

Die Übergabe von Java-Objekten als Parameter erfolgt lokal zwar im Client zwischen Client-Objekt und RMI-Stub über Referenzen, für die Übertragung werden die Objekte aber in den RMI-Strom kopiert und damit ebenfalls als Wertparameter (*Call-by-value*) übergeben. Man bezeichnet das Verfahren – wie bereits oben ausgeführt - auch mit *Call-by-copy*.

Die Übergabe von Referenzen auf Remote-Objekte erfolgt als Referenzparameter (*Call-by-reference*).

Dienstschnittstellen

Die Beschreibung der Schnittstelle eines verteilten Objektes geschieht bei Java-RMI direkt in der Programmiersprache. Es gibt keine sprachneutrale Möglichkeit wie dies in CORBA durch die IDL sichergestellt ist. Die Schnittstelle eines verteilten Objekts wird als Java-Interface beschrieben, das nur das in Java-RMI vorgegebene *Remote*-Interface beerben muss. Der Unterschied zu einem lokalen Interface ist also nicht groß, aber es ist dennoch einer vorhanden. Unterschiedlich ist auch, dass man an der Schnittstelle spezielle Exceptions (*RemoteException*) behandeln muss. Für den Programmierer ist die Nutzung verteilter Objekte also sichtbar und er muss im Vergleich zur Nutzung einer lokalen Schnittstelle das Exception-Handling anders vorsehen, wenn er die Schnittstelle eines entfernten Objekts nutzt.

Die Schnittstellenbeschreibung kann alle Möglichkeiten von Java vorsehen. Es gilt das Java-Typkonzept. Da die JVM ein eigenes Typsystem hat, das völlig unabhängig von der konkreten Hardware-Architektur ist, ist eine spezielle Umwandlung der Nachrichten in eine Transfersyntax nicht erforderlich. Bedingung für den Austausch von Objekten in Argumenten und Returnwerten ist lediglich, dass diese gemäß der Java-Objektserialisierung serialisierbar sein müssen. Beispiele für die Beschreibung von Schnittstellen verteilter Objekte sind im folgenden Absatz skizziert.

In neueren RMI-Versionen wird nun auch RMI/IIOP (RMI über IIOP) unterstützt. Mit dieser Variante kann man die Schnittstellenbeschreibung im RMI-Stil durchführen und auf Basis der RMI-Schnittstellendefinitionen Code generieren, der die Kommunikation über IIOP durchführt. Hierzu wird ein Compiler (*rmic*[6]) verwendet, der die Java-Datentypen auf CORBA-IDL-Typen abbildet. Die Nutzung dieser Möglichkeit ist dann sinnvoll, wenn man RMI-Programme entwickeln muss, die mit CORBA-Objekten kommunizieren sollen. Bei RMI/IIOP sind aber einige Restriktionen zu beachten, die sich aus der Abbildung der Java-Datentypen auf CORBA-IDL-Datentypen ergeben.

Ab der Java Standard Edition 5 (J2SE 5) ist der rmic-Compiler nicht mehr notwendig, da eine dynamische Generierung von Stub und Skeleton durchgeführt werden kann.

Die Remote-Aufrufe eines Clients sehen fast genauso aus wie lokale Methodenaufrufe mit dem Unterschied, dass wegen Netzwerkfehlern zusätzlich eine spezielle Ausnahme (*RemoteException*) erzeugt werden kann.

---

[6] Den Compiler *rmic* muss man mit der Option „-iiop“ aufrufen, um die Generierung von CORBA-Code zu bewirken. Die Option „-idl“ erzeugt aus den angegebenen RMI-Schnittstellendefinitionen darüberhinaus auch eine IDL-Notation. Umgekehrt gibt es bei RMI auch einen Compiler namens *idlj*, der aus einer IDL-Notation eine Java-RMI-Schnittstellenbeschreibung generiert. Diese Mechanismen gibt es seit J2SE V1.3.

Beispiel „Interface einer RMI-basierten Artikelverwaltung“: Die folgende Interface-Definition für einen einfachen RMI-basierten Artikelmanager soll die Nutzung der vorhandenen Java-Klassen aufzeigen:

```
package articlemanagement.server;
import articlemanagement.*;
import java.rmi.*;
import java.util.Vector;
public interface ArticleMgrRemoteI extends Remote
{
    public void createArticle(long katalogId)
        throws ArticleMgrExistsException, RemoteException;
    public void createArticle(Article a)
        throws ArticleMgrExistsException, RemoteException;
    public void updateArticle(long katalogId,
        String beschreibung, float preis, long warengruppe)
        throws ArticleMgrNotFoundException, RemoteException;
    public Article findArticleByKatalogId(long katalogId)
        throws ArticleMgrNotFoundException,
        ArticleMgrParamException, RemoteException;
    public Vector findArticlesByGroup(long warengruppe)
        throws ArticleMgrNotFoundException,
        ArticleMgrParamException, RemoteException;
    public Vector findAllArticles()
        throws ArticleMgrNotFoundException,
        ArticleMgrParamException, RemoteException;
    public void deleteArticle(long katalogId)
        throws ArticleMgrNotFoundException,
        ArticleMgrParamException, RemoteException;
}
```

Das Interface *ArticleMgrRemoteI* erbt vom Remote-Interface und beschreibt alle Operationen, die über den Artikelmanager ausgeführt werden können.

In einer Implementierungsklasse sind alle Methoden zu programmieren. Die im Folgenden skizzierte Klasse *ArticleMgrRemote* erbt von *UnicastRemoteObject* und implementiert das RMI-Interface *ArticleMgrRemoteI*. Die Artikel können ohne Veränderung der Schnittstelle z.B. im Filesystem oder in einer Datenbank persistent gespeichert werden. Die Klasse *Article* beschreibt einen Artikel und ist für das Beispiel nicht weiter von Interesse. Der Java-Code für die speziellen Exceptions ist aus Vereinfachungsgründen nicht aufgeführt.

```
package articlemanagement.server;
import articlemanagement.*;
import java.rmi.*;
import java.rmi.server.*;
```

```
…
public class ArticleMgrRemote
      extends UnicastRemoteObject implements ArticleMgrRemoteI
{
      ...
      // Konstruktor
      public ArticleMgrRemote()
          throws RemoteException
      {
          …
      }
      public void createArticle(long katalogId)
          throws ArticleMgrExistsException, RemoteException
      {
          …
      }
      public void createArticle(Article a)
          throws ArticleMgrExistsException, RemoteException
      {
          …
      }
      public void updateArticle(long katalogId,
          String beschreibung, float preis, long warengruppe)
          throws ArticleMgrNotFoundException, RemoteException
      {
          …
      }
      public Article findArticleByKatalogId(long katalogId)
          throws ArticleMgrNotFoundException,
          ArticleMgrParamException, RemoteException
      {
          System.out.println("start: findArticleByKatalogId");
          …
      }
      public Vector findArticlesByGroup(long warengruppe)
          throws ArticleMgrNotFoundException,
          ArticleMgrParamException, RemoteException
      {
          …
      }
      public Vector findAllArticles()
          throws ArticleMgrNotFoundException,
          ArticleMgrParamException, RemoteException
```

```
    {
        …
    }
    public void deleteArticle(long katalogId)
         throws ArticleMgrNotFoundException,
        ArticleMgrParamException, RemoteException
    {
        …
    }
}
```

Über den RMI-Compiler *rmic* (ab J2SE 5 dynamisch) werden Client-Stub und Server-Skeleton mit folgenden Dateinamen generiert:

- ArticleMgrRemote_Stub.class
- ArticleMgrRemote_Skel.class

rmic ist ab Java Version 1.5 nicht mehr erforderlich. Die Stubs und Skeletons werden dann dynamisch erzeugt.

Der im Folgenden skizzierte Client legt einen neuen Artikel an und löscht anschließend einen anderen Artikel.

```
…
import java.rmi.* ;
import articlemanagement.*;
public class ArticleMgrClient
{
    // Artikelmanager-Interface
    private static ArticleMgrRemoteI aMgr = null;
    public static void main (String args[]) throws IOException
    {
        // Lookup zum besorgen de Objektreferenz hier nur angedeutet
        // Im Lookup-Aufruf die RMI-URL des Servers mit Hostname,
        // Port und Servicename angeben
        aMgr = (ArticleMgrRemote) =
              java.rmi.Naming.lookuk(rmi://hostname/AreticleMgr");
        // Katalog-Id, Beschreibung, Warengruppe und Preis einlesen und
        // Artikel anlegen. Die Erfassung der Artikelattribute ist
        // im Beispiel nicht programmiert.
        Article a = new Article();
        try {
              aMgr.createArticle(a);
        }
        catch (ArticleMgrExistsException e)
        {
```

```
                System.out.println("Artikel bereits vorhanden");
            }
            // Weitere Katalog-Id einlesen und Artikel löschen
              try
              {
                  aMgr.deleteArticle(katalogId);
            }
            catch (ArticleMgrNotFoundException e)
            {
                  …
            }
            catch (Exception e)
            {
                  …
            }
            …
        }
}
```

Marshalling und Unmarshalling

In Abbildung 2-34 wird das Marshalling bei RMI vereinfacht dargestellt. Man sieht, dass man auf Client- und auf Serverseite durchaus komplexe Objektstrukturen aufbauen kann, die als Parameter bzw. Rückgabewerte über den Objektstrom gesendet werden können. Als Programmierer muss man sich hierum nicht kümmern, da RMI die Objekte selbstständig serialisiert.

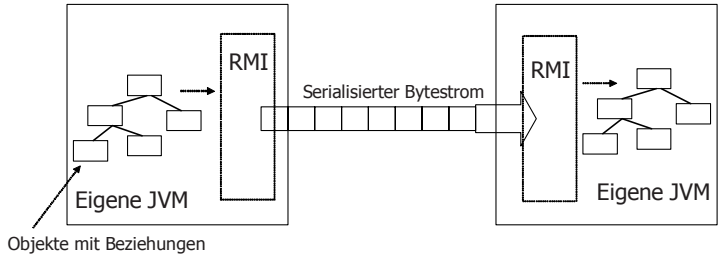


Abbildung 2-34: Serialisierung bei RMI

Die Serialisierung bei RMI ist einfach und leistungsfähig, was sich aus der einheitlichen Nutzung einer virtuellen Maschine ergibt, die eine übergreifende Präsentation aller Datentypen vorgibt. Die übertragenen Objekte sind hinsichtlich ihrer lokalen Präsentation aufgrund der JVM immer gleich aufgebaut und müssen damit nicht transformiert, sondern zur Übertragung nur in eine flache Struktur gebracht

werden. Dabei spielt es auch keine Rolle, auf welchen Rechnerarchitekturen die Client- und Serveranwendungen laufen. Die Präsentation der Datentypen in der JVM ist immer gleich. Für die Übertragung wird ein Java-Objektstrom verwendet, was bedeutet, dass alle übertragenen Objekte das Interface *Serializable* implementieren müssen.

Adressierung und Kommunikation

Namensauflösung und Naming-/Directory-Service: Die RMI-Registry ist ein sehr einfacher Namensdienst. Sie wird im Package *java.rmi.Naming* oder im Package *java.rmi.Registry* bereitgestellt. Mit den Klassen und Interfaces dieses Packages kann man ein Registry anlegen und auch einen eigenen Naming-Service implementieren, wenn man die Standard-Registry nicht für geeignet hält. Jeder Eintrag in der Registry hat einen Namen und eine Objektreferenz. Der Namensraum ist nicht wie bei anderen Naming-Services hierarchisch geordnet, sondern flach.

Wie bei Sun RPC's Portmapper wird auf jedem Serverrechner ein Hintergrundprozess mit dem Namen *rmiregistry* gestartet. Der Prozess belegt standardmäßig den TCP-Port 1099. *rmiregistry* verwaltet die Objektreferenzen aller Serverobjekte, die auf dem Serverrechner laufen. Eine Instanz der Registry, also ein *rmiregistry*-Prozess, kann mehrere RMI-Server und deren Objekte verwalten.

Die Adressierung von Serverdiensten erfolgt über URL-Adressen (Uniform Resource Locator). Der Client holt sich zunächst vom Naming-Server eine *Objektreferenz*, um ein entferntes Objekt zu adressieren. Das URL-Schema sieht wie folgt aus:

```
rmi://<server>;<port>/<object> mit rmi:// als Protokolltyp
```

Eine RMI-Objektreferenz hat das Schema <endpoint(IP-Adresse, TCP-Port); Objekt-Id>, was bedeutet, dass ein Serverobjekt zunächst durch die Transportadresse und innerhalb eines Serverprozesses mit einer eindeutigen Objekt-Id adressiert wird.

Das initiale Auffinden des RMI-Registries erfolgt beim Lookup-Aufruf, da die Adresse der Registry aus der Server-URL abgeleitet werden kann. Die zuständige Registry läuft auf dem gleichen Serverrecher. Dem Programmierer muss also der Server, auf dem die adressierten Serverbausteine ablaufen, bekannt sein.

Die öffentlich zugänglichen Methoden für das Binding und die Kontexterzeugung und damit für den Zugriff auf die Registry bzw. zur Aktualisierung der Registry sind in der Klasse *java.rmi.Naming* definiert. Folgende Methoden sind von Interesse:

- *bind*: Mit dieser Methode bindet ein RMI-Server einen Namen über einen URL an ein entferntes Objekt. Nach dem erfolgreichen Binden des Namens an das Objekt ist dem Objekt eine eindeutige Objektreferenz zugeordnet und ein Client kann sich diese besorgen.
- *rebind*: Mit dieser Methode kann einem bereits gebundenen Objekt ein neuer Name zugeordnet werden.

- *unbind*: Diese Methode dient dem Entfernen eines Objekts aus der Registry.
- *lookup*: Diese Methode dient dem Client dazu, mit Hilfe eines URL eine Objektreferenz aus einer Registry zu lesen.

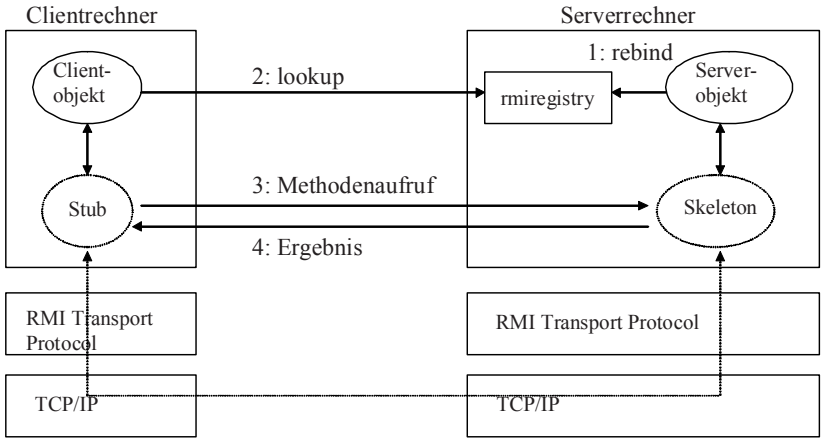


Abbildung 2-35: Ablauf der Kommunikation mit RMI-Registry

Ein Serverobjekt registriert sich lokal über einen *rebind*- oder *bind*-Aufruf bei der RMI-Registry. Ein Client ruft zur Ermittlung der Objektreferenz eine *lookup*-Methode auf und kann anschließend mit dieser die gewünschten Methodenaufrufe ausführen. Beim Lookup ist der vollständige Name des URL des Serverobjekts anzugeben. In Abbildung 2-35 ist der Zusammenhang nochmals dargestellt.

Kommunikationsprotokoll: Die Kommunikation erfolgt in Java-RMI grundsätzlich synchron mit einer unterstützen *At-Most-Once-* Fehlersemantik. Benötigt man asynchrone Aufrufe, lässt sich das nur über Threads oder über eigene, ereignisgesteuerte Callback-Methoden bewerkstelligen. Der Client muss dann auch ein Remote-Interface zur Verfügung stellen und ist in einer Zwitterstellung (siehe weiter unten). Aufrufe desselben Clients werden im Server sequentiell ausgeführt.

RMI unterstützt die Kommunikation über zwei Protokolle, die beide auf TCP/IP basieren. Es handelt sich um CORBA/IIOP und um das RMI Transport Protokoll, das auch als *JRMP* bekannt ist. Wir wollen den Protokollstack für das *RMI Transport Protocol* etwas näher betrachten. In Abbildung 2-36 sind zunächst die verschiedenen Layer dargestellt. Stub und Skeleton kommunizieren über dieses Protokoll, das der Anwendungsschicht zugeordnet ist. Die *Remote Reference Layer* ist ebenfalls der Anwendungsschicht zugeordnet und enthält die Adressierungsmechanismen für die Objekt-Adressierung.

Die *Remote Reference Layer* soll grundsätzlich verschiedene Protokolle zum Auffinden von Serverobjekten unterstützen und ist daher generisch konzipiert. Derzeit ist jedoch nur eine *Point-to-Point*-Kommunikation zwischen Client und Server imple-

mentiert und in der RMI-Klasse *UnicastRemoteObject* realisiert. Zukünftig sind auch andere Protokolle denkbar. Im Transportsystem wird derzeit TCP in Schicht 4, IP in Schicht 3 und - wie es im Internet üblich ist - ein beliebiger Netzwerkzugang genutzt.

Das RMI-Transport-Protokoll ist ein streamorientiertes Protokoll. Eine RMI-Instanz verwaltet je Verbindung einen Input- und einen Output-Socket-Stream. Beide Streams werden paarweise verwaltet. Jeder Output-Stream hat einen korrespondierenden Input-Stream.

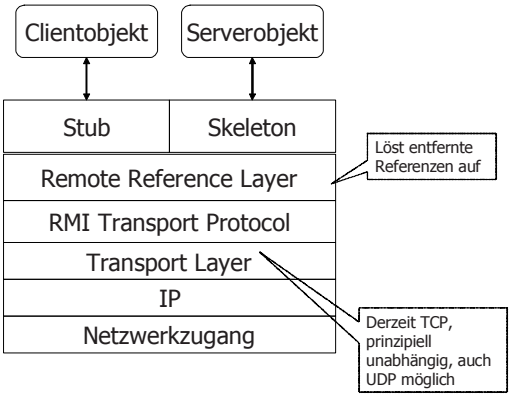


Abbildung 2-36: RMI-Protokollstapel

Die Abbildung 2-37 zeigt den RMI-Transport-Header, der nicht über den RMI-Objektserialisierungsmechanismus serialisiert wird. Nach einem *Version*- und einem *Protocol*-Feld werden die Nutzdaten übertragen. RMI unterstützt verschiedene Protokollvarianten. Die Nutzdaten-Nachrichten sind mit einem speziellen Header versehen. Im Feld *Protocol* ist angegeben, um welche Protokollvariante es sich handelt. Möglich sind hier:

- *StreamProtocol*: Dies ist ein streamorientiertes Protokoll für eine einzige Transportverbindung. Es können mehrere Nachrichten nach einem Header gesendet werden.
- *SingleOpProtocol*: Dieser Protokolltyp erlaubt das Senden einer einzigen Nachricht (Message) nach dem Header und wird für die Einbettung von RMI-Nachrichten in HTTP verwendet.
- *MultiplexProtocol*: Dieser Protokolltyp dient der Nutzung einer *realen* Transportverbindung (Socket-Verbindung) für das Multiplexieren mehrerer logischer RMI-Verbindungen. Für dieses Protokoll sind einige Zusatzmechanismen zum logischen Auf- und Abbau von Verbindungen und zum Senden von Requests für die verschiedenen logischen Verbindungen erforderlich. In der Spezifikation sind die Operationen *OPEN*, *CLOSE*, *CLOSEACK*, *REQUEST*, und *TRANSMIT* definiert.

Alle Protokolltypen sind in der RMI-Spezifikation beschrieben (WWW-005). Wir betrachten im Weiteren hauptsächlich das Stream-Protokoll. RMI unterscheidet aus der Sicht eines Clients zwischen ausgehenden (out) und ankommenden (in) Nachrichten. Es sind drei ausgehende RMI-Nachrichtentypen spezifiziert:

- *Call*: Normaler Request für einen Methodenaufruf
- *Ping*: Nachricht zur Überprüfung auf der Transportebene, ob eine Partner-JVM noch aktiv ist
- *DgcAck*: Nachricht zum Senden von Garbage-Collection-Informationen vom Client zum Server, wenn ein Client ein entferntes Objekt über einen Returnwert erhält

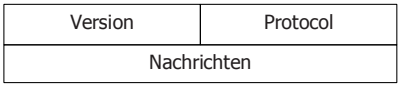


Abbildung 2-37: RMI Transport-Header

Es sind auch u.a. die folgenden ankommenden Nachrichten definiert:

- *ReturnData*: Nachricht für das Ergebnis eines Methodenaufrufs
- *PingAck*: Diese Nachricht enthält eine Antwort auf eine Ping-Nachricht.

Der Inhalt des Message-Teils ist in Abbildung 2-38 skizziert, wobei zwischen Request und Response unterschieden wird.

Eine Request-Nachricht enthält folgende Felder:

- *Message-Type*: In diesem Feld ist der Message-Typ kodiert (Call, Ping, DgcAck).
- *OID*: Objekt-Identifikation des adressierten Objekts
- *Operation:* Dies ist eine Nummer, welche die adressierte Servermethode identifiziert.
- *Hash Number:* Ein Hashcode, welcher der Verifikation von Stub- und Skeletons dient. Es wird damit überprüft, ob Stub und Skeleton zusammenpassen.
- (*Argument, Value),...:* Parameter und Parameterwert, der beim Aufruf übergeben wird. Hier ist eine Liste zugelassen. Die Argumente werden entsprechend der Java-Objektserialisierung serialisiert.

Eine Response-Nachricht enthält folgende Felder:

- *UniqueIdentifier:* Eindeutige Identifikation des Returnwerts oder bei negativem Ablauf der Exception. Diese Identifikation wird in einer folgenden *DgcAck*-Nachricht an den Server genutzt, um diesem bekanntzugeben, dass ein RMI-Objekt über den Returnwert an den RMI-Client übertragen wurde. Mit dieser Information kann der RMI-Server den entsprechenden Referenzzähler für das Garbage-Collection erhöhen.
- *Return Value:* Rückgabewert oder Wert der Exception

Der typische Ablauf einer Kommunikation zwischen RMI-Client und -Server erfordert zunächst einen TCP-Dreiwege-Verbindungsaufbau (Mandl 2008b), den der Client initiiert. Anschließend wird vom Client eine Request-Message gesendet, die vom Server bestätigt wird.

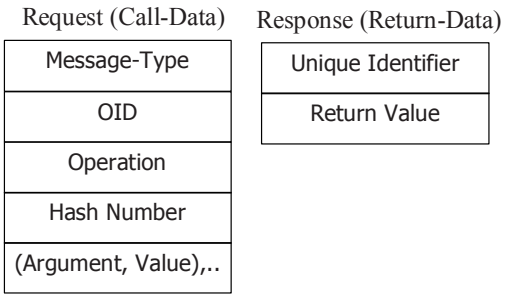


Abbildung 2-38: RMI Message-Inhalt (Output-Stream)

Die Antwort des Servers mit dem Ergebnis der Requestbearbeitung wird in einer Response-Message gesendet, die bei Bedarf vom Client nochmals mit einer *DgcAck*-Nachricht beantwortet wird. Die *DgcAck*-Nachricht wird vom Client nur gesendet, wenn im Returnwert der Response-Nachricht ein Objekt übertragen wurde, das bis dahin vom Client nicht genutzt wurde. Als Parameter wird die Object-Id (Unique Identifier) des empfangenen RMI-Objekts übergeben. Der Server kann daraufhin den zugehörigen Referenzzähler für das RMI-Objekt hochzählen (siehe auch Garbage-Collection). Eine dedizierte PDU für ein explizites Reduzieren des Referenzzählers im Server gibt es nicht. Wenn also ein Client ein Objekt nicht mehr benötigt, bemerkt der Server dies nicht.

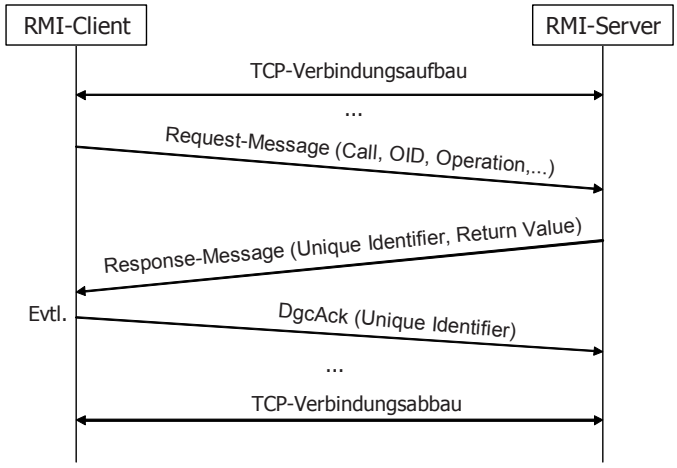


Abbildung 2-39: Typischer Ablauf einer RMI-Kommunikation

Der typische Ablauf einer RMI-Kommunikation ist in Abbildung 2-39 skizziert. Eine Verbindung kann für mehrere Requests verwendet werden. Es ist nicht festgelegt, wer diese wieder abbaut, also dürfen es vermutlich beide Partner. Üblicherweise wird der Client den Verbindungsabbau initiieren, wenn er die Dienste des Servers nicht mehr benötigt.

Mit *Ping*- und *PingAck*-Nachrichten können sich Client und Server gegenseitig überwachen (siehe Abbildung 2-40). In der RMI-Spezifikation ist allerdings nicht festgelegt, in welchen Zeitabständen eine Überwachung stattfindet und welche Konsequenzen sich daraus ergeben, wenn ein *PingAck* nicht rechtzeitig ankommt. Dies ist vermutlich einer RMI-Implementierung überlassen.

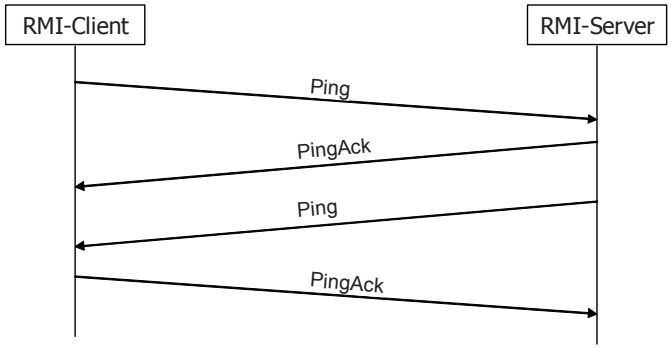


Abbildung 2-40: Lebendüberwachung zwischen RMI-Client und RMI-Server

Garbage-Collection

Die verteilte Speicherbereinigung oder auch *Garbage-Collection* erfolgt bei Java RMI auf Basis eines *Reference-Counting-Algorithmus* in Verbindung mit *Leases*, der im Laufzeitsystem (der JVM) integriert ist. Der Garbage-Collection-Mechanismus von RMI basiert auf dem Modulo-3-Algorithmus von Birell (Birell 1993). Das RMI-Laufzeitsystem führt Buch über alle lebenden Objektreferenzen. Wenn eine Objektreferenz eine Client-JVM zum ersten Mal „betritt", wird ein Zähler für diese Objektreferenz angelegt. Gleichzeitig wird die oben genannte DgcAck-PDU an den Server gesendet, damit der Server seinen Referenzzähler erhöhen kann. Auf der Serverseite werden nur die Client-JVMs gezählt, die ein Serverobjekt referenzieren.

Wenn die Verbindung zu einem Client abgebaut wird, kann im Server der Referenzzähler für alle von diesem Client benutzten Serverobjekte reduziert werden. Für diese Aufgabe implementiert jedes RMI-Objekt das Interface *java.rmi.server.Unreferenced*, das nur die Methode *unreferenced* enthält. Das RMI-Laufzeitsystem ruft diese Methode auf, wenn kein RMI-Client mehr das Objekt referenziert. Das Ereignis, über welches das RMI-Laufzeitsystem dies bemerkt, ist in der Spezifikation nicht festgelegt. Er ist vermutlich der Abbau der Transportverbindung. Wenn ein Referenzzähler eines Serverobjekts auf 0 steht und keine sonstige lokale Refe-

renz mehr auf das Objekt zeigt, kann man das Objekt beim nächsten Lauf entfernen.

Wenn zwischen Client- und Serverrechner ein Netzwerkproblem vorhanden ist, so dass sie nicht mehr miteinander kommunizieren können, kann es vorkommen, dass der Server das Objekt freigibt, obwohl der Client es noch nutzt. Somit ist keine referenzielle Integrität sichergestellt. Ist das Serverobjekt nicht mehr vorhanden, wenn ein Client eine Methode aufruft, so erhält dieser eine *RemoteException*.

Nebenläufigkeit

Die Parallelität der RMI-Server wird durch die Implementierung geregelt. Typischerweise wird für die Abwickung des Aufrufs im Server in der Regel ein eigener Thread verwendet (meist aus einem Threadpool), der den Methodenaufruf ausführt und das Ergebnis an den Client zurücksendet. Dies ist aber in der RMI-Spezifikation nicht zwingend vorgeschrieben. Die RMI-Spezifikation (Sun 2004b) überlässt die Implementierung vielmehr dem Hersteller. Es muss also kein eigener Thread sein und man kann sich auch nicht darauf verlassen. Daher muss der Programmierer eines RMI-Objekts dafür sorgen, dass der Code *thread-sicher* ist. In Abbildung 2-41 ist ein typischer Methodenaufruf eines Clients an einen Server dargestellt.

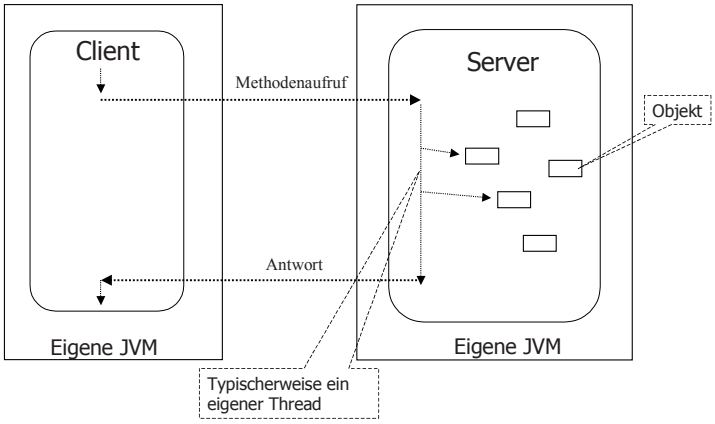


Abbildung 2-41: RMI-Client und -Server in eigenen JVMs

Beispiel „Server zur Artikelverwaltung“: In der unten skizzierten *main*-Methode wird eine Instanz des Artikelmanagers erzeugt und über die Methode *rebind* des RMI-Naming-Services wird das Remote-Objekt registriert. In einer praktisch sinnvollen Anwendung wird natürlich meist nicht „localhost“ als Serveradresse verwendet.

```
…
// Hier wird der eigentliche RMI-Server gestartet und das
// Remote-Objekt im RMI-Registry eingetragen.
public static void main(String args[])
{
    try {
      ArticleMgrRemote amgr = new ArticleMgrRemote();
      // Name des Service angeben und einer konkreten
      // Serverbaustein-Instanz zuordnen
      Naming.rebind("rmi://localhost/articlemgr", server, amgr);
      System.out.println("ArticleMgr gestartet");
    }
    catch(Exception e) {
      System.out.println("Fehler bei Start");
    }
}
```

Die weitere Abwicklung übernimmt das RMS-Laufzeitsystem und kann nicht beeinflusst werden.

Lebenszyklus von Serverbausteinen

Die RMI-Objekte werden in einem RMI-Server in der Regel zum Startzeitpunkt instanziiert und existieren so lange, bis der Server sie freigibt. Allerdings legt die RMI-Spezifikation dies nicht fest und überlässt es dem Hersteller.

Es wird auch eine Instanzierung festgelegt, die als *dynamische Serveraktivierung* bzw. *Lacy Activation* bezeichnet wird. Bei der dynamischen Serveraktivierung wird das Starten eines Servers mit einem Verwaltungsprozess namens *rmid* dynamisch bei Bedarf möglich. Ein Server registriert in diesem Fall seine Objekte nicht bei der RMI-Registry sondern beim *rmid,* der dann bei einem Request eine eigene JVM startet, um diesen abzuarbeiten.

Die Instanzierung und die Aktivierung werden in der Terminologie der einzelnen Lösungsansätze nicht immer eindeutig getrennt. Die Aktivierung gemäß unserer Definition, also die konkrete Zuordnung einer Serverbaustein-Instanz zu einem Client, erfolgt zum Zeitpunkt der Verbindungsaufnahme während der Lookup-Phase. In der Regel, aber eben doch implementierungsabhängig, wird eine RMI-Serverbaustein-Instanz in einem eigenen Serverthread ausgeführt.

Basisdienste

Java-RMI bietet neben dem einfachen Naming-Service keine weiteren Basisdienste für verteilte Anwendungen an.

Sonstige Unterscheidungsmerkmale

Bei Java-RMI gibt es keine Basismechanismen, die eine Skalierung von Servern unterstützen und auch keine Möglichkeiten, hochverfügbare Lösungen zu realisieren. Dies muss über eigene Mechanismen erfolgen.

Als besondere Merkmale sollen die Konzepte für das dynamische Laden von Klassen und die Möglichkeit, einen Callback-Mechanismus zu simulieren, erwähnt werden:

Dynamic Class Loading: Eine Besonderheit bei Java-RMI ist das *Dynamic Class Loading*. Java bietet hier generell die Möglichkeit, Softwarebausteine in Form von Klassen, Interfaces und ebenso Stub-Klassen und auch Klassen, die als Parameter und Returnwerte benötigt werden, von einem entfernten Rechner in eine lokale JVM zu laden. Hierzu muss eine URL bekanntgemacht werden, über die die Klassen und Interfaces angefordert werden können. Das bedeutet, dass man Software dynamisch zum Ablauf bringen kann, ohne diese vorher lokal zu installieren. Diese Aufgabe übernimmt der sog. RMI-ClassLoader. RMI stellt hierfür die Klasse *RMIClassLoader*, die dem Herunterladen von Klassen über das Netzwerk dient, bereit. Die Adresse, unter der man entfernte Softwarebausteine zum Herunterladen bereitstellt, wird als *Codebase* bezeichnet und ist vergleichbar mit dem lokalen Java-Klassenpfad (CLASSPATH-Environment-Variable), nur dass die Codebase einen Pfad auf ein entferntes (oder auch lokales) Verzeichnis angibt. Der Klassenpfad ist also im Prinzip eine lokale Codebase. Die Reihenfolge des Durchsuchens nach Klassen über den CLASSPATH und den URL wird aus der Angabe in *java.rmi.server.codebase* gesteuert.

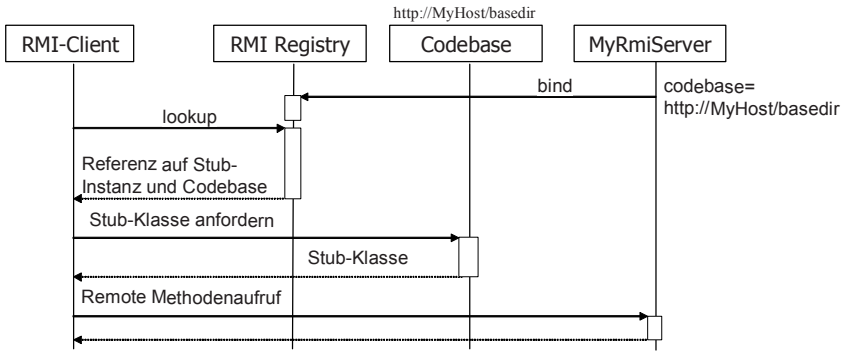


Abbildung 2-42: Nutzung der Codebase für das Klassenladen bei RMI

Um die *Codebase* bekanntzumachen, gibt es bei RMI eine Eigenschaft (Property), die mit dem URL oder einer Liste von URLs belegt werden kann. Diese Eigenschaft wird mit *java.rmi.server.codebase* bezeichnet und wird im Server und zwar in der Regel beim Start über Startoptionen angegeben. Sie wird bei der Registrierung des

Servers auch der RMI-Registry bekanntgegeben. Bei entfernten Objekten zeigt der URL üblicherweise auf einen über die Protokolle HTTP oder FTP erreichbaren Pfad. Die Zusammenhänge beim Laden von Java-Klassen über die Codebase sind in Abbildung 2-42 dargestellt.

Falls die benötigte Klasse im lokalen Klassenpfad liegt, wird diese immer bevorzugt von dort geladen. Ist dies nicht der Fall, wird die Codebase des entfernten Objekts durchsucht. Wenn ein Client eine Referenz auf ein entferntes Objekt erhält, wird ihm eine Stub-Klasse zugeschickt, von der er eine Instanz erzeugt. Diese Instanz dient als Proxy (Stub) des Serverobjekts.

Die *Codebase* kann auch von einem entfernten Objekt verwendet werden, wenn der RMI-Client eine Instanz einer Klasse als Parameter mitgibt, die das entfernte Objekt nicht kennt, also auch nicht in seinem Klassenpfad hat. Im Normalfall kommt das zwar nicht vor, da die Parameter auf der Serverseite bekannt sind, aber ein Client kann auch einen Subtyp eines in der Methodendeklaration festgelegten Parametertyps übergeben. Dieser Subtyp kann im Server nicht bekannt sein. Für diesen Fall muss der Client mit dem Methodenaufruf eine Codebase-Information an das entfernte Objekt mitgeben, damit dieses die Klasse des Parameters nachladen kann. Dieser Fall ist in Abbildung 2-43 skizziert. Diese Codebase wird als *client-spezifizierte* Codebase bezeichnet.

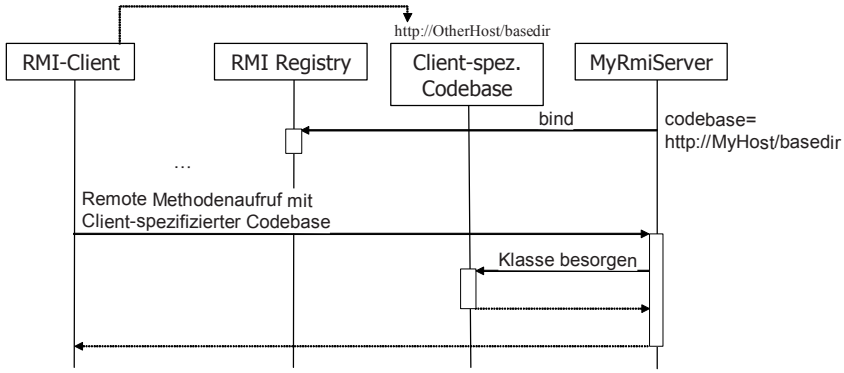


Abbildung 2-43: Nutzung der Codebase für das serverseitige Klassenladen bei RMI

Ein interessanter Aspekt, der noch erwähnt werden soll, ist das Sicherheitskonzept von RMI, das auf dem Java Sandbox-Modell basiert. Der *RMISecurityManager* wird von der JVM bereitgestellt. Er verhindert das Ausführen unerlaubter Operationen durch Klassen, die vom *RMIClassLoader* über das Netz geladen wurden. Diese Klassen gelten nicht als vertrauenswürdig. Vor der Java-Version 2 musste der RMI-Security-Manager zur Laufzeit installiert werden (mit Methode *System.setSecurityManager*). Auch das Überschreiben der Security-Politik durch die Definition eige-

ner Security-Manager ist möglich. Seit Java-Version 2 ist der Security-Mechanismus implizit in der JVM vorhanden.

Callback-Mechanismus im Client: In RMI ist auch eine ereignisgesteuerte Verarbeitung realisierbar. Diese Variante eignet sich für Anwendungen, bei denen ein Server bei bestimmten Ereignissen einen oder mehrere Clients informieren möchte. Wie wir bei der Betrachtung des Message-Passing-Modells noch sehen werden, handelt es sich in diesem Fall um die Realisierung einer *Publish-Subscribe*-Aufgabenstellung. Ein Server (Publisher) informiert einen oder mehrere Interessenten (Subscriber) über das Eintreffen bestimmter Ereignisse. Die Clients müssen sich vorher beim Server registrieren.

Ein Client kann zur Lösung dieser Aufgabenstellung mit RMI eine Callback-Methode festlegen (z.B *onMessage()*), die er selbst in einem eigenen Remote-Interface (abgeleitet vom Interface *Remote*) definiert. Eine Instanz des implementierten Remote-Interface (abgeleitet von *UnicastRemoteObject*) muss er über ein weiteres Remote-Interface des Servers sozusagen als *Listener* registrieren, indem er die Referenz auf das Remote-Interface des Clients als Parameter über einen Methodenaufruf an der entfernten Schnittstelle (z.B *registerListener())* an den Server übergibt. Der Server kann dann bei Eintreten des definierten Ereignisses die Methode *onMessage* (anderer Methodenname möglich) des Clients aufrufen. Was in dieser Methode gemacht wird, obliegt dem Programmierer der Anwendung.

Diese Möglichkeit hat eine Umkehrung der Client-Server-Beziehung zur Folge. Der Client wird in diesem Fall auch zum Server. Die Interfaces und Methoden können mit RMI-Mitteln beliebig benannt werden. Eine konkretere Vorgabe gibt es hierzu nicht. Ein Beispiel für diese Nutzungsvariante von RMI ist in (Abts 2003) zu finden.

2.3.4 Fallbeispiel .NET Remoting

.NET Remoting ist ein grundlegender Mechanismus für die Kommunikation verteilter Objekte in der .NET-Technologie von Microsoft. Ähnliche Mechanismen waren in Microsofts DCOM verfügbar. .NET Remoting ist vom Grundkonzept und der angebotenen Funktionalität her zu Java-RMI sehr ähnlich.

Architektur

.NET Remoting setzt auf die .NET-Plattform und damit auf dessen Laufzeitsystem, die *Common Language Runtime* (*CLR*) auf. Client und Server laufen meist, aber nicht zwingend in einem eigenen Betriebssystemprozess und innerhalb des Prozesses in einer *Application Domain* (siehe unten) ab. Client und Server sind über einen Kommunikationskanal verbunden. Da für das Verständnis von .NET Remoting grundlegende .NET-Kenntnisse erforderlich sind, wird vorab ein kurzer Überblick über dieses, noch stark in Entwicklung befindliche Framework gegeben. In Abbildung 2-44 ist die Einbettung von Client- und Serveranwendungen in die .NET-Laufzeitumgebung skizziert.

.NET-Framework-Begriffe: Die Laufzeitumgebung CLR ist eine virtuelle Maschine. Sie führt einen vom Compiler bzw. von den Compilern der unterstützten Programmiersprachen erzeugten Zwischencode aus, der als *Common Intermediate Language* (*CIL*) bezeichnet wird. Die Umwandlung in den Maschinencode erfolgt erst zur Laufzeit (Just-in-Time). Alle Microsoft .NET-Sprachen (Visual C#, Visual J#[7], Visual Basic,...) werden unterstützt. In vordefinierten Klassenbibliotheken wird Basisfunktionalität bereitgehalten. Zudem gibt es verschiedene Werkzeuge zur Entwicklung und für das Deployment. Es werden vorwiegend die neueren Windows-Derivate als Ablaufumgebung verwendet.

Die Laufzeitumgebung ist in Abbildung 2-45 mit ihren Einzelkomponenten skizziert. Sie enthält wie die JVM Mechanismen für das dynamische Laden von Klassen, Sicherheits-Features, Exception-Handling, einen Just-in-Time-Compiler sowie einen Garbage-Collector. Ebenso sind eine Thread-Unterstützung und eine Unterstützung für sog. virtuelle Prozesse vorhanden.

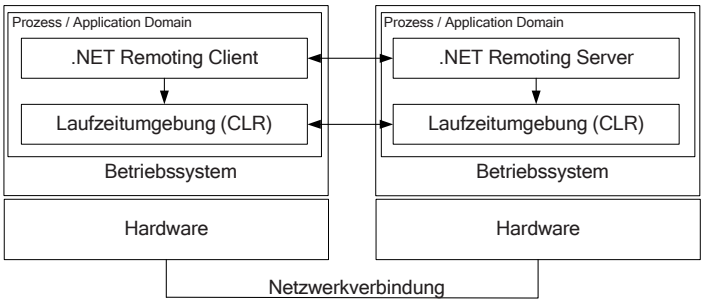


Abbildung 2-44:.NET-Remoting-Umgebung nach (Kuhrman 2004)

Der Code, der unter der CLR abläuft, wird als *managed* Code bezeichnet. Dies ist ein Code von Softwarebausteinen, die von der CLR verwaltet werden. Diese Bausteine enthalten Metainformationen zur Beschreibung der enthaltenen Klassen und den CIL-Code, der zur Laufzeit auf Maschinencode abgebildet wird. Ein oder mehrere verwaltete Bausteine werden zusammen mit dem CIL-Code und den Metadaten als *Assembly* bezeichnet. Eine Assembly enthält zudem auch ein sog. Manifest, das alle wichtigen Informationen zur Assembly enthält (Versionsinformation, Referenzen auf andere Assemblies, exportierte Typen usw.). Es lassen sich unter .NET Framework auch sog. *shared Assemblies* einrichten, die in einem *globalen Assembly Cache (GAC)* registriert werden müssen. Diese sind dann systemweit verfügbar. Assemblies sind ablauffähige Programme oder Dynamic Link Libraries (DLLs), die verteilbar sind.

[7] Visual J# ist die Java-Variante von Microsoft und in einigen Teilen nicht kompatibel zu Java.

Auch die Einbindung von sog. *unmanaged* Code, also Codeteilen, die nicht unter der CLR ablaufen (z.B. bestehende COM-/DCOM-Anwendungen), ist möglich.

Interessant ist vor allem das spezielle Prozesskonzept. .NET unterstützt sog. *Applikationsdomänen* (Application Domains). Dies sind Ablaufumgebungen für virtuelle Prozesse. Innerhalb eines Prozesses können mehrere Applikationsdomänen existieren (siehe Abbildung 2-46), die wiederum mehrere *Kontexte* beherbergen.

Ein Prozess enthält mindestens eine Applikationsdomäne. Man kann eine Applikationsdomäne als *virtuellen Prozess* bzw. *Subprozess* innerhalb eines Betriebssystemprozesses bezeichnen. Existieren mehrere Applikationsdomänen in einem Betriebssystemprozess, so sind diese untereinander isoliert. Jede Applikationsdomäne enthält mindestens einen Kontext. Auch Kontexte sind untereinander isoliert.

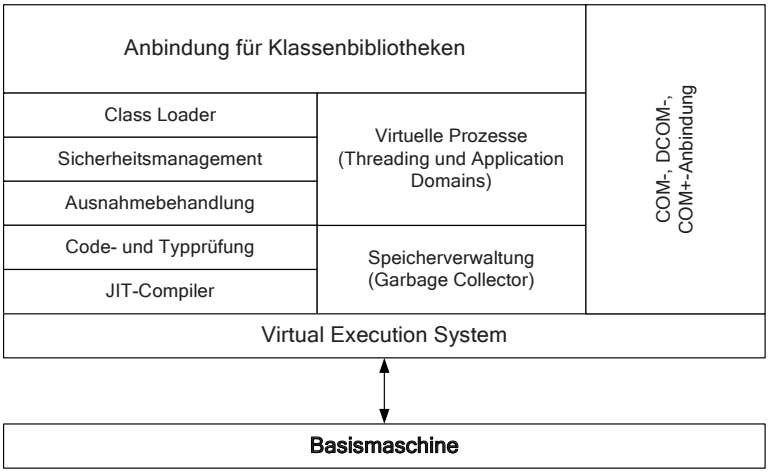


Abbildung 2-45: Laufzeitumgebung CLR nach (Kuhrmann 2004)

In einem Kontext ist eine Gruppe von Objekten zusammengefasst, die gleiche Anforderungen an die Laufzeitumgebung hat. Der Kontext wird auch als *Lebensraum* für die Objektgruppe bezeichnet. Der Zugriff auf einen Kontext erfolgt über ein Proxy-Objekt, das auch als *transparenter Proxy* bezeichnet wird. Um von einem Kontext auf einen anderen zuzugreifen, nutzt man .NET Remoting, wobei es keine Rolle spielt, ob die Kontexte in der gleichen Applikationsdomäne liegen oder nicht oder mehrere Betriebssystemprozesse beteiligt sind, die über Rechnergrenzen hinweg verteilt sind.

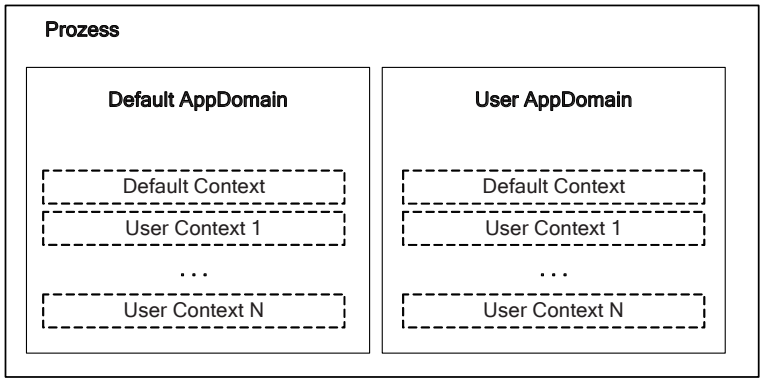


Abbildung 2-46: .NET Remoting Applikationsdomänen und Kontexte

Einem Kontext können Eigenschaften (Attribute) zugeordnet werden. Ein Kontext ist also demnach eine durch gemeinsame Eigenschaften gebildete Gruppe von Objekten. Ein Beispiel für ein Kontextattribut ist das Transaktionsattribut. Es gibt an, ob der Kontext transaktional ist, d.h. ob man Transaktionen darin ausführen kann.

Programmierung: Die wichtigsten .NET-Remoting-Klassen sind im Namespace mit der Bezeichnung *System.Runtime.Remoting* definiert. Eine Klasse, die verteilte Objekte repräsentieren soll, erbt von der Basisklasse *MarshalByRefObject* und implementiert mindestens ein Remote-Interface. Ein statisches Generieren von Stub und Skeleton ist nicht erforderlich. Diese werden zur Laufzeit bei der Aktivierung eines Serverbausteins erzeugt.

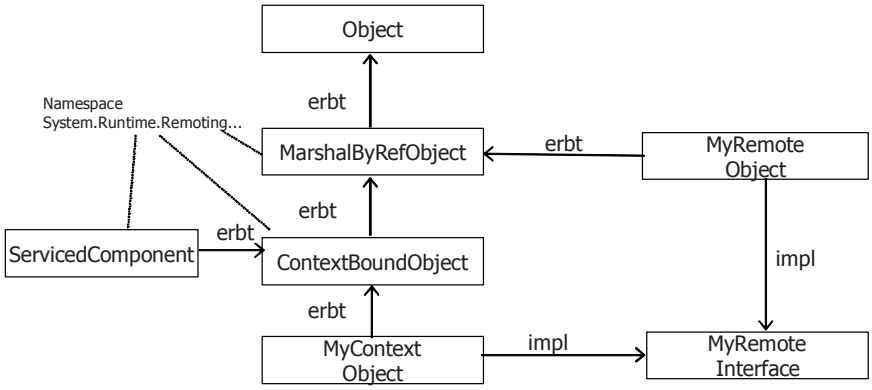


Abbildung 2-47: Wichtige Klassen von .NET Remoting und deren Nutzung

Die Vorgehensweise bei der Programmierung sieht etwa wie folgt aus:

- Man entwickelt zunächst das Interface, welches das verteilte System bereitstellen soll (in Abbildung 2-47 als *MyRemoteInterface* benannt).
- Serverseitig ist dann die Objektklasse zu realisieren, die die Methoden aus dem Interface implementiert. Diese Klasse (in Abbildung 2-47 als *MyRemoteObject* benannt) erbt von *MarshalByRefObject.*
- Neben der eigentlichen Klasse für das Remote-Objekt ist noch ein *Server-Startup-Code* zu implementieren, der das Remote-Objekt registriert.
- Anschließend kann ein Clientprogramm entwickelt werden.

Im Serverprogramm muss zunächst ein Kommunikationskanal eingerichtet und registriert werden. Eine vordefinierte Klasse namens *ChannelServices* im Namespace *System.Runtime.Remoting* stellt Dienste bereit, um einen Kommunikationskanal zu registrieren. Die verwendete Methode heißt *RegisterChannel*. Anschließend muss noch eine Registrierung des Dienstes mit Hilfe eines vordefinierten Objekts namens *RemotingConfiguration* erfolgen. Hier wird auch angegeben, wie die Aktivierung auszusehen hat. Ein Client, der die Dienste eines entfernten Objekts nutzen möchte, legt ebenfalls einen Kommunikationskanal an und registriert ihn.

Zustandsverwaltung

Man bezeichnet Objekte, die über Referenzen angesprochen werden, in .NET auch als Referenzobjekte. Sie erben von der .NET-Basisklasse *MarshalByRefObject* und sind nur über einen Proxy (Client-Stub) erreichbar. Instanziiert werden diese Objekte in den Adressräumen von Serverprozessen. Sie verfügen über Remote-Schnittstellen und können nur über eine Objektreferenz genutzt werden. Eine Objektreferenz wird auf der Clientseite immer durch einen Proxy verwaltet.

Serveraktivierte Objekte (SAO) sind Objekte, deren Lebensdauer direkt vom Server gesteuert wird. Fordert ein Client eine Instanz eines serveraktivierten Objekts an, wird in der Anwendungsdomäne des Clients ein Proxy erstellt. Es gibt zwei sog. Aktivierungsmodi für serveraktivierte Objekte:

- *Singletons*: Wie der Name sagt, wird bei diesem Aktivierungsmodus im Server nur eine Instanz des verteilten Objekts angelegt. Ein Singleton wird beim ersten Aufruf einer Methode instanziiert und kann von beliebig vielen Clients verwendet (aktiviert) werden.
- *SingleCall*-Objekte sind verteilte Objekte, bei denen für jeden Request eine eigene Instanz verwendet wird. SingleCall-Instanzen vom gleichen Typ sind untereinander isoliert. SingleCall-Objekte sind zustandslose Serverbausteine.

Clientaktivierte Objekte (CAO) sind dagegen Objekte, deren Lebensdauer von der aufrufenden Anwendungsdomäne (also vom Client) gesteuert wird. Die Objekte liegen aber nicht wirklich im Client. Nach der Registrierung und einem klassischen Aufruf des *new*-Operators im Client wird eine Konstruktion und damit Instanzierung des Serverbausteins im Server durchgeführt. Der Client erhält eine Objektreferenz auf das CAO, die nur er sieht, das CAO residiert aber im Server. CAOs sind

statusbehaftete Serverbausteine. Jeder Client erhält eine eigene Instanz des verteilten Serverbausteins. Die Lebensdauer eines CAO richtet sich nach der Lebensdauer des Clients, kann aber die Lebensdauer des Servers, in dem es residiert, nicht überdauern.

Für den Serverprogrammierer spielt es fast keine Rolle, ob er SAOs oder CAOs anlegt. Die Programmierung ist hier ähnlich. Lediglich bei der Registrierung im Server-Startup-Code sind leichte Unterschiede vorhanden. SAOs werden über die Objektklasse *RemotingConfiguration* mit der Methode *RegisterWellKnownServiceType*, CAOs werden über die Methode *RegisterActivatedServiceType* registriert.

Parameterübergabe

Werteobjekte bei einfachen Typen werden in .NET Remoting über *Call-by-value* übertragen. Objektreferenzen werden über *Call-by-reference* (genauer: *Call-by-copy/copy-back*) übermittelt.

Dienstschnittstellen

Bei .NET Remoting wird, wie bei Java-RMI, das Interface des Objekts direkt in der Hostsprache (z.B. C#) definiert. Ein Remote-Interface unterscheidet sich nicht von anderen Interfaces und wird erst durch eine Implementierung als Remote-Objekt abgeleitet von der Basisklasse *MarshalByRefObject* bekanntgemacht.

Beispiel: Das einfache Beispiel für ein Artikelmanager-Interface sieht folgendermaßen aus:

```
namespace ArticleManagementServer {
  public interface IArticleMgr{
    void createArticle(long katalogId);[8]
    void createArticle(Article a);
    void updateArticle(long katalogId, String beschreibung,
      float preis, long warengruppe);
    Article findArticleByKatalogId(long katalogId,
      ArrayList findArticlesByGroup(long warengruppe);
    ArrayList findAllArticles();
    void deleteArticle(long katalogId);
  }
}
```

Im Folgenden wird nur der Rahmen für das Remote-Objekt angegeben, in dem die Implementierung der Methoden stattfindet.

```
using System.Runtime.Remoting;
…
```

[8] Exceptions werden im Gegensatz zu Java in C# bei der Methodenbeschreibung nicht angegeben.

```
// Basisklassen zur Artikelverwaltung, die hier nicht betrachtet werden …
using ArticleManagementBase;
namespace ArticleManagementServer {
      public class ArticleMgr : MarshalByRefObject,IArticleMgr {
          public ArticleMgr() { … } // Konstruktor
          public void createArticle(long KatalogId) { … }
          public void createArticle(Article a) { … }
          public void updateArticle(long KatalogId,
           string Beschreibung, float Preis, long Warengruppe) { … }
          public Article findArticleByKatalogId(long KatalogId) { … }
          public ArrayList findArticlesByGroup(long Warengruppe) { … }
          public ArrayList findAllArticles() { … }
          public void deleteArticle(long KatalogId) { … }
      }
}
```

Marshalling und Unmarshalling

Die Argumente und Rückgabewerte der Methoden eines Remote-Objekts müssen serialisierbar, d.h. sie müssen entsprechend gekennzeichnet sein (Attribut „Serializable“). Der Client-Proxy und das Server-Skeleton werden implizit zur Laufzeit aus Metadaten generiert und sind für das Marshalling und Unmarshalling verantwortlich.

Adressierung und Kommunikation

Namensauflösung und Naming-/Directory-Service: Die Registrierung erfolgt durch einen Server über das .NET-Framework. Im Server laufen die .NET-Remote-Objekte innerhalb eines Kontextes in einer Applikationsdomäne ab. Der Server registriert für den Zugriff auf die Objekte eigene Kommunikationskanäle. Der Client wendet sich über die vorgegebene *Activator*-Klasse an den zuständigen Server, um eine Objekttreferenz zu ermitteln.

Kommunikation: NET Remoting ist so konzipiert, dass es auf allen Transportsystemen implementiert werden kann. Die Kommunikation wird über. sog. Kanäle (Channels) zwischen zwei Kommunikationsendpunkten ausgeführt. Die Endpunkte der Kommunikation werden als *Transportsenken* (*Transport Sinks*) bezeichnet. Sog. *Formatierungssenken* (*Formatter Sinks*) übernehmen an beiden Endpunkten die Abbildung der Parameter und Rückgabewerte sowie das Einfügen in den Datenstrom bzw. das Herauslesen aus dem Datenstrom in die lokale Syntax. Mehrere *Senken* können für spezielle Verarbeitungen auch hintereinandergeschaltet werden. Eigene Senken können ergänzt werden.

Das Zusammenspiel zwischen den einzelnen Bausteinen und die Anordnung der *Formatter* und *Senken* ist in Abbildung 2-48 dargestellt. Der Proxy gibt einen Methodenaufruf bzw. dessen Argumente an einen Formatter, der daraus eine Nach-

richt erzeugt, die über den Transportkanal gesendet wird. Auf der Empfängerseite wird nach Empfang der Nachricht am Transportkanal ein Dispatcher bemüht, der den Methodenaufruf schließlich an das adressierte verteilte Objekt weitergibt.

Es gibt mehrere Kanaltypen, die im .NET-Framework verfügbar sind und beim Aufbau der Kommunikation angelegt und registriert werden. Client und Server müssen beide die gleichen Kanaltypen anlegen, um zusammenarbeiten zu können, ein Remote-Objekt kann aber auch mehrere Kanaltypen unterstützen. Man unterscheidet:

- Transport über HTTP (HTTP-Kanal)
- Transport über TCP (TCP-Kanal)
- Lokaler Transport über IPC (auf Basis von *Named Pipes* realisiert)

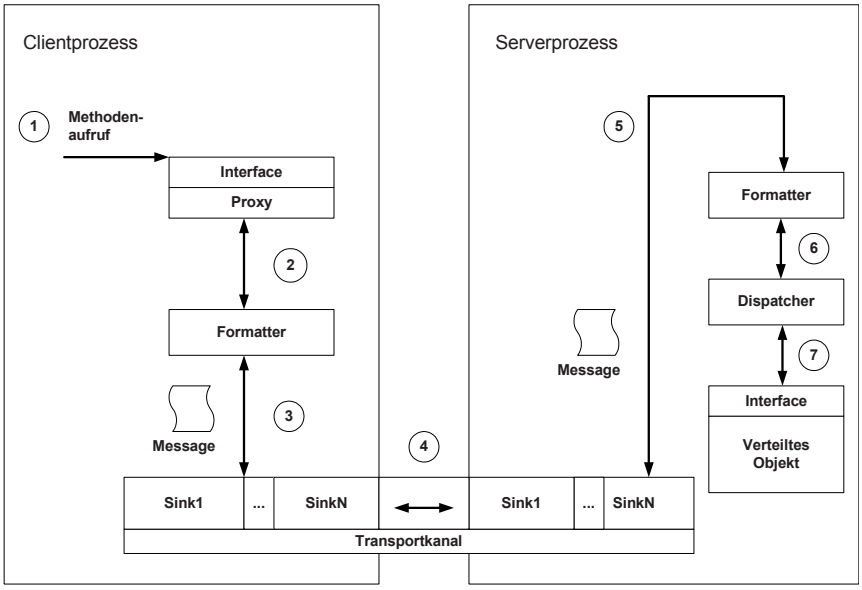


Abbildung 2-48: .NET Remoting Formatter und Kanäle nach (Kuhrmann 2004)

Der HTTP-Kanal verwendet standardmäßig eine SOAP-Formatierungssenke. Der TCP-Kanal verwendet eine Formatierungssenke, die einen binären Datenstrom serialisierter Objekte erzeugt. Auf das Kommunikationsprotokoll und auf den Nachrichtenaufbau der .NET Remoting-Nachrichten soll an dieser Stelle nicht weiter eingegangen werden (siehe WWW-006).

Als Fehlersemantik für die Kommunikation wird *At-Most-Once* unterstützt. Die Kommunikation zwischen Client und Server ist synchron (blockierend), es wird in .NET Remoting aber auch ein *asynchrones Kommunikationsmodell* unterstützt. Zudem wird ein *Callback*-Mechanismus bereitgestellt, der es einem Client ermög-

licht, bei Aufruf einer Servermethode nicht zu blockieren. Stattdessen wird beim Aufruf der Servermethode ein Callback-Objekt (Rückrufobjekt) übergeben, von dem bei Ankunft des Ergebnisses durch das Laufzeitsystem eine Methode aufgerufen wird. Diese Mechanismen werden hier nicht weiter betrachtet (WWW-06).

Garbage-Collection

Bei der ersten Nutzung eines entfernten Objekts erhält der Proxy eine Objektreferenz. Objektreferenzen sind in .NET nur für eine bestimmte Zeit gültig und müssen vom Client auf Basis eines Lease-Mechanismus immer wieder ausgeliehen werden. Diese Aufgabe übernimmt der Proxy. Es wird kein Referenzzähler (Reference Counting) für entfernte Objekte verwendet. In DCOM, das die Basis für .NET darstellt, wird aber ein Reference-Counting-Mechanismus eingesetzt.

Die Leases sind mit einer Zeitbegrenzung (TTL-Angabe = Time To Live) versehen, eine Verlängerung der Leases wird durch einen erneuten Methodenaufruf veranlasst. Ein Lease kann auch explizit verlängert werden. Dies wird als Sponsoring bezeichnet (Schwichtenberg 2007).

Nebenläufigkeit

Nebenläufigkeit der Serverbausteine wird durch das .NET-Remoting-Laufzeitsystem unterstützt, eine explizite Programmierung nebenläufiger Server ist nicht möglich. CAO-Instanzen sind Clients fest zugeordnet. Jeder Client erhält seine eigene CAO-Instanz durch das .NET-Framework. Beim Singlecall-Mechanismus (SAOs) werden ebenfalls vom Server mehrere SAO-Instanzen erzeugt. Singletons werden nur einmal instanziiert. Parallele Methodenaufrufe werden aber in nebenläufigen Threads ausgeführt, wobei keine Thread-Sicherheit beim Zugriff auf gemeinsame Datenbereiche unterstützt wird. Darum muss sich der Programmierer selbst kümmern.

Lebenszyklus von Serverbausteinen

Wie bereits erläutert, unterscheidet man bei .NET Remoting die clientseitige und die serverseitige Aktivierung, und bezeichnet die entsprechenden Objekte als *Client Activated Objects (CAO)* und *Server Activated Objects (SAO)*, wobei Instanzierung und Aktivierung nicht eindeutig voneinander getrennt sind.

Die vordefinierte Objektklasse *Activator* dient z.B. der Aktivierung eines SAOs (SingleCall-SAO). Wann die tatsächliche Instanzierung erfolgt, wird nicht offengelegt und ist der .NET-Implementierung überlassen. Folgende Beispiele sollen die Aktivierung näher beschreiben.

Beispiel-Aktivierung über Activator-Klasse: Unser Client wird in diesem Fall unter dem Namespace *ArticleManagementClient* programmiert. Zunächst wird ein HTTP-Kanal (siehe Adressierung und Kommunikation) angelegt und registriert. Anschließend wird das Objekt über die Klasse *Activator* aktiviert.

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using ArticleManagementBase;
using ArticleManagementServer;

namespace ArticleManagementClient
{
    class ArticleMgrClient
    {
      // Interface des Artikelmanagers
      static ArticleMgrI aMgr;
      static void Main (string[] args)
      {
        // Hostnamen erfassen
        string host = Console.ReadLine();
        // Adresse des Servers
        string url = "http://" + host + ":1234/ArticleMgr";
        // Kanal erzeugen und registrieren
        HttpChannel chnl = new HttpChannel();
        ChannelServices.RegisterChannel(chnl);
        // Erzeugen eines Proxy für den Zugriff auf einen
        // Artikelverwalter
        aMgr = (ArticleMgrI)
        // Aktivieren des Serverobjekts
        Activator.GetObject(typeof(ArticleMgrI), url);
        // Nutzung des entfernten Objekts …
        …
      }
    }
}
```

Beispiel für eine Singleton-Aktivierung: Eine Singleton-Aktivierung erfolgt im .NET-Remoting-Server, was am Beispiel eines Artikelverwalters grob skizziert wird. Das Singleton wird im folgenden Codestück instanziiert und registriert. Der Service ist über einen HTTP-Kanal erreichbar.

```
…
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using ArticleManagementBase; // Basisklassen für Artikel
namespace ArtikelManagementServer
{
```

```
public class ServerStartup
{
    public static void Main()
    {
      try {
        // ArtikelManager für Remoting als Singleton mit
        // einem HTTP-Channel deklarieren
        HttpChannel chnl = new HttpChannel(1234);
        ChannelServices.RegisterChannel(chnl);
        // Objekt als Singleton registrieren
        RemotingConfiguration.RegisterWellKnownServiceType
            (typeof(ArticleMgr), /* Typ des Objekts */
            "ArticleMgr", /* URI des Objekts */
            WellKnownObjectMode.Singleton); /* Aktivierungs-Modus */
            Console.WriteLine("ArticleMgr gestartet");
      }
      catch(Exception) {
        Console.WriteLine("Fehler bei Start");
      }
    }
}
```

Basisdienste

.NET Remoting bietet außer einem impliziten Naming-Service keine weiteren Dienste für eine verteilte Programmierung an. Darüber hinausgehende Dienste sind in den .NET Enterprise Services (siehe unten) zu finden.

Sonstige Unterscheidungsmerkmale

.NET Remoting ist eine recht einfach zu nutzende, objektbasierte Variante für eine Client-Server-Lösung, die nicht für hochskalierbare und hochverfügbare Anwendungen konzipiert wurde. Hierfür sind .NET Enterprise Services oder ältere Microsoft-Lösungen (COM+, DCOM) zu verwenden.

Die Zukunft von .NET Remoting ist aus heutiger Sicht aber unsicher, da Microsoft ab .NET V3.0 die *Windows Communication Foundation (WCF*) in den Vordergrund stellt. Die Kommunikationsmechanismen nutzen in erster Linie Webservices.

2.3.5 Zusammenfassung

In diesem Abschnitt wurden die Client-Server-Technologien CORBA, Java-RMI und .NET Remoting beschrieben. Dies sind im Prinzip Weiterentwicklungen von RPC. Grundsätzlich kann man sagen, dass die grundlegenden Mechanismen recht ähnlich sind, sich allerdings auch an einigen Stellen wie etwa der Beschreibung von Dienstschnittstellen unterscheiden. CORBA ist ein sprachunabhängiger Stan-

dard, ONC RPC arbeitet mit der Sprache C, Java-RMI funktioniert nur mit Java und .NET Remoting wird heute vorwiegend unter Windows-Betriebssystemen, dafür aber mit mehreren Microsoft-Sprachen unterstützt.

Ein Fehler der frühen Nutzung des Konzepts der verteilten Objekte war, dass Leistungsaspekte nicht richtig betrachtet wurden. Es funktioniert nicht so ohne weiteres, dass in einem verteilten System jedes Objekt auch mit all seinen Methoden so wie ein lokales Objekt aus der Ferne benutzt werden kann. Es liegen Netzwerkverbindungen zwischen den beteiligten Rechnersystemen, die es erfordern, auf die Netzbelastung und auf Verzögerungen zu achten und diese im Design verteilter Anwendungen zu berücksichtigen.

CORBA wird heute nur noch in heterogenen Umgebungen eingesetzt, oft auch „nur“ zur Entwicklung der Infrastruktur von Komponentensystemen, also im Verborgenen. In der Praxis ist CORBA heute eine Basistechnologie zur Implementierung von Application-Servern geworden. Der ursprüngliche Ansatz, CORBA als direkt zu nutzendes Werkzeug für die Anwendungsentwickler verteilter, objektbasierter Anwendungen zu platzieren, verlor aufgrund der schnellen Entwicklung der Komponententechnologie in letzter Zeit an Bedeutung.

ONC RPC ist immer noch in vielen Anwendungen im Einsatz, wird jedoch in der Regel nicht mehr für neue Entwicklungen verwendet. Java-RMI wird als leichtgewichtiger Mechanismus heute gerne eingesetzt, wenn man nicht größere Lösungen mit hohen Skalierbarkeitsanforderungen bauen muss. .NET Remoting ist erst am Anfang seiner Entwicklung und evtl. auch schon wieder auf dem Abstellgleis, wenn man die .NET-Strategie von Microsoft näher betrachtet. (siehe .NET 3.0)

Wir wollen abschließend in der Tabelle 2-1 eine zusammenfassende Gegenüberstellung der betrachteten Lösungsansätze in Bezug auf die betrachteten konzeptionellen Fragestellungen durchführen.

Wie man der Tabelle entnehmen kann, bietet CORBA die meisten Dienste an und ist damit die am weitesten entwickelte Objekttechnologie. Die Services sind aber zum Teil sehr komplex und haben wie der Persistenzdienst oder der Nebenläufigkeitsdienst nie praktische Bedeutung erlangt. Der Transaktionsdienst wird in Kapitel 3 noch besprochen. .NET Remoting und Java RMI sind nahezu gleichwertig.

Man erkannte nach einer ersten Nutzungsphase von CORBA in der Praxis, dass man grobkörnigere Objekte mit speziellen Schnittstellen konzipieren musste, um in verteilten Umgebungen bestehen zu können. Auch aus dieser Erkenntnis heraus hat der Begriff der Komponenten in verteilten Systemen immer mehr an Bedeutung gewonnen. Dies führte zu einer Entwicklung von verteilten Komponentensystemen, die im Folgenden betrachtet werden.

Tabelle 2-1: Gegenüberstellung der Client-Server-Lösungen

	Technologien			
Konzept-merkmal	CORBA	Java RMI	.NET Remoting	ONC RPC
Architektur	Objektbasiert, ORB im Mittelpunkt, mächtiger POA	Objektbasiert, Basis ist die JVM	Objektbasiert, Basis ist die CLR	Prozedural
Zustands-verwaltung	Keine explizite Unterstützung	Keine explizite Unterstützung	Stateful (CAO) Stateless (Single-Call) Singelton	Keine explizite Unterstützung
Parame-terübergabe	Call-by-value Call-by-reference für IOR-Referenzen	Call-by-value Call-by-copy/copy-back bei Referenzen Call-by-reference bei Remote-Objekten	Call-by-value Call-by-copy/copy-back für lokale Refe-renzen	Call-by-value für alle Parameter und Returnwerte
Dienst-schnittstellen	CORBA IDL	Über Java-Interface	Über .NET-Sprache (Inter-face)	RPC Sprache
Marshalling und Un-marshalling	CDR	Java-Objekt-serialisierung	.NET-Objekt-serialisierung	XDR
Adressie-rung und Kommuni-kation	Naming-Service IIOP At-most-once	rmiregistry JRMP und RMI/IIOP At-most-once	.NET Naming-Service .NET Protokoll At-most-once	Portmapper TCP/UDP-basiert At-most-once (TCP) At-Least-once (UDP)
Garbage-Collection	Keine explizite Unterstützung	Reference-Counting	Leases mit max. TTL + Sponsoring	Keine explizite Unterstützung
Nebenläu-figkeit	Thread Policies im POA	Implementierungs-abhängig	Implementie-rungsabhängig	Entwicklung über Low-Level-API
Lebens-zyklus	Lifecycle Service Activation Policies	Instanzierung beim Start, Aktivierung bei Verbindungs-aufbau	Instanzierung und Aktivierung abhängig von der Activation-Strategie	Instanzierung beim Start und Aktivie-rung beim Verbin-dungsaufbau
Basisdienste	Persistency Service Transaction Service und viele andere	Keine weiteren Basisdienste	Keine weiteren Basisdienste	Authentifizierung-dienste
Sonstiges	Hohe Skalierbarkeit über POA Policies konfigurierbar	Keine hohe Skalier-barkeit und Verfüg-barkeit machbar	Keine hohe Ska-lierbarkeit und Verfügbarkeit machbar	Keine hohe Skalier-barkeit außer bei Eigenentwicklung über Low-Level-API

2.4 Verteilte Komponenten

2.4.1 Grundlegendes Modell

Im praktischen Einsatz stellte sich heraus, dass objektbasierte, verteilte Systeme relativ aufwändig zu programmieren sind (siehe CORBA). Eine Vereinfachung für den Entwickler wurde gefordert. Die Verteilung relativ kleiner Objekte stellte sich zudem als nicht leistungsfähig genug heraus, wenn man heute verfügbare Netzwerke unterstellt. Ein entferntes Objekt kann (noch) nicht wie ein lokal vorhandenes Objekt genutzt werden. Objekte sind also feingranulare Softwarebausteine, die sich nicht so gut für eine Verteilung eignen. Man denke nur an ein Objekt mit vielen Setter- und Getter-Methoden, die alle über das Netz als entfernte Methodenaufrufe ausgeführt werden. Dies führt zu schlechter Performance. Weiterhin sind verteilte Objektsysteme recht kompliziert in der Handhabung.

Definitionen

Aus diesen Überlegungen heraus, wurden die Konzepte verteilter Komponenten entwickelt. Vorreiter waren hier Microsoft mit MTS bzw. COM+ und Sun Microsystems mit der Enterprise-Java-Beans-Technologie. Soweit es noch nachzuvollziehen ist, hat sich die Java-Gemeinde ein paar Anregungen von MTS/COM+ für die EJB-Spezifikation geholt, aber auch die umgekehrten Einflüsse sind erkennbar. Beide Technologien stellten Konzepte und Ablaufumgebungen zur Realisierung verteilter Komponenten bereit. Während Sun Microsystems eine Spezifikation auf Basis von Java grundsätzlich offen entwickelte, stellte Microsoft eine geschlossene Ablaufumgebung innerhalb des Betriebssystems Windows bereit. Komponenten kapseln Objekte und stellen eine oder mehrere Schnittstellen (Fassaden) für den Zugriff bereit. Der Komponentenbegriff ist sicher nicht eindeutig und es gibt viele Definitionen. Wir betrachten die Definitionen aus der UML (Jeckle 2004) und aus (Scyperski 1997).

Definition gemäß UML: *Eine verteilte Komponente stellt eine modulare, verteilbare und ersetzbare Einheit eines Systems dar, welche ihren Inhalt kapselt und eine odere mehrere Schnittstellen nach außen zur Verfügung stellt.*

Eine Komponente wird in UML mit speziellen Symbolen (siehe Abbildung 2-49) dargestellt.

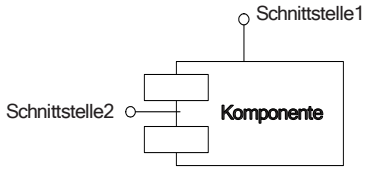


Abbildung 2-49: UML-Notation für eine Komponente

Definition nach (Scyperski 1997): *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Diese Definition macht mehr als die UML-Definition deutlich, dass hinter dem Konzept der Komponenten auch die Möglichkeit der *Komposition*, also der Zusammensetzung von Anwendungssystemen aus Komponenten, steht. Komponenten soll man kaufen und in eine eigene Anwendung integrieren können. Die Definition von Scyperski unterscheidet nicht zwischen lokalen und verteilten Komponenten, da hier auch prinzipiell die gleichen Anforderungen gelten. Verteilte Komponenten sind allerdings eher (im Sinne einer Client-Server-Beziehung) serverseitige Softwarebausteine. Vergleicht man sie mit verteilten Objekten, so kann man sagen, dass verteilte Komponenten in der Regel größere Softwarebausteine sind.

Komponentenmodelle und –systeme, Application-Server und Container

Komponentenmodelle und -systeme: Ein *Komponentenmodell* legt Regeln für die Anwendung und das Zusammenspiel von Komponenten fest. Systeme bestehend aus Komponenten, die einem Komponentenmodell folgen, werden auch als *Komponentensystem*e bezeichnet. Bekannte Komponentenmodelle sind Enterprise Java Beans (JEE/EJB), das Spring-Komponentenmodell (Wolff 2006), Jini (Haase 2001), CORBA Components (CORBA/CCM) und .NET Enterprise Services.

Application-Server: Komponenten benötigen eine Ablaufumgebung, die als Application-Server bezeichnet wird. Es gibt zwar keine eindeutige Festlegung, welche Funktionen ein Application-Server bereitstellen muss, im Prinzip handelt es sich aber um eine Weiterentwicklung der klassischen Transaktionsmonitore für verteilte Anwendungen. Application Server bieten Basisdienste, zu denen Naming- bzw. Directory-Dienste, Transaktionsdienste, Sicherheitsdienste und Persistenzdienste gehören. Während Transaktionsmonitore für die Mainframe-Welt entwickelt wurden, sind Application-Server für verteilte Anwendungssysteme konzipiert. Transaktionsmonitore unterstützen im Wesentlichen eine prozedurale Entwicklung von Transaktionsprogrammen. Application-Server nutzen die Konzepte der Objekt- und vor allem der Komponentenorientierung. Anwendungen werden in sog. Komponenten zerlegt, die über Schnittstellen nach außen verfügen sollen. Eine Komponente muss bestimmte Schnittstellen bereitstellen, die der Application-Server verwendet, um Aufrufe an die Komponente weiterzuleiten. Diese Schnittstellen werden dann über eine Netzwerkverbindung von den Clients benutzt.

Das Grundprinzip der Arbeitsweise eines Application-Servers ist in Abbildung 2-50 vereinfacht dargestellt. Die Clientanwendungen laufen, im Gegensatz zum klassischen Teilhaber- und auch Teilnehmerbetrieb, auf eigenen Rechnern mit eigenen Betriebssystemen ab. Auf den Clientrechnern muss eine entsprechende Software vorhanden sein, die den Zugang zum Application-Server ermöglicht. Im Server-Betriebssystem läuft der Application-Server ab. In diesem liegen, ähnlich

wie Transaktionsprogramme in Transaktionsmonitoren, die Softwarekomponenten der Anwendungen. Zum Ablaufzeitpunkt werden diesen Komponenten nach Bedarf Prozesse oder Threads zugeordnet, in denen der eigentliche Programmcode zum Ablauf kommt. Ein Application-Server verwaltet üblicherweise einen Pool an Prozessen bzw. Threads und ordnet diese den Komponenten dynamisch zu. Die benötigten Datenhaltungssysteme liegen meist, aber nicht zwingend, auf einem eigenen Serverrechner Die Zugriffe auf die Datenbank werden über den Application-Server kontrolliert.

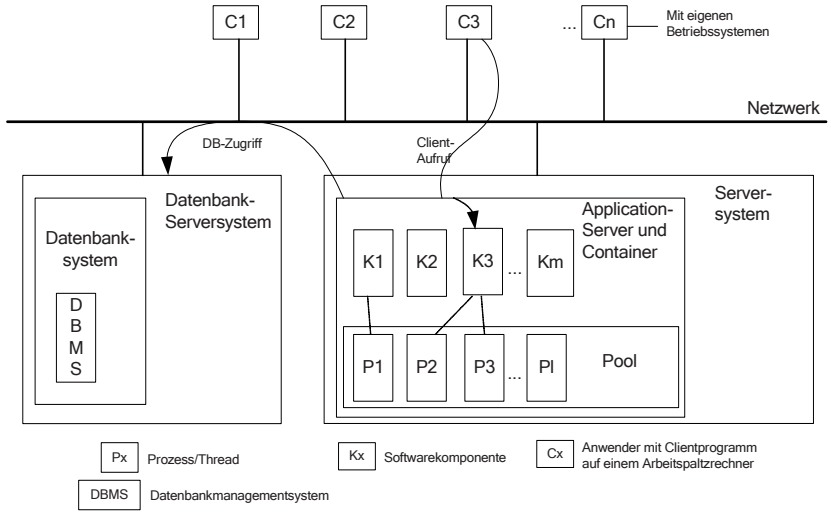


Abbildung 2-50: Architektur eines Application-Servers nach (Mandl 2008a)

Application-Server leisten wie Transaktionsmonitore umfangreiche Dienste. Sie verwalten die Komponenten, ordnen diese zur Laufzeit Prozessen zu, optimieren die Datenbankzugriffe und erleichtern die Programmierung durch vordefinierte Standardschnittstellen.

Container: In heutigen Technologien für verteilte Systeme wird der Begriff *Container* als konkrete Laufzeitumgebung für gleichartige Komponenten häufig verwendet. Container sind Bestandteile von Application-Servern. Je nach verwalteten Objekten spricht man auch von Web-Containern, EJB-Containern usw. Container laufen meist innerhalb der Umgebung eines Application-Servers ab und sind in Application-Server eingebettet. Container nehmen dem Programmierer und auch dem Systemintegrator (auch als Deployer bezeichnet) einige Arbeiten ab. Hierzu gehört:

- Der Programmierer muss sich nicht mehr um die Erzeugung von Komponenteninstanzen kümmern. Die Verwaltung des Lebenszyklus von Komponenten wird von Containern übernommen.

- Container ermöglichen den nebenläufigen Ablauf mehrerer Instanzen einer Komponente ohne zusätzliche Programmierung. Der Programmierer einer Komponente nutzt ein einfaches, sequentielles Programmiermodell. Die Parallelisierung übernimmt der Container. Komponenten werden vom Container bei Bedarf erzeugt und freigegeben.
- Container stellen eine Reihe von Systemdiensten bereit, die vom Komponentenentwickler verwendet werden können bzw. implizit verwendet werden. Hierzu gehören das Pooling von Datenbankverbindungen und Threadpooling.
- Container stellen Basismechanismen für die Einbettung von Komponenten dar. Über Regeln wird festgehalten, welche Schnittstellen eine Komponente bereitstellen muss, damit sie im Container ablauffähig ist.
- Mechanismen für die Skalierbarkeit und auch für Fehlertoleranz und Lastverteilung werden durch den Container unterstützt. Komponenten können mehrfach gestartet werden.

Inversion of Control: Die Programmierung von Komponenten ist im Vergleich zur Programmierung verteilter Objekte einfacher. Der Container kümmert sich um den gesamten Lebenszyklus einer Komponente, dient als Laufzeitumgebung und ruft bei einem ankommenden Request die Methode des entsprechenden Komponenten-Interface auf. Dies bezeichnet man auch als *Inversion of Control*[9].

2.4.2 Fallbeispiel CORBA Components (CCM)

Eine weitere Komponententechnologie namens CORBA Components Model (CCM) wurde von der OMG spezifiziert (OMG 2002a). Basis von CCM ist CORBA. CCM stellt also eine Erweiterung von CORBA um Möglichkeiten zur komponentenbasierten Entwicklung dar. Der in der OMA definierte Object Request Broker (ORB) dient zur Kommunikation zwischen den beteiligten Instanzen.

Ähnlich wie die EJB-Spezifikation stellt CCM ein Komponentenmodell für verteilte Komponenten bereit, das genaue Vorgaben zu den Komponentenverträgen macht und hierfür eine Menge von Schnittstellen festlegt. Eine Integration mit EJBeans ist ebenfalls möglich. CCM besteht im Wesentlichen aus dem *Komponentenmodell* selbst, der *Component IDL (CIDL)*, dem *Component Implementation Framework* (*CIF*) und dem *Container Programming Model.*

CCM definiert den grundsätzlichen Aufbau der Komponenten sowie Ihre Eigenschaften. Es legt auch fest, wie man Komponenten zu sog. *Assemblies* zusammenbauen kann und wie diese verteilt werden (Deployment). *CIDL* ist eine Erweiterung von IDL und *CIF* stellt ein Rahmenwerk für die Komponentenentwicklung und die Infrastruktur für die Laufzeit der Komponenten bereit.

---

[9] Man nennt dies auch das Hollywood-Prinzip („Don't call us, we call you").

Beispiel: Die folgende CIDL-Notation zeigt eine CORBA-Komponente namens *MyCCMComponent*, die zwei Interfaces bereitstellt und zum Modul *MyCCMModule* gehört. Das Schlüsselwort *session* gibt den Lebenszyklus-Typ an und ist vergleichbar mit dem Bean-Typ *Session-Bean* bei EJB (siehe weiter unten). Neben dem Komponententyp *session* gibt es auch noch die Typen *service*, *entity* und *process*.

```
module MyCCMModule {
        interface MyInterface1 {... };
        interface MyInterface2 { ... };
        component MyCCMComponent session {
                provides MyInterface1 i1;
                provides MyInterface2 i2;
        };
};
```

Das Container Programming Model definiert einen *Container* als Laufzeitumgebung für Komponenten, der in Application-Servern implementiert wird. Der Container dient der Verwaltung von Komponenten-Instanzen. Diese können mit Hilfe eines POA (Portable Object Adapter) referenziert werden. CORBA-Komponenten greifen auch über einen ORB auf die Standarddienste des CORBA-Modells zu.

CCM-Komponenten sind die wesentlichen Bausteine im CCM. Eine Komponente kann mehrere Schnittstellen bereitstellen. Aus der CIDL-Schnittstellenspezifikation generiert ein CCM-konformes System die Skeletons für Client und Server.

Die Schnittstellen zur Kommunikation mit den CORBA-Clients und anderen CORBA-Komponenten werden auch als *Ports* bezeichnet. Vier Port-Typen sind vorgegeben, von denen eine CORBA-Komponente jeweils mehrere anbieten kann:

- *Facets* sind die Schnittstellen, die einem Client zur Verfügung gestellt werden. Eine CORBA-Komponente kann verschiedene *Facets* anbieten, die jeweils durch eine Objekt-Referenz angesprochen werden können. Jedes *Facet* ist durch ein IDL-Interface beschrieben.
- *Receptables* sind Schnittstellen, die es einer CORBA-Komponente ermöglichen, mit anderen Komponenten eine Verbindung aufzubauen. *Receptables* sind benannte Verbindungsendpunkte (Named Connection Points).
- *Event Sources* sind auch benannte Verbindungsendpunkte, die es ermöglichen, typisierte Ereignisse an Clients über die Implementierung des *Observer Patterns* bereitzustellen. Ereignisse können an diesem Port-Typ publiziert (Publish-Subscribe-Modell) und gesendet werden.
- *Event Sinks* dienen dazu, definierte Ereignisse entgegenzunehmen. Es sind auch benannte Verbindungsendpunkte. Komponenten können gemäß dem Publish-Subscribe-Modell Ereignisse abonnieren. Tritt ein konkretes Ereignis für einen abonnierten Typ ein, wird es am *Event Sink* zugestellt. Von der Komponente kann an diesen Port-Typ eine Aktionsroutine bereitgestellt werden, die auf das Ereignis adäquat reagiert.

Wie in Abbildung 2-51 zu sehen ist, verfügen CORBA-Komponenten über ein Home-Interface, das hier mit IDL definiert wird. Es wird für das Lifecycle-Management verwendet (Erzeugen und Löschen von Komponenten-Instanzen). Die Referenzen der Home-Schnittstellen werden in einer Datenbasis verwaltet. Eine *HomeFinder*-Schnittstelle dient dem Auffinden der Komponenten.

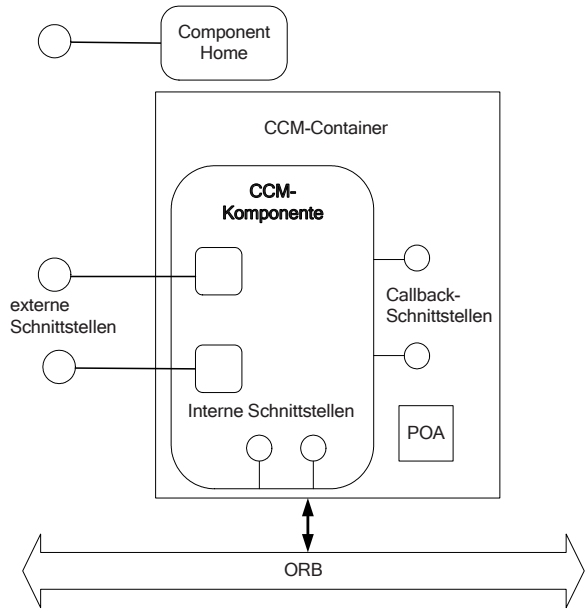


Abbildung 2-51: CCM-Container-Programmiermodell nach (Andresen 2003)

Beispiel: In Abbildung 2-52 ist eine Komponente zur Artikelverwaltung skizziert, die über alle vier Port-Typen verfügt. Die Adressierung einer Komponente erfolgt, wie in der Abbildung dargestellt, über eine CORBA-Objektreferenz. Im Bild stellt eine Komponente namens *Artikelverwaltungs-Komponente* ein Facet *Artikel* für Clients bereit, um Methoden für den Zugriff auf Artikel aufzurufen. Dieses *Facet* wird von den Clients verwendet. Über das Receptable *Lager* kann eine andere Komponente (z.B. ein Lagerverwaltungssystem) auf Bestandsdaten von Artikeln zugreifen. Über den *Event Sink* Auftrag können z.B. neu im System ankommende Aufträge an die Bestandsverwaltung gemeldet werden. Damit kann der Bestand aktualisiert werden. Das *Event Source Preisänderung* dient der Propagierung von Preisänderungen von Artikeln an andere betroffene Komponenten (z.B. eine Auftragsverwaltung).

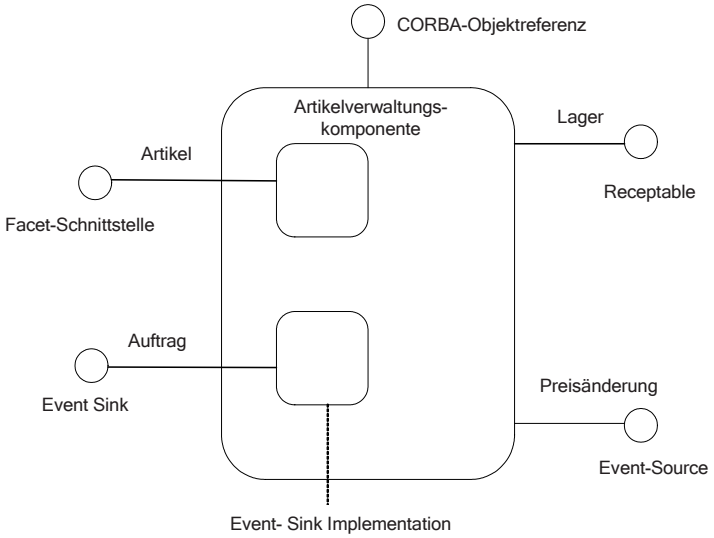


Abbildung 2-52: CCM-Komponente und deren Ports nach (Andresen 2003)

Die Entwicklung von eigenen Komponenten wird mit Hilfe eines CIDL-Compiler durchgeführt, der aus einer CIDL-Datei eine Komponentenbeschreibung sowie ein Komponenten-Gerüst erzeugt.

Die Bedeutung von CCM ist in der Praxis relativ gering. Es gibt nur wenige Implementierungen wie etwa *OpenCCM* (WWW-039). Für eine tiefergehende Betrachtung wird auf die Literatur verwiesen (WWW-009). Das CCM ist allerdings im Vergleich zu anderen Komponentenmodellen mächtiger und daher relativ aufwändig zu implementieren. Möglicherweise ist dies auch ein Grund für die geringe Verbreitung.

2.4.3 Fallbeispiel .NET Enterprise Services

Das Komponentenmodell von Microsoft heißt COM+, basiert auf der Windows-*DNA*-Architektur (*Windows Distributed interNetwork Applications Architecture*). Es baut auf COM und DCOM auf. Windows-DNA wurde von Microsoft im Jahre 1999 mit COM+ vermarktet und enthält neben COM+ auch eine Reihe von Produkten. DNA war in der Praxis nie relevant, da es im Wesentlichen ein Satz von Dokumentationen bzw. Spezifikationen war (Kuhmann 2004). COM steht für *Component Object Model* und wurde aus der *OLE*-Technologie (*Object Linking and Embedding*), die der Verküpfung von Dokumenten unter Windows dient, weiterentwickelt. Vorgänger von OLE war wiederum das Clipboard von Windows, also die klassische Windows-Zwischenablage aus den späten 80er Jahren. *DCOM* (*Distributed* COM) erweiterte COM um verteilte Kommunikationsmechanismen auf Basis

des Remote Procedure Calls. Verteilte Komponenten kommunizieren bei DCOM über eine Implementierung von DCE/RPC.

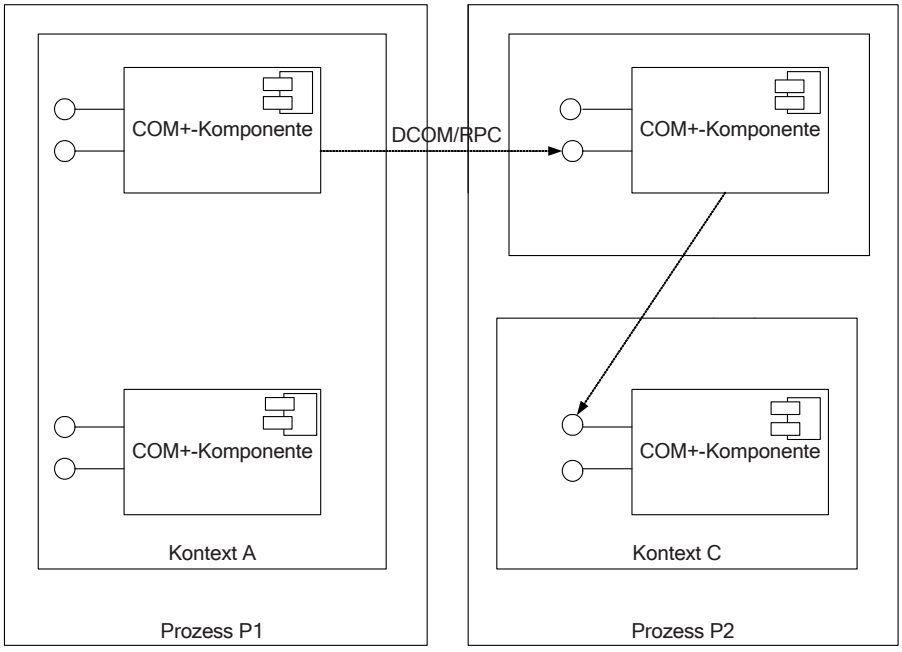


Abbildung 2-53: COM+-Komponenten, Kontexte und Prozesse nach (Andresen 2003)

Bevor COM+ freigegeben wurde, veröffentlichte Microsoft in den 90er Jahren noch einen Application-Server namens *MTS (*MTS = *Microsoft Transaction Manager).* MTS gilt heute als der erste Application-Server. Weiterhin entwickelte Microsoft einen verteilten Transaktionskoordinator namens *MSDTC* und einen Message-Queue-Server namens *MSMQ* jeweils als eigenständige Produkte auf dem Markt. All diese Produkte wurden dann Ende der 90er Jahre in COM+ vereint.

COM+-Komponenten werden in dynamischen Laufzeitbibliotheken (DLL) abgelegt und zur Laufzeit vom sog. *Service Control Manager* (SCM) auf eine Clientanfrage hin geladen. Die prozessübergreifende Kommunikation erfolgt über DCOM und damit über DCE/RPC.

COM+ sorgt für die Aktivierung von Objekten (Just-in-Time-Activation), für das Objekt-Pooling, die Transaktionsunterstützung usw. Die COM+-Laufzeitumgebung unter Windows erfüllt also im Wesentlichen die Dienste eines Containers bzw. eines Application-Servers. In Abbildung 2-53 ist eine Komponentenverteilung unter Windows auf zwei Betriebssystemprozesse dargestellt. Mehrere Komponen-

ten können beim COM+ unter einem gemeinsamen Kontext ablaufen und ein Prozess kann mehrere Kontexte aufnehmen.

Nachfolger von COM+ sind die *.NET Enterprise Services.* Der Anfang 2000 vorgestellte „Komponentenstandard" der Windows-Welt stellt auch eine Basis für komponentenbasierte verteilte Systeme dar und setzt selbst auf COM+ auf. Komponenten laufen auch in Containern ab.

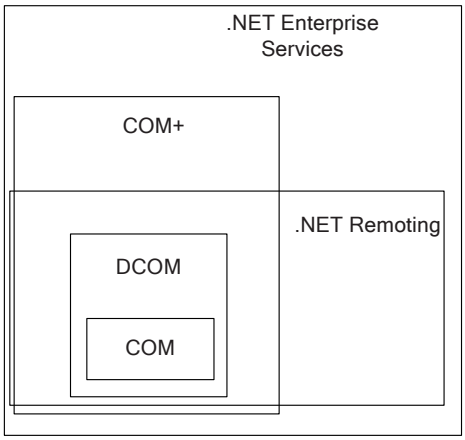


Abbildung 2-54: .NET Enterprise Services, Einordnung nach (Kuhrmann 2004)

.NET Enterprise Services basiert auf der *Common Language Runtime (CLR). CLR* stellt - wie auch die *Java Virtual Machine (JVM)* - eine Laufzeitumgebung für Bytecode-orientierte Sprachen dar. *CLR* dient der Interpretation mehrerer Sprachen, weshalb der Windows-Komponentenstandard verschiedensprachige Komponenten unterstützt (C++, C#, J#, VB,...).

Die Zusammenhänge der einzelnen Windows-Komponentenlösungen sind in Abbildung 2-54 aufgezeigt. COM+ vereint die älteren Technologien COM und DCOM und die .NET Enterprise Services nutzen sowohl COM+ als auch .NET Remoting. Das Ziel von Microsoft zur weiteren Entwicklung verteilter Basissysteme ist die Vereinheitlichung der Technologien. Derzeit wird noch COM+ benötigt, um verteilte Komponenten zum Ablauf zu bringen.

Im Folgenden wird kurz auf das Progammiermodell und die Laufzeitumgebung von *.NET Enterprise Services* eingegangen. Weitere Informationen sind in der angegebenen Literatur enthalten.

Jede Komponente ist als sog. *Serviced Component* zu implementieren, indem sie von der Basisklasse *ServicedComponent* (Namensraum *System.EnterpriseServices*) abgeleitet wird. Alle *Serviced Components* werden in Containern verwaltet, die heute noch übergangsweise auf die COM+-Infrastruktur abgebildet werden. Container sind „Lebensräume für Komponenten". Sie werden in einem Katalog, dem sog. *COM+-*

Katalog mit all ihren Metadaten registriert und damit für ihre Nutzer bekannt gemacht. Die Klasse *ServicedComponent* leitet sich wiederum aus der Klasse *ContectBoundObject* und indirekt aus der Basisklasse *MarshalByRefObject* ab, womit die Kommunikationsmechanismen von .NET Remoting geerbt werden. *Serviced Components* sind also auch entfernte Objekte. Komponenten müssen in einer Bibliotheks-Assembly (DLL) abgelegt sein.

Die Objektklasse *ContextBoundObject* (Kontextgebundenes Objekt) ist eine Erweiterung der Objektklasse *MarshalByRefObject*. Über einen Kontext können mehrere Komponenten zu einer gemeinsamen Gruppe zusammengefasst werden. Dieser Gruppe können dann gemeinsame Eigenschaften, sog. *Kontextattribute*, zugewiesen werden.

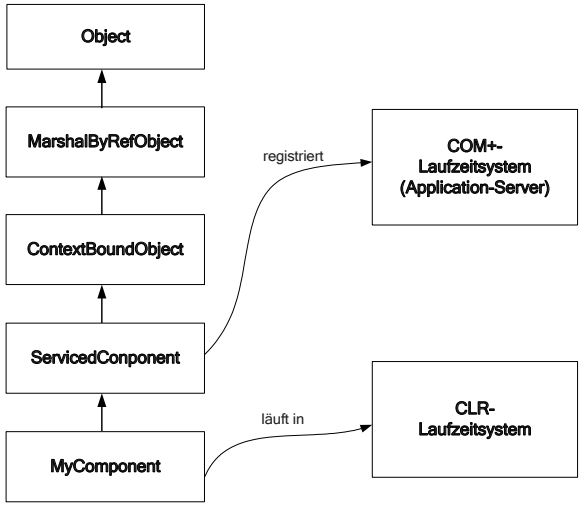


Abbildung 2-55: Klassenhierarchie für .NET-Komponenten

Eine verteilte Anwendung, die auf Basis der .NET Enterprise Services läuft, besteht aus einer Gruppe von Komponenten, der gemeinsame technische Eigenschaften wie der Aktivierungsmechanismus, das Prozessmodell und der Mechanismus zum Nachrichtentransport zugewiesen werden können. Das Prozessmodell sieht unter .NET auch noch sog. *Anwendungsdomänen* vor. Anwendungsdomänen stellen eine logische Aufteilung von Prozessen dar. In einer Anwendungsdomäne laufen mehrere Anwendungen ab. Ein Prozess kann mehrere Anwendungsdomänen enthalten. Damit können also mehrere Anwendungen in einem Prozess verwaltet werden.

Die .NET Enterprise Services stellen den eingebetteten Komponenten einige Basisdienste bereit, die im Wesentlichen aus COM+ stammen. Hierzu gehören die verteilte Transaktionsunterstützung, Ressourcen-/Objekt-Pooling-Mechanismen und

eine Just-In-Time-Aktivierung für die Komponenten (JITA). Eine Komponente registriert sich beim COM+-Laufzeitsystem und läuft in der CLR ab (siehe Abbildung 2-55).

Ähnlich wie in anderen Komponentenmodellen (siehe EJB) müssen auch .NET-Komponenten einige Methoden implementieren, die für den Lebenszyklus wichtig sind. Beispielmethoden der Klasse *ServiceComponents* sind z.B. *Activate* zur Kompontentenaktivierung und *Deactivate* zur Deaktivierung. Die Methoden werden durch das Laufzeitsystem aufgerufen. Eigenschaften der Komponenten, wie z.B. das Transaktionsverhalten, können über Attribute im Programm eingestellt werden. Das Transaktionsverhalten der .NET Komponententechnologie wird in Kapitel 4 näher betrachtet.

2.4.4 Fallbeispiel JEE / Enterprise Java Beans

JEE bzw. J2EE (vor der Version 1.5) beinhaltet neben vielen anderen Standards für verteilte Systeme die Enterprise-Java-Beans-Spezifikation (EJB). EJB dient der Entwicklung von serverseitigen Komponenten und ist mittlerweile in der Praxis ein verbreiteter Ansatz. Wir versuchen uns bei dieser Betrachtung auf die EJB-Grundkonzepte zu beschränken. Eine detaillierte Technologiebetrachtuung ist in der EJB-Spezifikation (Sun 2007a) bzw. in (Burke 2006) oder (Backschat 2007) zu finden.

Idee und Ziele

Bei Enterprise JavaBeans handelt es sich um ein verteiltes Komponentenmodell, bei dem die Komponenten auch als Enterprise Java Beans[10] (im Weiteren auch EJ-Beans) bezeichnet werden. Ziel der EJB-Spezifikation ist es, einen „Standard" für verteilte, transaktionsorientierte, objektorientierte und serverbasierte Komponenten, die mit Java entwickelt werden, zu schaffen. Enterprise JavaBeans sind dabei die Serverkomponenten. Diese werden nach bestimmten Regeln entwickelt und müssen einige sog. Contracts (Verträge) erfüllen, damit sie in jeder beliebigen Umgebung, welche die EJB-Spezifikation unterstützt, ablaufen können.

Die Definition von Sun Microsystems für Enterprise JavaBeans (EJBeans) lautet wie folgt (WWW-004): *"The Enterprise JavaBeans architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multiuser secure. These applications may be written once, and deployed on any server platform that supports the Enterprise JavaBeans specification."*

Unter Nutzung der EJB-Technologie soll es also möglich sein, serverseitige Komponenten, die mit Werkzeugen unterschiedlicher Hersteller entwickelt werden, miteinander zu kombinieren. Das Prinzip „Write Once Run Anywhere" trägt dem

[10] Wir verwenden im Weiteren den Begriff Bean im Femininum.

Portabilitätsgedanken Rechnung. Die EJB-Spezifikation zielt also nicht nur auf die Phase der Entwicklung ab, sondern regelt auch die Aspekte Deployment und Runtime.

Enterprise Java Beans sind also serverseitige Komponenten und laufen in EJB-Application-Servern bzw. in EJB-Containern innerhalb von EJB-Application-Servern ab. Wir bezeichnen im weiteren Enterprise Java Beans auch in Abgrenzung zum Komponentenmodell kurz als *EJBeans*. Konkrete Instanzen von EJBeans bezeichnen wir mit EJB-Instanzen., sofern der Kontext nicht eindeutig ist.

Eine Bean soll laut EJB-Spezifikation auf einem EJB-Application-Server jedem Ablauf kommen können, ohne den Code noch einmal zu verändern. Der Hersteller des EJB-Application-Servers muss sich an die EJB-Spezifikation halten. Es muss lediglich ein sog. *Deployment Descriptor (DD)*, der außerhalb des Anwendungscodes liegt, angepasst werden. Dies ist eine XML-basierte Konfigurationsdatei.

Weitere wichtige Aspekte der EJB-Spezifikation sind vor allem die Transaktionsunterstützung, die Bereitstellung von Security-Mechanismen und die Unterstützung bei konkurrierenden Zugriffen auf persistente Daten. Auf welche Weise solche aufwändigen Mechanismen wie Transaktionsunterstützung und persistente Datenhaltung implementiert werden sollen, bleibt den Herstellern überlassen. Hieraus ergibt sich auch, dass Hersteller von EJB-Produkten das Rad nicht neu erfinden. Hersteller entwickelten und entwickeln vielmehr vorhandene Application-Server, Datenbankmanagementsysteme oder transaktionale Basissysteme zu EJB-Application-Servern weiter.

EJB-Application-Server und EJB-Container

Ein EJB-Container ist für die Erzeugung und die Verwaltung der Beans in einer nebenläufigen Umgebung zuständig. Die Infrastruktur, in der ein EJB-Container in ein Betriebssystem eingebettet wird, wird als *EJB-Application-Server* oder kurz als *EJB-Server* bezeichnet. In die Sprache Java ist die EJB-Technologie über das Java-Package *javax.ejb* eingebettet. Es enthält alle Klassen und Interfaces für die Zusammenarbeit zwischen EJB-Client und EJB-Container sowie zwischen EJBeans und EJB-Container. Damit sind gewissermaßen auch die Verträge der Zusammenarbeit zwischen den beteiligten Softwarebausteinen definiert.

Ein EJB-Container verwaltet den Kontext von EJBeans. Es gibt mehrere Bean-Typen. Der Zugang einer Client-Anwendung zu einem EJBean erfolgt über dedizierte Komponentenschnittstellen. In früheren EJB-Versionen (< 3.0) verfügte jede EJBean über zwei vordefinierte Schnittstellen, ein sog. *Home*- und ein *Remote*-Interface. Der Bean-Entwickler musste beide Interfaces definieren und die eigentlichen Methoden-Implementierungen bereitstellen. Das Home-Interface diente als Factory zur Erzeugung einer EJBean. Das *Remote*-Interface stellte die Funktionalität der Komponente über Business-Methoden bereit. Das Home- und das Remote-Interface repräsentierten die Sicht eines EJB-Clients. In neueren Versionen (ab EJB

3.0) wurde die Unterscheidung zwischen Home- und Remote-Interface aufgehoben und soll daher nicht weiter betrachtet werden. Der Fokus unserer Betrachtung liegt auf der EJB-Version 3.0.

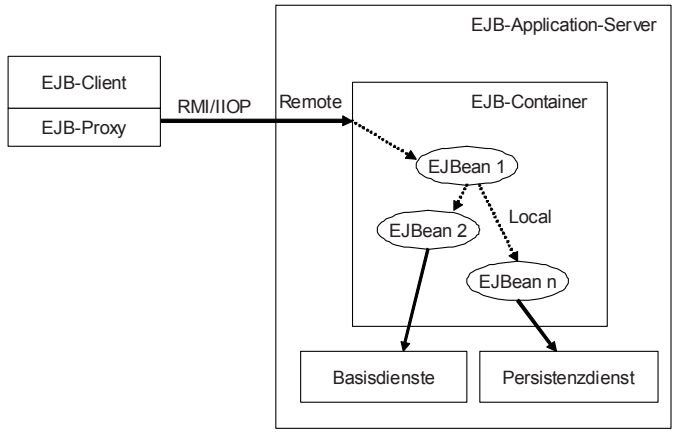


Abbildung 2-56: EJB-Architektur im Groben

In Abbildung 2-56 wird ein grober Überblick über die serverseitige EJB-Architektur gegeben. Innerhalb eines EJB-Containers kommen beliebig viele EJ-Beans zum Ablauf, die über Remote-Schnittstellen von außen zugänglich sind. Innerhalb des EJB-Containers können die EJBeans auch über Local-Schnittstellen genutzt werden. EJBeans können sich also im Container auch gegenseitig nutzen. In der Spezifikation ist die Art der Implementierung des EJB-Containers nicht vorgegeben. Er kann in einer oder mehreren JVMs ablaufen. Der EJB-Container ist in einen EJB-Application-Server eingebettet, der die Schnittstellen von Basisdiensten (Naming, Transaktionen,…) und einen Persistenzdienst bereitstellt. Die Schnittstelle zwischen EJB-Container und EJB-Application-Server ist nur über die Dienstschnittstellen vorgegeben. Im Client ist ein EJB-Proxy erforderlich, der den Zugriff und das Marshalling regelt.

In der EJB-Spezifikation sind die Bean-Typen *Session-Bean* (stateless und stateful), *Entity-Bean* und *Message-Driven-Bean* (MDB) definiert. Damit können eine Reihe von grundlegenden Client-Server-Mechanismen (siehe Abschnitt 2.1) abgebildet werden. Diese Session-Beans und ihre Besonderheiten sollen im Weiteren betrachtet werden. MDBs kamen erst später hinzu, um auch die nachrichtenbasierte Kommunikation zu unterstützen. Sie werden in Abschnitt 2.5 erläutert. Entity-Beans dienen dem Zugang zu Datenbank-Entitäten und werden in Kapitel 5 im Zuge der Architekturdiskussion nochmals aufgegriffen.

Rollen

Die EJB-Architektur legt Benutzerrollen für die Entwicklung, die Inbetriebnahme und die Administration fest. Die Rollen sollen kurz beschrieben werden:

- Eine EJBean wird von einem *Enterprise Bean Provider* (kurz Bean-Provider) erstellt. Dieser ist ein Spezialist einer Anwendungsdomäne und liefert im Wesentlichen die eigentliche Anwendungslogik.
- Ein *Application Assembler* ist ein in seiner Anwendungsdomäne erfahrener Anwendungsentwickler, der sich für eine Anwendung die notwendigen Komponenten z.B. in einem zukünftigen EJBean-Markt oder im Unternehmen zusammensucht und daraus seine Applikation aufbaut.
- Ein *Deployer* ist gewissermaßen ein Systemintegrator, der die fertige Applikation in einer IT-Umgebung zum Einsatz bringt bzw. installiert. Damit eine automatische Generierung von Code möglich ist, muss der *Component Contract* eingehalten werden. Der Bean-Provider muss die entsprechenden Methoden wie im Contract vorgegeben implementieren.
- Ein *EJB-Server-Provider* (Server-Provider) ist typischerweise ein Hersteller von Middleware- oder Basissoftware, der die notwendigen Basisdienste wie Transaktionsmechanismen usw. bereitstellt.
- Ein *EJB-Container-Provider* (Container-Provider) liefert die erforderlichen Deployment-Werkzeuge sowie die Laufzeitumgebung, um EJBeans zum Einsatz zu bringen.
- Ein *System-Administrator* ist für die Konfigurierung und Administration der verteilten Umgebung verantwortlich, in der die EJBeans ablaufen sollen.

Die Schnittstelle zwischen einem EJB-Container und einem EJB-Application-Server ist nicht in einem *Contract* festgelegt. Die EJB-Spezifikation geht davon aus, dass in der Praxis die beiden Rollen EJB-Server- und Container-Provider zusammenfallen. Dies sind die typischen Aufgaben des Herstellers eines EJB-Application-Servers. Hier wird auch nochmals deutlich, dass typische EJB-Application-Server als umfassende Laufzeitumgebungen angesehen werden, die sowohl die Aufgaben des EJB-Server-Providers, als auch die eines EJB-Container-Providers übernehmen. Nicht in der EJB-Spezifikation vorgeschrieben, aber in der Regel ebenfalls von einem EJB-Application-Server angeboten, sind die Werkzeuge zur Entwicklung von EJBeans, die vom Bean-Provider verwendet werden.

Session-Beans

Lokale und entfernte Schnittstellen: Session-Beans benötigen ein oder mehrere Business-Interfaces. Die Kennzeichnung, ob es sich um ein lokales oder entferntes Interface handelt, erfolgt über die Java-Annotationen *@Local* und *@Remote*.

Erst ab EJB-Version 2.0 gibt es auch lokale Interfaces für EJBeans. Aufrufe über lokale Interfaces sind auf die virtuelle Maschine (JVM) beschränkt, in der die EJBean läuft. Der Grund für die Einführung von lokalen Interfaces liegt in der Effizienz der Aufrufe. Ohne diese Möglichkeit konnten sich EJBeans nämlich nur über

die Remote-Interfaces gegenseitig nutzen, was ineffizienter ist, als ein lokaler Aufruf. Der gesamte Remote-Mechanismus muss hier nämlich durchlaufen werden. Eine Architektur, in der sich EJBeans auch gegenseitig nutzen, war damit für größere Anwendungen praktisch nicht möglich. Allerdings haben lokale Interfaces den Nachteil, dass die Ortstransparenz nicht mehr gegeben ist. Der Nutzer der Schnittstelle muss wissen, wo die EJBean liegt.

Ein wichtiger Unterschied zwischen Remote- und Local-Interface ist die Übergabe der Parameter bei den Methodenaufrufen. Während Remote-Interfaces die *Call-by-value*-Semantik (Call-by-copy) verwenden, nutzen Local-Interfaces für die Parameterübergabe *Call-by-reference*, d.h. statt Werte von Objekten werden Referenzen auf Objekte (im lokalen Adressraum der JVM) übergeben.

Ein Business-Interface einer Session-Bean wird nun als . *POJI* (Plain Old Java Interface) angegeben. Dies sind klassische Java-Interfaces ohne EJB-Besonderheiten. Die aus EJB 2.1 bekannten Interfaces *EJBObject* oder *EJBLocalObject* müssen nicht mehr implementiert werden. Eine typische Notation für die Implementierung einer Bean wird an einem Beispiel skizziert.

Beispiel: Am Beispiel unseres Artikelmanagers soll gezeigt werden, wie man Interfaces für Session-Beans notiert. Man definiert zunächst beispielsweise ein klassisches POJI für die Business-Methoden und erweitert dies dann zu Remote- und Local-Interfaces.

```
// Klassisches POJI mit Business-Methoden
package …;
import javax.ejb.*;
…
public Interface ArticleMgrI {
     public Article updateArticle(Article a);
     public Article deleteArticle(long catalogId);
     …
}
…
// Remote-Interface abgeleitet vom klassischen POJI
@Remote
public Interface ArticleMgrRemoteI extends ArticleMgrI {
     …
}
// Local-Interface abgeleitet vom klassischen POJI
@Local
public Interface ArticleMgrLocalI extends ArticleMgrI {
     …
}
```

Eine Implementierung der Interfaces erfolgt in . POJOs (Plain Old Java Objects), wie wir noch weiter unten sehen werden.

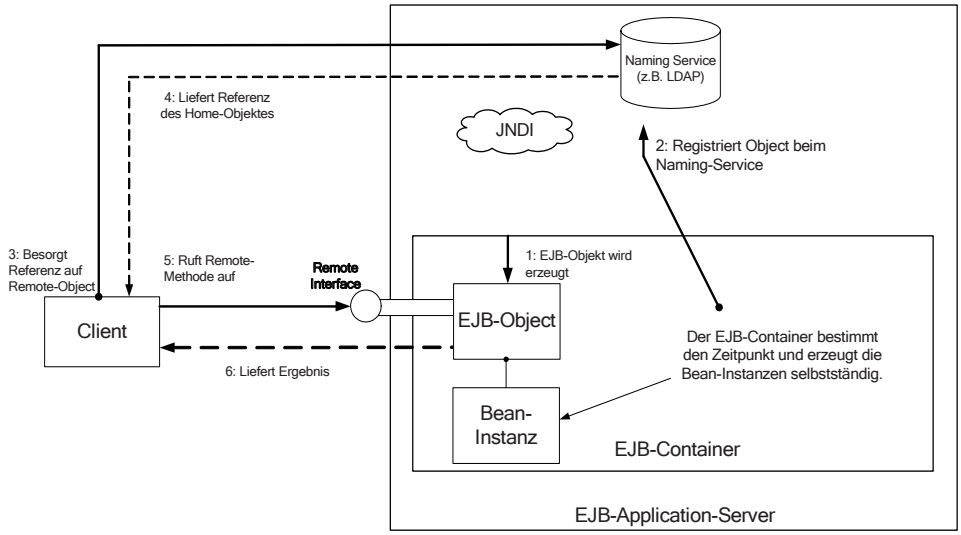


Abbildung 2-57: Abwicklung eines synchronen Aufrufs einer Session-Bean

In Abbildung 2-57 ist das Zusammenspiel von Application-Server, EJB-Container und EJB-Client skizziert, wobei es keine Rolle spielt, um welche Art von Session-Bean es sich handelt. Zunächst muss der EJB-Container ein EJB-Objekt erzeugen und eine Bean-Instanz zuordnen (1). Das EJB-Objekt wird bei einem Naming-Service über das JNDI-API registriert (2). Ein Client muss, bevor er die Dienste einer EJBean nutzen kann, dessen Adresse (Referenz) über JNDI besorgen (3)(4). Anschließend wendet er sich an den zuständigen Application-Server, um über das Remote-Interface der EJBean zu adressieren.

Der Vorgang des Erzeugens, bei EJB als Aktivieren bezeichnet, und des Entfernens (Passivierens) einer Bean-Instanz ist für den Client transparent. Das Erzeugen oder das Verwenden einer bereits bestehenden Bean-Instanz bestimmt der Container.

Bei Session-Beans unterscheidet man *stateful* und *stateless* Session-Beans. Session-Beans werden nicht persistent verwaltet. Stateful Session-Beans ordnen einem EJB-Client einen Zustand zu. Sie werden benötigt, wenn ein Client einen Session-Kontext benötigt. Dies ist beispielsweise erforderlich, wenn mehrere Methodenaufrufe zusammenhängend verwendet werden und ein Aufruf einer Methode auf den Zustand des vorherigen Methodenaufrufs aufsetzt. Für die EJBean wird vom Container ein Konversationszustand verwaltet. Stateful Session-Beans realisieren zustandsbehaftete Serverobjekte. Die Lebensdauer einer stateful Session-Bean ist an

die Client-Session gebunden und damit auch durch diese begrenzt. Eine stateful Session-Bean kann nicht mehreren Clients zugeordnet werden.

Stateless Session-Beans: Stateless Session-Beans sind Implementierungen zustandsloser Serverobjekte, die von mehreren (auch vielen) Clients gleichzeitig genutzt werden können. Ein Methodenaufruf eines Clients wird vollständig abgearbeitet. Die Lebensdauer einer stateless Session-Bean ist nicht an die Client-Session gebunden. Sie wird durch den Container implementierungsabhängig bestimmt. In Abbildung 2-58 ist der Lebenszyklus einer stateless Session-Bean dargestellt. Entweder die Session-Bean ist nicht vorhanden oder sie liegt in einem Bean-Pool für die Bearbeitung neuer Methodenaufrufe bereit. Der Zustandsübergang vom Zustand *Pooled-Ready* nach *Does-Not-Exist* wird durch den Container initiiert. Der Pool wird auch durch den Container verwaltet. Beim Zustandsübergang vom Zustand *Does-Not-Exist* nach *Pooled-Ready* wird ggf. eine Bean-Instanz erzeugt und der Session-Kontext belegt. Jeder Methodenaufruf wird komplett unabhängig von einem anderen Methodenaufruf ausgeführt. Während der Zustandsübergänge können Methoden aufgerufen werden, die vom EJB-Provider mit Hilfe der Annotationen *@PreDestroy* und *@PostConstruct* definiert werden *können. Mehr hierzu bei der Erläuterung der Konzepte Callback und Interceptor*.

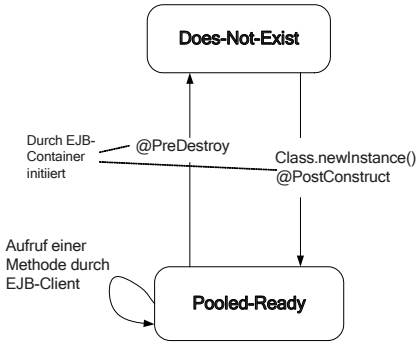


Abbildung 2-58: Zustandsdiagramm für stateless Session-Bean

Stateful Session-Beans: Der Zustandsautomat für eine stateful Session-Bean ist in Abbildung 2-59 skizziert. Es gibt hier einen Zustand mehr als bei stateless Session-Beans. Der Zustand *Passive* wird eingenommen, wenn die Session-Bean vom Container passiviert wird. Sie wird dort vom Container in einen externen Speicherbereich gelegt und bei Aufruf einer Business-Methode durch den Client implizit wieder aktiviert.

Bei stateful Session-Beans wird für jeden Client eine eigene Instanz zugeordnet. Es wird ein Konversationszustand verwaltet, was z.B. bei der Realisierung eines Warenkorbs für eine Web-Anwendung sinnvoll ist. Stateless Session-Beans ermögli-

chen im Vergleich hierzu eine wesentlich effizientere Abarbeitung der Methoden und sind auch robuster als stateful Session-Beans.

Der Container kann die Bean-Instanz zu einem beliebigen Zeitpunkt vor der eigentlichen Nutzung durch einen Client anlegen. Der Client bemerkt davon nichts. Nach dem Passivieren und erneuten Aktivieren ist es möglich, dass eine andere Bean-Instanz die Aufgabe übernimmt. Nicht die physikalische Bean-Instanz, sondern der Zustand (Konversationszustand) der stateful Session-Bean ist dem Client zugeordnet.

Beim Zustandsübergang vom Zustand *Passive* in den Zustand *Method-Ready* ruft der Container beispielsweise bei einer stateful Session-Bean die über die Annotation *@PrePassivate* definierte Methode auf. Weitere Callback-Routinen können über die Annotationen *„@PostConstruct“*, *@PreDestroy*, usw. angegeben werden. Die Bezeichnungen der Annotationen geben meist schon das Callback-Ereignis an. Die mit *@PostConstruct* angegebene Methode wird nach der Konstruktion einer Session-Bean aufgerufen, die mit *@PostActivate* angegebene Methode wird vor einer Aktivierung aufgerufen usw. Die Annotationen *@PrePassivate* und *@PostActivate* können nur bei stateful Session-Beans angewendet werden. Ein weiterer Zustand kommt noch hinzu, wenn die Session-Bean in einer Transaktion verwendet wird. In diesem Fall gelten auch noch weitere Regeln bzgl. der Passivierung (siehe Kapitel 4).

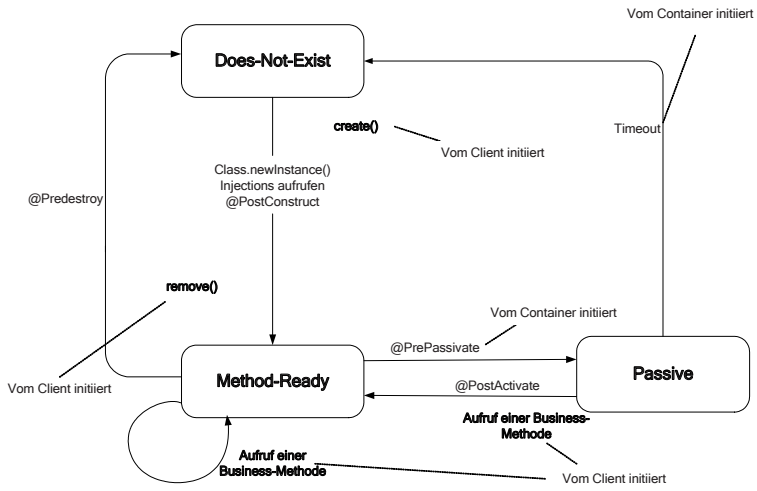


Abbildung 2-59: Zustandsdiagramm für stateful Session-Bean

Beispiel: Am Beispiel unseres Artikelmanagers soll gezeigt werden, wie eine stateless Session-Bean notiert wird. Man implementiert Remote- und Local-Interfaces, die vorher mit entsprechenden Annotationen definiert wurden. Remote- und Local-Interfaces können dabei gemeinsam oder auch in eigenen Session-Bean-Klassen

implementiert werden. Eine stateless Session-Bean, die die *ArticleMgr*-Interfaces unterstützt, sieht dann wie folgt aus:

```
package …;
import javax.ejb.*;
…
@Stateless
public class ArticleMgrSessionBean implements
        ArticleMgrRemoteI,ArticleMgrLocalI {
     public void updateArticle(Article a) {…}
     public void deleteArticle(long catalogId) {…}
     …
}
```

Der Name der Session-Bean könnte in der Annotation überschrieben werden. Wird dies nicht gemacht, so wird standardmäßig der Name der Klasse, in diesem Fall also *ArticleMgrSessionBean* verwendet. Alternativ könnte man auch auf die explizite Definition von POJIs verzichten und die Bean-Klasse gleich wie folgt programmieren:

```
package …;
import javax.ejb.*;
…
@Stateless
@Remote(ArticleMgrLocalIclass)
@Local(ArticleMgrRemoteI.class)
public class ArticleMgrSessionBean
{
   public void updateArticle(Article a) {…}
   public void deleteArticle(long catalogId) {…}
   …
}
```

Eine explizite Trennung von Interface und Implementierung ist aber die bevorzugte Variante. Eine stateful Session-Bean wird auf ähnliche Weise notiert:

```
import javax.ejb.*;
…
@Stateful public class MySessionBean implements ArticleMgrRemoteI ….
{
   public Article deleteArticle(…) { … }

   @Remove
   public void removeArticleMgr(…) {
        // Code wird vor dem Zerstören der Instanz ausgeführt
   }
}
```

Für jede Session-Bean wird vom Container ein Kontext verwaltet, der durch die Klasse *SessionContext*, die wiederum von der Klasse *EJBContext* abgeleitet ist, definiert wird. Das *SessionContext*-Interface ist wie folgt definiert:

```
public interface javax.ejb.SessionContext extends javax.ejb.Context {
{
   EJBLocalObject getEJBLocalObject() throws …; // EJB 2.1
   EJBObject getEJBObject() throws …; // EJB 2.1
   MessageContext getMessageContext() throws …;
   <T> getBusinessObject(Class <T> businessInterface) throws …;[11]
   class getInvokedBusinessInterface();
}
```

Die Methoden *getEJBLocalObject* und *getEJBObject* gelten nur noch für EJB 2.1. Die Methode *getBusinessObject* liefert eine Referenz auf die aktuelle Bean-Instanz, die mit dem Java-Pointer *this* vergleichbar ist. Diese Referenz kann z.B. genutzt werden, um sie als Parameter für einen Methodenaufruf einzusetzen, da die Weitergabe von *this* in EJBeans nicht zulässig ist. Über die Methode *getInvokedBusiness-Interface* lässt sich herausfinden, über welches Interface (lokal, remote, Webservice) die EJBean aktuell aufgerufen wurde.

Bei einer stateful Session-Bean wird also für jeden Client ein Konversationszustand verwaltet. Ein EJB-Objekt ist zumindest logisch einem Client zugeordnet. Wie aus der Darstellung des Zustandsautomaten hervorgeht, gibt es im Container bei stateful Session-Beans die Vorgänge *Passivieren* und *Aktivieren*. Der Konversationszustand darf hierbei nicht verlorengehen. Die beiden Aktivitäten werden ausschließlich vom Container gesteuert. Nur er bestimmt, wann ein Passivieren erforderlich ist. Ein Grund kann sein, dass die Session-Bean längere Zeit nicht angesprochen wurde. Durch ein Passivieren kann man Hauptspeicher-Ressourcen freimachen, da der Container bei diesem Vorgang laut Spezifikation die Session-Bean aus dem Hauptspeicher entfernt. In diesem Fall muss die Session-Bean serialisiert werden, was bedingt, dass einige Regeln bei der Entwicklung einer Session Bean zu beachten sind. Um die Vorgänge besser zu verstehen, wollen wir noch genauer klären, aus welchen Bestandteilen der Konversationszustand besteht und welche Konsequenzen sich aus einer Passivierung ergeben.

Konversationszustand: Die EJB-Spezifikation definiert den Konversationszustand einer instantierten stateful Session-Bean als die aktuellen Werte aller Instanzvariablen der entsprechenden Bean-Klasse einschließlich aller Objekte, die über die Instanzvariablen per Referenz erreichbar sind. Dies kann ein komplexer Objektgraph sein. Ausgeschlossen sind die Variablen, die als *transient* gekennzeichnet sind.

[11] <T> ist ein Platzhalter für den Typ der Bean.

Passivierung: Bei der Passivierung wird eine Session-Bean-Instanz einschließlich des gesamten referenzierten Objektgraphen gemäß den Regeln der Java-Objektserialisierung oder auch einem anderen herstellerspezifischen Verfahren serialisiert und als Byte-Strom auf einen externen Speicher geschrieben. Das Speichermedium ist herstellerspezifisch festgelegt. Der Speicherbereich, den die EJBean samt Objektgraph belegte, wird freigegeben (Backschat 2007). Die Passivierung erfolgt in der Regel nach Ablauf eines definierten und konfigurierbaren Timers.

Aktivierung: Bei der Aktivierung wird der zur Passivierung inverse Vorgang durchgeführt. Der Byte-Strom wird wieder deserialisiert, im Hauptspeicher abgelegt und einer Session-Bean-Instanz zugeordnet. Dies setzt voraus, dass die Instanzvariablen einer stateful Session-Bean und alle referenzierten Objekte auch serialisierbar sind, also das Java-Interface *Serializable* implementieren. Der Container kümmert sich bei der Passivierung um die von ihm verwalteten Objekte wie SessionContext, Entity Manager (siehe Entity Beans in Kapitel 3), DataSources für den Zugriff auf externe Systeme, Timer usw.

Zusammenfassung: Zusammengefasst kann festgehalten werden, dass Session-Beans für die Realisierung von Client-Server-Beziehungen spezifiziert wurden. Folgende Konzepte wurden eingearbeitet:

- *Stateless Session-Bean* entsprechen zustandslosen *shared* Servern. Die Erzeugung regelt der EJB-Container in Eigenregie, wofür ein Bean-Pool verwaltet wird. Die Aktivierung und Zuweisung zum Client erfolgt bei einem Methodenaufruf eines Clients. Nach dem Methodenaufruf entzieht der EJB-Container dem Client die Bean wieder (SingleCall-Mechanismus).
- *Stateful Session Beans* sind zustandsbehaftete, *unshared* Server. Die Erzeugung regelt ebenfalls der EJB-Container. Die Aktivierung und Zuweisung der Bean-Instanzen zu Clients erfolgt auf Initiative des Clients. Logisch bleibt die EJBean einem Client zugeordnet, bis die Verbindung abgebaut wird. Durch einen Passivierungsmechanismus kann die EJBean nach einiger Zeit der Untätigkeit physikalisch einem Client entzogen werden. Bei erneutem Zugriff auf die „logische“ Bean wird der Konversationszustand durch den EJB-Container wieder hergestellt.
- Das Singleton-Konzept wird bis zur EJB-Version 3.0 noch nicht unterstützt, sondern erst ab Version 3.1. Einige Hersteller bieten aber Möglichkeiten, Singletons zu definieren. Im JBoss-Application Server kann man z.B. im Deployment Descriptor eine Session-Bean als „Singleton-Bean“ konfigurieren. Sie wird dann nur einmal instanziiert. Dazu sind im Deployment Descriptor spezielle Einträge notwendig, es wird allerdings empfohlen, sich mit dem genauen Eintrag im DD zu befassen. Er könnte etwa wie folgt aussehen:

```
<container-configuration>
  <container-pool-conf>
    <MaximumSize>1</MaximumSize>
    <strictMaximumSize>true</strictMaximumSize>
  </container-pool-conf>
</container-configuration>
```

EJB-Clients

EJB-Clients greifen über einen generierten Stub (Proxy-Pattern) auf entfernte Session-Beans zu. Vom Container-Hersteller wird erwartet, dass er die nötigen Tools dafür bereitstellt, um die Stubs dynamisch oder statisch zu erzeugen. Als Protokoll wird entweder das RMI Transport Protokoll oder RMI/IIOP (RMI over IIOP) verwendet. Letzteres bildet die RMI-Mechanismen auf das standardisierte Protokoll IIOP ab. Dies ist sinnvoll, wenn ein Client mit einem CORBA-Objekt kommunizieren möchte oder ein CORBA-Objekt mit einer EJBean.

Bevor eine Methode einer EJBean aufgerufen werden kann, muss die EJBean zunächst über einen *JNDI-Lookup* gefunden werden. Über einen initialen Kontext, den der Container-Hersteller bereitstellt, wird ein konkreter Kontext einer EJBean ermittelt. Anschließend steht das Remote-Interface für die Nutzung der Business-Methoden zur Verfügung. In früheren EJB-Versionen musste zunächst ein Home-Interface erzeugt werden, um ein Remote-Interface für eine EJBean zu instanziieren.

Die Kommunikation zwischen einem Client und einer EJBean ist synchron, also blockierend. Der Client wartet immer auf das Ergebnis, bevor die nächste entfernte Methode ausgeführt werden kann.

Beispiel: Das folgende Beispiel soll die prinzipielle Programmierung eines EJB-Clients zeigen, der die Dienste einer Session-Bean nutzt. Ein EJB-Client, der den weiter oben skizzierten Artikelmanager verwendet, soll hier als Beispiel dienen. In einer eigenen Methode namens *getHome* wird der Namenskontext hergestellt und es wird ein *JNDI-Lookup* auf das Remote-Interface des Artikelmanagers durchgeführt. Der JNDI-Code ist proprietär und kann bei den verschiedenen Application-Server-Produkten unterschiedlich sein. Im Beispiel werden zwei Methoden des Artikelmanagers aufgerufen (*updateArticle* und *deleteArticle*).

```
…
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import MyArticleMgr;
…
public class MyClient {
    public static void main(String[] args) throws RemoteException
    {
```

```
        // Referenz auf eine entfernte Bean
        MyArticleMgr manager = null;

        // Lookup durchführen
        try {
            Hashtable env = new Hashtable();
            // Setzen von Environment-Parametern in Variable env
            // und Namen der gesuchten Bean als Parameterliste aufbauen
            // (Code abhängig vom EJB-Produkt)
            InitialContext ctx = new InitialContext(env);
            manager = ctx.lookup(env, "MaBean");
            ctx.close();
        } catch(Exception e){…}
        …
        try {
            // Beispielhafte Bean-Methodenaufrufe
            Article a = new Article;
            // Artikel-Instanz füllen und über Bean aktualisieren …
            manager.updateArticle(a);
            // Einen Artikel löschen
            long catalogId = 121212;
            manager.deleteArticle(catalogId);
        …
        } catch (RemoteException e) {
            System.out.println("RemoteException!");
            e.printStackTrace();
        …
        }
        …
    }
}
```

Wie man sieht, muss sich der Programmierer des EJB-Clients nur an den Schnittstellen der EJBean orientieren und diese nutzen. Er muss sich nicht um die Details der Kommunikation zwischen Client und EJB-Container kümmern. Dies wird durch den Stub erledigt.

Auch ein Webservice-Client kann die Methoden einer EJBean über einen Webservice-Endpunkt nutzen.[12] Diese Mechanismen sollen aber hier nicht weiter ausgeführt werden. Weitere nützliche Hinweise zur Programmierung von EJB-Clients sind der EJB-Spezifikation (WWW-004) oder den Produktbeschreibungen der einzelnen Application-Server-Produkte zu entnehmen.

[12] Siehe hierzu die JAX-API.

Timer Service

In EJB wird ein serverseitiger, zeitgesteuerter Event-Benachrichtigungsdienst bereitgestellt, der in EJBeans verwendet werden kann. Man kann damit eine Aktion implementieren, die nach einer vorgegebenen Zeit (Single-Action-Timer) oder periodisch (Intervall-Timer) zum Ablauf gebracht (getriggert) wird. Bei Single-Action-Timern wird ein einziges Timeout-Event erzeugt, bei Intervall-Timern immer wieder eines in vorgegebenen Zeitabständen. Der EJB-Container erzeugt nach der gewünschten Zeit eine sog. *Timer-Notification,* die durch eine entsprechende Methode bearbeitet werden muss.

Der Entwickler muss lediglich einen Timer erzeugen und für diesen eine Aktionsmethode (Callback-Routine) über die vorgegebene *TimeoutObject*-Schnittstelle implementieren. In der Aktionsroutine wird die Business-Logik programmiert, die bei Ablauf des Timers durchlaufen werden soll. Diese Schnittstelle enthält genau eine Methode namens *ejbTimeout*, welche die Aktionsroutine repräsentiert.

Das folgende Codestück zeigt das zu implementierende Interface:

```
…
public interface javax.ejb.TimedObject
{
     // Aktionsroutine mit Geschäftslogik
     public void ejbTimeout(Timer timer);
}
```

Die Erzeugung eines Timers erfolgt über das Interface TimerService. Dieses Interface verfügt über mehrere Erzeugungsmethoden:

```
public interface javax.ejb.TimerService
{
        // Single Action Timer, der an nach einer bestimmten Zeit abläuft
        public Timer createTimer(long duration,
        java.io.Serializable info) throws …;

        // Intervall Timer, der nach einer vorgegebenen Zeit startet
        public Timer createTimer(long initialDuration,
               long intervalDuration,
               java.io.Serializable info) throws …;

        // Single Action Timer, der an einem festen Termin abläuft
        public Timer createTimer(java.util.Date expiration,
               java.io.Serializable info) throws …;

        // Intervall Timer, der an einem festen Termin startet
```

```
        public Timer createTimer(java.util.Date initialExpiration,
                long intervalDuration,
                java.io.Serializable info) throws …;

        // Alle aktiven Timer auslesen
        public Collection getTimers() throws …;
}
```

Mit der ersten *createTimer*-Methode ist es möglich, einen Timer zu erzeugen (Returnwert vom Typ Timer), der nach einer bestimmten Zeit (in ms) abläuft. Mit der zweiten Methode kann ein Intervall-Timer erzeugt werden, der immer wieder in einem angegebenen Abstand abläuft. Bei Ablauf der Zeit wird die Methode *ejbTimeout* des Timer-Objekts aufgerufen (Aktionsroutine), das bei Aufruf einer der *create*-Methoden zurückgegeben wurde.

Sobald ein Timer erzeugt worden ist, wird er persistent gemacht und überlebt auch Systemausfälle.[13] Der Timer selbst ist ein Objekt einer Objektklasse, welche das Interface *javax.ejb.Timer* implementiert. Das Interface sieht wie folgt aus:

```
public interface javax.ejb.Timer
{
    public void cancel() throws …;
    public java.util.Date getNextTimeout() throws …;
    public long getTimeRemaining() throws …;
    public TimerHandle getHandle() throws…;
    …
}
```

Ein Timer kann z.B. über das Timer-Interface wieder entfernt werden. Hierzu wird die Methode *Timer.cancel()* verwendet. Ein Timer-Objekt wird der Methode *ejbTimeout* im *ejbTimeou*t-Interface angegeben.

Beispiel: Die Erzeugung eines Timers und das Definieren einer Aktionsroutine sieht wie folgt aus:

```
…
@Stateless
public class ArticleMgrSessionBean implements …
{
    …
    public void … {
```

[13] Allerdings ist hier nicht ganz klar, was passiert, wenn der Timer während der Ausfallzeit eines Systems abläuft. Es ist nicht ganz eindeutig spezifiziert, ob dann die Timeout-Methode nach dem erneuten Start aufgerufen wird. Dies muss der Hersteller eines Application-Servers definieren.

```
    @Resource javac.ejb.TimerService timerService
    // Timer erzeugen
    timerService.createTimer(…);
  …
  }
  @Timeout
  public void MyTimeoutAction (javax.ejb.Timer timer) {
      …
      // Geschäftslogik beim Ablauf des Timers
  }
}
```

In diesem Beispiel wird in der Artikelmanager-Bean auch das TimedObject-Interface implementiert. Über den Aufruf der Methode *createTimer* wird ein Timer „aufgezogen“ und der Callback-Routine *ejbTimeout* übergeben.

Weitere Details zu Timern insbesondere zur Nutzung von Timern innerhalb von Transaktionen und zur Nutzung in message-driven und stateless Session Beans sind der Literatur zu entnehmen.

Enterprise Naming Context (JNDI ENC) und Bean-Kontext

Eine wichtige Aufgabe eines EJB-Containers ist die Verwaltung von Ressourcen. Jeder EJB-Container verfügt über ein eigenes internes Registry (siehe Abbildung 2-60), das als Enterprise Naming Context (ENC) bezeichnet wird. In diesem Registry werden verschiedenste Referenzen verwaltet, die über das JNDI-API abgefragt werden können.

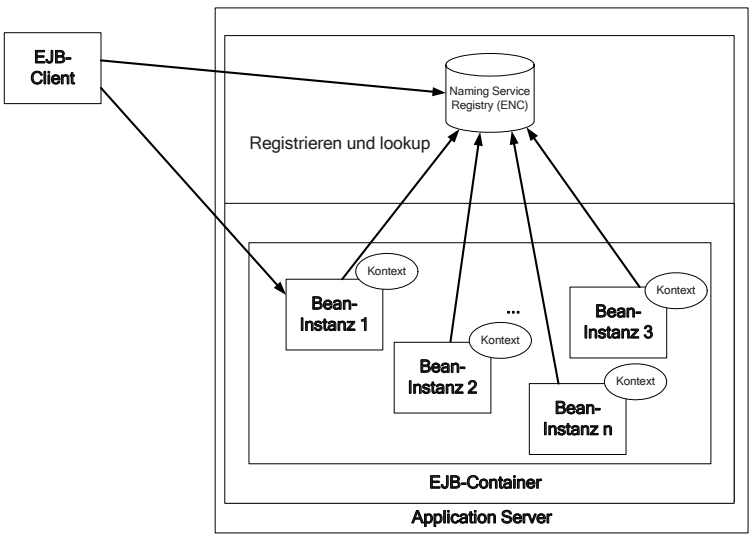


Abbildung 2-60: JNDI ENC im EJB-Container, Kontext je Bean

Das JNDI-API stellt eine Schnittstelle für einen Naming- und Directory-Service zur Verfügung. Sowohl das Registrieren von Ressourcen als auch das Auffinden dieser (Lookup) ist über die API möglich. Referenzen werden zum Teil automatisch vom Container erzeugt, können aber auch explizit angegeben werden und verweisen u.a. auf EJB-Interfaces, JMS-Queues, JMS-Topics und auf Datasources für Datenbankzugriffe. Jedes „Objekt", das dem ENC bekanntgegeben wird, kann über einen JNDI-Lookup-Aufruf abgerufen werden.

Referenzen als Ressourcen injizieren: Das ENC ist bereits seit der ersten EJB-Version vorhanden. Durch Nutzung von Annotationen wurde der Zugang ab EJB 3.0 noch etwas komfortabler gelöst. In EJB 3.0 wurde ein *Injection*-Mechanismus eingeführt, der es ermöglicht, Referenzen direkt in Variablen von Bean-Klassen einzubringen.

Einträge im ENC können im DD oder direkt über eine Annotation bekanntgemacht werden, wobei Einträge im DD Annotationen überschreiben. Für die Bekanntmachung in einem DD stehen verschiedene XML-Elemente zur Verfügung. Eine lokale Interface-Referenz wird z.B. über das XML-Tag *<ejb-name-ref>* in der Konfigurationsdatei *ejb-jar.xml* definiert. Im folgenden XML-Code wird einer SessionBean mit dem Namen *CustomerMgrSessionBean* eine lokale Referenz auf eine *ArticleMgrSessionBean* zugeordnet. Dies bedeutet, die *CustomerMgrSessionBean* kann die *ArticleMgrSessionBean* lokal im Container nutzen. Der Klassenname der referenzierten Bean wird mit dem Tag *<local>* angegeben.

```
<ejb-jar>
      <enterprise-bean>
        <session>
        <ejb-name>CustomerMgrSessionBean</ejb-name>
           …
           <ejb-ref-type>Session</ejb-ref-type>
           <ejb-local-ref>[14]
                <ejb-ref-name>ArticleMgrSessionBean</ejb-ref-name>
                <ejb-ref-type>Session</ejb-ref-type>
                <local>articlemanagement.ArticleMgrSessionBean</local>
                …
           </ejb-local-ref>
        </session>
      </enterprise-bean>
</ejb-jar>
```

[14] Es gibt verschiedene Varianten für Referenzen. In diesem Fall handelt es sich um eine lokale Referenz. Andere Referenzen beziehen sich auf Webservices, MDBs, Datenbanken usw.

In EJB 3.0 können für die Bekanntgabe von Ressourcen die speziellen Annotationen „@RESOURCE“ und „@EJB“ verwendet werden. Nutzt man eine Annotation anstelle eines DD, kann man die Referenz wie folgt im Coding der *CustomerMgrSessionBean* bekanntgeben:

```
import javax.annotation.ejb;
@stateless
@EJB(name=”ArticleMgrSessionBean”,
       beanInterface=ArticleMgrSessionBean.class
       beanName=”ArticleMgrSessionBean”)
public class CustomerMgrSesssionBean implements …
{
   …
```

Ein *lookup*-Aufruf in der *CustomerMgrSessionBean* liefert eine Referenz auf die lokale *ArticleMgrSessionBean*:

```
   public void customerMgrMethod(…) {
        ArticleMgrSessionBean articleMgrBean = null;
        …
        try {
           javax.naming.InitialContext initialContext = new InitialContext();
           articleMgrBean = (ArticleMgrSessionBean) initialContext.lookup
              (“java:comp/env/ejb/ArticleMgrSessionBean”);[15]
        }
        catch … {…}
        // Aufruf einer Methode des Artikelmanagers
        articleMgrBean.createArticle(…)
        …
   }
}
```

Bevor auf die *ArticleMgrSessionBean* zugegriffen werden kann, muss über einen initialen Kontext ein *lookup* durchgeführt werden, der die Referenz besorgt. Danach können Methoden der *ArticleMgrSessionBean* aufgerufen werden (hier *createArticle*). Nachteilig an dieser Vorgehensweise ist, dass überall im Bean-Coding Lookup-Code eingebaut werden muss. Dies widerspricht auch einem grundlegenden Prinzip in der Softwaretechnik und zwar dem Prinzip des „Separation of Concerns“ (Trennung der Zuständigkeiten). Technische Belange sollten von der Geschäftslogik getrennt werden, damit eine mögliche Technologie-Portierung nicht unnötig erschwert wird (siehe Kapitel 5).

Um das Problem zu vermeiden nutzt man heute auch gerne *Dependency Injection* (DI) als Spezialform des Inversion-of-Control-Prinzips. DI überträgt die Verant-

[15] java:comp/env verweist auf den Environment Naming Context (ENC).

wortung für das Erzeugen und die Verknüpfung von Serverobjekten/-komponenten an ein konfigurierbares Framework.

Über die Annotation „@RESOURCE“ können weitere Referenzen in das Bean-Coding injiziert werden, beispielsweise Referenzen auf verschiedene Kontext-Informationen, die der EJB-Container für Beans verwaltet. Hierzu gehören die Kontexte von Beans (*EJBContext*, *SessionContext*, *MessageDrivenBeanContext*) aber auch der oben im Beispiel verwendete *InitialContext*. Durch Nutzung von „@RESOURCE“ kann das oben gezeigte Code-Beispiel etwas vereinfacht werden:

```
…
@stateless
@EJB(name=”ArticleMgrSessionBean”, …
public class CustomerMgrSesssionBean implements … {
   …
   @RESOURCE private javax.ejb.SessionContext ejbContext;
   public void customerMgrMethod(…) {
        ArticleMgrSessionBean articleMgrBean = null;
        …
        try {
           articleMgrBean = (ArticleMgrSessionBean)
                ejbContext.lookup(“ejb/ArticleMgrSessionBean”);
        }
        catch … {…}
        …
   }
}
```

Da dies immer noch kompliziert ist, gibt es eine einfachere Variante. Eine Referenz auf die *ArticleMgrSessionBean* kann direkt in den Bean-Code der *CustomerMgrSessionBean* injiziert werden. In diesem Fall übernimmt der EJB-Container die Injizierung in die Variable *articleMgrBean* automatisch. Diese Variante kommt dem *Dependency-Injection* schon recht nahe:

```
…
import javax.annotation.ejb;
@stateless
public class CustomerMgrSessionBean implements … {
    @EJB private ArticleMgrSessionBean articleMgrBean;
    …
    articleMgrBean.createArticle(…);
    …
   }
}
```

Zur Injizierung von Referenzen gibt es in EJB 3.0 noch weitere Varianten, die hier nicht mehr ausgeführt werden sollen.

Konfigurationsparameter als Ressourcen definieren und injizieren: Eine interessante Anwendungsmöglichkeit, die die Annotation „@RESOURCE" mit sich bringt, ist die Definition von Variablen, die von außen über den DD initialisiert und „injiziert" werden können. Man definiert z.B. in einer EJBean eine Resource-Variable mit der Bezeichnung *konfigParameter* wie folgt:

```
@stateless
public class ArticleMgrSesssionBean implements … {
        …
        @RESOURCE (name="param1") int konfigParameter;
        …
}
```

Der Inhalt der Variable kann dann über den DD vorbelegt werden. Dazu wird ein Tripel der Form (Name, Typ, Wert) im XML-Tag *<env-entry>* der zugehörigen Bean festgelegt. Der DD kann wie folgt angegeben werden:

```
<ejb-jar>
    <enterprise-bean>
      <session>
      <ejb-name>ArticleMgrSessionBean</ejb-name>
      …
      <env-entry>
           <env-entry-name>param1</env-entry-name>
           <env-entry-type>java.lang.Integer</env-entry-type>
           <env-entry-value>1000</env-entry-value>
      </env-entry>
      …
      </session>
    </enterprise-bean>
</ejb-jar>
```

Der Wert von *param1* (hier 1000) wird bei der Initialisierung des Containers in das ENC geladen und bei jeder Instanzierung einer *ArticleMgrSessionBean* verwendet, um die Variable *konfigParameter* zu setzen. Damit ist eine Möglichkeit gegeben, typische Konfigurationsparameter aus dem Bean-Coding zu eliminieren.

EJBContext: Wie oben bereits angedeutet, verwaltet der Container für alle Bean-Typen eine Kontextinformation, die auch den Bean-Instanzen zur Verfügung gestellt wird. Die zugrundeliegende Schnittstelle hat den Namen EJBContext und ist wie folgt definiert:

```
public interface javax.ejb.EJBContext extends
{
    public Object lookup(String name);
```

```
    …
    public TimeService getTimerService() throws ..;
    …
    // Transaktionsmethoden
    javax.transaction.UserTransaction getUserTransaction() throws ..;
    public Boolean setRollbackOnly() throws ..;
    …
    public EJBHome getEJBHome() throws ..;
    public EJBLocalHome getEJBLocalHome() throws ..;
}
```

Wie schon unter EJB 2.1 kann über den Kontext auf die Home-Objekte der Bean zugegriffen werden. Dazu gibt es die Methoden *getEJBHome* und *getEJBLocal*. Die wichtigste Methode ist die Methode *lookup*. Sie dient dem Auflösen von Referenzen auf Objekte, die im Bean-Coding angesprochen werden. Der Aufruf wird auch über das container-interne ENC bedient. Schließlich kann über *EJBContext* auch auf den Timer-Service und auf den Kontext einer laufenden Transaktion zugegriffen werden. Mit der Methode *setRollbackOnly* kann man für eine Transaktion auch vorgeben, dass der Container ein Rollback durchzuführen hat.

Callbacks und Interceptoren

Der EJB-Standard bietet die Möglichkeit, an vorgegebenen Stellen eigene Methoden einzubringen, die bei Eintreffen bestimmter Ereignisse vom EJB-Container aufgerufen werden. Dies wird über Callbacks und Interceptoren erreicht. Callbacks wurden bereits bei der Diskussion der Bean-Zustandsautomaten angerissen.

Callbacks: Callbacks können prinzipiell für alle Bean-Typen eingesetzt werden. Betrachtet man die Lebenszyklen von stateless und stateful Session-Beans, so sieht man, dass bei einigen Zustandsübergängen in den ausgeführten Aktionen Callback-Methoden aufgerufen werden. Bei einer stateless Session-Bean wird z.B. die mit der Annotation *@PreDestroy* markierte Methode aufgerufen, wenn ein Zustandsübergang von *Ready Pooled* nach *Does Not Exist* durchgeführt wird. Beim Zustandsübergang vom Zustand *Passive* in den Zustand *Method-Ready* ruft der Container bei einer stateful Session-Bean die über die Annotation *@PreActivate* definierte Methode auf. Weitere Callback-Routinen können über die Annotationen *@PostConstruct* und *@PrePassivate*, usw. angegeben werden. Die Bezeichnungen der Annotationen geben meist schon das Callback-Ereignis an.

Beispiel: Die Angabe einer Callback-Routine für das Ereignis *PreDestroy* sieht wie folgt aus:

```
@Stateful public class ArticleMgrBean implements ArticleMgrRemoteI
{
    …
    public Article findArticleByKey(int catalogId) {…}
```

```
    @PreDestroy endOfArticleManagement() {…}
    …
}
```

Interceptoren: Interceptoren[16] sind zustandslose Klassen und dienen, wie in CORBA, dem Abfangen von Methodenaufrufen, die ein EJB-Client an eine Bean richtet. Es gibt viele Einsatzmöglichkeiten für Interceptoren. Der Client-Request wird abgefangen und man kann vor und nach der Ausführung des Requests eigene Aktionen einbauen. Beispielsweise können ankommende Client-Requests über spezielle Check-Routinen untersucht werden, die Parameter können geprüft werden, eine Zugangskontrolle kann realisiert werden, bestimmte Aufrufe können verhindert werden oder es wird einfach ein Logging bzw. eine Zeitmessung eingebaut. Der Sinn des Interceptor-Konzepts ist es, allgemeine Routinen aus der Business-Logik in zentrale Klassen auszulagern.

Ein EJB-Container ruft wie bei Callback-Methoden einen Interceptor automatisch und zwar nach dem Eintreffen eines Client-Requests und vor der eigentlichen Ausführung auf. Interceptoren werden als Objektklassen programmiert. Die Interceptor-Methode wird über die Annotation *@AroundInvoke* gekennzeichnet. Die Nutzung eines Interceptors muss dann über eine weitere Annotation *(@Interceptor* oder *@Interceptors* für die Angabe einer Liste von Interceptoren) in der nutzenden Bean-Klasse angegeben werden.

Der eigentliche Aufruf des Requests wird im Interceptor-Code über den Methodenaufruf *proceed* ausgeführt.

Beispiel: Zunächst ist eine Interceptor-Klasse zu definieren. Die folgende Klasse *ArticleMgrInterceptor* kann dann von beliebig vielen Bean-Klassen verwendet werden:

```
import javax.ejb.*;
public class ArticleMgrInterceptor {
    @AroundInvoke
    public Object doMyWork (InvocationContext ctx)
    {
       // Tue etwas vor der eigentlichen Bearbeitung des Methodenaufrufs
       return ctx.proceed();// Aufruf der eigentlichen Methode
       …
       // Tue etwas nach der eigentlichen Bearbeitung des Methodenaufrufs
    }
}
```

Die Methode *doMyWork* erhält als Parameter vom Container einen Aufruf-Kontext übergeben, in dem Informationen über den aktuellen Aufrufkontext des Requests

[16] Engl. Interceptor = Abfänger, Abfangjäger; to intercept = abfangen.

enthalten sind. Dieser Kontext wird allen Interceptoraufrufen der gleichen Bean weitergereicht, Beans können aber nicht darauf zugreifen. Der Aufruf-Kontext ist über ein spezielles Interface zugänglich:

```
public interface InvocationContext {
     public Object getBean()
     public Method getMethod()
     public Object[] getParameters();
     public EJBContext getEJBContext();
     public Object proceed();
     …
}
```

Über die Methode *getBean* kann man sich z.B. die aktuelle Instanz der Bean besorgen, die den Request bearbeitet, über die Methode *getMethod* die aktuell aufgerufene Methode. Auch die Methode *proceed* ist in dem Interface definiert. Mit der Methode *getEJBContext* kann man sich im Interceptor auch den aktuellen EJB-Kontext besorgen, der jeder Bean zugeordnet ist und dann Operationen auf den Kontext (z.B. eine Transaktion mit *setRollbackOnly* markieren) ausführen.

Für die Nutzung eines Interceptors ist in der Bean-Klasse, die ihn verwenden soll, ein Verweis auf die Interceptorklasse über die Annotation *@Interceptor* einzubauen. Werden mehrere Interceptoren zugeordnet, wird die Annotation *@Interceptors* verwendet. Dabei werden die Klassennamen der Interceptorklasse angegeben:

```
import javax.ejb.Interceptor;
…
@Stateless
// Interceptorklasse nutzen, referenziert durch den Klassennamen
@Interceptor(ArticleMgrInterceptor.class)
public class ArticleMgrSessionBean implements …{
     …
}
```

In diesem Fall wird der Interceptor der ganzen Bean-Klasse und damit allen Methoden der Bean-Klasse zugeordnet. Es ist auch möglich einzelnen Methoden dediziert einen Interceptor zuzuordnen. In diesem Fall sieht die Definition wie folgt aus:

```
import javax.ejb.Interceptor;
…
@Stateless
public class ArticleMgrSessionBean implements …{
     …
     // Interceptor wird einer Methode zugeordnet (hier updateArticle)
     @Interceptor(ArticleMgrInterceptor.class)
     public void updateArticle(Article a) {…}
}
```

Der konkrete Ablauf für das Beispiel sieht bei Aufruf einer Methode (hier *updateArticle*) wie folgt aus:

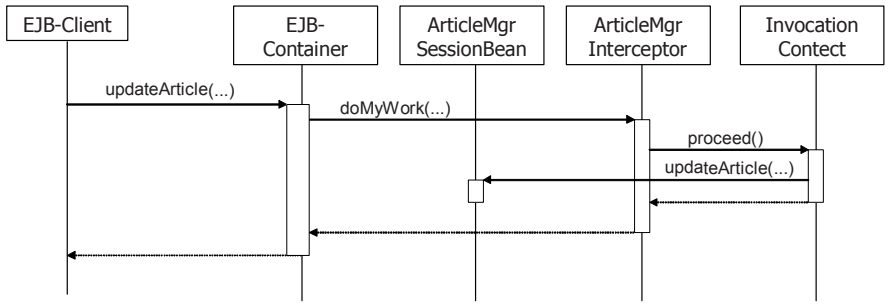


Abbildung 2-61: Interceptor-Einbindung bei Requestbearbeitung in EJB 3

Wie man sieht, wird der Interceptor-Aufruf im aktuellen Stack der Requestbearbeitung aufgerufen. Er erhält damit auch mögliche Exceptions der Bean-Methode (hier der Methode *updateArticle*) als Ergebnis des Aufrufs *proceed*, die bearbeitet werden müssen. Auch der aktuelle Transaktionskontext wird nicht verlassen.

Als weitere Variante, einen Interceptor zu nutzen, soll noch die Möglichkeit aufgezeigt werden, eine Interceptor-Methode direkt in eine Bean-Klasse einzubauen:

```
import javax.ejb.Interceptor;
…
@Stateless
public class ArticleMgrSessionBean implements …{
     …
     // Interceptormethode direkt in der Bean-Klasse
     @AroundInvoke
     public Object doMyWork (InvocationContext ctx) {…}
     …
}
```

In diesem Fall wird die Interceptor-Methode als letzter Interceptor einer möglichen Liste von Interceptoren aufgerufen.

Weitere interessante Features sind die Definition von Default-Interceptoren, die Zuordnung einer Interceptor-Klasse an eine ganze Bean (alle Methoden der Bean) oder sogar an alle Bean-Klassen und das Unterdrücken von Interceptoren für bestimmte Methodenaufrufe. Es soll noch erwähnt werden, dass Interceptor-Methoden auch andere Beans nutzen, JMS-Nachrichten versenden oder über JDBC bzw. den Entity-Manager auf eine Datenbank zugreifen können. Ein Interceptor hat also Zugriff auf alle Resourcen und kann sich die erforderlichen Referenzen auch über das JNDI-API besorgen. Interceptoren können anstelle der Nutzung von Annotationen auch über den Deloyment-Descriptor außerhalb des Codings defi-

niert werden. Hierzu wird auf die Literatur verwiesen (Burke 2006). Die Zuordnung einer Interceptor-Klasse auf alle Bean-Klassen erfolgt am besten über den DD.

Einige EJB-Programmierregeln und Erfahrungswerte

Wie wir gesehen haben, sind bei der Programmierung von EJBeans eine Reihe von Konventionen und Regeln zu beachten. Programmier müssen definierte Verträge (Contracts) einhalten, damit sich die EJBeans auch EJB-konform verhalten.

Zudem gibt es noch einige allgemeine Programmierkonventionen, die darauf zurückzuführen sind, dass der Container die Kontrolle über alle EJBeans behalten muss. Vor allem kann man nicht davon ausgehen, dass die EJBeans vom Container in genau einer JVM gestartet werden und damit immer eine prozess-interne Kommunikation stattfindet. Dies kann der Hersteller eines EJB-Containers selbst entscheiden.

Programmierer von EJBeans (Bean-Provider) müssen also einige Enschränkungen akzeptieren. Zu diesen gehören u.a.:

- Es dürfen keine Synchronisationsprimitive von Java (*wait*, *notify*, *notifyAll*, *join*,…) verwendet werden. Der Grund dafür liegt darin, dass nicht festgelegt ist, in welchen JVMs welche Bean-Instanzen zum Ablauf kommen. Diese Methoden funktionieren nur innerhalb der gleichen JVM.
- Alle statischen Felder sind aus dem gleichen Grund als *final* zu deklarieren.
- Es dürfen auch keine eigenen Threads erzeugt werden. Dies ist Aufgabe des Containers.
- *sleep*-Aufrufe bringen den Container ebenfalls aus dem Konzept.
- Eine Nutzung des Package *java.io* und damit eine eigenständige Nutzung von Ein-und Ausgabeoperationen (etwa für das Fileshandling) ist in einer EJBean nicht zugelassen. Stattdessen ist ein entsprechendes Ressourcenmanager-API zu nutzen (wie JDBC für Datenbanken).[17]
- Socketaufrufe wie *listen* sind nicht gestattet, das Erzeugen eines aktiven Sockets ist aber zulässig.

Auch aus Sicherheitsgründen gibt es einige Einschränkungen, die der EJB-Spezifikation entnommen werden können. Generell gilt, dass der Container die Kontrolle hat und nach jedem Methodenaufruf diese auch schnell wieder erhalten muss, um effizient zu sein. Sonst kann es passieren, dass manche Clients zu lange auf die Bearbeitung ihrer Methodenaufrufe warten müssen.

Es gibt auch schon einige Erfahrungswerte für die Nutzung der verfügbaren Bean-Typen, die aber keine Vorgabe im Sinne der EJB-Spezifikation darstellen. So kön-

[17] Für das Filehandling ist keine API definiert. Hierfür ist ein Connector entsprechend der JCA-API (Java Connector API) zu entwickeln. File-Connectors sind im Open-Source-Umfeld verfügbar.

nen Session-Beans Anwendungsfälle oder Teile davon repräsentieren oder Service-Klassen darstellen. Sie sollten vorzugsweise eingesetzt werden, um Dienste eines Anwendungssystems (Business-Logik) bereitzustellen. Stateful Session-Beans sind notwendig, wenn EJB-Clients unbedingt mehrere zusammenhängende Methoden ausführen müssen, die sich einen bestimmten Session-Kontext teilen. Wenn möglich sollte man sie aus Gründen der Robustheit eines Systems vermeiden, was aber nicht immer möglich ist. Zumindest sollte man aber einen minimalen Einsatz von stateful Session-Beans anstreben, der sich auf die Verwaltung spezieller Session-Kontexte beschränkt. Der Einsatz von MDBs und Entity-Beans wird weiter unten erläutert.

2.5 Message-Passing

2.5.1 Grundlegendes Modell

Nicht alle Anwendungen lassen sich auf das asymmetrische Client-Server-Modell abbilden. Oft ist auch eine gleichberechtigte (symmetrische) Beziehung zwischen den Kommunikationspartnern oder aber eine andere Art der Kommunikation erforderlich. Folgende Kommunikationsbeziehungen sind u.a. möglich:

- Eine Gruppe von Kommunikationspartnern (oder nur einer) möchte etwas konsumieren, was andere produzieren. Hier spricht man von einem Produzenten-Konsumenten-Modell (Producer-Consumer), das oft auch im lokalen Bereich als klassisches Synchronisationsproblem vorkommt. Das Modell ist auch als Publish-Subscribe-Modell bekannt.
- Ein Partner sendet an viele andere bei Auftreten eines Ereignisses bestimmte Nachrichten. Dies ist auch eine Art Produzenten-Konsumenten-Modell mit nur einem Produzenten. Die Konsumenten möchten erst dann benachrichtigt werden, wenn eine Nachricht zur Bearbeitung ansteht. Eine Erweiterung dieser Kommunikationsart ist, dass alle Nachrichten auf alle Fälle ankommen müssen und evtl. sogar in der richtigen Reihenfolge.
- Ein Teilnehmer möchte eine Nachricht an einen anderen Teilnehmer senden, der nicht zwangsläufig aktiv ist. Die Nachricht soll zwischengespeichert werden und wenn der Empfänger wieder aktiv ist, dann soll sie ihm zugestellt werden. Diese Variante ist auch über Systemausfälle hinweg denkbar, die Nachrichten sind in diesem Fall persistent zu machen.

Man spricht in diesem Zusammenhang auch vom Point-to-Point-Modell (P2P) und vom Publish-Subscribe-Modell. Bei einer P2P-Kommunikation kommunizieren zwei Kommunikationsparter gleichberechtigt (symmetrisch) und keiner ist, wie im asymmetrischen Client-Server-Modell, in einer speziellen Rolle. Im Publish-Subscribe-Modell registrieren (subscribe) sich Konsumenten von bestimmten Nachrichtentypen für einen bestimmten Kommunikationsaspekt (Topic). Produ-

zenten erzeugen diese Nachrichtentypen und senden (publish) sie an die Konsumentengruppe.

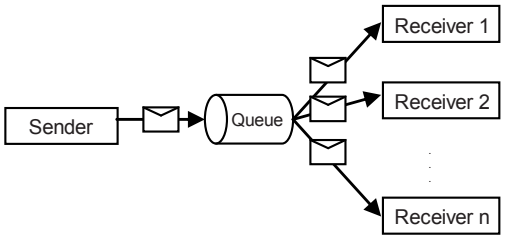


Abbildung 2-62: Point-to-Point-Modell

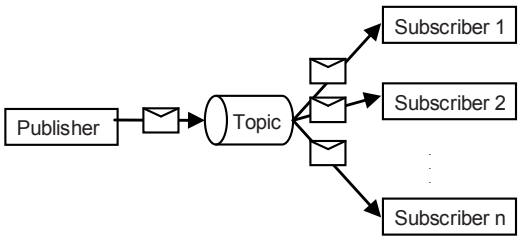


Abbildung 2-63: Publish-/Subscribe-Modell

2.5.2 Klassifikation nachrichtenorientierter Kommunikation

Diese verschiedenen Spielarten der Kommunikation werden – sofern sie nicht in das Client-Server-Modell passen - auch unter dem Begriff der *nachrichtenorientierten Kommunikation* oder unter dem Begriff *Message-Passing* zusammengefasst. Die Systeme, die in der Regel über bestimmte Middleware-Lösungen implementiert sind, werden auch als *Nachrichtensysteme* bezeichnet. Spezielle Nachrichtensysteme sind *Nachrichtenwarteschlangensysteme* (*Message-Queuing*-Systeme). Letztere unterstützen z.B. die Kommunikation, auch wenn gerade ein Partner oder mehrere Partner nicht aktiv sind, im sog. *store-and-forward*-Mechanismus. Sie werden auch häufig eingesetzt, wenn Prozesse (ohne Einfluss der Benutzeroberfläche) miteinander kommunizieren müssen. Wichtigster Aufgabenbereich ist die Integration von Anwendungen in betrieblichen Informationssystemen, wenn z.B. eine Anwendung eine Transaktion anstößt, die auch weitere Transaktionen in anderen Systemen bzw. deren Datenbanken erfordert.

Nachrichtenwarteschlangen können auch persistent (dauerhaft) sein, also Systemausfälle oder das bewusste Stoppen eines Teilnehmersystems tolerieren. Ein typisches Beispiel für ein verteiltes Anwendungssystem, das persistente Warteschlangen verwendet, ist das heute weit verbreitete E-Mail-System.

Es gibt zu Message-Passing-Systemen keine einheitliche Theorie. Tanenbaum unterscheidet bei diesen Systemen generell *transiente* und *persistente* sowie (wie bei

der Client-Server-Kommunikation) *synchrone* und *asynchrone* Kommunikation. Diese Unterscheidung soll uns hier als Diskussionsgrundlage dienen (Tanenbaum 2003):

- Wie bereits erläutert kann der Sender bei der asynchronen Kommunikation nach dem Absenden einer Nachricht sofort weiterarbeiten, ist also nicht blockiert. Die Nachricht wird dem Empfänger z.B. über eine Middleware zugestellt. Bei der synchronen Kommunikation wird der Sender blockiert, bis der Empfänger die Nachricht erhalten bzw. gar verarbeitet hat oder diese zumindest im Empfängerrechner angekommen ist.
- Bei transienter Kommunikation wird eine Nachricht nur so lange gespeichert, wie das verteilte System (bzw. Sender und Empfänger) aktiv ist. Im persistenten Fall dagegen wird eine Nachricht solange gespeichert, bis sie beim Empfänger oder den Empfängern ausgeliefert ist.

Kombiniert man nun die Attribute, die die Kommunikationsart beschreiben, so können folgende Fälle vorkommen:

- *Transient asynchrones* Message-Passing ist dem Einwege-RPC sehr ähnlich. Eine gesendete Nachricht wird vom Kommunikationssystem lokal (in einem Puffer) abgelegt. Der Sender arbeitet weiter und das Kommunikationssystem sendet die Nachricht parallel dazu zum Empfänger. Die Datagrammorientierte Kommunikation über UDP auf Basis von Datagram-Sockets kann hier als Beispiel dienen. Bei Systemausfällen ist kein Recovery möglich, um den Zustand vor dem Ausfall wiederherstellen zu können.
- Bei der *transient synchronen* Kommunikation gibt es mehrere Formen. In der strengsten Form blockiert der Sender bis das Ergebnis vom Empfänger zurück ist. In der einfacheren Form wird der Sender so lange blockiert, bis die Nachricht im Puffer des Empfängers gelandet ist. Systemausfälle überlebt das System nicht.
- Bei der *persistent synchronen* Kommunikation speichert das Kommunikationssystem eine Nachricht beim Empfänger (in einem Puffer) ab, bevor der Sender weiterarbeiten kann.
- Beim *persistenten asynchronen* Message-Passing wird eine gesendete Nachricht dauerhaft vom Kommunikationssystem zwischengespeichert, bis der adressierte Empfänger sie liest, während der Sender gleich weiterarbeiten kann. Eine Anwendung dieses Typs ist wiederum das E-Mail-System der Internet-Welt mit SMTP (Simple Message Transfer Protocol) als Protokoll.

Die Begriffswelt ist bzgl. der Kommunikationsarten uneinheitlich. Weber unterscheidet in (Weber 1998) im Gegensatz zu Tanenbaum (Tanenbaum 2003) zwischen *dienst- und meldungsorientierter* Kommunikation und diese beiden Begriffe werden mit den Attributen synchron und asynchron kombiniert. Im Prinzip ist die Unterscheidung nicht so wesentlich, da man anhand der Anwendung am besten ersehen kann, welchen Kommunikationsmechanismus man braucht, wie er heißt ist sekundär. Welche Variante verwendet wird, hängt von der

konkreten Anwendung ab und ist vom Architekten der verteilten Anwendung zu entscheiden.

Neu an der Klassifikation von Tanenbaum sind die Attribute *transient* und *persistent*. Die transiente Kommunikation lässt sich mit Hilfe der klassischen Transportzugriffsmechanismen wie die Berkeley-Sockets, über die von AT&T entwickelte TLI (Transport Layer Interface) oder über XTI (X/Open Transport Interface), einem Standard der ehemaligen X/Open[18], realisieren. Auch das hier nicht weiter diskutierte MPI (Message-Passing-Interface) mit über 100 hochwertigen Funktionen für das Message-Passing dient ähnlichen Zwecken. MPI ist ein Standard für ein Message Passing Interface, der heute auch in Grid-Systemen Anwendung findet (Bauke 2006). Der Standard wurde von der University of Tennessee, Knoxville sowie von vielen Herstellern und Anwendern Mitte der 90er Jahre entwickelt. MPI ist frei verfügbar und steht als Bibliothek zur Verfügung. Die Bibliothek stellt Funktionen wie MPI_send, MPI_recv usw. bereit. Weitere Informationen hierzu finden sich in der Home-Page des MPI-Forums (WWW-008) und in (Tanenbaum 2003).

2.5.4 Persistente Kommunikation

Die nachrichtenorientierte persistente Kommunikation soll im Weiteren etwas ausführlicher betrachtet werden. Diese Form der Kommunikation wird über Nachrichtenwarteschlangensysteme oder sog. MOM-Systeme (Message-Oriented Middleware) abgewickelt. Diese Systeme stellen Mechanismen zur Nachrichtenübertragung im Wesentlichen für die persistente asynchrone Kommunikation bereit. Als Abgrenzung zu den klassischen RPC-Mechanismen und der objektbasierten Kommunikation kann man festhalten, dass bei nachrichtenorientierter persistenter Kommunikation keine entfernten Funktions- oder Methodenaufrufe unterstützt werden, sondern die Nachrichten durch die Kommunikationspartner explizit aufzubauen und zu übertragen sind.

Nachrichtenwarteschlangensystem: In einem Nachrichtenwarteschlangensystem kommunizieren die beteiligten verteilten Anwendungsbausteine nicht direkt miteinander. Die Kommunikation erfolgt indirekt über Nachrichtenwarteschlangen, in die ein Sender eine Nachricht einträgt und ein Empfänger diese abholt. Der Sender erhält die Zusicherung, dass seine Nachricht irgendwann in die Warteschlange des Empfängers gestellt wird, wann dies geschieht, ist nicht festgelegt und wann der Empfänger diese ausliest ebenfalls nicht. Die Verwaltung der Warteschlangen übernimmt das Kommunikationssystem bzw. die Middleware.

Die allgemeine Architektur eines Warteschlangensystems kann nach Tanenbaum wie folgt skizziert werden (vgl. Abbildung 2-64):

[18] X/Open war eine Herstellervereinigung, die in die Open Group eingegangen ist.

- *Quell- und Zielwarteschlangen,* wobei jede Warteschange einen Namen bzw. eine Adresse hat. Die Quellwarteschlange ist örtlich in der Nähe des Senders, evtl. sogar auf dem gleichen Rechnersystem. Eine Nachricht enthält die Adresse der Zielwarteschlange, an die sie übertragen werden soll. Das Nachrichtenwarteschlangensystem ist für die Bereitstellung von Warteschlangen zuständig.
- *Warteschlangen-Manager* verwalten die Warteschlangen. Sie kommunizieren direkt mit den Anwendungsbausteinen (im Bild die Applikationen), die Nachrichten senden und empfangen. Sender stellen ihre Nachrichten immer in eine lokale Warteschlange oder eine Warteschlange, die sich in der Nähe des Senders befindet, ein. Die Warteschlange wird vom lokalen Warteschlangen-Manager verwaltet.
- Spezielle Warteschlangen-Manager arbeiten als *Router* (auch *Relays* genannt). In dieser Funktion geben Sie eingehende Nachrichten an andere Warteschlangen-Manager, die auf dem Weg zum Ziel liegen, weiter.
- Die Applikationen können über eine einfach zu bedienende Schnittstelle (API) mit Operationen wie *put* und *get* (entspricht *send* und *receive*) Nachrichten in die Quellwarteschlange einstellen bzw. aus der Zielwarteschlange auslesen. Weiterhin können Sie die Zielwarteschlange auf anstehende Nachrichten abfragen (Operation *poll*) bzw. auf ankommende Ereignisse (Nachricht in der eigenen Zielwarteschlange) über die Bekanntgabe einer selbst definierten Funktion mit Hilfe der Operation *notify* reagieren.

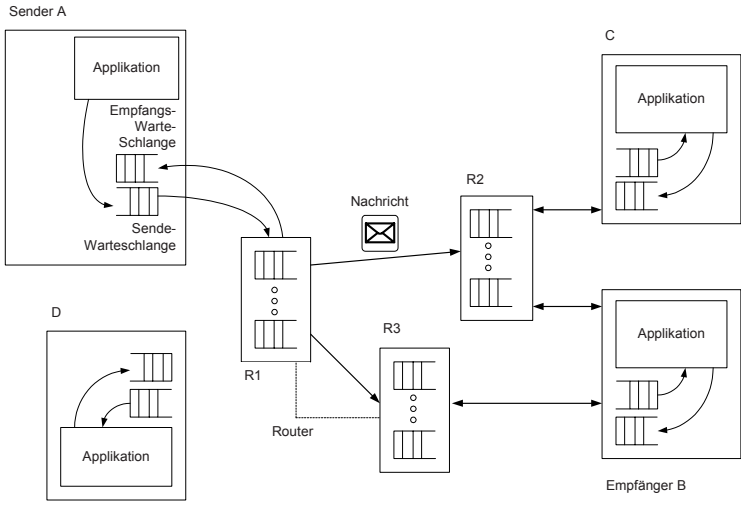


Abbildung 2-64: Allgemeiner Aufbau eines Nachrichtenwarteschlangensystems nach (Tanenbaum 2003)

In Abbildung 2-64 ist ein Nachrichtenwarteschlangensystem mit vier Rechnersystemen A, B, C, und D, auf denen jeweils eine Applikation mit zugeordneten Quell- und Zielwarteschlangen abläuft, skizziert. Weiterhin sind drei Router R1, R2 und R3 eingezeichnet. Diese Architektur ist vom Grundaufbau her dem Mbone-Netz für Multicasting im Internet sehr ähnlich.

Beispiele für MOM-Produkte sind IBM Websphere *MQ*, Microsoft *MSMQ* und BEA *MessageQ*. Interessant ist in diesem Zusammenhang auch die Java-API *JEE/JMS (*Java Message Service), die eine Schnittstelle für Message-Passing-Systeme bereitstellt und auch die bereits erwähnten *Message-driven Beans* (MDB) der EJB-Spezifikation, die nachrichtenorientierte Komponenten darstellen. Wir betrachten im Weiteren JMS und MDBs als Fallbeispiele.

2.5.6 Fallbeispiel: Java Messaging Service

JMS ist eine Java API, die einen Satz allgemeingültiger Schnittstellen für die Kommunikation über MOM-Systeme bereitstellt. Sowohl die Point-to-Point- (P2P) als auch die Publish-/Subscribe-Kommunikation (Pub/Sub) wird über JMS ermöglicht. JMS stellt allerdings nur eine API für den Zugang zu MOM-Systemen bereit. Die Implementierung eines MOM-Systems und auch die Kommunikation zwischen MOM-Systemen werden nicht festgelegt. Es gibt bis heute kein Standardprotokoll, das alle Hersteller von MOM-Systemen nutzen, um miteinander zu kommunizieren.

Die JMS-Spezifikation legt fest, dass ein System, welches die JMS-API bereitstellt, als JMS-Provider bezeichnet wird. Eine JMS-Anwendung, die über JMS kommunizieren möchte, muss sich mit dem JMS-Provider verbinden, also eine *Connection* einrichten. Eine JMS-Anwendung wird auch als JMS-Consumer bezeichnet. Zusätzlich muss eine Session erzeugt werden. Connections und Sessions bilden also die Basis für die Kommunikation zwischen JMS-Anwendung und JMS-Provider. Sessions sind unabhängig voneinander. Jede Anwendung baut eine eigene Session mit dem JMS-Provider auf. Eine Connection kann als Kommunikationskanal zum JMS-Provider verstanden werden, während eine Session einen Kommunikationskontext für Sender, Empfänger und Nachrichten repräsentiert (Abbildung 2-65).

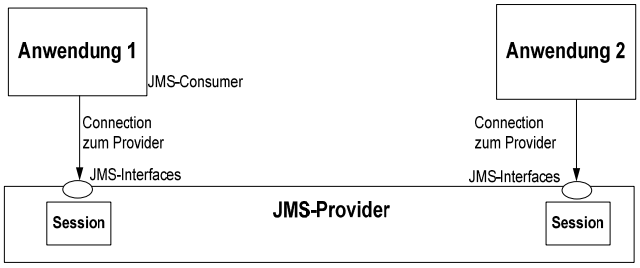


Abbildung 2-65: Kommunikation zwischen Anwendung und JMS-Provider

Ein JMS-Provider bietet Kommunikationsdienste an, die in den Schichten 5 – 7 des ISO/OSI-Referenzmodells anzusiedeln sind und setzt auf einen Transportdienst wie etwa TCP auf (siehe Abbildung 2-66). In der Abbildung wird noch deutlicher, dass jeder CMS-Consumer völlig unabhängig von anderen Consumern eine Verbindung und eine Session mit dem JMS-Provider unterhält. Die JMS-Anwendungen sind in einer Client-Server-Beziehung mit dem JMS-Provider und stellen in dieser Beziehung den Client dar, während der JMS-Provider in der Serverrolle ist. Die Abbildung zeigt auch deutlich die völlige Entkopplung der Anwendungen, die miteinander Nachrichten austauschen wollen.

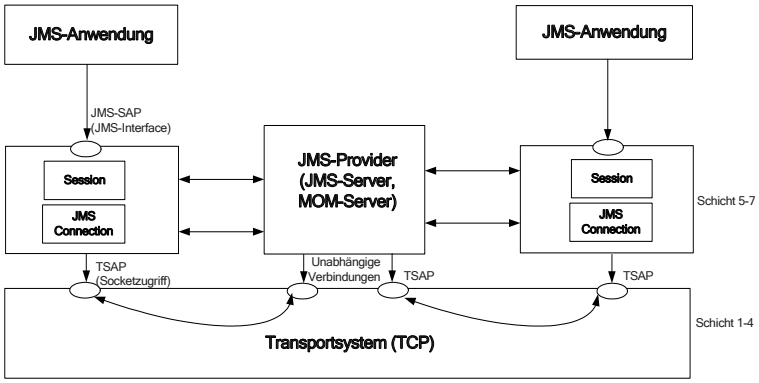


Abbildung 2-66: Kommunikation bei JMS

JMS unterstützt sowohl die Kommunikation im Point-to-Point- als auch im Publish-Subscribe-Modell. Der JMS-Provider stellt hierfür die nötigen Ressourcen bereit. Er verwaltet Queues und Topics, die auch als administrative Objekte bezeichnet werden (siehe Abbildung 2-67).

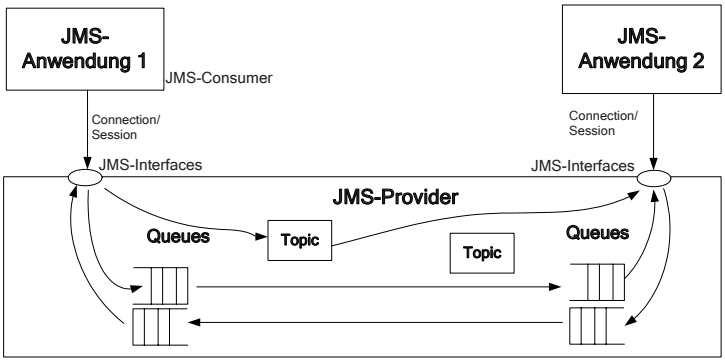


Abbildung 2-67: Topics und Queues als administrative Objekte

Über JMS können aber auch *temporäre* Topics und Queues angelegt werden (Methoden *createTemporaryTopic* und *createTemporaryQueue*).

JMS Point-to-Point-Modell

Das JMS-P2P-Modell legt fest, wie eine Anwendung mit Warteschlangen (*Queues*) arbeiten kann. Dazu gehört das Auffinden von Queues sowie das Senden und Empfangen von Nachrichten in/aus Queues.

JMS-P2P definiert keine Mechanismen für das Erzeugen und Administrieren von Warteschlangen (Queues). Eine Ausnahme bilden temporäre Queues. Das Anlegen von dauerhaften Queues ist Aufgabe des Administrators eines Nachrichtenwarteschlangensystems. Eine Queue wird üblicherweise durch einen Administrator erzeugt und lebt dann für einen längeren Zeitraum. Queues werden in der Regel vom JMS-Provider immer verfügbar gehalten, also dauerhaft eingerichtet, und Anwendungen können Nachrichten ohne Unterbrechung einstellen oder auslesen.

Es ist in JMS nicht definiert, wie lange der Transport einer Nachricht dauert. Die Verzögerung hängt von der Implementierung des JMS-Providers ab.

JMS-PTP bietet zwei verschiedene Typen von Interfaces an: Das *PTP Domain Interface* und das *JMS Common Interface*. Beide Schnittstellen sind semantisch gleich und bieten mehrere Einzel-Interfaces an (siehe Tabelle 2-3).

Über das Interface *QueueConnectionFactory*[19] bzw. *ConnectionFactory* kann eine Verbindung zu einem JMS-PTP-Provider erzeugt werden. Es wird ein Objekt instanziiert, das ein Interface vom Typ *QueueConnection* (bzw. *Connection*) bereitstellt. JMS unterstützt auch temporäre Queues (WWW-004). Ein Objekt, das ein *Queue*- bzw. ein *Destination*-Interface bereitstellt, kapselt eine Queue. Über dieses Interface kann eine Anwendung eine in einem Nachrichtenwarteschlangensystem eingerichtete Queue ansprechen.

Tabelle 2-3: JMS-PTP-Interfaces

PTP Domain Interfaces	JMS Common Interfaces
QueueConnectionFactory	ConnectionFactory
QueueConnection	Connection
Queues	Destination
QueueSession	Session
QueueSender	MessageProducer
QueueReceiver	MessageConsumer

[19] Hier wird das Factory-Pattern genutzt.

Über eine *Connection* können eine oder mehrere *Sessions* zum JMS-Provider angelegt werden. Sessions stellen das Interface *QueueSession* (bzw. *Session*) zur Verfügung. Innerhalb einer Session kann man Objekte mit dem Interface *QueueReceiver* (bzw. *MessageConsumer*) und *QueueSender* (*MessageProducer*) erzeugen, über die sowohl aus Queues gelesen als auch in Queues geschrieben werden kann.

JMS Publish-/Subscribe-Modell

Das JMS-Pub/Sub-Modell stellt einen abstrakten, allen Beteiligten bekannten „Knotenpunkt" zur Verfügung, der als *Topic* bezeichnet wird. Dieser Knotenpunkt kann Nachrichten empfangen und weiterleiten. Ein Topic kann auch als Message-Broker verstanden werden, der Nachrichten sammelt und wieder verteilt. Die Nutzer eines Topics heißen auch Produzenten und Konsumenten. Die Anzahl der Produzenten und Konsumenten kann sich dynamisch verändern.

Man erkennt die Analogie zum PTP-Modell. Die *Common Interfaces* entsprechen den Interfaces des PTP-Modells. Über die *Common Interfaces* kann also sowohl das PTP- als auch das Pub/Sub-Modell genutzt werden (siehe Tabelle 2-4). Im Pub-/Sub-Modell unterscheidet man Produzenten und Konsumenten von Nachrichten, die völlig unabhängig voneinander arbeiten.

Tabelle 2-4: JMS-Pub/Sub-Interfaces

Pub/Sub Domain Interfaces	JMS Common Interfaces
TopicConnectionFactory	ConnectionFactory
TopicConnection	Connection
Topic	Destination
TopicSession	Session
TopicSender	MessageProducer
TopicReceiver	MessageConsumer

JMS-Nachrichten

JMS gibt einen bestimmten Nachrichtenaufbau vor (siehe Abbildung 2-68). Die Nachricht besteht aus einem Header, der sich wiederum aus drei Bestandteilen zusammensetzt, sowie einem Message-Body. Im Header sind vordefinierte JMS-Message-Headerfelder und optionale Headerfelder als Properties zu finden.

Bei den Properties unterscheidet man wiederum vordefinierte JMSX-Properties und eigene Properties, die auch eigene Namen haben können.

Properties sind als (name, value)-Paare mit unterschiedlichen Typen (boolean, byte, long, String, …, Object) definierbar. JMSX-Properties sind in der JMS-Spezifikation vordefinierte Message-Eigenschaften. Beispiele hierzu sind:

- *JMSXUserid*: Eindeutiger String zur Identifikation eines JMS-Clients, wird vom Provider beim Senden gesetzt
- *JMSXApplId*: Identifiziert die Anwendung, wird vom Provider beim Senden gesetzt

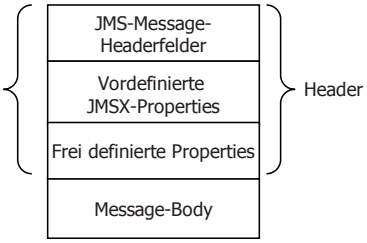


Abbildung 2-68: JMS-Message-Aufbau

Eine Nachricht kann über das vordefinierte Interface *Message* bearbeitet werden. So können z.B. mit der Methode *getPropertyNames* alle Property-Namen aus einer Nachricht ausgelesen werden. Die Methode *clearProperties* dient dem Löschen aller Properties einer Nachricht. Mit *get*- und *set*-Methoden können die Properties gelesen und verändert werden. Die Methode *clearBody* löscht den ganzen Message-Body. Beim Erzeugen einer Nachricht innerhalb einer Session werden schon einige Felder des Headers vorbelegt.

JMS-Message-Header-Felder können ebenfalls mit *get*/*set*-Methoden gelesen und verändert werden In Tabelle 2-5 sind die wichtigsten Header-Felder aufgeführt. Im Property-Teil werden optionale JMS-eigene Felder und Provider-spezifische Felder gesetzt. Weiterhin können eigene, anwendungsspezifische Felder definiert werden.

Der optionale Message-Body beinhaltet die applikationsspezifischen Daten, welche mit der Nachricht übertragen werden sollen. Im JMS-Standard werden aufbauend auf einen generischen Nachrichtentypen (Klasse *Message*) fünf spezielle Nachrichtentypen bereitgestellt:

- *BytesMessage*: Mit diesem Nachrichtentyp können Daten byteweise übertragen werden. Damit lassen sich auch Nachrichten lesen, die keine Standard-JMS-Datentypen nutzen (etwa von Fremdsystemen).
- *MapMessage*: Der Nutzlastteil bei diesem Nachrichtentyp enthält nur Properties mit (name,value)-Paaren. Die Namen werden als Strings übergeben.
- *ObjectMessage*: Dieser Nachrichtentyp erlaubt die Übertragung serialisierter Java-Objekte (Vorsicht: Kompatibilität).
- *StreamMessage*: Mit diesem Nachrichtentyp können große Mengen primitiver Datentypen versendet.
- *TextMessage*: Hier wird die Nachricht als String übertragen.

Tabelle 2-5: JMS-Pub/Sub-Interfaces

Header-Feld	Bedeutung	Wird belegt
JMSDestination	Ziel-Topic oder –Queue	implizit beim Senden
JMSDeliveryMode	Nicht-persistenter und persistenter Modus	implizit beim Senden
JMSMessageID	Eindeutige Nachrichten-ID (String)	implizit beim Senden
JMSTimeStamp	Zeitpunkt der Übergabe an den JMS-Provider)	implizit beim Senden
JMSRedelivered	Hinweis, dass möglicherweise schon ausgeliefert wurde	durch JMS-Provider
JMSRedelivered	Erneute Zustellung einer Nachricht	durch Message-Consumer
JMSExpiration	Angabe, wie lange eine Nachricht leben soll	implizit beim Senden
JMSPriority	Prioritäten, 0-4 = normal, 5-9 = hoch	implizit beim Senden
JMSCorrelationID	Verlinken von Nachrichten möglich	durch JMS-Client
JMSReplyTo	Destination (Topic, Queue), zu der die Antwort geschickt werden soll	durch JMS-Client
JMSType	Frei vergebener Typ der Nachricht (Client vergibt	durch JMS-Client

Filterung von Nachrichten

Ein wichtiger Aspekt bei pub/sub-Anwendungen stellt die Filterung von Nachrichten dar. Wenn mehrere Konsumenten vorhanden sind, stellt sich die Frage, ob alle Nachrichten an alle Konsumenten verteilt werden sollen oder ob eine Filterung bzw. Selektion vorzunehmen ist. Beispielsweise kann es in einem Handelssystem für Derivate sinnvoll sein, dass bestimmte Händler auch nur dedizierte Informationen über bestimmte Aktienkurse oder Kurse anderer Finanzinstrumente zugestellt bekommen.

In JMS wird hierfür der Selektor-Mechanismus bereitgestellt. Dies ist ein Mechanismus zum Filtern von Nachrichten auf der Providerseite, also üblicherweise auf dem JMS-Server. Bei entsprechender Implementierung hat dies den Vorteil, dass auch nur die Nachrichten an einen JMS-Konsumenten zugestellt werden, die er wirklich braucht. Der Selektor-Mechanismus ermöglicht die Definition von Filtern in einer SQL-ähnlichen Notation. In den Filtern werden Bedingungen erfüllt, die sich auf Headerfelder und Properties beziehen, nicht auf den Message-Body. Es können sowohl die JMSX-Properties, als auch die selbst definierten Properties he-

rangezogen werden. Wir werden den Filtermechanismus weiter unten anhand eines Beispiels erläutern.

JMS-Interfaces im Überblick

Die JMS-API stellt einige Basis-Interfaces bereit. Die wichtigsten sind in Abbildung 2-68 zusammengefasst. Dabei ist auch die typische Nutzungsreihenfolge angegeben. Zunächst wird typischerweise über das Interface *ConnectionFactory* ein Topic oder eine Queue erzeugt und anschließend eine *Connection* angelegt. Danach kann über das *Connection*-Interface eine Session erzeugt und innerhalb der Session je nach Anwendungsteil *Producer* oder Consumer instanziiert werden. Diese können dann Message-Objekte erzeugen und versenden sowie, evtl. über ein Listener-Objekt, Messages empfangen.

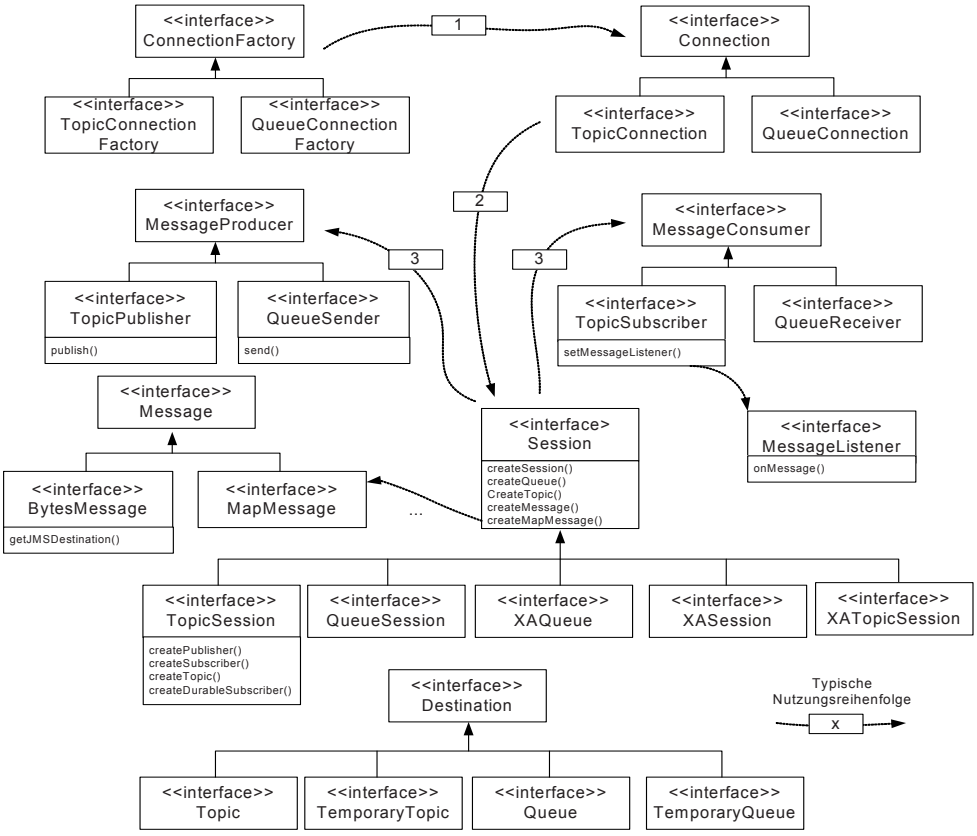


Abbildung 2-69: Interfaces der JMS-API

Wir werden die Nutzung der Basisschnittstellen im Folgenden an weiteren einfachen Beispielen mit konkreten Einsatzszenarien skizzieren.

Beispiel 1 „Producer-Consumer-Lösung über P2P“: Die Nutzungsreihenfolge ist in Abbildung 2-70 am Beispiel einer Producer-Consumer-Lösung über eine Queue skizziert. Im Beispiel kommunizieren ein Produzent und ein Konsument über JMS miteinander. Der Konsument liest Nachrichten synchron aus, d.h. er ist nach Aufruf einer *receive*-Methode blockiert, bis eine Nachricht angekommen ist. Asynchrones Empfangen ist über JMS auch möglich. Produzent und Konsument können völlig unabhängig voneinander arbeiten. Es wird davon ausgegangen, dass das Topic „Auftragsqueue“ bereits vorher vom Administrator erzeugt worden ist. Das vorgestellte Codebeispiel ist stark vereinfacht und enthält keine Ausnahmebehandlung.

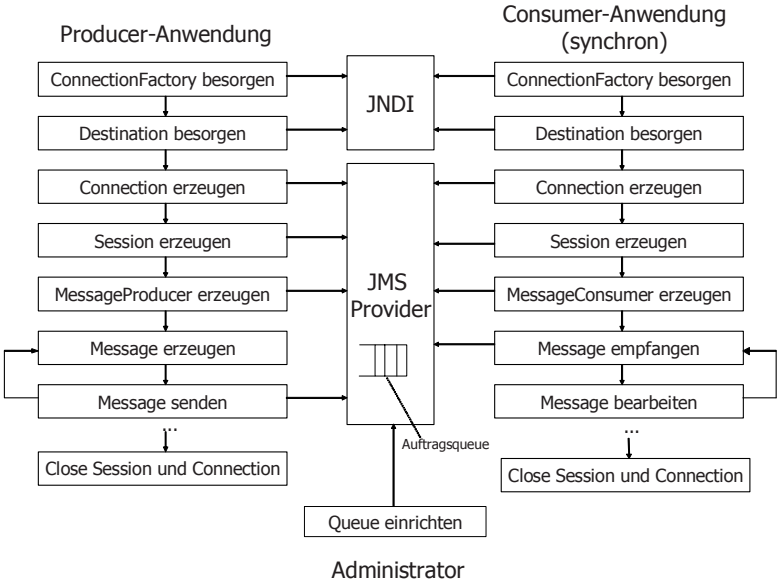


Abbildung 2-70: Ablauf einer Producer-Consumer-Kommunikation über JMS

Zunächst müssen Produzent und Konsument jeweils eine *Connection* erzeugen. Dies geschieht über die *ConnectionFactory*. Um die Factory zu finden, wird JNDI benutzt:

```
import javax.naming.*;
import javax.jms.*;
ConnectionFactory connFactory;
Context messaging = new InitialContext();
connFactory = (ConnectionFactory) messaging.lookup(“ConnectionFactory”);
```

Über einen JNDI-*Lookup* kann auch die Auftragsqueue adressiert werden:

```
Queue auftragQueue;
auftragQueue = (Queue) messaging.lookup("Auftragsqueue");
```

Danach kann eine Verbindung und einer Session angelegt werden: Der erste Parameter der Methode *createSession* gibt im Beispiel an, ob die Session transaktionsgesichert sein soll (*false* bedeutet, dass dies nicht der Fall ist). Der zweite Parameter legt die Vorgehensweise beim Bestätigen einer empfangenen Nachricht fest. In unserem Beispiel werden diese automatisch bestätigt.

```
Connection conn;
conn = ConnectionFactory.createConnection();
Session session;
Session = conn.createSession(false, Session.AUTO_ACKKNOWLEDGE);
```

Über eine Connection können auch mehrere Sessions initiiert werden, jede Session stellt aber für sich einen unabhängigen Produzenten oder Konsumenten dar. Anschließend wird auf der einen Seite ein Produzent, auf der anderen ein Konsument erzeugt.

Auf der Produzentenseite muss durch Aufruf der Methode *createProducer* innerhalb der Session ein Produzent angelegt werden. Dabei wird die Ziel-Queue als Parameter übergeben:

```
MessageProducer sender;
sender = session.createProducer(auftragQueue);
```

Auf der Konsumentenseite wird über den Aufruf der Methode *createConsumer* innerhalb der Session ein Konsument erzeugt. Hier wird ebenfalls die Queue als Parameter übergeben:

```
MessageConsumer receiver;
receiver = session.createConsumer(auftragQueue);
```

Danach können vom Produzenten Nachrichten erzeugt und diese über die definierte Queue den JMS-Provider zur Weiterleitung übergeben werden. Dies wird hier am Beispiel von einfachen Text-Nachrichten gezeigt. Vorher muss der Nachrichtenfluss noch gestartet werden.

```
connection.start();
String auftrag;
// Auftrag befüllen ...
TextMessage auftragsMessage;
auftragsMessage = session.createTextMessage();
auftragsMessage.setText(auftrag);
sender.send(auftragsMessage);
```

Auf der Konsumentenseite wird die Nachricht (hier synchron) empfangen und ausgepackt. Der Konsument wartet in diesem Beispiel genau 4 Sekunden (4000 ms) synchron auf eine Nachricht. Ist keine Nachricht da, wird der *receive*-Aufruf nach dieser Zeit abgebrochen:

```
TextMessage auftragsMessage;
auftragsMessage = (TextMessage)receiver.receive(4000);
String auftrag;
Auftrag = auftragsMessage.getText();
```

Der Produzent und der Konsument können unabhängig voneinander die Session und die Connection am Ende ihrer Kommunikationsabläufe wieder abbauen.

```
session.close();
conn.close();
```

Beispiel 2 „Durable Subscriber für ein Topic (pub/sub)“: In diesem Beispiel wird ein dauerhafter Subcriber skizziert, also ein Konsument, der sich an ein Topic anmeldet und den Abonnement-Modus (Diskussion erfolgt weiter unten) auf „durable“ setzt. Der Konsument erhält alle Nachrichten des Topics, auch wenn er zwischenzeitlich nicht aktiv ist. Der JMS-Provider speichert die Nachrichten solange, bis sie abgeholt wurden.

Bei Abmelden und bei erneuter Anmeldung ist es wichtig, dass der JMS-Provider den Konsumenten wiedererkennen kann. Dazu muss die gleiche Verbindung mit der Angabe der gleichen Destination angegeben werden. Auch evtl. Message-Selektoren müssen identisch sein. Die Zusammenhänge des Beispielszenarios sind in Abbildung 2-71 skizziert.

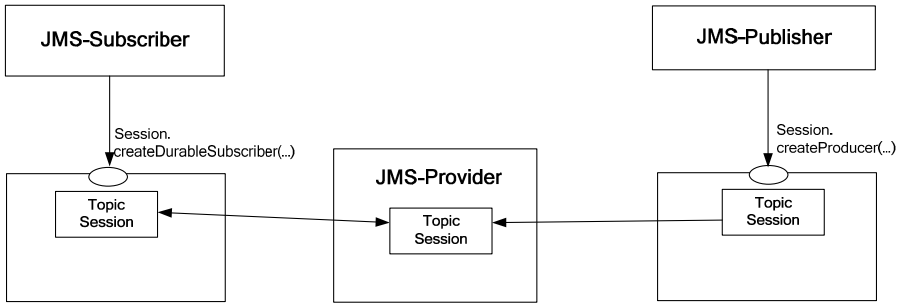


Abbildung 2-71: Durable Subscriber

Die Anmeldung beim Topic erfolgt mit der Methode *createDurableSubscriber* und wird im Folgenden stark vereinfacht gezeigt. Es wird ein dauerhafter Abonnent erzeugt und ein Listener eingerichtet, der die im Topic ankommenden Nachrichten bearbeitet.

```
TopicConnection topicConnection = null;
TopicSession topicSession = null;
Topic topic = null;
TopicSubscriber topicSubscriber = null;
TextListener topicListener = null;
…
```

```
// JNDI abfragen ….
// Verbindung aufbauen …
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(false,
       Session.AUTO_ACKNOWLEDGE);
topicSubscriber = topicSession.createDurableSubscriber(topic,
       "MyDurableTopic");
// Topic-Listener erzeugen
topicListener = new TextListener();
topicSubscriber.setMessageListener(topicListener);
topicConnection.start();
// Nachrichten können nun über das Topic empfangen werden
```

Auf der Produzentenseite ist entsprechend ein Produzent zu erzeugen, der Nachrichten in das Topic schreibt.

Beispiel 3 „Filterung von Nachrichten bei Topics (pub/sub)“: Das Filtern von Nachrichten bei Nutzung von Topics kann über die Definition eines eigenen Topics für jedes auszufilternde Themengebiet oder über den Einsatz eines Message-Selektors erfolgen. Die Filterung über Topics kann aufwändig werden, wenn es viele verschiedene Filterkriterien gibt, da dann für jedes Kriterium ein Topic erzeugt werden muss. Eine andere Variante wäre, die Nachrichten im Konsumenten über spezielle Nachrichteninhalte direkt im Konsumenten auszufiltern. Dies hat aber den Nachteil, dass alle Nachrichten an den Konsumenten gesendet werden, obwohl er gar nicht alle benötigt. Eine Selektion im JMS-Provider ist also sinnvoller.

In Abbildung 2-72 ist die Filterung über Topics und über einen Message-Selektor skizziert.

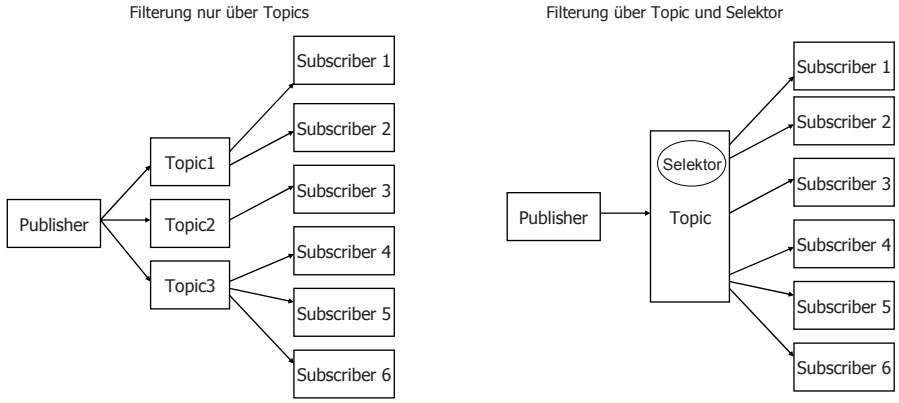


Abbildung 2-72: Filterung von Nachrichten bei Topics

Wir betrachten im Folgenden die Nutzung eines einfachen Selektors für die Auswahl bestimmter Aktienkurse. Im Coding aller Kommunikationspartner sind zunächst eine Destination (StockSource), eine Connection und eine Session zu erzeugen:

```
…
Context c = new InitialContext();
ConnectionFactory cf = (ConnectionFactory) c.lookup("ConnectionFactory");
Queue stockQueue = (Queue)c.lookup("StockSource");
Connection connection = ConnectionFactory.createConnection();
Session session;
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Im Produzentencode sind Vorbereitungen für das Senden von Nachrichten zu treffen. Ein Produzent ist anzulegen (*createProducer*):

```
MessageProducer sender = session.createProducer(stockQueue);
String boersenkurse;
TextMessage message;
message = session.createTextMessage();
message.setText(boersenkurse);
// Property 'Boerse' setzen
message.setStringProperty("Branche", "Technology");
```

Im Konsumentencode ist über einen entsprechenden Aufruf (*createConsumer*) innerhalb der Session ein Konsument anzulegen. Beim Anlegen wird der Selektor angegeben, der eine Filterung auf Technologieaktien vornimmt:

```
String selector;
selector = new String("(Branche = 'Technology')");
MessageConsumer receiver;
receiver = session.createConsumer(boerse, selector);
```

Als Nächstes können Nachrichten gesendet und empfangen werden, wobei der Selektor bereits bei JMS-Provider angewendet wird.

Zuverlässigkeit der Kommunikation und Fehlersemantik bei JMS

Bei der Betrachtung der Zuverlässigkeit der Übertragung und der unterstützten Fehlersemantik ist bei JMS eine andere Sicht als bei RPC einzunehmen. Es wird nämlich bei JMS im Gegensatz zu RPC oder ähnlichen Ansätzen keine Ende-zu-Ende-Übertragung zwischen Sender und Empfänger durchgeführt, da Konsumenten und Produzenten jeweils eigene und unabhängige Sessions mit dem JMS-Provider aufbauen. Im Mittelpunkt der Betrachtung steht also die Kommunikation zwischen JMS-Client (Produzent, Konsument) und JMS-Provider. Die Zuverlässigkeit der Kommunikation lässt sich bei JMS über mehrere Mechanismen beeinflussen. Hierzu gehören der Transaktionsmodus, der Delivery-Modus, der Acknowle-

dement-Modus, und der sog. Abonnement-Modus. Diese Mechanismen werden im Folgenden erläutert.

Transaktionsmodus: Hier handelt sich um einen Mechanismus, der es ermöglicht, mehrere Nachrichten sende- oder empfangsseitig in einer Transaktion zu bündeln. Die Einstellung gilt für die Session. Sind alle Nachrichten innerhalb der Transaktion erfolgreich versandt und gelesen worden, so wird die Transaktion mit dem Aufruf der Methode *commit* abgeschlossen. Man spricht in JMS auch von der sog. *atomaren Unit of Work*. Diese umfasst alle gesendeten und empfangenden Nachrichten einer Transaktion. Eine Transaktion teilt eine Session in mehrere atomare Einheiten auf. Unterhält ein JMS-Client mehrere Sessions, so sind die Transaktionen innerhalb der Sessions völlg unabhängig voneinander.

Die gesamte Transaktion kann aber auch durch Aufruf der *rollback*-Methode rückgängig gemacht werden. In diesem Fall wird im JMS-Provider auch wieder der Zustand vor der Transaktion hergestellt. Die bereits gesendeten Nachrichten sind also wieder in die entsprechenden Queues und Topics einzustellen. Ein transaktionsorientiertes Senden (vom Produzenten zum JMS-Provider) oder ein transaktionsorientierter Empfang (vom JMS-Provider zu Konsumenten) gewährleistet, dass alle in der Transaktion gebündelten Nachrichten oder gar keine gesendet bzw. empfangen werden, also eine atomare Ausführung erfolgt. Damit ist eine *exactly-once*-Semantik möglich, die allerdings nicht die Bearbeitungslogik anderer JMS-Clients mit einbezieht, sondern nur die Nachrichtenübertragung zwischen einem JMS-Client und JMS-Provider. In der JMS-Spezifikation (Sun 2007b) ist nicht die Rede von ACID-Transaktionen, die neben der Atomarität auch weitere Korrektheitskriterien wie die Sicherstellung der Isolationseigenschaft ansetzt. Die Rede ist ausschließlich von atomaren Transaktionen. ACID-Transaktionen werden in Kapitel 4 noch näher betrachtet.

Delivery-Modus: Dieser Modus regelt die Maßnahmen bei einem Providerausfall, wobei die Einstellung für die Destination vorgenommen wird (*createDurableSubscriber*). Es gibt zwei Delivery-Modi, die als *persistent* und *nicht-persistent* bezeichnet werden. Bei persistener Lieferung werden die Nachrichten dauerhaft in einen Speicher beim JMS-Provider gelegt. Dies bedeutet, dass die Nachrichten im Falle eines JMS Provider-Ausfalls nicht verloren gehen. Bei nicht-persistent ausgelieferten Nachrichten kann nach einen Ausfall des JMS-Providers keine Zustellung erfolgen. Die Nachrichten gehen verloren. Ein JMS-Provider muss gemäß Spezifikation eine at-most-once-Fehlersemantik für eine nicht persistente Auslieferung und eine once-and-only-once Fehlersemantik (exactly-once) bei persistenter Auslieferung garantieren. Im letzteren Fall darf also auch ein Providerfehler nicht zu einer Verletzung der unterstützten Fehlersemantik führen. Die Nachricht wird also in jeden Fall genau einmal ausgeliefert. Wie bereits erläutert, ist zu beachten, dass die Fehlersemantik nicht für die Ende-zu-Ende-Beziehung zwischen JMS-Clients gilt, sondern nur innerhalb der Beziehung eines JMS-Clients und dem JMS-Provider.

Acknowledgement-Modus: Dieser Modus bezieht sich auf die Art und Weise, wie der Konsument den Empfang einer Nachricht bestätigt. Die Einstellung gilt für die Session und es gibt drei Modi:

- *Auto_Ack:* In diesem Modus wird der Empfang einer Nachricht automatisch nachdem die Nachricht beim Konsumenten angekommen ist, bestätigt. Bei Einsatz von Auto_Ack wird eine Duplikatskontrolle ausgeführt. Dies gewährleistet, dass die Nachricht genau einmal an den Konsumenten. Gesendet wird (at-most-once-Semantik).
- *Dups_Ok_Ack:* Eine Empfangsbestätigung wird in diesem Fall automatisch nach dem Empfang von mehreren Nachrichten durch den Konsumenten abgesetzt. Die Anzahl der Nachrichten ist vom JMS-Provider abhängig, also herstellerspezifisch Dieser Modus führt keine Duplikatskontrolle durch, wodurch nur eine at-least-once Semantik möglich wird.
- *Client_Ack:* Bei diesem Modus geschieht der Empfang einer Nachricht nicht automatisch. Vielmehr muss der Konsument explizit eine Bestätigung senden. Hierzu verwendet der Konsument, also ein JMS-Client, die Methode *acknowledge()*. Es ist nicht nötig, dass der Konsument jede Nachricht bestätigt, da durch Aufruf der Methode *acknowledge()* alle vorher gesendeten Nachrichten ebenfalls bestätigt werden.

Wenn die Session transaktional ist, werden Nachrichten ohne Berücksichtigung des Acknowledgement-Modus bestätigt. Acknowledgement- und Transaktionsmodus werden also nicht gemeinsam verwendet.

Die Spezifikation schreibt vor, dass ein JMS-Provider grundsätzlich keine Nachrichtenduplikate senden darf. Allerdings gibt es eine Reihe von Einschränkungen bzw. Aufweichungen, die trotzdem dazu führen können. Nicht-persistent zugestellte Nachrichten (Delivery-Modus), temporäre Destinations wie temporäre Queues und nicht dauerhafte Topics (siehe Abonnement-Modus) nämlich sind per Definition unzuverlässig. Die verschiedenen Einstellmöglichkeiten für die einzelnen Modi sind also genau zu betrachten, um die eigenen Zuverlässigkeitskriterien sicherzustellen. Je nach Einstellung muss also ein JMS-Client auf Duplikate vorbereitet sein oder nicht.

Abonnement-Modus: Dieser Modus regelt die Maßnahmen bei einem Konsumentenausfall. Es gibt zwei Abonnement-Modi, nämlich *dauerhaft* und *nicht-dauerhaft*. Mit einem dauerhaften Abonnement wird sichergestellt, dass Nachrichten, die an nicht verfügbare Konsumenten gerichtet sind, in einen vorläufigen Speicher abgelegt und zu einem späteren Zeitpunkt zugestellt werden. Im anderen Fall werden die Nachrichten nicht übermittelt. Man unterscheidet dauerhafte und temporäre (nicht-dauerhafte) Queues sowie dauerhafte und nicht dauerhafte Topics.

Reihenfolgegarantie: Abschließend wollen wir noch auf die Maßnahmen zur Garantie der Reihenfolge von Nachrichten, die über JMS gesendet werden, eingehen. Die Reihenfolge der Nachrichtenzustellung an einen Konsumenten für eine dedizierte

Destination innerhalb einer Session wird durch JMS zugesichert. Wenn aber innerhalb einer Session mehrere Destinations adressiert werden, wird eine übergreifende Nachrichtenreihenfolge im Sinne einer totalen Ordnung aller gesendeten Nachrichten aber nicht gewährleistet. Die Reihenfolge kann auch durch einige Aspekte beeiflusst werden. So können Nachrichten priorisiert werden, womit ein Überholen von vorher gesendeten Nachrichten möglich wird. Bei nicht-dauerhaften Queues oder Topics können Nachrichten aufgrund von Providerfehlern verlorengehen. Auch wenn sowohl persistente als auch nicht-persistente Nachrichten an eine Destination gesendet werden, so ist die Reihenfolge der Nachrichten nicht garantiert. Eine Reihenfolgegarantie ist nur innerhalb der persistenten Nachrichten und innerhalb der nicht-persistenten Nachrichten gegeben. Schließlich sind bei Nutzung des Transaktionsmodus spezielle Aspekte zu betrachten, die die Nachrichtenreihenfolge beeinflussen (Sun 2007b).

Resümee:

Es soll nochmals festgehalten werden, dass die Transaktionsorientierung und der Acknowledgement-Modus zwei verschiedene Möglichkeiten für die Gewährleistung von Zuverlässigkeit bei der Übertragung von Nachrichten darstellen. Sie schließen sich gegenseitig aus. Während die Nutzung von Transaktionen für eine Erhöhung der Zuverlässigkeit einer Session innerhalb eines Produzenten oder Konsumenten geeignet ist, kann man mit dem Acknowledgement-Modus eine Session-übergreifende Bestätigung erreichen.

Während Transaktionen die Lieferung und den Empfang einer Menge zusammengehöriger Nachrichten in einer Session garantieren, sichert der Acknowledgement-Modus die Auslieferung von jeweils genau einer Nachricht an den Empfänger, also über die Session hinaus, ab.

Die zuverlässigste Variante scheint die Nutzung von Transaktionen zu sein. Möchte man eine Ende-zu-Ende-Semantik, so empfiehlt sich der Einsatz von Transaktionen beim Sender und beim Empfänger. Dabei sind allerdings Leistungsaspekte zu betrachten.

2.5.7 Fallbeispiel: Message-driven Beans

Message-driven Beans (MDB) entsprechen im Prinzip stateless Session-Beans (siehe Abschnitt 2.4), können aber nicht direkt von außen durch die Clients angesprochen werden. Sie warten auf JMS-basierte Nachrichten und verarbeiten diese, indem Sie als „JMS-Listener“ implementiert werden. Clients können beliebige JMS-Anwendungen und auch andere EJBeans sein.

Message-driven Beans müssen das Interface *javax.ejb.MessageDrivenBean* implementieren. Der Container legt für jede MDB einen Kontext an, der von der MDB über ein Interface namens *MessageDrivenBeanContext* verwendet werden kann.

In Abbildung 2-73 ist das Zusammenspiel zwischen Client, EJBean, EJB-Container und JNDI-naming-Service skizziert. Eine MDBean-Instanz wird vom Container verwaltet. Der Container registriert sich beim JMS-Provider als Listener für eine Queue oder ein Topic (2). Diese müssen vorab beim JMS-Provider angelegt werden (1) und über JNDI bekannt gemacht worden sein. Ein Client kann nun, nachdem er sich die entsprechende Referenz auf die Queue oder das Topic besorgt hat (3)(4), eine Nachricht in die Queue oder für das Topic einstellen (5). Diese wird wiederum vom JMS-Provider an den Container weitergereicht (6), der sie einer entsprechenden Bean-Instanz übergibt. Der JMS-Provider ist üblicherweise ein Bestandteil des EJB-Application-Servers.

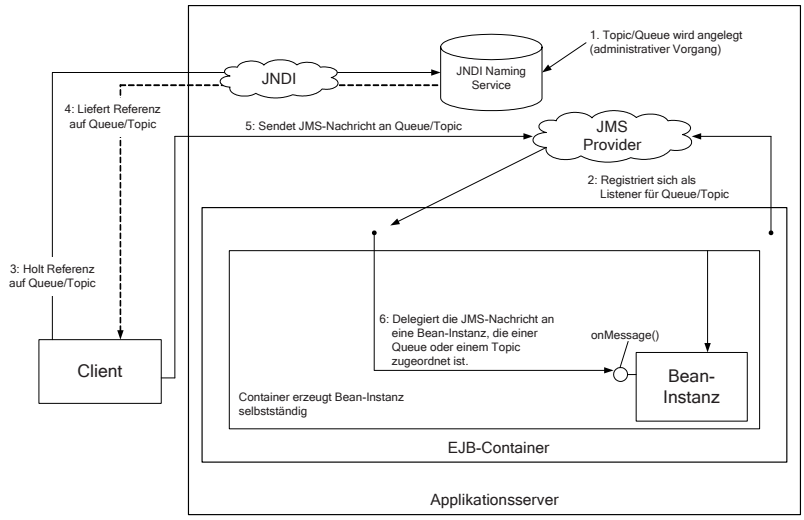


Abbildung 2-73: Abwicklung eines asynchronen Aufrufs einer Message-driven Bean

Das Interface *MessageDrivenBean* spiegelt den Vertrag zwischen Container und MDB wider. Die wichtigste Methode der Schnittstelle *MessageDrivenBean* ist *onMessage*. Hier wird der Code implementiert, der auszuführen ist, wenn eine JMS-Nachricht eintrifft. Die JMS-Nachricht vom Typ *javax.jms.Message* wird als Argument übergeben. Message-driven Beans können mit Hilfe der Annotation „@MessageDriven“ oder über eine Konfigurationsdatei (Deployment Descriptor, óder kurz DD) definiert werden. Weiterhin können verschiedene Eigenschaften der Kommunikation über die Annotation „@ActivationConfigProperty“ angegeben werden.

Beispiel: Der Destination-Typ und der JMS-Acknowledgement-Modus einer MDB können wie folgt festgelegt werden:

```
…
@MessageDriven (
     @ActivationConfig={
        @ActivationConfigProperty(
          propertyName=”acknowledgeMode”, property-value=”Auto-acknowledge”),
        @ActivationConfigProperty(
          propertyName=”destinationType”, property-value=”javax.jms.Queue”);
     }
)
public class MyMessageDrivenBean implements javax.jms.MessageListener {
     …
     public void onMessage(Message message) {
        …
     }
}
```

Der Zugriff auf den EJBean-Kontext wird über eine Ressource ermöglicht. Die Ressourcen-Definition sieht wie folgt aus:

```
@Resource MessageDrivenContext context;
```

Der Kontext einer MDB wird im Interface *MessageDrivenBeanContext* definiert. Dieses Interface erweitert das bereits in Abschnitt 2.4 erläuterte Interface *EJBContext*.

2.5.8 Zusammenfassung

Über eine nachrichtenorientierte Kommunikation kann man einige Kommunikationsvarianten realisieren, die nicht dem strengen Client-Server-Modell entsprechen. Es wird eine asynchrone Kommunikation gleichberechtigter Partner ermöglicht. Aber auch die Entwicklung von Client-Server-Anwendungen ist über diese Form der Kommunikation möglich, ist aber unüblich. Queues dienen der Entkopplung von Anwendungssystemen und eignen sich sehr gut für die Integration von „neu entwickelten“ Anwendungen in bestehende IT-Landschaften. Auch komplexe Mechanismen können damit umgesetzt werden. Nachrichtenwarteschlangensysteme zwingen keine Kommunikation nach dem Client-Server-Modell mit einem klaren Rollenkonzept (Diensterbringer und Dienstnehmer) auf und ermöglichen daher auch die Entwicklung von Kommunikationslösungen, bei denen z.B. ein Partner mehrere andere Partner (one-to-many) über bestimmte Ereignisse informiert. Auch Point-to-Point-Kommunikation zwischen gleichberechtigten Partnern ist über Message-Passing-Systeme realisierbar. Damit lassen sich einige Architekturvarianten unterstützen. Zudem bieten einige Middleware-Produkte die Möglichkeit der transaktionsgesicherten Kommunikation, bei der Nachrichten, die schon in einer Queue sind, Ausfallsituationen überleben.

Es soll noch abschließend erwähnt werden, dass heutige Message-Passing-Basisprodukte unterschiedlicher Hersteller nicht miteinander interagieren können,

da es keinen anerkannten Protokollstandard für Message-Passing-Systeme gibt. Dies gilt auch für JMS-basierte Lösungen. Eine Standardisierungsbemühung zu diesem Problem stellt das *Advanced Message Queuing Protocol* (AMQP) der AMQP-Working-Group dar, das aber noch nicht sehr verbreitet ist (WWW-037).

2.6 Übungsaufgaben

1. Wozu wird Threadpooling eingesetzt und welche Aufgabe hat ein Request-Dispatcher im Client-Server-Modell?
2. Erläutern Sie die Aufgaben eines Objektservers bei der Implementierung eines verteilten Objektsystems!
3. Welche beiden grundlegenden Konzepte werden für die Realisierung von verteilten Objektsystemen herangezogen?
4. Was ist ein Proxy-Objekt und welche Aufgaben hat es?
5. Erläutern Sie das verteilte Garbage-Collection am Beispiel des Reference-Counting-Algorithmus!
6. Welche Vorteile bieten verteilte Komponentensysteme im Vergleich zu verteilten Objektsystemen für den Anwendungsprogrammierer?
7. Wie läuft ein Remote-Procedure-Call aus Sicht eines Clients prinzipiell ab? Erläutern Sie den Ablauf anhand des Fallbeispiels ONC RPC kurz und gehen Sie dabei auch auf die Nutzung des zugehörigen Naming-Service ein!
8. Wie wird eine entfernte Schnittstelle eines Servers bei ONC RPC beschrieben? Was wird aus der Schnittstellenbeschreibung über einen speziellen Compiler generiert?
9. Was steckt hinter dem sog. Bootstrapping-Problem bei verteilten Anwendungen, die einen Naming- oder Directory-Service nutzen?
10. Wie funktioniert bei Java RMI und bei CORBA ein „Lookup“?
11. Wie funktioniert das verteilte Garbage-Collection unter Java-RMI und unter CORBA?
12. Wie wird bei Java-RMI ein Interface eines verteilten Objekts definiert? Grenzen Sie die Vorgehensweise zu der in CORBA ab.
13. Wie wird in Java-RMI ein Remote-Objekt programmiert? Gehen Sie dabei auf die Nutzung der RMI-Basis-Interfaces- und Klassen ein.
14. Welche Channel-Typen sind bei .NET Remoting für die Kommunikation zwischen Client und Server möglich?
15. Was versteht man bei .NET Remoting unter serveraktivierten und clientaktivierten Objekten (SAO und DAO)?
16. Was ist bei .NET Remoting ein Singleton und was ist der Unterschied zu einem SingleCall-Objekt? Handelt es sich hier um SAOs oder CAOs?
17. Was versteht man unter RMI/IIOP (RMI over IIOP) und wozu ist es nützlich?
18. Skizzieren Sie typische Protokollstacks für das RMI-Transportprotokoll!
19. Wie werden bei Java-RMI die Argumente und Returnwerte eines Methodenaufrufs an ein entferntes Objekt übertragen (by-reference, by-value)?

20. Wann verwendet man stateless und wann stateful Session-Beans?
21. Wozu dient das JNDI-API?
22. Warum kann man sich als Bean-Provider bei EJB nicht darauf verlassen, dass alle Beans in einer JVM ablaufen? Nennen Sie Restriktionen, die sich für den Programmierer daraus ergeben!
23. Skizzieren und beschreiben Sie den Zustandsautomaten einer stateless Session-Bean!
24. Was ist im Messsage-Passing-Modell eine synchrone, persistente Kommunikation im Vergleich zu einer asynchronen persistenten Kommunikation?
25. Nennen Sie Einsatzmöglichkeiten bzw. Anwendungen für Nachrichtenwarteschlangensysteme!
26. Erläutern Sie das P2P-Modell im Vergleich zum Publish-Subscribe-Modell!
27. Was ist ein JMS-Topic?

3 Verteilte Dienstaufrufe und Webservices

Webservices haben in den letzten Jahren als eine konkrete Implementierung von serviceorientierten Architekturen (SOA) einen hohen Bekanntheitsgrad erreicht. Viele Unternehmen beschäftigen sich mit dieser Technik der serverseitigen Diensteimplementierung und sind meist prototypisch dabei die ersten Webservices in Einsatz zu bringen. In diesem Kapitel geben wir einen Überblick über Standards und Technologien für die Entwicklung von Webservices als eine Implementierung für verteilte Dienstaufrufe. Nach einer Einführung in grundlegene Konzepte verteilter Dienste wird zunächst auf XML, SOAP als Übertragungsprotokoll und auf weitere Standards wie WSDL, UDDI und den WS-I-Standard eingegangen. Anschließend wird die Vorgehensweise bei der Entwicklung von Webservices diskutiert. Anhand von Java-Standards wie JAX-RPC und JAX-WS wird gezeigt, wie man Java-Webservices programmieren kann. Webservices werden auch über eine Interface-Beschreibungssprache, die WSDL, definiert. Es gibt zwei Möglichkeiten in der Vorgehensweise bei der Programmierung von Webservices, die als Bottum-Up oder Top-Down-Ansatz bezeichnet werden. Diese Ansätze werden vorgestellt. Anhand von Fallbeispielen für konkrete SOAP-Engines (Laufzeitumgebungen für Webservices) wie Apache Axis wird skizziert, wie man Webservices in Einsatz bringt.

Zielsetzung des Kapitels

Der Studierende soll grundlegende Konzepte und Entwicklungstechnologien für verteilte Dienste und Webservices verstehen und beurteilen können. Weiterhin soll er befähigt werden, Anwendungen mit Webservices zu entwickeln.

Wichtige Begriffe

Verteilter Dienst, Service, Webservice, SOAP, WSDL, UDDI, WS-I, JAX-RPC, JAX-WS, SOAP-Engine, Axis, XML

3.1 Verteilte Dienste

Im Zuge der Diskussion um Webservices wurde auch der allgemeine Begriff der verteilten Services bzw. des dienstorientierten Modells eingeführt. In diesem Zusammenhang entstand das SOA-Prinzip (Service-orientierte Architektur), das aber grundsätzlich nicht in Verbindung mit einer konkreten Technologie steht. SOA ist vielmehr ein Basiskonzept, das mit mehreren Technologien realisiert werden kann.

Ein Service stellt hier ganz allgemein einen oder mehrere Dienstschnittstellen (Services) zur Verfügung, die jeweils mehrere Methoden bzw. Operationen haben

können. Die Operationen sind für die Dienstnutzer in einem Service-Vertrag genau spezifiziert. Die Implementierung einschließlich der implementierten Business-Logik und der Datenzugriffe bleiben nach außen verborgen.

Das Konzept der verteilten Services ist nicht neu. Im Prinzip stellt ein Softwarebaustein als Diensterbringer seine Schnittstellen für Dienstnutzer bereit. Das Client-Server-Modell kann damit auch als ein Basismodell für Services verstanden werden. Ein Dienst im Sinne des dienstorientierten Modells steht aber üblicherweise auf einer etwas höheren Abstraktionsstufe wie die Dienste von verteilten Prozeduren, Objekten oder Komponenten.

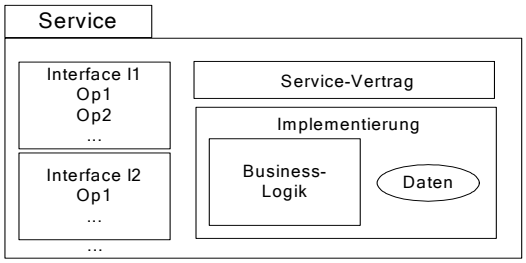


Abbildung 3-1: Service im dienstorientierten Modell

Wichtig ist auch die Verfügbarkeit von Services über eine technische Plattform hinaus. Während man beim komponentenorientierten Modell eine technische Plattform wie einen Container benötigt, der einem Dienstnutzer bekannt sein muss, wird im dienstorientierten Modell eine plattformübergreifende Interoperabilität fokussiert. Dienste evtl. sogar von verschiedenen Unternehmen können dabei zu Anwendungen zusammengefasst (komponiert) werden. Man verwendet in diesem Zusammenhang auch gerne die Begriffe *Orchestrierung* und *Choreographie*:

- Als *Orchestrierung* bezeichnet man die Zusammenstellung eines komplexeren Dienstes aus mehreren vorhandenen Diensten. Damit werden Dienste wiederverwendet und es wird ein „Mehrwertdienst“ geschaffen.
- Unter *Choreographie* versteht man die Kombination von Diensten zu Geschäftsprozessen. Dies wird heute insbesondere mit neuen Sprachen wie BPEL (Business Process Execution Language) im Umfeld der Businessprozessmodellierung toolunterstützt angestrebt (Schill, 2007).

Verteilte Services kann man prinzipiell mit allen Technologien realisieren. Realisiert man z.B. einen Service als CORBA-Objekt mit einem Remote-Interface, fällt es allerdings einem Client, der in einer anderen Technologie entwickelt wurde, schwer, den Dienst zu nutzen. Die Technologiewelten passen nicht zusammen. Moderne Basissysteme für verteilte Services müssen daher eine Plattform bereitstellen, die einen anerkannten, hersteller- und technologieübergreifenden Mechanismus zum Betrieb von Services ermöglichen.

Dienste muss man aber auch finden können. Für diese Aufgabe werden Verzeichnisdienste benötigt, die ein schnelles Auffinden eines Dienstanbieters ermöglichen.

Wie wir noch sehen werden, findet das Konzept der verteilten Services mit Webservices und den zugehörigen Standards WSDL, SOAP und UDDI eine weitgehend technologie- und herstellerunabhängige Realisierung, die von vielen Herstellern unterstützt wird. Die praktische Nutzung in unternehmensübergreifenden Anwendungen wirft allerdings auch einige Probleme auf, die noch zu lösen sind.

3.2 Grundlegende Konzepte von Webservices

In diesem Abschnitt werden die Basistechniken und Standards für die Entwicklung von Webservices vorgestellt. Da die Basistechniken derzeit noch stark entwickelt werden, sollen nur die grundlegenden Konzepte erläutert werden. Webservices nutzen XML in vielen Bereichen, weshalb ein grundlegendes Verständnis dieser Sprache erforderlich ist. In diesem Abschnitt kann allerdings hierzu nur eine kurze Einführung gegeben werden. Das W3C definiert einen Webservice wie folgt:

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

Ein Webservice ist also eine Technik, welche die Zusammenarbeit bzw. den Austausch von Informationen unterschiedlicher Softwareprogramme auf verschiedensten Plattformen ermöglicht (WWW-010). Die Implementierungen der einzelnen Programme sehen die Partner nicht. Die Komponenten werden über eine plattformunabhängige Diensteschnittstelle gekapselt. Mit Webservices lassen sich also Bausteine, die mit beliebiger Technologie entwickelt wurden, mit einer Schnittstelle versehen, die eine Nutzung über einen neutralen Mechanismus ermöglicht.

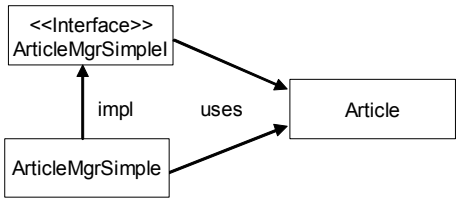


Abbildung 3-2: Klassenmodell für den vereinfachten Artikelmanager

Das Konzept sowie dessen Implementierung mit konkreten Plattformen soll an dem bereits verwendeten Beispiel eines Artikelmanager-Bausteins skizziert wer-

den. Wir bezeichnen das etwas vereinfachte Interface des Artikelmanagers mit *ArticleMgrSimpleI* und eine konkrete Implementierung mit *ArticleMgrSimple*. Der Artikelmanager verwaltet Artikel, die über die Objektklasse *Article* beschrieben werden. Ein Klassenmodell dieses sehr vereinfachten Softwarebausteins zeigt Abbildung 3-2.

Es soll im Weiteren gezeigt werden, wie man einen derartigen Baustein auch als Webservice bereitstellen kann. Das Beispiel zieht sich also durch die Erläuterung der wesentlichen Bestandteile von Webservices.

Services als grundlegendes Konzept

Wie bei serviceorientierten Architekturen üblich, stellt ein Diensterbringer (Service Provider) die Dienste bereit, ein Client ruft einen Dienst über ein standardisiertes Protokoll (hier SOAP) auf und die Syntax eines Webservice ist in einer festgelegten Sprache (WSDL[1], XML-basiert) beschrieben. Für das Auffinden von Diensten ist ein sog. Dienstregister vorgesehen, das mit UDDI[2] bezeichnet wird. Damit sind die drei wesentlichen Standards zur Unterstützung von Webservices bereits genannt: SOAP, WSDL und UDDI. Grundlage für SOAP und WSDL ist die deskriptive Sprache XML, die zur Beschreibung von Schnittstellen verwendet wird.

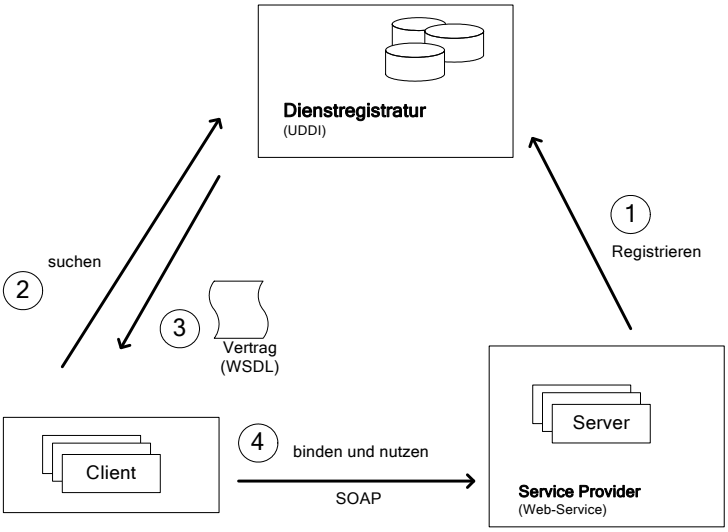


Abbildung 3-3: Service-orientierte Architektur mit Webservices nach (Kuhrmann 2004)

[1] WSDL steht für Web Service Description Language.

[2] UDDI steht für Universal Description, Discovery and Integration.

Die Kommunikation zwischen den Instanzen Server, Client und Dientregistratur ist in Abbildung 3-3 dargestellt. Ein Service-Provider registriert seinen Webservice bei einer Dienstregistratur (1). Registrierte Dienste kann ein Client ebenfalls über die Dienstregistratur suchen (2). Das Ergebnis der Suche ist eine Beschreibung des Dienstes, die man auch als „Schnittstellenvertrag" sehen kann (3). Mit der Dienstebeschreibung kann ein Client dann den Dienst verwenden (4).

Eine wesentliche Motivation für die Verwendung von Webservices zum Dienstaufruf ist der plattformübergreifende und sprachunabhängige Zugriff auf Funktionalität über das Internet (WWW) oder im Intranet. Ein Dienst kann in einer beliebigen Sprache programmiert sein. Um ihn anzusprechen, muss ein Clientprogramm lediglich seine Adresse und die Schnittstelle kennen.

Unternehmen können damit Dienste eines Drittanbieters nutzen, ohne sich mit den Technologien, die intern benutzt werden, befassen zu müssen. Ein Beispiel dafür ist das Abfragen der aktuellen Börsenkurse über einen Webservice-basierten Dienst, den eine Bank seinen Kunden anbietet.

Die wesentlichen Bestandteile von Webservices sowie deren Entwicklung sollen im Folgenden skizziert werden. Die in der Praxis übliche Nutzung erfordert in der Anwendung einen SOAP-Client und auf der Diensteseite einen SOAP-Prozessor, was in der Abbildung 3-4 nochmals skizziert ist: Wir werden weiter unten noch auf SOAP-Prozessoren eingehen.

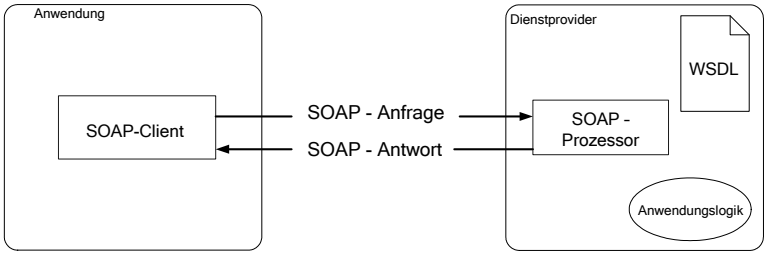


Abbildung 3-4: SOAP-Client und SOAP-Prozessor im Zusammenspiel

3.3 Technologien für SOA und Webservices

3.3.1 XML als Basistechnologie

Für das Verständnis des Aufbaus von SOAP-Nachrichten ist es notwendig, XML grundlegend zu verstehen. Daher wird an dieser Stelle ein kleiner Exkurs in die Konzepte von XML vorgenommen. Weiteres zu XML kann z.B. in (Wöhr 2004) nachgelesen werden.

XML ist eine Abkürzung für *Extensible Markup Language* und der Sprache HTML recht ähnlich. Im Gegensatz zu HTML wurde XML allerdings nicht dafür entworfen, um Daten anzuzeigen, sondern um Daten zu beschreiben. In XML gibt es kei-

ne vordefinierten *Tags* wie in HTML. Man muss im Prinzip eigene *Tags* definieren, um spezielle Sprachen zu entwerfen. XML dient also als Basissprache, um konkrete Sprachen mit einfachen und komplexen Datentypen darauf aufzubauen.

Die Beschreibung von XML-basierten Sprachen erfolgt über die etwas ältere DTD-Technik (Data Type Definition) oder über den neueren Standard namens *XML-Schema*. Beide Beschreibungsvarianten können alternativ eingesetzt werden, XML-Schema ist allerdings etwas umfassender und ermöglicht auch die Definition einer Fülle eigener Datentypen.

Beispiel: Die Syntax eines typischen XML-Dokuments für eine konkrete, XML-basierte Sprache (man spricht hier wie bei HTML auch von Dokumenten) soll an einem einfachen Beispiel (hier an einem Artikel unseres Artikelmanager-Bausteins) demonstriert werden:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Article>
   <KatalogId>10002</KatalogId>
   <Beschreibung>Mantel</Beschreibung>
   <Warengruppe>4711</Warengruppe>
   <Preis>239</Preis>
</Article>
```

XML-Dokumente sind in zwei Bereiche gegliedert. Der Kopf bzw. die sog. XML-Deklaration enthält allgemeine Informationen zur XML-Version und zum Zeichensatz. Im Body (Rumpf) sind die eigentlichen Sprachelemente einer XML-basierten Sprache definiert. Die XML-Deklaration legt im Beispiel die verwendete XML-Version „1.0“ und die verwendete Zeichenkodierung „ISO-8859-1“ für das XML-Dokument fest. Die folgende Zeile definiert das sog. *root*-Element, von dem es genau eines pro XML-Dokument gibt. XML-Dokumente sind baumartig aus *XML-Elementen* aufgebaut, die wiederum rekursiv Subelemente enthalten können. Die vier Zeilen nach <*Article*> sind demnach Kindelemente (*child*) des *root*-Elementes.

Jedes XML-Element ist durch ein öffnendes und ein schließendes *XML-Tag* gekennzeichnet. XML-Tags und XML-Elemente sind „*case-sensitive*“ und somit sind *artikel* und *Artikel* zwei unterschiedliche Tags. XML-Elemente können auch Eigenschaften besitzen. Soll das Element *Article* das Attribut „lagernd“ bekommen, würde dies folgendermaßen aussehen:

```
<Article lagernd = "ja">[3]
```

XML-Dokumente werden als *wohlgeformt* bezeichnet, wenn alle Regeln zur Formatierung eingehalten werden. Zu den Regeln gehört, dass

- das XML-Dokument genau eine Wurzel hat,

[3] Es ist zu beachten, dass der Eigenschaftswert immer in Anführungszeichen stehen muss.

- Attribute als Name-Wert-Paare definiert werden,
- Elemente immer paarweise definiert (öffnende und schließende Tags) sind und sich auch nicht überlappen dürfen.

Dies kann syntaktisch geprüft werden, wozu eine Software namens XML-Parser eingesetzt werden kann. Einer der bekanntesten XML-Parser ist Xerces aus dem Apache-Projekt (WWW-007).

XML unterstützt, wie höhere Programmiersprachen[4], auch Namensräume, die dazu dienen, Namenskonflikte und Mehrdeutigkeiten in XML-Dokumenten zu vermeiden. Folgende Problemstellung soll die Notwendigkeit von Namensräumen verdeutlichen, wobei wir wieder das Artikelbeispiel und ein weiteres Beispiel zur Beschreibung einer Person verwenden:

Beispiel: Konkreter Artikel

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Article>
   <Beschreibung>Mantel</Beschreibung>
   <Warengruppe>4711</Warengruppe>
   <Preis>239</Preis>
</Article >
```

Beispiel: Konkrete Person

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Person>
     <Beschreibung>1.70 m groß, blondes Haar, … </Beschreibung>
     <Alter>23</Alter>
     ...
</Person>
```

Die Problematik wird bei der Betrachtung der beiden XML-Beschreibungen deutlich. Beide root-Elemente verfügen über ein Subelement namens *Beschreibung*, jede Beschreibung hat jedoch eine eigene Bedeutung. Im XML-Element *Article* bezieht sich die Beschreibung auf einen Artikel, im XML-Element *Person* auf eine Person. Um nun das XML-Element *Beschreibung* mehrfach verwenden zu können, wird das Konzept der XML-Namensräume (Namespace) verwendet. Es ist möglich einen Namensraum ab Höhe des Wurzelelementes zu definieren. Das XML-Dokument für den *Article* würde dann folgendermaßen aussehen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
     xmlns: Vertrieb="http://meinBetrieb.de/Article">
<Vertrieb:Article
     <Vertrieb:Beschreibung>Mantel</Vertrieb:Beschreibung>
```

[4] In Java wird dies über Packages gelöst, in C# über Namespaces.

```
    <Vertrieb:Warengruppe>4711</Vertrieb:Warengruppe>
    <Vertrieb:Preis>239</Vertrieb:Preis>
</Vertrieb: Article>
```

Der Namensraum wird als Attribut der Wurzel (des root-Elements) definiert. *xmlns* steht dabei für XML-Namespace. Dieses Attribut ist ein (Name, Wert)-Paar und definiert den Namensraum *Vertrieb*:

```
Vertrieb=„http:://meinBetrieb.de/Article“
```

Über diesen Namensraum lässt sich nun die Beschreibung für den Artikel von der Beschreibung der Person abgrenzen:

```
<Vertrieb:Beschreibung>Mantel</Vertrieb:Beschreibung>
```

Entsprechend könnte man einen eigenen Namensraum für das XML-Dokument zur Definition der Person festlegen.

Die XML-Beispiele für einen Artikel stellen bereits konkrete Ausprägungen von XML-Beschreibungen dar. Die Datentypen werden - wie gesagt - z.B. in einem XML-Schema festgelegt. Im Weiteren werden wir bei der Beschreibung der WSDL noch auf die Datentypen eingehen.

3.3.2 Simple Object Access Protocol

Das *Simple Object Access Protocol* (SOAP) ist gemäß der Spezifikation des World Wide Web Consortiums (W3C) als ein einfacher Mechanismus für den Austausch von strukturierten und typisierten Informationen auf Basis von XML zwischen Rechnern in einer dezentralen verteilten Umgebung beschrieben.[5] Vorgänger war XML-RPC[6], das ursprünglich entwickelt wurde, um XML-basierte Nachrichten über das Protokoll HTTP zu senden. SOAP hat eigentlich nichts mit Objektorientierung zu tun, weshalb der Name ab der Version 1.2 nicht mehr als Abkürzung betrachtet wird. SOAP ist also heute keine Abkürzung mehr für „Simple Object Access Protocol“.

SOAP kann man als zustandsloses Protokoll verstehen. Es legt sich prinzipiell nicht auf eine Interaktionsform fest. Man kann über SOAP grundsätzlich eine Einwege-Kommunikation zwischen zwei SOAP-Knoten (*nodes*) abwickeln, jedoch können Einwege-Nachrichten in komplexeren Szenarien kombiniert werden. So ist es z.B. möglich, eine Einweg-Nachricht an mehrere Empfänger zu senden oder das

---

[5] Die Entwicklung von SOAP wurde 1998 unter Federführung von Microsoft und anderen Firmen begonnen. Eine erste SOAP-Version wurde Ende 1999 über die Internet Engineering Task Force (IETF) veröffentlicht. Im Mai 2000 wurde die SOAP-Spezifikation in der Version 1.1 beim W3C angemeldet. Siehe auch (W3C 2003a) und (W3C 2003b).

[6] XML-RPC wurde 1998 von Microsoft entwickelt und spielt heute keine wesentliche Rolle mehr.

klassische Request-/Response- bzw. ein RPC-Modell zu realisieren. Auch ein Weiterleiten einer SOAP-Nachricht über einen Zwischenknoten (*intermediary*) an einen Empfänger ist möglich.

SOAP regelt im Wesentlichen den Aufbau von Nachrichten für Webservices. Eine SOAP-PDU ist ein XML-Dokument, in dem der Aufruf eines Webservice mit all seinen Parametern beschrieben wird. Im Falle einer Response-Nachricht sind die Ergebnisse im XML-Dokument enthalten.

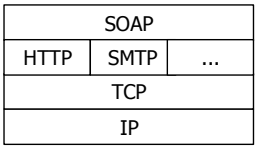


Abbildung 3-5: Typischer SOAP-Protokollstack

SOAP ist grundsätzlich unabhängig vom darunterliegenden Transportmechanismus, meist wird im WWW aber als Basis das Anwendungsprotokoll HTTP verwendet, um SOAP-Nachrichten zu transportieren. Die Möglichkeit der Nutzung von HTTP zeigt auch, dass SOAP ein recht einfaches Protokoll ist. SOAP-PDUs werden in einem Textformat kodiert und nutzen ein spezielles XML-Schema für den Nachrichtenaufbau. Ein typischer Protokollstack für die Nutzung von SOAP ist in Abbildung 3-5 skizziert.

SOAP-PDU: In eine SOAP-PDU können beliebige Anwendungsdaten wie z.B. eine Warenbestellung, eine Anfrage nach einem Artikel, eine Anfrage an eine Suchmaschine usw. sowie entsprechende Antworten auf die Anfragen transportiert werden. Die Inhalte einer Nachricht können in einem vorgegebenen Rahmen in der Nutzlast der SOAP-PDU frei definiert werden.

Die SOAP-Spezifikation aus (WWW-010) legt eine XML-Grammatik zur Spezifikation von SOAP-PDUs als XML-Schema fest. Mit dieser Grammatik lassen sich strukturierte und typisierte Informationen zu einer Nachricht zusammenfassen. Eine SOAP-Nachricht (ein XML-Dokument) besteht grob aus einem *Header* und einem sog. *Body*. Beide Nachrichtenteile beinhalten beliebige Elemente, sind anwendungsspezifisch und im sog. Umschlag (*envelope*) zusammengefasst.

Header-Elemente sind optional und beinhalten Kontrollinformationen, die nicht zur Nutzlast gehören. Die Kindelemente des Headers werden als Headerblöcke bezeichnet (*header blocks*). Headerblöcke sind dazu da, bestimmte, zusammengehörige Informationen zwischen den Knoten, also zwischen Sender und Empfänger, aber auch zu Zwischenknoten auf dem Nachrichtenpfad zu übermitteln. Das Header-Element beinhaltet also applikationsspezifische Informationen und Anweisungen wie die Nachricht verarbeitet werden soll. Dazu gehören z.B. Angaben zum Routing (Weiterleiten von SOAP-Nachrichten), zur Auslieferung, zur Au-

thentifizierung, zur Autorisierung und zu Transaktionskontexten. Der Header muss das erste Kind-Element des Umschlags sein. Das Header-Element liefert einen Erweiterungspunkt zur Definition anwendungsspezifischer Protokolle und ist vom Aufbau her vergleichbar mit der Funktionsweise von HTTP-Headern.

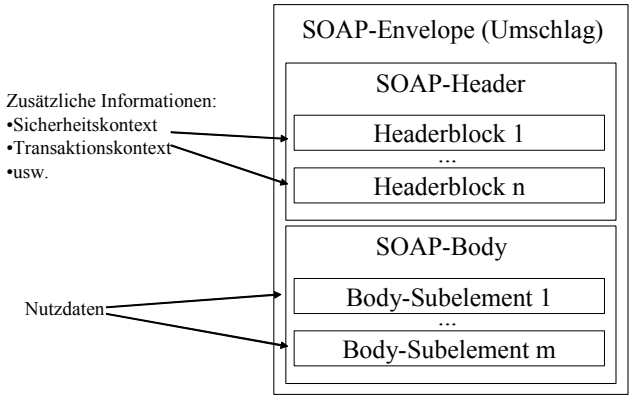


Abbildung 3-6: Aufbau einer SOAP-PDU

Der SOAP-Body ist ein Pflichtelement der SOAP-PDU und enthält die eigentliche Nutzlast, die zwischen Sender und Empfänger ausgetauscht werden soll.

Ein sog. *Fault*-Element mit Informationen über Fehler, die während der Bearbeitung auftreten können, kann optional ergänzt werden.

Nachrichtenpfade: SOAP-Nachrichten passieren dem SOAP-Nachrichtenflussmodell entsprechend auf der Route vom Sender zum Empfänger ggf. mehrere Vermittler oder Zwischenknoten (engl. *intermediaries*). SOAP-Anwendungen, die zwischen Sender und Empfänger liegen und die Nachrichten verarbeiten und weiterleiten, heißen Akteure (früher actors, heute role). Der Pfad vom Sender über diverse Akteure zum Empfänger heißt Nachrichtenpfad (siehe Abbildung 3-7). Wie im Bild dargestellt ist, können die Einzelverbindungen auch unterschiedliche Transportmechanismen verwenden.

Alle Beteiligten (Zwischenstation, Sender und Empfänger) werden durch URIs eindeutig identifiziert. SOAP kann über das Attribut *actor/role* in Headerblöcken festlegen, welche Teile der Nachricht von welchem Dienst verarbeitet werden. Der Wert von *actor/role* kann eine URL oder ein anderer Bezeichner sein.

Es ist zulässig, dass nicht alle Teile der SOAP-Nachricht für den entgültigen Empfänger bestimmt sind, sondern nur für Zwischenstationen auf der Route, die von der Nachricht zurückgelegt wird. Eine Zwischenstation muss die Teile der Nach-

richt, die nur für sie bestimmt sind, aus der Nachricht entfernen, bevor sie weitergeleitet wird.

Header-Informationen können also von den SOAP-Knoten auf dem Nachrichtenpfad verarbeitet werden. Sind Informationen nicht für den endgültigen Empfänger bestimmt, sondern für eine Zwischenstation, kann dieser die Informationen auslesen und darauf reagieren. Zusammengefasst können folgende vordefinierte Informationen als Attribute der Headerblöcke übertragen werden[7]:

- Attribut *actor* oder *role*: SOAP kann über das Attribut actor/*role* in Headerblöcken festlegen, welche Teile der Nachricht von welchem SOAP-Knoten auf dem Nachrichtenpfad verarbeitet werden sollen. Mögliche Werte sind *next* für den nächsten Knoten, *none* wenn kein Knoten die SOAP-PDU verarbeiten soll und *ultimateReceiver*, wenn die PDU vom adressierten Empfänger verarbeitet werden soll.
- Attribut *mustUnderstand*: In diesem Attribut wird festgelegt, ob ein Headerblock zwingend verarbeitet werden muss (Wert "true") oder ob er auch ignoriert werden darf (Wert "false" ist der Standardwert).
- Attribut *relay*: Dieses Attribut legt fest, ob ein nicht verarbeiteter Headerblock weitergeleitet werden muss (Wert "true") oder nicht (Wert "false" ist der Standardwert).

Auf der Route muss ein SOAP-Knoten, der eine Nachricht empfängt, festgelegte Verarbeitungsrichtlinien einhalten. Wird ein mit *mustUnderstand*="true" gekennzeichneter Teil des SOAP-Headers vom Knoten nicht verstanden, muss eine Fehlernachricht zurückgesendet werden. Ein Knoten auf der Route muss vor der Weiterleitung einer Nachricht alle für ihn bestimmten Headerblöcke aus der Nachricht entfernen.

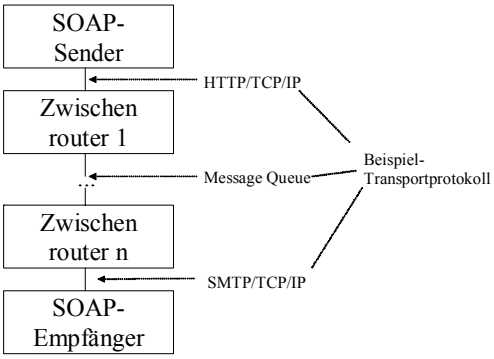


Abbildung 3-7: Ein potenzieller SOAP-Nachrichtenpfad

[7] Die Attribute werden im SOAP-Headerblock in Form von vollständigen URIs übermittelt.

Ein Methodenaufruf für unseren Artikelmanager könnte in einer konkreten SOAP-Nachricht folgendermaßen aussehen:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<!-- SOAP-Header nicht vorhanden -->

<!-- SOAP-Body -->
  <soap:Body>
    <CreateArticle xmlns="http://…/">
      <ID>long</ID>
      <Warengruppe>long</Warengruppe>
      <Beschreibung>string</Beschreibung>
      <Preis>float</Preis>
    </CreateArticle>
  </soap:Body>
</soap:Envelope>
```

Der SOAP-Umschlag (SOAP-Envelope) liegt im XML-Namespace "http://schemas.-xmlsoap.org/soap/envelope/". Im obigen Beispiel erkennt man, dass die eigentlichen Daten im Body-Element der SOAP-Nachricht liegen. Die obige SOAP-Nachricht soll noch erläutert werden:

- In der ersten Zeile werden die Version und die Kodierung für das XML-Dokument festgelegt.
- In der zweiten Zeile befindet sich der Tag für den SOAP-Umschlag als Wurzelelement des Dokumentes.
- Im Wurzelelment werden die drei Namensräume *soap, xsi* und *xsd* deklariert.
- Innerhalb des Body-Tags wird das *CreateArticle*-Element definiert, das die eigentliche Nutzlast der SOAP-Nachricht enthält. Zur Nutzlast der SOAP-PDU gehören eine Identifikation, die Beschreibung, die Warengruppe und der Preis eines Artikels mit den entsprechenden Werten.

SOAP-Fehlerbehandlung: In der SOAP-Spezifikation ist das SOAP-Body-Element zur Übertragung von Fehlerinformationen, *SOAP-Faults* genannt, definiert. SOAP-Faults geben Fehlermeldungen zurück, die beim fehlerhaften Verarbeiten von SOAP-Nachrichten auftreten. Falls ein SOAP-Fault-Element verwendet werden soll, muss es das einzige direkte Kindelement des Body-Elements sein. Da die SOAP-Spezifikation ständig weiterentwickelt wird, gibt es Unterschiede in den einzelnen Versionen. Ein SOAP-Fault-Element besteht u.a. aus mehreren Kindelementen. Hierzu gehören die Elemente *code* und *Reason*. Das Element *code* dient als Fehlercode und kann u.a. die Werte *VersionMismatch*, *MustUnderstand*, *DataEncoding*

Unknown enthalten. Im *Reason*-Element wird ein Text übertragen, der den Grund des Fehlers näher erläutert. Neben diesen beiden Pflichtangaben können noch Informationen zu dem Knoten, der den Fehler verursacht hat (node-Element), eine Rollenangabe (Role-Element und eine Detailfehlerangabe (Detail-Element) angegeben werden. Das folgende Beispiel soll die SOAP-Fehlermeldung verdeutlichen:

```
<env:Envelope xmlns:env=„...">
<env:Body>
    <env:Fault>
      <env:Code>
          <env:Value>env:Sender</env:Value>
      </env:Code>
      <env:Reason>
          <env:Text xml:lang=”en”>Sender timeout</env:Text>
      </env:Reason>
    </y:Fault>
</y:Body>
</y:Envelope>
```

Trägerprotokolle für SOAP

Trägerprotokoll für SOAP: Mit SOAP können verschiedene Interaktionsmodelle unterstützt werden. Die wichtigste Verwendung von SOAP ist wohl der klassische entfernte Prozeduraufruf (RPC). Dabei werden in der Anfrage Eingabeparameter übermittelt, um damit eine entfernte Prozedur aufzurufen. Als Antwort wird das Ergebnis zurückgesendet. Einen RPC abzusetzen bedeutet also nichts weiter als eine SOAP-Nachricht zu bauen und an den Diensterbringer zu senden. Die Nutzlast der Anfrage enthält den Methodenaufruf.

In Abbildung 3-8 ist der Aufruf eines Webservice auf Basis von SOAP mit HTTP als „Trägerprotokoll“ skizziert. Bevor die in eine HTTP-PDU eingebettete SOAP-PDU versendet werden kann, muss zunächst eine Transportverbindung auf Basis von TCP aufgebaut werden. Der SOAP-Request und die SOAP-Response werden jeweils in einer HTTP-GET- oder in einer HTTP-POST-PDU übertragen.

Wie bereits dargestellt ist in der SOAP-Spezifikation kein spezielles Transportprotokoll festgelegt. Die Benutzung von HTTP als „Transportprotokoll“ ist zwar am gängigsten, jedoch nicht unbedingt vorgeschrieben. Der Transport kann auch direkt auf Basis von TCP oder auf Basis anderer Anwendungsprotokolle wie SMTP oder FTP erfolgen. Sollte das zugrundeliegende Kommunikationsprotokoll keine entsprechende Unterstützung für eine synchrone Interaktion bieten, so wie es z.B. bei SMTP der Fall ist, müssen SOAP-Request und SOAP-Response in zwei voneinander unabhängigen Transportverbindungen übermittelt werden.

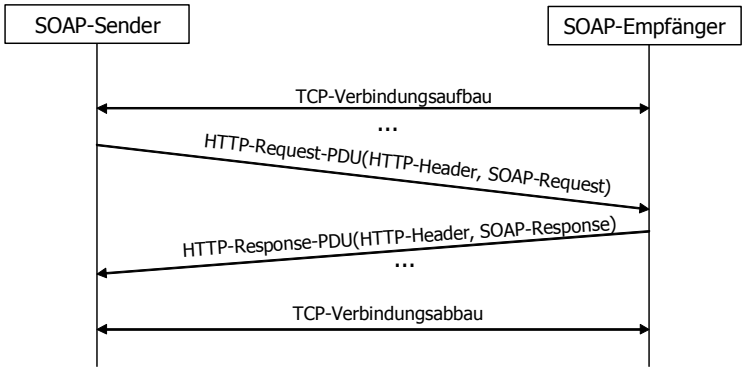


Abbildung 3-8: SOAP-Protokollablauf über HTTP

Da HTTP das Request-/Response-Modell beherrscht und fast überall verfügbar ist, liegt eine Verwendung dieses Protokolls zum Transport von SOAP-Nachrichten sehr nahe. Ein HTTP-Request dient zur Versendung einer SOAP-Anfrage. Die SOAP-Antwort kann im HTTP-Response gesendet werden. Die Interaktion erfolgt synchron, nach einer Anfrage wartet der Client also blockierend auf die Antwort.

Eine SOAP-PDU wird bei Verwendung von HTTP als Trägerprotokoll vollständig in eine HTTP-PDU eingebettet. Da beide Protokolle textbasiert sind, ist das weiter kein großes Problem. Wir wollen das HTTP-Bindung am Beispiel unseres Artikelmanagers verdeutlichen und betrachten zunächst einen SOAP-Request zum Erzeugen eines neuen Artikels (*CreateArticle*) eingebettet in eine HTTP-POST-PDU:[8]

```
POST /WebServiceAM/ArticleMgr.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: <length>
SOAPAction: "http://…/CreateArticle"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CreateArticle xmlns="http://…
      <ID>2000</ID>
      <Warengruppe>20</Warengruppe>
      <Beschreibung>Vespa</Beschreibung>
```

[8] URI des Webservice, hier in Microsoft-Umgebung. Webservices werden hier über Dateien mit der Endung .asmx erzeugt.

```
      <Preis>2500</Preis>
    </CreateArticle>
  </soap:Body>
</soap:Envelope>
```

Als Medientyp (Content-Type) muss nach RFC 2376 der Typ "text/xml" angegeben werden, wenn SOAP-Nachrichten per HTTP transportiert werden. Die aufzurufende Webservice-Operation wird dem Empfänger im HTTP-Header über das zusätzliche *SOAPAction*-Feld angezeigt. In diesem Feld wird der Name der Operation, die in der Definition des Webservice festgelegt wurde, angegeben. Der Wert dieses Feldes ist ein URI. Ein HTTP-Client, der einen SOAP-Request senden will, muss dieses Header-Feld benutzen. Ein leerer String im Feld *SOAPAction* bedeutet dabei, dass die aufzurufende Operation des Requests aus dem HTTP-Request-URI hervorgeht. Wie man sieht wird die SOAP-PDU im HTTP-Body übertragen. Die Länge der SOAP-PDU ist im HTTP-Header *Content-Length* eingetragen.

Die Antwort-Nachricht *CreateArticleResponse* ist ebenfalls in eine HTTP-PDU verpackt und sieht wie folgt aus:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: <length>
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CreateArticleResponse xmlns="http://…/" />
     <!-- Je nach Definition der Operation wird hier eine Nachricht mit -->
     <!-- Rückgabewerten eingetragen -->
  </soap:Body>
</soap:Envelope>
```

Zur Anzeige des Erfolgs oder Misserfolgs des SOAP-Requests wird der HTTP-Statuscodes in der ersten Zeile der HTTP-PDU (im Beispiel 200) genutzt. Ein Statuscode aus dem Bereich 200-299 in der Antwort zeigt das erfolgreiche Bearbeiten eines Requests an.

Wenn in der SOAP-Anwendung beim Verarbeiten der Nachricht ein Fehler auftritt, muss von der Anwendung mit einem HTTP 500 "Internal Server Error" geantwortet werden. Weiterhin muss die Antwort eine SOAP-Nachricht sein, welche das SOAP Fault-Element enthält.

3.3.4 Web Services Description Language (WSDL)

Die *Web Services Description Language* (WSDL)[9] definiert einen plattform-, programmiersprachen- und protokollunabhängigen XML-Standard zur Beschreibung von Netzwerkdiensten (Webservices). WSDL ist eine Metasprache, die in einem XML-Dokument einen Webservice beschreibt. WSDL definiert zudem, wo der Webservice erreichbar ist und die Methoden, die über den Service angesprochen werden können. Es werden im Wesentlichen die Methoden definiert, die von außen zugänglich sind, sowie die Parameter und Rückgabewerte dieser Operationen. Mit WSDL werden also sowohl die auszutauschenden Daten als auch die Operationen beschrieben. Ein Webservice beinhaltet eine oder mehrere Dienstprimitive (Web-Methoden oder Operationen genannt).

WSDL beschreibt einen Webservice über folgende Sprachbestandteile:

- Datentypen: Dies sind die Typen der einzelnen Übergabe- und Rückgabewerte (Tag *types*).
- Interface aller Funktionen: Hierzu gehören die Eingabe- und Ausgabeparameter (Tag *message*) und die Methodensignaturen (Tag *operations*).
- Binding: Hier wird das Trägerprotokoll, z.B. HTTP angegeben (Tag *binding*).
- Adresse des Services: Angabe zur Adressierung eines Webservice (Tags *service* und *port*).

Für die Beschreibung eines Webservice ergibt sich folgender Aufbau:

```
<definitions>
<types>
     Definition eines Typs
</types>
<message>
     Definition einer Nachricht
</message>
<portType> oder seit V1.2 <interface>
     Definition einer Schnittstelle
     <operation>
          Definition einer Methodensignatur
     </operation>
</portType> oder seit V1.2 </interface>
<binding>
     Definition einer Bindung an ein Trägerprotokoll
</binding>
```

[9] Die Version WSDL 1.1 wurde dem W3C im März 2001 von den Unternehmen Ariba, Microsoft und IBM vorgelegt und als Spezifikation durchgesetzt. Mittlerweile gibt es Weiterentwicklungen.

```
<service>
     Definition eines Service
       <port>
         Definition eines Ports
       </port>
</service>
</definitions>
```

Die beteiligten WSDL-Elemente sind in Abbildung 3-9 skizziert. Eine WSDL-Beschreibung kann beliebig viele Typen definieren. Alle Nachrichten, die in den Porttypen benötigt werden, sind im WSDL-Dokument zu beschreiben. In einer WSDL-Datei können auch mehrere Interfaces (Porttypen) definiert werden.

Ein WSDL-Dokument kann auch noch weitere, hier nicht erwähnte Elemente beinhalten (WWW-010). Die einzelnen Tags sollen im Folgenden anhand einfacher Beispiele näher erläutert werden. Es würde allerdings den Rahmen dieses Buches sprengen, wenn die komplette WSDL-Syntax beschrieben würde. Es werden daher nur wichtige Elemente und Attribute erläutert und für den Rest auf die entsprechende Literatur verwiesen.

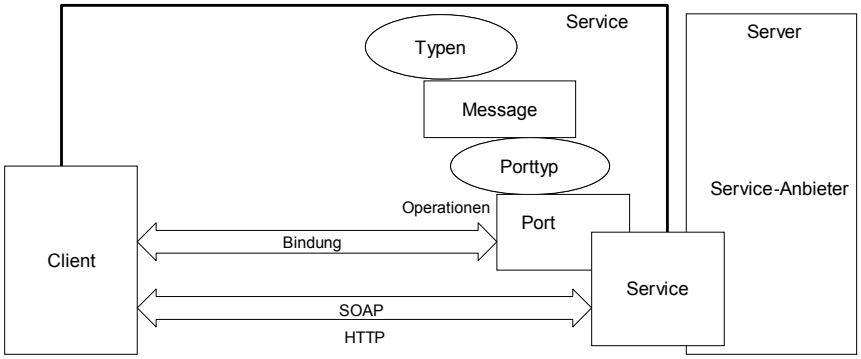


Abbildung 3-9: WSDL-Elemente und deren Zusammenspiel

Das *definitions*-Element

Dieser Tag bildet das Wurzelelement eines WSDL-Dokuments und definiert die Namensräume die im WSDL-Dokument gebraucht werden. Folgendes Beispiel soll das Element konkretisieren:

```
<definitions>
     xmlns:http=http://schemas.xmlsoap.org/wsdl/http/
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:s="http://www.w3c.org/2001/XMLSchema"
```

```
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    …
</definitions>
```

Die Namensräume werden mit Kurzbezeichnungen versehen. Hierzu gehören im Beispiel *soap* für den SOAP-Namensraum, *s* für den Namensraum des XML-Schemas selbst, *soapenc* und *tm*. Bei der Nutzung von Typen aus den angegebenen Namensräumen wird dann nur noch die Abkürzung als Präfix zum Namen des Typen ergänzt.

Insbesondere werden folgende Namensräume sehr häufig verwendet (WWW-012):

- wsdl: http://schemas.xmlsoap.org/wsdl/
- soap: http://schemas.xmlsoap.org/wsdl/soap/
- http: http://schemas.xmlsoap.org/wsdl/http/
- soapenc: http://schemas.xmlsoap.org/soap/encoding/
- soapenv: http://schemas.xmlsoap.org/soap/envelope/
- xsd: http://www.w3.org/2001/XMLSchema (oder aktuellere Stände)[10]

Die Abkürzung *tns* (sozusagen als „*this-Namespace*“) wird oft verwendet, um den Bezug zum aktuellen Dokument zu gewährleisten.

Wir wollen im Folgenden auf die einzelnen Komponenten der WSDL mit den Service-bezogenen Merkmalen näher eingehen.

Das *types*-Element

Dieser Tag ist optional und nicht erforderlich, falls im XML-Dokument nur einfache Datentypen zum Einsatz kommen. Einfache Datentypen werden bereits vom entsprechenden XML-Schema für Webservices angeboten (z.B.: string, integer, float, double). Um komplexe Datentypen zu beschreiben, muss dieses Tag verwendet werden. Die Typdefinition wird immer mit einem <*schema*>-Element (XML-Schema) begonnen.

Alle komplexen Datentypen müssen über den Tag *complexType* beschrieben werden. Wenn alle komplexen Typen gemäß dem Standard auf den vorhandenen Datentypen aufgebaut werden, ist eine Interoperabilität zwischen Partnersystemen, die auch unterschiedliche Programmiersprachen verwenden, gewährleistet.

Das folgende Beispiel zeigt die Definition einer Struktur namens *Article* als komplexen Datentypen mit vier Attributen (Subelemente). Mit dem Bezeichner *s* wird der in der Definition angegebene Namensraum für das XML-Schema angesprochen. Im Beispiel ist das der Namensraum mit dem URL http://www.w3c.org/-2001/XMLSchema.

```
<types>
```

[10] Ändert sich mit aktuellen Versionen wegen der Jahresangabe.

```
<s:schema xmlns="http://www.w3.org/2001/XMLSchema">
     targetNamespace="http://ArticleMgr/">
     <s:complexType name="Article">
     <s:sequence>
       <s:element minOccurs="1" maxOccurs="1" name="id" type="s:int" />
       <s:element minOccurs="0" maxOccurs="1" name="Warengruppe"
          type="s:string" />
       <s:element minOccurs="0" maxOccurs="1" name="Beschreibung"
          type="s:string" />
       <s:element minOccurs="1" maxOccurs="1" name="Preis"
          type="s:double" />
     </s:sequence>
</s:complexType>
…
</s:schema>
<types>
```

Hier wird der komplexe Datentyp *Article* beschrieben. In dem Tag *sequence* werden alle Elemente des zusammengesetzten Typs als Gruppe definiert. Es gibt auch noch andere Varianten für die Bildung von Element-Gruppen.

Die Typbeschreibung beginnt, wie bei XML-Schemabeschreibungen üblich, immer mit einem *<schema>*-Element. Der Namensraum *www.w3.org/2001/XMLSchema* ist in diesem Beispiel der Default-Namensraum. Im Attribut *targetNamespace* wird die URI des Zielnamensraumes definiert. Dieser Namensraum wird automatisch für alle deklarierten Elemente und Attribute des Schemas verwendet. Weiterhin kann man über die im Beispiel nicht einbezogenen Attribute *elementFormDefault* und *attributeFormDefault* den Zielnamensraum lockern. Gibt man hier den Wert „qualified" an, so kann ein Element oder ein Attribut auch außerhalb des Zielnamensraums auftreten.

Im Attribut *type* werden jeweils die Typen der Elemente des komplexen Datentyps *Article* angegeben. Jedes Element besitzt auch einen Namen. Ein Artikel besteht im Beispiel aus einem Attribut vom Typ *int*, zwei Attributen vom Typ *string* und einem Attribut des Typs *double*. Bei den einzelnen Elementen des Datentyps *Article* ist angegeben, wie oft sie in einem konkreten XML-Dokument vorkommen dürfen. Hierzu dienen die Attribute *minOccurs* und *maxOccurs*. Wenn beliebig viele Einträge zugelassen sind, wird im Attribut *maxOccurs* der Wert „unbounded" angegeben. Wird im Attribut *minOccurs der Wert* „0" angegeben, kann das Element in einer konkreten Ausprägung des Typs auch fehlen.

Das *message*-Element

Dieser Tag beschreibt die Daten, die zwischen einem Client und dem Webservice-Provider ausgetauscht werden. Zu jeder Methode eines Webservice (Tag *WebMethod*) gibt es üblicherweise (aber nicht zwingend) zwei Nachrichten (Tag *Messa-*

ge). Eine Nachricht für den Request mit Eingabeparametern (*input*) und eine Response-Nachricht mit Rückgabeparametern (*output*). Jede Nachricht beinhaltet eine oder mehrere *part*-Elemente, je eines für jeden Parameter einer Webservice-Methode. Allen Parametern sind Typen zugeordnet, die entweder Basistypen der WSDL oder eigene Typen (siehe *types*-Element) sein können.

An einem einfachen Beispiel soll das *message*-Element verdeutlicht werden. Diese Nachricht könnte die Request-Nachricht für die Methode *deleteArticle* unseres Artikelmanagers sein. Als Übergabe ist im *part*-Tag ein Parameter namens *PLZ* (Postleitzahl) vom Typ *string* definiert worden: [11]

```
<message name="deleteArticleInputMessage">
    <part name="katalogId" type="s:long"/>
</message>
```

Das nächste Beispiel definiert die Übergabeparameter an eine Methode unseres Artikelmanagers namens *findArticleByKatalogId*. Das part-Attribut definiert dabei die Eingabeparameter der *findArticleByKatalogId*-Methode. Dieses Element kann auf folgende Weise definiert werden: [12]

```
<message name="findArticleByKatalogIdInputMessage">
<part name="parameters" element="s0:findArticleByKatalogId" />
</message>
```

In der Message wird folgender Elementtyp als Parameter verwendet:

```
<s:element name="findArticleByKatalogIdInput">
  <s:complexType>
     <s:sequence>
    <s:element minOccurs="1" maxOccurs="1 name="KatalogId"
     type="s:long" />
    </s:sequence>
  </s:complexType>
</s:element>
```

Das Struktur-Element mit dem Namen *findArticleByKatalogIdInput* beinhaltet verpackt in einen komplexen Typ lediglich ein einfaches Element vom Typ *long*[13].

Die Rückgabe der Methode *findArticleByKatalogId* kann folgendermaßen definiert werden:

```
<message name="findArticleByKatalogIdOutput">
  <part name="returnvalue" element="s0:findArticleByKatalogIdResult" />
</message>
```

---

[11] Der Namensraum http://www.w3c.org/2001/XMLSchema wird im Beispiel mit „s“ abgekürzt.

[12] Die Abkürzung „s0“ wird in den Beispielen für den eigenen Namensraum genutzt.

[13] Man könnte dies sicher auch einfacher definieren und auf die Sequenz verzichten.

Hier ist Element im part-Attribut ein Verweis auf ein schon definiertes Element namens *findArticleByKatalogIdResult*, das wie folgt definiert werden kann:

```
<s:element name="findArticleByKatalogIdResult">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1"
        name="findArticleByKatalogIdResult"
        type="s0:Article" />
      </s:sequence>
  </s:complexType>
</s:element>
```

Der Rückgabewert der Methode *findArticleByKatalogId* ist also ein Datenobjekt vom Typ *Article*, der wiederum als Typ deklariert werden muss.

Das Element portType (in V1.2 interface)

Im Abschnitt *portType* werden Operationen deklariert. Eine Operation wird im *operation*-Tag beschrieben und besteht aus bis zu zwei Nachrichten. Sollen an eine Web-Methode Parameter übergeben werden (siehe *message*-Tag) wird im *input*-Element diese Nachricht festgelegt. Die Operation kann auch eine Nachricht für die Übermittlung eines Ergebnisses definieren. Diese Nachricht wird dann im *output*-Element bestimmt. Es werden nur die im Abschnitt *message* beschriebenen Nachrichten verwendet. Mehrere Operationen bilden schließlich einen Port-Typ, also ein Interface. Eine WSDL-Beschreibung kann mehrere *portType*- bzw. Interface-Elemente enthalten. Man unterscheidet vier Operationstypen in WSDL:

- *One-Way*: Einwegenachricht an den Diensterbringer ohne Antwort. Der Dienstnehmer schickt also keine Nachricht zurück.
- *Request-Response*: Klassische RPC-Kommunikation mit Anfrage des Dienstnehmers und Antwort des Diensterbringers.
- *Solicit-Response*: Senden einer Anfrage durch den Diensterbringer und Warten auf eine Antwort vom Dienstnehmer (also umgekehrt zum Request-Response-Operationstyp).
- *Notification*: Senden einer Nachricht durch den Diensterbringer ohne Warten auf eine Antwort vom Dienstnehmer.

Die Operationen unterscheiden sich im Wesentlichen in der Anzahl der Nachrichten (nur eine Input-Nachricht, eine Input- und eine Output-Nachricht). Im folgenden Beispiel ist die Operation *findArticleByKatalogId* einem Interface *ArticleMgrI* zugeordnet und es wird der Operationstyp „Request-Response“ verwendet.

```
<portType name="ArticleMgrI">
  <operation name="findArticleByKatalogId">
    <input message="s0:findArticleByKatalogIdRequestMessage" />
    <output message="s0:findArticleByKatalogIdResponseMessage" />
```

```
  </operation>
</portType>
```

Das *binding*-Element

Der Abschnitt *binding* beschreibt die Art und Weise wie der Webservice mit dem Client kommuniziert. Es wird festgelegt, welches Träger- bzw. Transportprotokoll[14] (z.B. SOAP auf Basis von HTTP) verwendet werden soll und wie die Daten der einzelnen Operationen kodiert (serialisiert) werden sollen. Für jedes unterstützte Transportprotokoll gibt es ein eigenes *binding*-Element, in dem das Protokoll im *transport*-Attribut festgelegt wird. Zudem wird für jede Operation das Kommunikationsmodell und die Art der Kodierung (Kodierungsstil) angegeben (siehe die Attribute *Style* und *Use*).

Im *soap:operation*-Element werden schließlich die Operationen angegeben, die dem Binding zugeordnet sind. Die Kodierung der Parameter und der Rückgabewerte wird mit dem *Use*-Attribut für jede Nachricht festgelegt. Der Wert „*encoded*“ bedeutet, dass das in der SOAP-Spezifikation festgelegte SOAP-Encoding verwendet wird. Der Wert „*literal*“ gibt dagegen an, dass die Daten gemäß dem XML-Schema-Standard serialisiert werden. Der Standard „*XML-Schema*“ war zum Zeitpunkt der SOAP-Spezifikation noch nicht fertig spezifziert, daher wurde in der SOAP-Spezifikation ein eigener Serialisierungsstandard entwickelt.

Das folgende Beispiel soll die Nutzung des Binding-Elements verdeutlichen. Der Name des Bindings wird mit *ArticleMgrSoapBinding* festgelegt und für den Transport wird HTTP definiert. Die Daten werden als RPC-Aufruf strukturiert und es gibt nur eine Operation, die dem Binding zugeordnet ist.

```
<binding name="ArticleMgrSoapBinding" type="s0:ArticleMgrI">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="rpc" />
    <soap:operation
     soapAction=http://isys-software.de/findArticleByKatalogId/>
     <input>
      <soap:body use = "literal"/>
     </input>
     <output>
      <soap:body use = "literal"/>
     </output>
    <soap:operation>
</binding>
```

[14] HTTP ist eigentlich kein Transportprotokoll (Schicht 4). Daher verwenden wir gelegentlich den Begriff Trägerprotokoll. Man weiß aber üblicherweise, was gemeint ist.

Im Beispiel wird das Binding für den Porttypen *ArticleMgrI* deklariert. Für jede Operation wird die korrespondierende SOAP-Aktion (im Beispiel nur eine) definiert und es ist die Kodierung der Nachrichten anzugeben (hier „use="literal"). „literal" bedeutet in diesem Zusammenhang, dass die Daten im SOAP-Body nach dem XML-Schema kodiert werden. Mit „use=encoded" würde man eine Kodierung nach den SOAP-Kodierungsregeln festlegen. Die Diskussion zu den Kodierungsstilen wird weiter unten wieder aufgegriffen.

Das *service*-Element

Das *service*-Element dient dazu die Lokation des Webservice für den Client transparent zu machen. Um den Service ansprechen zu können, muss dem Client dessen URI angegeben werden. Dafür ist das *soap:address*-Element zuständig. Im *port*-Element wird der Registrierungsname des Webservice angegeben, wobei das *biding*-Element sich auf die zuvor definierte Bindung bezieht.

```
<service name="ArticleMgrService">
  <port name="ArticleMgrI" binding="s0:ArticleMgrSoapBinding">
    <soap:address
     location="URL der Service-Lokation" />
  </port>
</service>
```

3.3.5 Universal Description, Discovery and Integration (UDDI)

Die erste Spezifikation von UDDI stammt aus dem Jahr 2000 und wurde in einer Zusammenarbeit der Firmen Microsoft, IBM und Ariba standardisiert. Seit dem Jahr 2002 obliegt die Weiterentwicklung dem Standardisierungskonsortium OASIS[15]. UDDI ist ein Standard für Webservice-Verzeichnisse (und auch für andere Verzeichnisse), mit denen Webservices verwaltet und gefunden werden können.

Einträge eines UDDI-Verzeichnisses sind in einem XML-Format beschrieben. Hierarchisch an erster Stelle steht ein *businessEntity*. Darin werden Informationen zum Dienstanbieter und UDDI-spezifische Daten wie beispielsweise der eingetragene Operator und ein UUID (eindeutige User-Id) für den Dienstanbieter festgelegt.

Der Zugriff auf das Verzeichnis zum Registrieren von Webservices und zum Suchen eines Webservice erfolgt über spezielle UDDI-APIs mit vordefinierten Funktionen. Dies sind eine sog. Publishing-API, über die ein Service-Provider einen Webservice veröffentlichen kann und eine Inquiry-API für den Client zum Suchen eines Webservice.

Wenn ein Dienstnutzer einen Webservice sucht, wendet er sich an ein UDDI-Verzeichnis und holt sich von dort die Schnittstellenbeschreibung in einer WSDL-

[15] OASIS steht für "Organization for the Advancement of Structured Information Standards"

Datei. Die WSDL-Beschreibung kann der Dienstnutzer dann verwenden, um eine entsprechende SOAP-Nachricht zu erzeugen und an den Diensterbringer zu senden.

Heute gibt es bereits einige Webservice-Verzeichnisse im Internet, wie etwa das Microsoft-UDDI (WWW-013) oder das IBM-UDDI (WWW-014). Die kommerzielle Nutzung von Webservices über das Internet ist allerdings noch nicht etabliert. Daher wird auf die weitere Beschreibung von UDDI verzichtet. Wenn man die Adresse eines Webservice (dessen URL) kennt, muss man UDDI nicht unbedingt nutzen. Heutige Anwendungen, die Webservices verwenden, arbeiten meist ohne UDDI.

3.3.6 Kommunikationsmodell, Nachrichtenformat und Kodierungsstil

Der Zusammenhang zwischen Kommunikationsmodell, Nachrichtenformat und Kodierungsstil ist aus historischen Gründen und aufgrund der Begriffswahl sehr verwirrend. Daher sollen die Zusammenhänge nochmals zusammengefasst werden. Bei der Erläuterung des WSDL-Binding-Elements wurde bereits die Kodierung der Nutzdaten angesprochen. Das Kommunikationsmodell, also die Art und Weise, wie die Kommunikationspartner miteinander kommunizieren, ist unabhängig vom Nachrichtenformat.

Beim Nachrichtenformat gibt es zwei Varianten, die mit „RPC“ und „Document“ bezeichnet werden (siehe Binding-Element, Style-Attribut):

- Im Nachrichtenformat „RPC“ steht im SOAP-Body genau ein XML-Element, das auch noch den Namen der Operation enthält, die aufgerufen werden soll. Zusätzlich enthält der SOAP-Body eine Liste von Parametern. Für jeden Parameter wird ein XML-Subelement übertragen. Nur dieses Nachrichtenformat wurde in der ersten SOAP-Version unterstützt.
- Im Nachrichtenformat „Document“ enthält der SOAP-Body ein XML-Kindelement oder sogar mehrere Kindelemente, die als Parts bezeichnet werden. Es werden keine SOAP-Kodierungsregeln für den Body angewendet. Der Aufbau der SOAP-Nachricht muss zwischen den Kommunikationspartnern abgestimmt werden und wird in XML formuliert.
- Der Wert „wrapped“ im Style-Attribut gibt an, dass alle Parameter einer Operation in ein Root-Schema-Element mit dem gleichen Namen wie die Operation eingebettet sind.

Die Bedeutung des Style-Attributs lässt sich aus der Namensgebung nicht exakt ableiten, denn man kann - wie bereits angedeutet - auch mit dem Dokumentenstil einen Request-Response-Ansatz realisieren oder mit dem RPC-Stil asynchrone Nachrichten versenden.

Kodierungsstile beschreiben das Marshalling von SOAP-Nachrichten und legen daher die Art der Kodierung fest (Marshalling, siehe Kapitel 2). WSDL nennt hierfür zwei Möglichkeiten (Kodierungsstile):

- „SOAP-Encoding“ wird im Attribut *encodingStyle* für beliebige Blöcke im SOAP-Body oder im *Use*-Attribut (siehe Binding) angegeben. In diesem Fall werden spezielle Serialisierungsregeln aus der SOAP-Spezifikation verwendet.
- „Literal“ wird im *Use*-Attribut (siehe Binding) angegeben. Bei dieser Art der Kodierung werden die Daten im SOAP-Body gemäß der Festlegung in der XML-Schema-Spezifikation kodiert.

Die Kombination aus Nachrichtenformat und Kodierungsstil ergibt mehrere Möglichkeiten (Nachrichtenformat = *Style*-Attribut, Kodierungsstil = *Use*-Attribut):

- *Style=Document/Use=Literal:* Der SOAP-Message-Body enthält hier ausschließlich Dokumente nach XML-Schema, d.h. die ganze Nachricht kann vollständig validiert werden. Dieser Stil wird innerhalb der Webservice-Community als favorisierte Lösung propagiert.[16]
- *Style=RPC/Use=Literal*: Hier werden die SOAP-Nachrichten als RPC-Aufrufe mit Parametern und Returncodes realisiert. Jede Nachricht hat einen eigenen Schematyp. Die Datentypen der Parameter werden in der Nachricht nicht explizit angegeben, sondern sie ergeben sich implizit aus der konkreten Schema-Beschreibung.
- Auch die Kombination Style=*RPC/Use=Encoded* ist möglich. Diese Kombination wird aufgrund der mangelnden Interoperabilität nicht mehr von allen Standards und Herstellern unterstützt. Heute verfügbare Webservice-Implementierungen nutzen diesen Kodierungsstil aber noch.
- Weiterhin kann man auch noch *Style=Document/Use=Literal wrapped* als Kombination verwenden (Beispiel siehe unten).

Das Ganze ist - wie man beim Lesen merkt – ziemlich verwirrend und nicht besonders glücklich gelöst. Einige Beispiel sollen daher die Nutzung der verschiedenen Nachrichtenformate und Kodierungsstile verdeutlichen:

Beispiel RPC/encoded: Die Nachrichten für einen *CreateArticle*-Dienst mit drei Eingabeparametern in der *CreateArticleRequestMessage* und einem Returnwert in der *CreateArticleResponseMessage* sehen in WSDL-Notation wie folgt aus:

```
<message name=CreateArticleRequestMessage”>
    <part name=“id”  type="xsd:int" />
        <part name="Warengruppe“ type="xsd:string" />
        <part name="Beschreibung“ type="xsd:string" />
        <part name="Preis"  type="xsd:double" />
</message>
<message name=CreateArticleResponseMessage”>
```

[16] Der Kodierungsstil wird im Binding-Element angegeben.

```
        <part name="return"  type="xsd:int" />
</message>
<portType name="ArticleMgrI">
    <operation name="createArticle">
        <input message="CreateArticleRequestMessage" />
        <output message="CreateArticleResponseMessage" />
</operation>
</portType>
<binding .../>
```

Eine konkrete SOAP-Nachricht für den Aufruf des *CreateArticle*-Dienstes kann dann folgendermaßen aufgebaut sein:

```
<soap:Envelope>
    <soap:Body>
        <CreateArticle>
          <ID>xsi:type="xsd:int">2000</ID>
          <Warengruppe> xsi:type="xsd:string">20</Warengruppe>
          <Beschreibung> xsi:type="xsd:string"> Vespa</Beschreibung>
          <Preis> xsi:type="xsd:float">2500</Preis>
    </soap:Body>
</soap:Envelope>
```

Aufgrund der in der Nachricht enthaltenen WSDL-Definitionen ist das XML-Dokument für einen Empfänger etwas schwer zu validieren. Weiterhin entsteht durch die Übertragung der Typdefinitionen ein gewisser Overhead.

Beispiel RPC/literal: Im Literal-Stil sieht eine Nachricht etwas anders aus. Anhand der *CreateArticle*-Request-Nachricht ist ersichtlich, dass sich *RPC/Literal* von *RPC/encoded* durch die fehlenden Typ-Angaben bei den Parametern unterscheidet. Die Nachricht wird damit kürzer und ist auch leichter zu validieren. Der Name des Dienstes (*CreateArcicle*) ist auch hier in der Nachricht, was sich vorteilhaft auf die Diagnose und auch auf das Dispatching im Zielknoten auswirkt.

```
<soap:Envelope>
    <soap:Body>
        <CreateArticle>
          <ID>2000</ID>
          <Warengruppe>20</Warengruppe>
          <Beschreibung>Vespa</Beschreibung>
          <Preis>2500</Preis>
          </CreateArticle>
    </soap:Body>
</soap:Envelope>
```

Beispiel Document/literal: In der WSDL wird in diesem Fall für jeden Parameter ein eigenes WSDL-Element definiert. In den Nachrichtenbeschreibungen wird darauf verwiesen. Wir betrachten wieder den *CreateArticle*-Dienst:

```
<types>
<schema>
    <element name="ID" xsi:type="xsd:int">2000</ID>
    …
</schema>
</types>
<message name=CreateArticleRequestMessage”>
    <part name="id” element="ID" />
    …
</message>
<message name=CreateArticleResponseMessage”>
    …
</message>
<portType name="ArticleMgrI">
…
```

Eine konkrete Nachricht kann dann wie folgt aussehen:

```
<soap:Envelope>
    <soap:Body>
        <ID>2000</ID>
      <Warengruppe>20</Warengruppe>
      <Beschreibung>Vespa</Beschreibung>
      <Preis>2500</Preis>
    </soap:Body>
</soap:Envelope>
```

Das übertragene XML-Dokument ist auch hier leicht zu validieren, da Standard-XML verwendet wird. Zudem entsteht kein Overhead durch die Übertragung von Typ-Information in der Nachricht.

Beispiel Document/literal wrapped: Im Unterschied zu *Document/literal* wird bei der Methode *Document/literal wrapped* nur noch ein XML-Element meist als komplexer Typ übergeben. Dieses XML-Element fasst alle Parameter einer Nachricht zusammen.

```
<types>
    <schema>
      <element name=„createArticle"
      <complexType>
        <sequence> ... Alle Parameter ... </sequence>
      </complexType>
      </element>
    </schema>
```

```
</types>
<message name=CreateArticleRequestMessage”>
      <part name=“createArticleRequest" />  …
</message>
…
```

Grundsätzlich sollte man sich bei der Entwicklung von Webservices auf die Kombination *Style=Document/Use=Literal* oder *Style=RPC/Use=Literal* beschränken. Alle Webservice-Plattformen dürften diese beiden Kombinationen kennen. Will man ganz sicher gehen, dass Webservices auch in heterogener Umgebung funktionieren, so ist die Kombination *Style=Document/Use=Literal* am besten, da diese auch in .NET-Webservices verwendet wird. SOAP-Encoding war historisch begründet und sollte nicht mehr verwendet werden.

3.3.7 WS-I-Standard

WS-I steht für *Web Service Interoperability Organisation* und ist ein offenes Herstellerkonsortium (WWS-001). WS-I hat die Aufgabe übernommen, die Interoperabilität von Webservice-Implementierungen zu verbessern. WS-I hat sich daher zum Ziel gesetzt, Standards im Webservice-Umfeld zu Profilen zusammenzustellen, Implementierungshinweise zu geben und Testtools für Conformance-Tests bereitzustellen. Insbesondere wurde ein „Webservice-Standards-Stack“ konzipiert (siehe Abbildung 3-10), der ständig weiterentwickelt werden soll. Wie man erkennen kann, werden hier neben den Kommunikationsmechanismen auch wichtige Aspekte des Zusammenspiels wie Security, Reliability, Transaktionshandling und Management von Webservices adressiert.

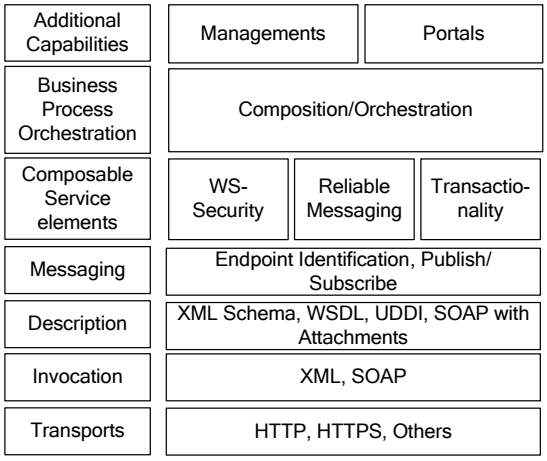


Abbildung 3-10: Webservice-Standards-Stack der WS-I

Durch WS-I werden spezielle Profile des Stacks definiert, die auch eine klare Zusammensetzung der Versionen der Einzelstandards vorgeben. So gibt es z.B. das sog. *Basic-Profile*, das die unteren Schichten *Transports*, *Invocation* und *Description* definiert.

Das Basic-Profile erlaubt in der Version 1.1 folgende Kombinationen aus Nachrichtenformat und Kodierungsstil:

- Style=Document/Use=Literal
- Style=RPC/Use=Literal

Die Kombination (Style=*RPC/Use=Encoded)* wird von WS-I nicht unterstützt.

3.3.8 Vorgehensweise zur Entwicklung von Webservices

Häufig werden bestehende Softwarebausteine in eine Webservice-Umgebung eingebettet. Abbildung 3-11 zeigt die Einbettung einer vorhandenen Softwarekomponente (im Beispiel unser Artikelmanager) in eine Webservice-Laufzeitumgebung. Im Wesentlichen ist ein Wrapping der bestehenden Komponente erforderlich, um die angebotenen Dienste als Webservice darzustellen. Hier eignet sich die Webservice-Technik also sehr gut für die Integration der bestehenden Komponente in neue Applikation.

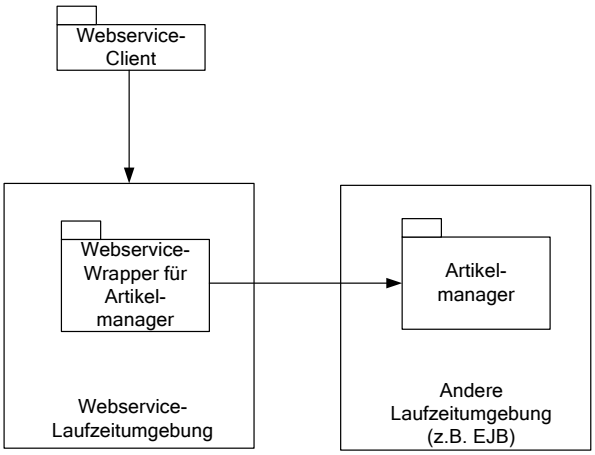


Abbildung 3-11: Einbettung in die Laufzeitumgebung

Damit die Methoden des Artikelmanagers über Webservices verwendet werden können, sind die Datentypen und Methoden in WSDL zu beschreiben. Dabei muss überlegt werden, welche Ein- und Ausgabeparameter für die einzelnen Methoden (Operationen) notwendig sind.

Am Beispiel unseres Artikelmanagers soll im Folgenden die Vorgehensweise bei der Entwicklung von Webservices skizziert werden. Wir gehen bei unserem Beispiel von dem typischen Szenario aus, in dem bereits eine Artikelmanager-Komponente mit der oben angegebenen Schnittstelle existiert und diese um eine Webservice-Schnittstelle erweitert werden muss.

Zunächst einmal kann man festhalten, dass die Dienste des Artikelmanagers auf einen Webservice mit einem Porttyp, der mehrere Operationen enthält, abgebildet werden können. Eine Java- oder sonstige Methode oder Prozedur entspricht dabei einer Webservice-Operation. Wir betrachten im Folgenden (etwas vereinfacht mit nur zwei Operationen) das Articlemanager-Interface. Die serverseitig implementierte Komponente bietet folgende Schnittstelle (hier in Java-Notation):

```
public interface ArticleMgrSimpleI extends Remote
{
    …
    public void createArticle(Article a)
      throws ArticleMgrExistException, RemoteException;
    public Vector findArticlesByGroup(long warengruppe) throws
      throws ArticleMgrNotFoundException, RemoteException;
    …
}
```

Für die Kommunikation erscheint das Request-Response-Modell angemessen. Dies entspricht dem Konzept des Artikelmanagers. Die beiden Methoden haben jeweils Eingabeparameter, Returnwerte und Exceptions. Es muss nun überlegt werden, wie diese auf die Mechanismen von Webservices abbildbar sind. Das Exception-Handling wird dabei zur Vereinfachung ausgeblendet:

- Wie bereits angedeutet wird das Interface auf genau einen Webservice und dessen Java-Methoden auf Webservice-Operationen abgebildet.
- Die Eingabeparameter lassen sich relativ einfach auf WSDL-Datentypen abbilden, da es sich nur um *long*-Werte handelt.
- Der Rückgabewert der Methode *CreateArticle* ist *void*. Hierfür ist also in WSDL nichts festzulegen.
- Etwas schwieriger gestaltet sich die Rückgabe des Artikel-Vectors bei der Methode *findArticlesByGroup*. Eine Übergabe von sprachabhängigen Objektinstanzen (Java- oder C#-Objekte) als Eingabe- und Ausgabeparameter einer Operation ist nicht sinnvoll, um Sprachabhängigkeiten zu vermeiden. Ein serialisiertes Java-Objekt kann in einer C#-Umgebung nicht bearbeitet werden und umgekehrt. Der Vektor muss also auf einen eigenen WSDL-Datentypen abgebildet werden. Hier sollte allerdings beachtet werden, dass eine Erzeugung einer WSDL-Datei durch vorhandene Tools problematisch ist. JAX-RPC (siehe unten), eine API-Implementierung von Webservices in

Java unterstützt z.B. keine Vektoren, sondern nur klassische Java-Arrays (wie z.B. Article[]).[17]

– Jede Operation benötigt aufgrund des gewählten Request-Response-Modells eine Request- und eine Response-Nachricht.

Die WSDL-Datei stellt das Verbindungsstück zwischen dem Diensterbringer (Server) und dem Dienstnehmer (Client) dar. Prinzipiell gibt es zwei Möglichkeiten, eine WSDL-Datei zu erstellen. Entweder über einen Top-Down-Ansatz, indem zunächst eine WSDL-Datei erstellt wird und dann aus dieser automatisch der notwendige Code (Stubs und Skeletons bzw. Ties[18]) zum Einsatz in Programmen generiert wird. Danach wird in diesem Fall der Code für die eigentliche Business-Logik programmiert bzw. eingebunden werden. Die zweite Variante ist ein Bottom-Up-Ansatz. In diesem Fall wird zunächst die Business-Logik programmiert und anschließend daraus bzw. aus den Schnittstellenbeschreibungen eine WSDL-Datei mit Stubs und Skeletons generiert. Die Elemente der WSDL sind zwar einfach zu verstehen, dennoch ist das Formulieren eines WSDL-Schemas relativ aufwändig. Darum gibt es bei den führenden Technologien (Java, .NET) Tools, die zu bestehenden Webservices die WSDL automatisch erzeugen. Auch die umgekehrte Richtung ist mit Tools automatisch zu erzeugen. Aus einer bestehenden WSDL lässt sich wieder ein Code generieren gegen den ein Client entwickelt werden kann.

Typisch für den Bottom-Up-Ansatz ist es, dass ein Webservice-Anbieter eine WSDL-Beschreibung bereitstellt, die vom WSDL-Nutzer zur Programmierung seiner Client-Anwendung verwendet wird. Die WSDL-Datei kann nun als Input für einen *Stub-Compiler* dienen, der daraus Client-Stubs und Server-Skeletons ähnlich wie bei RPC, CORBA oder RMI generiert. Die Zusammenhänge der Generierung sind in Abbildung 3-12 dargestellt.

Wie im Bild gezeigt wird, stellt der Service-Anbieter eine WSDL-Beschreibung für seinen Service bereit. Aus dieser WSDL-Beschreibung kann dann sowohl für den Client als auch für den Server ein Stub bzw. ein Skeleton erzeugt werden. Der Vorgang wird üblicherweise über einen WSDL-Compiler automatisiert. Das Skeleton stellt den Aufrufmechanismus und die Marshalling-/Unmarshalling-Mechanismen sowie die Kommunikationsbausteine bereit. Die spezielle Programmierung des Dienstes erfolgt in einem Anwendungsobjekt. Der Zugriff auf den Dienst wird für den Client durch den Stub, der ein Proxy-Objekt darstellt, gewährleistet. Im

---

[17] Hinweis: Generell sollten die Datentypen von Parametern und Rückgabewerten für Webservice-Operationen genau betrachtet werden, wenn die WSDL-Notation aus bestehenden Anwendungskomponenten generiert werden. *Java2WSDL (Tool von Axis), wscompile* oder sonstige Tools erkennen nicht alle Typen, was dann dazu führt, dass es Laufzeitprobleme geben kann. Dies gilt nicht nur für Java, sondern auch für andere Progammiersprachen. Es empfiehlt sich, möglichst einfache Parametertypen zu nutzen.

[18] Engl. Tie = Verbindung.

Bild wird auch gezeigt, dass im Service-Nutzer und im Service-Anbieter eine SOAP-basierte Middleware als Laufzeitumgebung vorhanden sein muss.

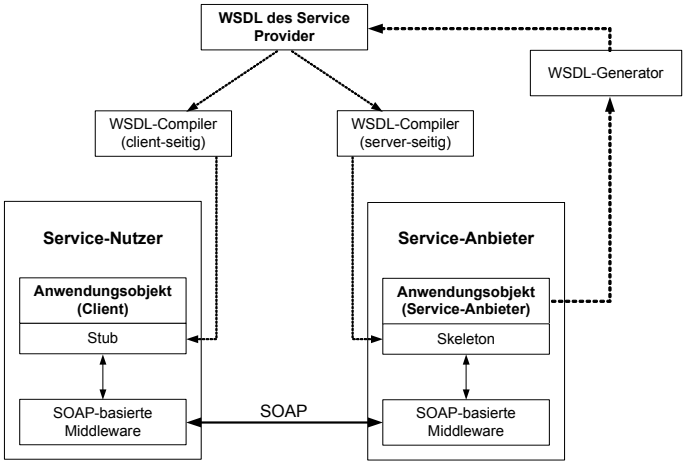


Abbildung 3-12: Generierung von Stub und Skeleton aus WSDL-Beschreibungen nach WWW-012

Eine manuelle Umsetzung des vereinfachten Java-Interface in eine WSDL-Notation wird im Folgenden unter Berücksichtigung der genannten Entscheidungen skizziert[19]. Eine konkrete Übersetzung von Java nach WSDL hängt von dem eingesetzten Übersetzungswerkzeug ab. Das Ergebnis kann beispielseweise wie folgt aussehen (Nachrichtenformat *document*, Kodierungsstil *literal*):

```
<!--Version und Kodierung -->
<?xml version="1.0" encoding="utf-8"?>

<!-- WSDL Beschreibung für eine einfache Artikelmanager-Schnittstelle. -->

<!--Benötigte Namensräume angeben: Als Namensraum für den Artikelmanager
wird http://ArticleMgr festgelegt und mit s0 abgekürzt -->
<definitions
       xmlns:http=http://schemas.xmlsoap.org/wsdl/http/
       xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
       xmlns:s="http://www.w3.org/2001/XMLSchema"
       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
```

[19] Der Leser möge selbst einmal eine manuelle Umsetzung versuchen, um festzustellen, wie wichtig eine automatisierte Umsetzung mit Werkzeugen ist.

```
        xmlns:tm=http://microsoft.com/wsdl/mime/textMatching/
        xmlns:s0=http://ArticleMgr/
        ...>

<!-- Eigene Typen definieren -->

  <types>

<s:schema elementFormDefault="qualified"
         targetNamespace=http://ArticleMgr/
         xmlns="http://www.w3.org/2001/10/XMLSchema">

    <s:complexType name="Article">
         <s:sequence>
             <s:element minOccurs="1" maxOccurs="1" name="Group"
                   type="s:long" />
             <s:element minOccurs="0" maxOccurs="1" name="Beschreibung"
                   type="s:string" />
             <s:element minOccurs="1" maxOccurs="1" name="Preis"
                   type="s:float" />
             <s:element minOccurs="1" maxOccurs="1" name="Warengruppe"
                   type="s:long" />
         </s:sequence>
    </s:complexType>
    …
    <s:complexType name="ArticleMgrResult">
         <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="Returnvalue"
            type="s:long" />
         </s:sequence>
    </s:complexType>

    <s:complexType name="ArticleList">
         <s:sequence>
            <s:element minOccurs="0" maxOccurs="unbounded"
            type="s0:Article" />
         </s:sequence>
    </s:complexType>

  </s:schema>

   </types>
```

```
<!-- Nachrichten für die Requests und Responses festlegen -->

   <message name="createArticleRequest">
       <part name="Article" element="s0:Article" />
   </message>
   <message name="createArticleResponse">
       <part name=" Result" element="s0:ArticleMgrResult" />
   </message>
   <message name="findArticlesByGroupRequest">
       <part name="Warengruppe" element="s0:long" />
   </message>
   <message name="findArticlesByGroupResponse">
       <part name="Result" element="s0:ArticleList" />
   </message>

<!-- Interface mit zwei Operationen -->

  <portType name="ArticleMgrSimpleI">
    <operation name="createArticle">
      <input message="s0:createArticleRequest" />
      <output message="s0:createArticleResponse" />
    </operation>
    <operation name="findArticlesByGroup">
      <input message="s0:findArticlesByGroupRequest" />
      <output message="s0:findArticlesByGroupResponse" />
    </operation>
  </portType>

<!—Bindings -->

  <binding name=" ArticleMgrSoapBinding " type="s0:ArticleMgrSimpleI">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
              style="document" />

    <operation name="createArticle">
       <soap:operation soapAction="http://…/createArticle"
       style=" document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
```

```
    </operation>

    <operation name="findArticlesByGroup">
      <soap:operation soapAction="http://…/findArticlesByGroup"
       style="document"/>
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>

<!—Service-Definition -->

  <service name="ArticleMgrService">
    <port name="ArticleMgrI" binding="s0:ArticleMgrSoapBinding">
       <soap:address location=
              "URL, unter der der Service zu finden ist"/>
    </port>
  </service>
</definitions>
```

Selbstverständlich sind auch andere Varianten für eine Umsetzung nach WSDL denkbar. Generierungstools verwenden ganz eigene Namenskonventionen. Wie man nun einen Webservice konkret auf Basis von vorhandenen Webservice-, Entwicklungs- und Laufzeitumgebungen realisieren kann, soll im Folgenden anhand von Fallbeispielen erläutert werden.

3.4 Java-Technologien für Webservices

3.4.1 Unterstützte Standards und Ablaufumgebungen

In Java werden einige API-Standards vorgegeben, die zum Ziel haben, Java-Komponenten unterschiedlicher Container (EJB, Servlet) mit einem Zugang über eine Webservice-Schnittstelle zu versehen. In J2EE 1.4 und auch in Java EE 5 und EJB 3.0 sind folgende Standards integriert:

- SAAJ: SOAP with Attachments API for Java
- JAXR: Java API for XML Registries
- JAXB: Java Architecture for XML Binding
- JAX-RPC: Java API for XML-bases RPC
- JAX-WS: Java API for XML Web Services.

SAAJ dient vor allem dazu, den internen Aufbau von SOAP-PDUs von eigenen Programmen aus zu beeinflussen (Header setzen usw.). Über JAXR kann der Zugriff auf UDDI-Verzeichnisse programmiert werden. JAXB dient dem Marshalling von SOAP-Requests. JAX-RPC und das neue JAX-WS dienen dazu, Anwendungen mit Webservices für alle Kommunikationspartner, also z.B. sowohl für die Server- als auch für die Clientseite zu entwickeln. Wir werden uns im Folgenden mit JAX-RPC und JAX-WS befassen.

Im EJB-Standard 3.0 wird sowohl der *Document/Literal-* als auch der *RPC/Literal-*Style unterstützt. Basis ist hierfür die Webservice-API JAX-WS (Java API for XML Web Services). Zudem wird auch noch die Stilform *RPC/Encoded* unterstützt, aber nicht mehr empfohlen.

Für die Entwicklung von Webservices ist eine Entwicklungsumgebung (ein Toolkit) und für den Betrieb eine Laufzeitumgebung (Webservice-Plattform) erforderlich. Es gibt sowohl für Java-, als auch für .NET-Umgebungen entsprechende Entwicklungs- und Laufzeitsysteme. Zudem gibt es verschiedene Toolkits und Laufzeitumgebungen. In der Webservice-Laufzeitumgebung ist ein sog. *SOAP-Prozessor* erforderlich, der SOAP-Nachrichten bearbeiten kann. Ein SOAP-Prozessor (auch SOAP-Engine) verarbeitet ankommende und abgehende SOAP-Nachrichten. Das Verarbeiten einer Nachricht bedingt, dass der SOAP-Prozessor über Informationen zu den konkret ablaufenden Webservices verfügen muss. Hierzu gehört das Wissen über das Kommunikationsmodell (Request-/Response-Modell, Einwegnachricht,...) und die verwendeten Transportmechanismen.

Weiterhin müssen die Nachrichtenbeschreibungen vorliegen. Heute unterstützen fast alle Application-Server-Lösungen auch Webservices. Ein typischer SOAP-Prozessor ist z.B. *Axis*, der in den Servlet-Container Apache Tomcat eingebunden werden kann. Axis enthält zudem Werkzeuge zur Generierung von Java-Klassen aus WSDL-Dateien und umgekehrt. Andere Implementierungen der Webservices-Spezifikationen sind JWSDP (Java Web Services Development Pack (Sun Microsystems) und die JBoss/WS im JBoss Application-Server.

3.4.2 JAX-RPC

JAX-RPC steht für *Java APIs for XML based RPC*, ist mit JSR 101 (Java Specification Request) spezifiziert worden und stellt eine API für die Entwicklung von Java-basierten Webservices dar. In Abbildung 3-13 ist die Bearbeitung eines Webservice auf Basis vonJAX-RPC skizziert. Ein Client sendet einen RPC-Request über einen generierten oder dynamisch erzeugten Stub und über das JAX-RPC-Laufzeitsystem an den Webservice, der seinerseits über ein Skeleton (auch Tie genannt) und eine JAX-RPC-Ablaufumgebung verfügen muss. Die Stubs und Skeletons sind wie beim klassischen RPC-Mechanismus für die Serialisierung und Deserialisierung der übermittelten Parameter und Returnwerte zuständig. Neben einer statischen Erzeugung von Stubs und Skeletons ist auf eine dynamische Vari-

ante vorhanden, in der Proxies zur Laufzeit erzeugt werden. Diese Varianten werden als *Dynamic Proxies* und *Dynamic Invocation* bezeichnet.

Die Übergabe der Parameter erfolgt per Value (call-by-value). Als Transportprotokoll wurde HTTP festgelegt. Wenn ein Client einen Request an den Server senden möchte, muss er diesen zunächst in eine SOAP-Nachricht packen und anschließend die SOAP-PDU in eine HTTP-PDU. Auf der Serverseite muss der Request ausgepackt werden und dann über einen Dispatching-Mechanismus an den adressierten Endpunkt übergeben werden. Die Antwortnachricht wird entsprechend verarbeitet. Diese Arbeiten werden vom JAX-RPC-Laufzeitsystem ausgeführt.

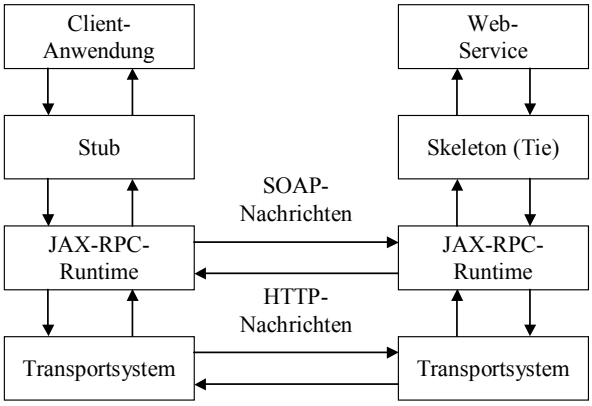


Abbildung 3-13: Webservice-Aufrufs mit JAX-RPC

Die Abbildung der Java-Objekte bzw. Datentypen auf WSDL-konforme Datentypen und umgekehrt wird von den Stubs und Skeletons durchgeführt. Ein Codegenerator erzeugt üblicherweise auch die Serialisierungs- und Deserialisierungsmethoden für diese Aufgabe.[20]

Zur Bereitstellung eines Webservice wird serverseitig ein sog. *Service-Endpoint* definiert. Ein Service-Endpoint stellt eine Schnittstelle für Webservice-Clients bereit, wobei die Technologie, die der Client nutzt, unerheblich ist. Die Schnittstelle wird mit Java-Hilfsmitteln als Remote-Interface beschrieben. Die Implementierung des Service-Endpoints erfolgt über die Implementierung des Remote-Interface als Java-Remote-Objekte (im Sinne der RMI-Programmierung). Aus der Java-Interface-Beschreibung können dann eine WSDL-Beschreibung (als XML-Datei) und die erforderlichen Stubs bzw. Skeletons erzeugt werden. Die WSDL-Beschreibung

[20] In JWSDP wird hierfür das Tool *wscompile*, in Apache Axis das Tool *Java2WSDL* bereitgestellt.

kann auch von den Client-Programmierern genutzt werden, um eine Stub-Generierung für eine andere Technologie (z.B. .NET Webservices) vorzunehmen.

Aus unserem bekannten Beispiel des Java-Artikelmanagers kann ein Webservice generiert werden, der mehrere Operationen (*createArticle*,...) enthält. Das Auffinden des Webservice und das Dispatching der entsprechenden Operationen zur Laufzeit werden durch das JAX-RPC-Laufzeitsystem ermöglicht. Auch eine vorhandene WSDL-Beschreibung kann auf eine Java-Schnittstelle abgebildet werden.

JAX-RPC legt fest, wie das Mapping der WSDL-Datentypen auf Java-Datentypen erfolgen soll. Tabelle 3-1 zeigt die wesentlichen Java-Datentypen und die zugehörigen Datentypen aus der XML-Schema-Spezifikation. Komplexe Java-Datentypen wie Arrays werden auf komplexe WSDL-Typen (wsdl:complexType) abgebildet. Weiterhin werden Java-Exceptions auf das *fault*-Element von WSDL abgebildet.

Java-Objektklassen können auch auf WSDL-Datentypen abgebildet werden. Beim Mapping sind die in der JAX-RPC-Spezifikation definierten Regeln zu beachten, sonst kann ein *WSDL*-Generator keine vernünftige Codegenerierung durchführen. Ein Ausschnitt aus dem Regelwerk soll erwähnt werden:

- Die Java-Objektklassen, aus denen eine WSDL-Beschreibung erzeugt werden soll, müssen einen Standard-Konstruktor, der als *public* definiert wird, implementieren.
- Die Java-Objektklassen müssen die Java-Bean-Konventionen einhalten. Beispielsweise müssen sie für alle Attribute Getter- und Setter-Methoden bereitstellen.
- Java-Objektklassen dürfen nicht das Remote-Interface des Package java.rmi implementieren.

Tabelle 3-1: Mapping der von JAX-RPC unterstützten Datentypen auf WSDL-Typen

Java-Typ	**XML-Schema-Typ**
boolean	xsd:boolean
byte	xsd:byte
double	xsd:double
short	xsd:short
java.lang.String	xsd:string
java.lang.Calendar	xsd: dateTime
byte[]	xsd:base64Binary
…	

Alle genannten Datentypen, auch die komplexeren Klassen, die sich an die genannten Konventionen halten, können sowohl als Parameter als auch als Return-

werte verwendet werden. Zudem ist die Nutzung dieser Datentypen in Arrays (Beispiele: int[], String[], Kunde[]) möglich. Eine Unterstützung von Collector-Klassen (Java-Vector, usw.) ist nicht sinnvoll, da Nicht-Java-Kommunikationspartner damit nicht umgehen können. Gemäß Spezifikation muss ein Hersteller eines JAX-RPC-Laufzeitsystems folgende Kombinationen aus Nachrichtenformat und Kodierungsstil (in der Spezifikation als *operation modes* bezeichnet) implementieren:

- Style=Document/Use=Literal
- Style=RPC/Use=Literal
- Style=RPC/Use=Encoded

Optional ist die Kombination *Style=Document/Use=Encoded* möglich.

Eine ausführliche Beschreibung zum WSDL-to-Java-Mapping ist in der JAX-RPC-Spezifikation nachzulesen (WWW-020).

Alle JAX-RPC-Klassen sind im Java-Package javax.xml.rpc enthalten. Auf der Client-Seite sind vor allem folgende Interfaces und Klassen relevant:

- Stub-Interface: *javax.xml.rpc.Stub*
- Schnittstellen für den dynamischen Aufruf eines Websservice: *javax.xml.rpc.Call* und *javax.xml.rpc.Service*
- Factory für die Erzeugung von Service-Objekten: *javax.xml.rpc.ServiceFactory*
- Ausnahmebehandlung: *javax.xml.rpc.JAXRPCException*

Hinzu kommen einige Basisklassen und Interfaces, die für eine Generierung von Stubs und Skeletons verwendet werden. Ein JAX-RPC-Laufzeitsystem muss diese Klassen und Interfaces implementieren. JAX-RPC hat verschiedene Möglichkeiten, als Webservice-Client auf Webservices zuzugreifen. Der Zugriff kann von einem J2SE-Client oder auch von einer JEE-Komponente, wie z.B. aus einer EJB-Session-Bean heraus erfolgen. Hierfür stellt JAX-RPC Schnittstellen bereit. Auf der Client-Seite ist ebenfalls ein JAX-RPC-Laufzeitsystem erforderlich, das im JEE-Fall auch im EJB-Container integriert sein kann. Die konkrete Implementierung ist hier allerdings nicht festgelegt. In JAX-RPC gibt es für die Implementierung eines Webservice-Clients folgende Möglichkeiten:

- Nutzung von generierten Stubs mit dynamischer Konfigurierungsmöglichkeit
- Nutzung von sog. Dynamic Proxies
- Nutzung des sog. Dynamic Invocation Call Interface (DII)

Diese Möglichkeiten sollen im Folgenden kurz vorgestellt werden.

Statische Stubs mit dynamischer Konfigurierungsmöglichkeit:

Eine Stub-Instanz repräsentiert einen clientseitigen Proxy für den Service-Endpunkt. Das Protokoll-Binding und der Service-Endpoint sind hier bereits im Stub-Code festgelegt. Der Client-Stub für einen Webservice kann statisch vorkonfiguriert werden und kann auch vom Webservice-Client dynamisch über das Stub-

Interface nachkonfiguriert werden. Diese Variante entspricht dem CORBA- oder RMI-Stub-Modell und kann als Standardvariante betrachtet werden.

Ein generierter Stub ist üblicherweise an ein vorgegebenes Transportprotokoll gebunden. Die Spezifikation verlangt hier keine generischen Stubs. Das Stub-Interface enthält aber auch Methoden zum Setzen und Auslesen von Eigenschaften (Properties) zur Laufzeit und sieht wie folgt aus:

```
package javax.xml.rpc;
public interface Stub {
     …
     void _setProperty(String name, Object value);
     Object _getProperty(String name);
     java.util.Iterator _getPropertyNames();
     …
}
```

Man kann also bestimmte Eigenschaften über die Stub-Instanz dynamisch über die Methode *setProperty* konfigurieren. Typische Eigenschaften, die verändert werden können, sind z.B. das Protokoll-Binding und der Service-Endpunkt.

Konkrete Produkte, die JAX-RPC unterstützen, erweitern das Stub-Interface bei der Codegenerierung in der Regel um die speziellen Methoden (Operationen) eines in WSDL notierten Webservice.

Beispiel: Eine generierte Klasse sieht dann beispielsweise wie folgt aus:

```
public class ArticleMgrSimpleIGenerierterStub[21]
   extends Basispackage.Stub implements ArticleMgrSimpleI {
     …
     void _setProperty(String name, Object value);
     Object _getProperty(String name);
     java.util.Iterator _getPropertyNames();
     // Individuelle Webservice-Methoden (Operationen)
     public void createArticle(…);
     …
}
```

Die Nutzung eines Stubs könnte dann, stark vereinfacht, wie folgt aussehen:

```
…
public class MyArticleMgrSimpleClient {
     …
     public static void main(String[] args) {
        …
```

[21] Vom Generierungstool frei vergebener Name.

```
        // Stub instanziieren, implementierungsabhängig
        ArticleMgrSimpleIGenerierterStub stub =
           (ArticleMgrSimpleIGenerierterStub) new
               ArticleMgrSimpleIGenerierterStub;
        stub._setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY, …);
           …
        // Konkrete Operation aufrufen
        stub.createArticle(…);
     …
     }
     …
}
```

Dynamic Proxy:

JAX-RPC unterstützt auch die dynamische Erzeugung eines Zugangs zu einem Webservice. Eine Variante hierzu ist die Nutzung eines dynamischen Proxies. Dieser erlaubt es, einen Webservice zur Laufzeit aufzurufen, ohne vorher eine Stub-Generierung vornehmen zu müssen. Über einen Aufruf der *getPort*-Methode aus dem Service-Interface kann man sich eine Referenz auf einen dynamischen Proxy besorgen. Das Service-Interface sieht vereinfacht wie folgt aus:

```
package javax.xml.rpc;
public interface Service {
     java.rmi.Remote getPort(QName portName,
           class serviceEndpointInterface) throws ServiceException;
     ...
}
```

Beispiel: Die Nutzung dieser Variante soll der folgende Codeausschnitt zeigen:

```
public class MyArticleMgrSimpleClient {
     …
        public static void main(String[] args) {
           …
           // Service-Instanz erzeugen
           Service ArticleMgrSimpleService = …; // implementierungsabhängig
           …
           ArticleMgrSimpleI port = ArticleMgrSimpleService.getPort(…);
           // Konkrete Operation aufrufen
           port.createArticle(…);
     }
}
```

In der JAX-RPC-Spezifikation ist nicht festgelegt, dass das Service-Endpunkt-Interface dynamisch gegen die WSDL-Definition validiert werden muss. Dies kann

der Hersteller selbst entscheiden. Dadurch sind im Gegensatz zur Nutzung von statischen Stubs Laufzeitfehler möglich.

Dynamic Invocation Call Interface (DII):

Als weitere dynamische Variante ist noch das *Dynamic Invocation Call Interface (DII)* zu nennen. Dieses Modell entspricht dem CORBA-DII-Modell und ist dann sinnvoll, wenn zum Zeitpunkt der Entwicklung eines Clients die Serverseite nicht feststeht, also die Methodensignaturen unbekannt sind und daher erst zur Laufzeit ermittelt werden müssen. In diesem Fall wird auch vorab keine Generierung von clientseitigen Klassen vorgenommen.

Über das *Service*-Interface wird mit Hilfe der Methode *createCall* eine *Call*-Instanz erzeugt, die schließlich der Ausführung einer Webservice-Methode dient. Die Call-Instanz ermöglicht dann das Setzen von Properties und sonstigen Parametern. *Service*- und *Call*-Interface sind, sehr vereinfacht, wie folgt definiert:

```
public interface Service {
     Call createCall() throws ServiceException;
     Call createCall(QName portName, String OperationName)
        throws ServiceException;
     …
}

public interface Call {
     void addParameter(…);
     QName getParameterTypeByName(…);
     void setOperationName(…);
     void setPortByName(…);
     void setProperty(…);
     void setTargetEndpointAddress(…):
     Object invoke(QName operationName, Object[] inputParams)
        throws java.rmi.RemoteException;
     …
}
```

Über die Call-Instanz können z.B. die aufzurufende Operation (*setOperationName*), der Porttyp (*setPortTypeName*) und die Endpunkt-Adresse (*setTargetEndpointAddress*) mit Werten belegt werden. Weiterhin können Parameter und alle Eigenschaften gesetzt werden, die auch bei statischen Stubs verfügbar sind. Über die Methode *invoke* wird schließlich der Aufruf der Webservice-Methoden initiiert.

Beispiel: Ein konkreter DII-Client könnte wie folgt programmiert werden:

```
public class MyArticleMgrSimpleClient {
     public static void main(String[] args) throws Exception {
     // Service-Instanz erzeugen
     Service ArticleMgrSimpleService = …; // implementierungsabhängig
```

```
    // Erzeugung einer Call-Instanz
    Call call = (Call) ArticleMgrSimpleService.createCall();
    // Service-Endpunkt-Adresse setzen
    call.setTargetEndpointAddress(…);
    // Definition der Eingabeparameter, des Operationsnamens
    // und des Ergebnistyps
    call.addParameter(…);
    call.setReturnType(XMLType.XSD_STRING);
    call.setOperationName(…);
    …
    // Aufruf der Operation
    call.invoke(…);
    …
}
```

Diese Variante bietet am meisten Möglichkeiten, da alles zur Laufzeit belegt wird. Die Programmierung wird dadurch entsprechend komplexer.

Fallbeispiel: Apache Axis

Apache Axis (kurz Axis) stellt eine Weiterentwicklung von Apache SOAP dar, das wiederum aus SOAP4J (IBM)[22] hervorging. Als SOAP-Engine implementiert sie die JAX-RPC-API und auch die SAAJ-API. Axis beinhaltet auch einige Tools für die Entwicklung von Webservices

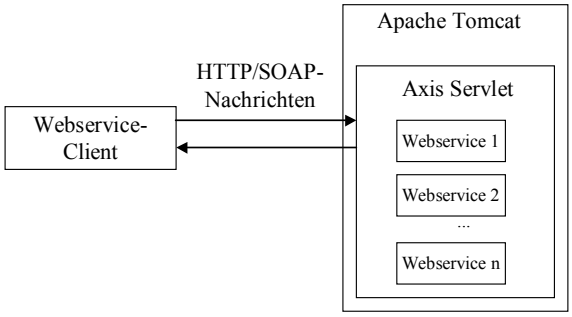


Abbildung 3-14: Axis-Architekturüberblick

Axis realisiert den Zugang zu Service-Endpunkten über ein Servlet und ist daher in einem Servlet-konformen Container wie Apache Tomcat (WWW-007) ablauffähig. Die eingehenden Aufrufe von Webservices (HTTP-Requests mit SOAP-PDUs) werden über das Servlet entgegengenommen und an die SOAP-Engine zur

[22] Die Entwicklung begann im Jahre 2000.

Bearbeitung weitergeleitet. Ankommende SOAP-PDUs sind in HTTP-PDUs verpackt. Der Servlet-Container entpackt diese und leitet sie an das Axis-Servlet weiter. Die Weiterleitung erfolgt mit Hilfe eines *doPost*-Aufrufs an das Axis-Servlet. Dort wird auch die SOAP-PDU entpackt und der Operationsaufruf an den adressierten Service-Endpunkt weitergeleitet. Die grobe Architektur von Axis ist in Abbildung 3-14 dargestellt.

Gemäß JAX-RPC kann die Programmierung von Webservices entweder Bottom-Up oder Top-Down erfolgen. Die Generierung einer WSDL-Datei mit Axis kann aus einer Java-Klasse oder einem Java-Interface im Top-Down-Ansatz durchgeführt werden. Hierzu wird das Tool *Java2WSDL* eingesetzt. Dieses Tool ist auch ein Java-Programm, das über verschiedene Optionen gesteuert werden kann. Bei Aufruf des Tools mit Angabe eines Java-Interface wird die Generierung durchgeführt. Es wird eine Datei erzeugt, die eine vollständige WSDL-Beschreibung für das Java-Interface enthält. Die Datei enthält neben den Beschreibungen aller Operationen, Messages, Porttypes, Services usw. auch für jede Java-Klasse, die als Parameter Verwendung findet, eine Beschreibung als komplexer WSDL-Datentyp.

Serverseitige Programmierung:

Bei der Bottom-Up-Vorgehensweise kann aus einer WSDL-Beschreibung eine Generierung von Stubs, Skeletons und Serialisierungs- sowie Deserialisierungscode über das Axis-Tool *WSDL2Java* erfolgen. Der Aufruf des Tools mit einer WSDL-Beschreibung für den einfachen Artikelmanager erzeugt folgende Filestruktur:

```
ArticlemanagementSimple (Verzeichnis)
        ArticlemanagementSimple (Verzeichnis)
        Article.java
        ArticleMgrSimpleI_pkg (Verzeichnis)
        ArticleMgrSimpleI.java
        ArticleMgrSimpleIService.java
        ArticleMgrSimpleIServiceLocator.java
        ArticleMgrSimpleISoapBindingStub.java
ArticleMgrSimpleISoapBindingSkeleton.java
ArticleMgrSimpleISoapBindingImpl.java
deploy.wsdd
undeploy.wsdd
```

Die Dateien können nun für die Entwicklung der Business-Logik auf der Serverseite, für den Aufruf des Webservice auf der Clientseite und für das Deployment verwendet werden. Abbildung 3-15 zeigt nochmals im Zusammenhang, welche Dateien über den Aufruf des Tools *WSDL2Java* erzeugt werden. Der Webservice-Client (*MyClient.java*) muss zusätzlich programmiert werden und die Server-Implementierung muss um die Business-Logik erweitert werden. Alle anderen Dateien werden automatisch generiert und müssen nicht mehr verändert werden.

Ein Java-Client muss z.B. mit Hilfe der JAX-RPC-API zunächst ein *ServiceLocator*-Objekt instanziieren, um dann eine Referenz auf den Artikelmanager-Service-Endpunkt zu bekommen. Über eine zusätzlich zu erzeugende Instanz eines Client-Stubs können die Methoden des Webservice aufgerufen werden.

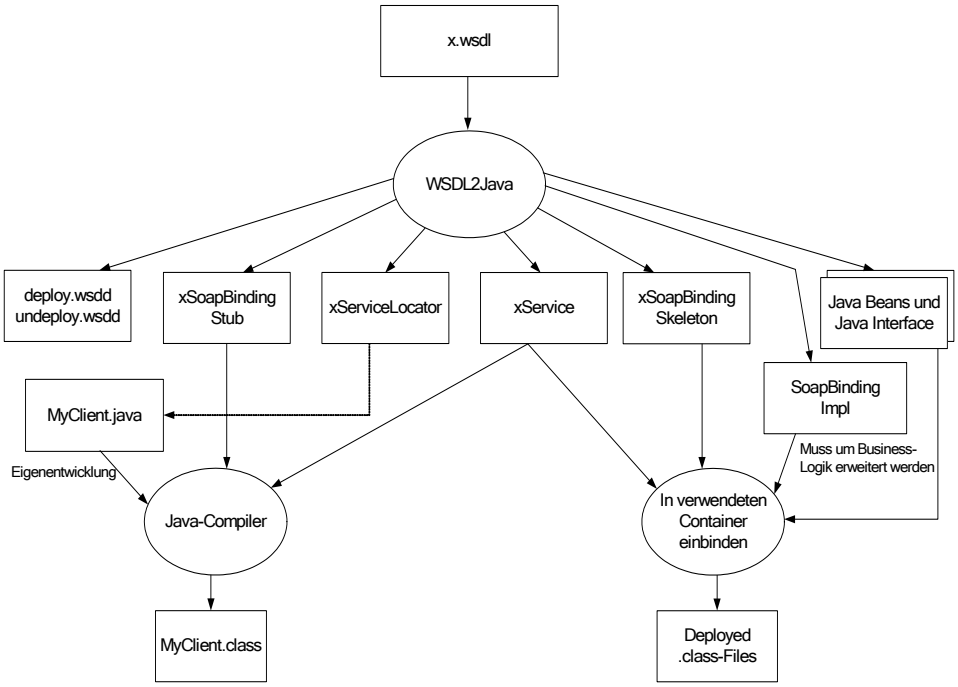


Abbildung 3-15: Codegenerierung mit Axis-WSDL2Java-Tool

In einer anderen Client-Umgebung (.NET, ...) können diese generierten Programme natürlich nicht verwendet werden. In diesem Fall muss über dort vorhandene Möglichkeiten aus der WSDL-Beschreibung eine eigene Stub-Generierung durchgeführt werden.

Das in Abbildung 3-16 skizzierte Klassendiagramm zeigt wichtige Basisklassen und Schnittstellen des Axis-Systems und die für unseren speziellen Webservice generierten Wrapperklassen. So wie dies in Axis realisiert ist, muss es nicht in anderen Webservice-Plattformen sein. Die Implementierung ist hier weitgehend frei. Die JAX-RPC-Spezifikation muss aber eingehalten werden, da sonst eine Kommunikation nicht möglich ist. In der Abbildung sieht man auch nochmals, dass serverseitig lediglich die Business-Logik implementiert werden muss. Alle anderen Interfaces und Klassen werden generiert bzw. sind im Axis-Basissystem enthalten und werden nur verwendet.

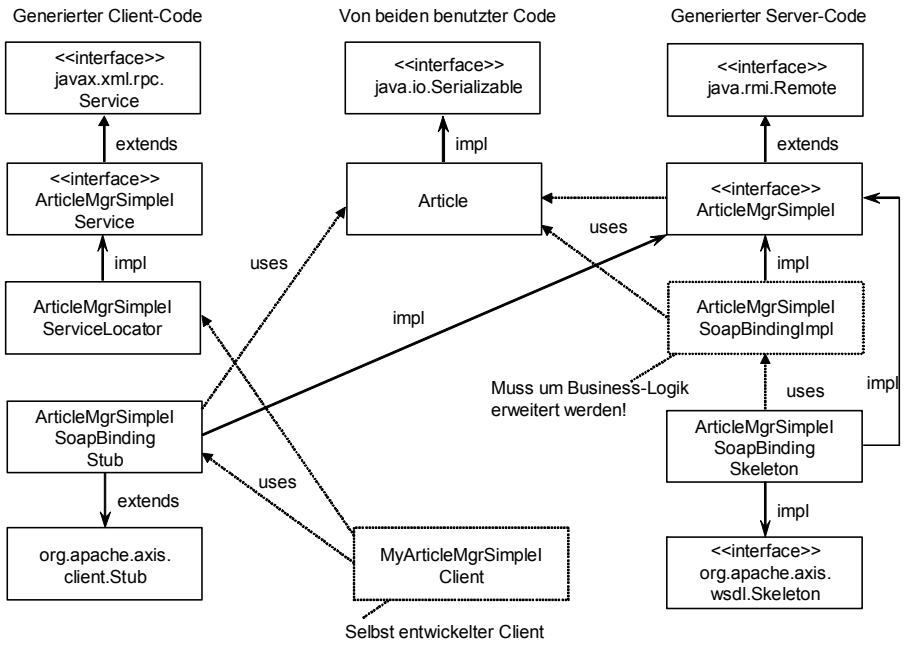


Abbildung 3-16: Klassenmodell für mit WSDL2Java generierte Klassen

Clientseitige Programmierung:

Für die Nutzung eines Webservice muss ein Webservice-Client (in Abbildung 3-16 *MyArticleMgrSimpleIClient* bezeichnet) entwickelt werden, der die generierten Java-Klassen nutzen kann. Aber auch ein Client, der in einer anderen Programmiersprache entwickelt wurde, kann - bei Einhaltung der Standards - den Webservice nutzen. Hierzu sind die entsprechenden Webservice-Mechanismen der jeweiligen Entwicklungsumgebung zu verwenden. Axis weicht allerdings vom Standard ab, wenn man spezielle Java-Mechanismen wie Exceptions und Collections nutzt. Java-Vektoren können z.B. durch die Tools bearbeitet werden, allerdings kann ein Kommunikationspartner, der nicht in Java realisiert ist, damit nicht sehr viel anfangen.

Bei der Implementierung eines Webservice-Clients in Java genügt die Kenntnis von einigen wenigen Axis-Basisklassen. Die wichtigsten Klassen *Stub*, *Call*, *ServiceFactory*, *Service*, *MessageContext* und *AxisFault* sind im Paket *org.apache.axis* enthalten. Die Klassen werden je nach Client-Variante (statischer Stub, dynamischer Proxy,…) verwendet. In Axis gibt es für die Implementierung eines Webservice-Clients die verschiedenen, im JAX-RPC-Standard spezifizierten Möglichkeiten.

Deployment:

Schließlich sollen noch einige Hinweise zum Deployment gegeben werden. Bei Axis muss ein Webservice mit einem Deployment-Descriptor[23] beschrieben werden. Die Webservice-Klassen müssen in die Axis-Umgebung eingebracht werden. Die Vorgehensweise für das Deployment ist in der Toolbeschreibung nachzulesen (WWW-007). Es gibt grundsätzlich zwei Deployment-Varianten:

- *Instant Deployment*: Dies ist ein vereinfachter Deploy-Mechanismus. Eine Java-Klasse wird in das Verzeichnis wepapps (*WEBAPPS/axis*) von Tomcat kopiert und umbenannt in eine Datei mit dem Suffix *.jws*. Diese Variante ist nur für einfachste Klassen ohne Java-Packages geeignet. Der Sourcecode muss verfügbar sein. Axis erzeugt beim ersten Aufruf alle notwendigen Klassen und die WSDL-Datei selbstständig.
- *Custom Deployment*: Bei dieser Deployment-Variante erfolgt das Deployment über Deployment-Descriptoren (Web Service Deployment Descriptors = WSDD). WSDDs können über das Tool *WSDL2Java* generiert werden. Über ein weiteres Axis-Tool namens *AdminClient* kann eine WSDD-Datei bei der Axis-Engine bekannt gemacht werden. Die generierten .class-Dateien werden unter dem Tomcat-Verzeichnis *\webapps\axis\WEB-INF\classes* abgelegt.

3.4.3 JAX-WS

Ziel von JAX-WS ist es, den Entwicklungsaufwand von JAX-RPC zu reduzieren und das Deployment von Webservices einfacher zu gestalten[24]. JAX-WS wird im JSR 181 spezifiziert. JAX-WS nutzt den JAXB-Standard für das Marshalling und Unmarshalling (Serialisierung/Deserialisierung). Dies bedeutet, dass das klassische XML-Parsing mit SAX/DOM-Parsern nicht mehr notwendig ist.

Serverseitige Programmierung:

JAX-WS unterstützt die Webservice-Entwicklung über spezielle Java-Annotationen (Burke 2006). Die wichtigsten Annotationen sind „@WebService“ und „@WebMethod“. Die Annotation „@WebService“ dient der Kennzeichnung einer stateless Session-Bean als Webservice. Die Annotation „@WebMethod“ wird verwendet, um einzelne Methoden der Session-Bean als Webservice-Operation zu bekanntzumachen. Wird die Annotation nicht verwendet, so werden alle Methoden als Webservice-Operationen bekanntgemacht, im anderen Fall nur die gekennzeichneten.

Die Annotationen werden in dem Java-Package *javax.jws* bereitgestellt. Die Annotation „@WebService“ ist mit entsprechenden Defaults für ihre Parameter definiert,

---

[23] Siehe hierzu z.B. die Dateien *webservices.xml* und *jaxrpc_mapping.xml*.

[24] Der Name der API ist auch nicht so irreführend wie JAX-RPC, bei dem man meinen könnte, es wäre nur Request-Reponse-Kommunikation möglich, was nicht der Fall ist.

so dass man sich auch nur auf die Angabe der Annotationen ohne weitere Parameter beschränken kann. Man kann aber auch einige Parameter wie den Namen oder den Servicenamen speziell angeben. Gleiches gilt für die Annotation „@WebMethod“. Wird hier der Operationsname nicht angegeben, so wird der Name der Java-Methode verwendet. Eine WSDL-Codegenerierung erfolgt durch die Entwicklungsumgebung, die durch den Container-Hersteller bereitgestellt wird.

Weitere Annotationen können optional verwendet werden. Hierzu gehören:

- „@SOAPBinding“
- „@WebParam“
- „@WebResult“
- „@OneWay“

Mit der Annotation „@SOAPBinding“ kann der Kodierungsstil beeinflusst werden. Standardmäßig wird als Kodierungsstil Document/Literal festgelegt. Es sind drei Parameter verfügbar: *Style* mit den Werten „DOCUMENT“ und „RPC“, *Use* mit den Werten „LITERAL“ und „ENCODED“ und *ParameterStyle* mit den Werten „BARE“ und „WRAPPED“ (Standardwert).

Die Angabe *Use*=“ENCODED“ wird nur aus Kompatibilitätsgründen erlaubt, sollte aber nicht mehr verwendet werden. Die Angabe des Parameters *ParameterStyle* beeinflusst die Notation im XML-Schema des Webservice (Burke 2006).

Über die Annotationen „@WebParam“ und „@WebResult“ können Angaben zu Parametern und Returnwerten der einzelnen Methoden gemacht werden. Über die Annotation „@OneWay“ kann festgelegt werden, ob eine Webmethode eine leere Nachricht als Antwort erhalten soll. Damit kann eine Optimierung durch eine asynchrone Ausführung der Methode ermöglicht werden.

Beispiel: Die Implementierung eines Artikelmanagers mit JAX-WS innerhalb einer EJB-Session-Bean kann grob wie folgt aussehen:

```
package…;
import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.jws.WebMethod;
@Stateless
@WebService (Name = “ArticleMgrSimple”,
  serviceName=”ArticleMgrSimpleService”)
@SOAPBinding (Style = Document, Use = Literal,
  ParameterStyle = Wrapped)
public class ArticleMgrSimple
{
  @WebMethod (OperationName = “createNewArticle”)
  public void createArticle(…) {…}
  @WebMethod
  public Article findArticleByKatalogId(…) {…}
```

```
…
}
```

Aus dem Namen des Webservice (Annotation „@WebService“) wird der WSDL-Porttyp generiert und die Namen der Methoden (Annotation „@WebMethod“) werden als WSDL-Operationsnamen verwendet. Der WSDL-Service-Name wird aus dem im Code angegebenen Service-Namen übernommen. Der Kodierungstil und das Nachrichtenformat wird aus der Annotaton „@SOAPBinding“ übernommen.

```
<portType name= "ArticleMgrSimple">
  <operation name = "createNewArticle">
  …
  </operation>
  …
</portType>
…
<binding name= … Style = "Document"…>
    <encodingStyle … use = "Literal">
<\binding>
…
<service
  name = "ArticleMgrSimpleService">
…
</service>
```

Webservices können in EJB-Session-Beans implementiert werden, sind aber nicht zwingend an diese gekoppelt. Es ist zwar nicht unbedingt erforderlich, dass ein Hersteller dies unterstützt, aber es ist durchaus sinnvoll, da nicht immer die EJB-Technologie verwendet wird.

Clientseitige Progammierung:

Für die Client-Programmierung stellt JAX-WS ähnlich wie JAX-RPC eine Java-Klasse *Service* bereit, über die ein Webservice aufgerufen werden kann. Eine eigene Klasse ist von der Klasse *Service* abzuleiten. Dabei ist eine Methode bereitzustellen, die einen Service-Endpunkt erzeugt. Eine weitere Annotation mit dem Namen „@WebServiceClient“ kann genutzt werden, um den Namen, den Namespace und die WSDL-Location des zu verwendenden Webservice anzugeben. Zudem ist die Annotation „@WebEndpoint“ erforderlich, um den Port des Webservice zu bestimmen.Webservice-Clients können beliebige Anwendungen sein.

- JAX-WS-Clients, die nicht in einem JEE-Container ablaufen und das JAX-WS-API direkt nutzen. Diese Client-Variante kann statisch generierte Stubs oder dynamische Service-Proxies verwenden.
- JEE-Clients, die in einem JEE-Container ablaufen. Eine Referenz auf den Service-Endpoint wird über JNDI ermittelt.

Beispiel 1: Die grobe Struktur eines Webservice-Clients, der einen dynamischen Proxy nutzt, zeigt das folgende Beispiel. Mit der Methode *create* wird ein Service erzeugt und mit der Methode *getPort* wird eine Verbindung zu einem Endpunkt hergestellt.

```
package…;
import javax.xml.ws.Service;
…
class MyClient {
    static String portType = "…";
    static String serviceName = "…";
    static String serviceEndpointAddress = "…";
    static String nameSpace = "…";

    public void main(….) {
        Url wsdlLocation = "…";
        QName serviceNameQ = new QName (nameSpace, serviceName);
        Service s = Service.create(wsdlLocation, serviceNameQ);
        ArticleMgrSimpleServiceI port =
          s.getPort(ArticleMgrSimpleI.class);

        // WS-Operationen über Port aufrufen
        …
}
```

Beispiel 2: Ähnliches kann einfacher über Annotationen erreicht werden. Der Client muss in diesem Fall aus der WSDL-Beschreibung Java-Klassen erzeugen. Weiterhin ist die Anwendung der zusätzlichen JAX-WS-Annotationen „@WebServiceClient“ und „@WebEndpoint“ erforderlich:

```
import javax.xml.ws.WebServiceClient;
import javax.xml.ws.WebEndpoint;
…
@WebServiceClient (name = "ArticleMgrSimpleService"
targetNamespace = "de.fhm.webservice.ArticleMgrSimple" wsdlLocation = "…")
public class ArticleMgrSimpleService extends javax.xml.ws.Service {
    public ArticleMgrSimpleService (…) {…}
    …
@WebEndpoint (name = "ArticleMgrSimplePort")
    public ArticleMgrSimpleI = getArticleMgrSimplePort() {…}
}
```

Erwähnt sei noch, dass für das Deployment in einem JEE-fähigen Container auf einen Deployment-Descriptor verzichtet werden kann. Wie bei EJB 3.0 werden alle Angaben, die für ein Deployment von Interesse sind, bereits in den Annotationen gemacht.

3.5 Zusammenfassung

Serviceorientierte Architekturen verbreiten sich zunehmend. Nahezu alle großen Unternehmen setzen sich mit dem Thema auseinander. SOA wird allerdings sehr stark mit Webservices in Verbindung gebracht, tatsächlich sind die Webservice-Technologien aber nur eine konkrete Implementierungsbasis für serviceorientierte Architekturen. Eine Umsetzung von SOA im Unternehmen erfordert zunächst eine Definition von Services, die völlig unabhängig von Webservices sein sollte. Dies ist ein rein konzeptueller Aspekt.

Technologisch wurde das Thema von allen namhaften Herstellern aufgegriffen. Im Java-Umfeld scheint sich die JAX-WS-API als Standard durchzusetzen. Im Zuge der Weiterentwicklung des .NET-Frameworks hat Microsoft die *Windows Communication Foundation (WCF)* eingeführt. WCF orientiert sich stark an SOA und wird als zentrale Implementierungsbasis für serviceorientierte Architekturen in Microsoft-Umgebungen dargestellt. Diese Technologie ist beispielsweise in (Schwichtenberg 2007) ausführlich erläutert und kann dort nachgelesen werden.

In Kapitel 2 wurden einige konzeptionelle Fragestellungen für Client/Server-Systeme diskutiert. Wendet man diese auf Webservices in der heutigen Form an, so muss festgestellt werden, dass einige Aspekte noch rudimentär berücksichtigt werden. Wir gehen kurz auf die in Kapitel 2 eingeführten Konzeptfragen ein:

- *Architekturkonzepte*: Die Architekturkonzepte heutiger Basissysteme für Webservice nutzen vorhandene Application-Server und damit auch deren Dienste. In diesem Bereich wurde also für Webservices nichts Neues erfunden.
- *Zustandsverwaltung*: Webservices sind vom Grundkonzept her zustandslose Dienste.
- *Diensteschnittstellen*: Die Diensteschnittstellen werden über WSDL spezifiziert, wobei eine Generierung aus einer Programmiersprache meist unterstützt wird, so dass man den komplizierten Code nicht selbst entwickeln muss.
- *Parameterübergabe*: Parameter und Returnwerte werden über Call-by-value übergeben.
- *Marshalling und Unmarshalling*: Das Marshalling erfolgt je nach Kodierungsstil, wobei die Nachrichten üblicherweise im Textformat übertragen werden. Die Marshalling-Routinen können automatisch erzeugt werden (Beispiel: JAXB-Standard). Die Übertragung von serialisierten Objekten (wie etwa Java-Objekte) als Parameter ist grundsätzlich möglich, fördert aber nicht die Sprachunabhängigkeit.
- *Adressierung und Kommunikation*: Die Kommunikation zwischen den beteiligten Partnern kann synchron oder aysnchron erfolgen, eine spezielle Unterstützung einer Fehlersemantik ist nicht festgelegt. Als Schnittstelle zu einem

Naming-Service wird UDDI empfohlen, es kann aber auch jeder andere Naming-Service verwendet werden.

- *Garbage-Collection, Nebenläufigkeit und Lebenszyklus:* Ein spezieller Garbage-Collection-Mechanismus ist in der Webservices-Spezifikation nicht vorgesehen. Ebenso wird keine Festlegung für die Nebenläufigkeitsunterstützung und für den Lebenszyklus eines Softwarebausteins im Server vorgenommen. Dies wird ebenso wie eine Unterstützung der Skalierbarkeit und Verfügbarkeit der herstellerspezifischen Plattform überlassen.
- *Basisdienste*: Spezifikationen von Basisdiensten für die Sicherheit (WS Security) und für Transaktionen (WS-Coordination) wurden durch OASIS vorgelegt, sind aber noch stark in Diskussion.

Eine Weiterentwicklung von Methoden zur Entwicklung verteilter Dienste, insbesondere von Webservice-Technologien ist aus heutiger Sicht vorhersehbar. Derzeit beschäftigen sich viele Softwareleute mit diesem Thema.

3.6 Übungsaufgaben

1. Was ist der große Vorteil von Webservices im Vergleich zu RMI oder CORBA?
2. Diskutieren Sie den Einfluss von SOAP auf die Leistungsfähigkeit einer Client-Server-Anwendung!
3. Welche Aufgabe hat UDDI bei Webservices?
4. Mit welchen Transportprotokollen arbeitet SOAP?
5. Was ist ein SOAP-Prozessor?
6. Erläutern Sie die Einbettung von SOAP-PDUs in HTTP-PDUs! Warum ist HTTP als „Transportprotokoll“ für SOAP so gut geeignet?
7. Wie wird einem Webservice ein Port zugeordnet?
8. Erläutern Sie die Mechanismen der Serialisierung bei Webservices?
9. Was ist der Unterschied zwischen JAX-RPC und JAX-WS?
10. Was versteht man unter einer serviceorientierten Architektur? Diskutieren Sie den Ansatz anhand von Webservices! Kann man eine serviceorientierte Architektur auch auf Basis von JEE/EJB realisieren?
11. Was bedeutet bei SOAP Document/Literal im Unterschied zu RPC/literal?

4 Konzepte, Modelle und Standards verteilter Transaktionsverarbeitung

Nachdem in Kapitel 2 die grundlegenden Konzepte und Implementierungsfragen für die verschiedenen Varianten von Client-Server-Systemen eingeführt wurden, soll nun ein bisher nur am Rande erwähntes und für betriebliche Anwendungen enorm wichtiges Konzept besprochen werden. Es handelt sich um das Transaktionskonzept für verteilte Umgebungen.

In diesem Kapitel werden wir, aufbauend auf den Grundlagen der Transaktionsverarbeitung im lokalen Umfeld, den Begriff der verteilten Transaktion einführen. Im Mittelpunkt der Transaktionsverarbeitung stehen die geforderten Transaktionseigenschaften, die mit dem Akronym ACID umschrieben werden. Ausgehend von den Anforderungen an lokale Transaktionen werden die Anforderungen an globale (verteilte) Transaktionen diskutiert. Es wird aufgezeigt, welche Funktionalitäten für die Implementierung eines Transaktionssystems erforderlich sind. Hierzu gehören vor allem Transaktions-Logging, Recovery, Concurrency-Control und verteilte Koordination der beteiligten Instanzen. Verschiedene Varianten der Implementierung werden gezeigt und es werden neben den klassischen flachen Transaktionen (Flat Transactions) auch andere Transaktionsmodelle skizziert.

An konkreten Standards wie dem DTP-Modell sowie CORBA OTS, die heute eine tragende Rolle bei der Transaktionsverarbeitung spielen, werden die Probleme der praktischen Umsetzung von Transaktionskonzepten in verteilten Umgebungen besprochen. Die Darstellung von Möglichkeiten zur Transaktionsunterstützung in heutigen Middleware-Plattformen und auch in Web-basierten Umgebungen rundet das Thema ab. Zusammenfassend werden schließlich einige Hinweise zur Nutzung von Transaktionen in verteilten Anwendungen gegeben.

Zielsetzung des Kapitels

Der Studierende soll verstehen, wie das Transaktionskonzept in verteilten Systemen grundsätzlich funktioniert und sinnvoll angewendet werden kann. Er soll ein Gefühl dafür entwickeln, wann dieses Konzept für verteilte Anwendungen sinnvoll bzw. notwendig ist. Weiterhin soll ein Verständnis der wichtigsten Transaktionsstandards, vor allem des DTP-Modells vermittelt werden.

Wichtige Begriffe

ACID-Transaktionseigenschaften, lokale und globale Transaktionen, flache und geschachtelte Transaktionen, Transaktionskontext, Serialisierbarkeit, Two-Phase-Locking (2PL), Two-Phase-Commit (2PC), Undo/Redo-Logging und Write-Ahead-

Logging (WAL), DTP-Modell und XA-Schnittstelle, EJB Transactions, CORBA OTS, WS, WS-AT und WS-BA, WS-C, JTA, JTS.

4.1 Transaktionen: Idee und Herausforderungen

Die Idee der Transaktion ist schon weit über dreißig Jahre alt und dient dazu, eine endliche Folge von Operationen zusammenhängend in einem Arbeitsschritt auszuführen. Eine Transaktion beginnt mit einer Operation *begin* und endet mit *commit* bzw. *commit work* oder wird mit einer Operation *rollback* abgebrochen. Der Beginn und das Ende einer Transaktion werden häufig auch als *Transaktionsgrenzen* bezeichnet.

Eine Transaktion ist eine Folge von Aktivitäten oder Operationen auf Daten bzw. Objekte. Ein konsistenter Zustand eines Systems wird über diese Folge von Operationen in einen neuen konsistenten Zustand überführt, wie dies in Abbildung 4-1 verdeutlicht wird.

Der Zustand zwischen dem Beginn und dem Ende einer Transaktion ist nicht konsistent, weshalb einige Probleme gelöst werden müssen. Beispielsweise darf kein anderer Benutzer inkonsistente Objekte sehen bzw. bearbeiten. Entweder es werden alle Operationen der Transaktion ausgeführt oder gar keine. Um dies sicherzustellen, sind entsprechende Mechanismen erforderlich.

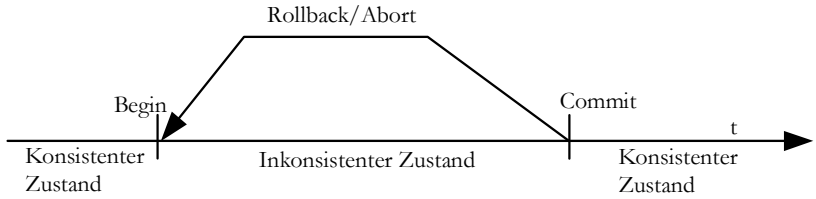


Abbildung 4-1: Idee einer Transaktion

Transaktionen bilden eine wichtige Grundlage für die Entwicklung fehlertoleranter Softwaresysteme. Das Konzept wurde ursprünglich für klassische OLTP-Anwendungen[1] auf Hostsystemen entwickelt.

Wie bereits angedeutet, sind bei der Implementierung des Transaktionskonzepts einige Probleme zu lösen. Zu den wesentlichen Teilproblemen, die bei verteilten Transaktionen behandelt werden müssen, gehören:

- Nebenläufigkeitskontrolle (Concurrency Control)
- Wiederanlaufstrategien (Recovery) bei Fehlersituationen sowie hierfür erforderliche Logging-Mechanismen

[1] OLTP steht für Online Transaction Processing.

– Koordination (Commit-Protokolle) für Transaktionen, in denen mehrere Instanzen beteiligt sind

Diese sollen im Weiteren näher betrachtet werden. Weiterhin sollen verschiedene Transaktionsmodelle erörtert werden. Zunächst sollen die besagten Teilprobleme einführend vorgestellt werden. Vorab sollen aber noch einige Begriffe definiert werden:

Lokale Transaktion: Als *lokale Transaktion* bezeichnet man eine Transaktion, die zu ihrer Ausführung nur auf lokale, also auf demselben Rechnersystem liegende, Ressourcen zugreifen muss und keine Abstimmung zwischen mehreren Rechnersystemen erfordert.

Globale oder verteilte Transaktion: Dagegen wird eine Rechner-übergreifende Transaktion als *globale* oder als *verteilte Transaktion* bezeichnet. Eine globale Transaktion wird auf einem Knoten gestartet und breitet sich dann auf andere Knoten aus.

Transaktionskontext: Eine lokale oder verteilte Transaktion wird im Transaktionssystem durch bestimmte Informationen eindeutig identifiziert. Diese Informationen werden als *Transaktionskontext* bezeichnet. Der Transaktionskontext muss für die gesamte Dauer der Transaktion verwaltet und allen Beteiligten kommuniziert werden. Im verteilten Umfeld bedeutet dies, dass entsprechende Protokolle zur Übermittlung des Kontextes (*Kontextpropagierung)* erforderlich sind. Im Prinzip ist die Verwaltung des Transaktionskontextes der Verwaltung eines TCP-Verbindungskontextes ähnlich, mit dem Unterschied, dass im Recovery-Fall abhängig vom Zustand in der Fehlersituation der Transaktionskontext wiederherstellbar sein muss. Der TCP-Verbindungskontext wird dagegen völlig neu erzeugt. Im Transaktionskontext ist eine für das Transaktionssystem global eindeutige Identifikation für die Transaktion enthalten. Ein Transaktionssystem ist für die eindeutige Vergabe zuständig.

Die verschiedenen Teilprobleme, die bei der Realisierung des Transaktionskonzeptes auftreten, sollen nun kurz skizziert werden.

Nebenläufigkeitskontrolle: In betrieblichen Informationssystemen und auch in anderen Systemen erfolgt meist ein nebenläufiger Zugriff auf verteilte Objekte durch mehrere oder sogar viele Anwender. Das System soll so arbeiten, dass es für den Benutzer so aussieht, als ob er es für sich alleine zur Verfügung hat. Wie beim Zugriff nebenläufiger Threads oder Prozesse auf gemeinsam genutzte Daten sind auch in verteilten Anwendungen die Zugriffe auf die gemeinsamen Ressourcen (meist Objekte in Datenbanken) zu synchronisieren.

Die Synchronisation wird bei der Programmierung nebenläufiger Threads, die auf einen gemeinsamen Adressraum zugreifen, meistens über Mechanismen wie Locks, Monitore oder Semaphore gelöst. Kritische Abschnitte sind zu definieren und isoliert zu durchlaufen. In verteilten Systemen sind hier ähnliche Probleme zu lösen. Sie sind nur noch komplexer, da ein Netzwerk zwischen den verteilten

Komponenten liegt und man daher aufwändige Protokolle benötigt. Zudem handelt es sich bei den bearbeiteten Daten meistens um sog. *persistente* Objekte, die auch sicher auf eine Datenbank oder einen sonstigen persistenten Speicher geschrieben werden müssen.

Erschwerend kommt hinzu, dass mehrere Operationen an Daten bzw. Objekten gemeinsam, am besten nach dem Prinzip „ganz oder gar nicht" ausgeführt werden müssen. Man denke hier nur an das Beispiel einer einfachen Buchung in einem Kontoführungssystem, bei der von einem Konto etwas abgebucht wird, das einem anderen Konto zugebucht werden soll.

Das Konzept der Transaktionen soll das Problem lösen. Transaktionen sind ein hochwertiges Programmierparadigma zur fehlertoleranten Programmierung und in klassischen Datenbankanwendungen wohlbekannt. Datenbankmanagementsysteme (DBMS) unterstützen dieses Konzept schon lange. In SQL-ähnlichem Pseudocode könnte eine Kontoumbuchungs-Transaktion etwa wie folgt notiert werden:

```
Begin Transaction
SELECT * FROM Konto WHERE Kontonummer = 100
SELECT * FROM Konto WHERE Kontonummer = 200
Betrag von Konto 1 abziehen
Betrag zu Konto 2 zubuchen
UPDATE Konto WHERE Kontonummer = 100 SET ...
UPDATE Konto WHERE Kontonummer = 200 SET ...
if (alles ausgeführt)
 Commit Work
else
 Rollback Work
End Transaction
```

Die Operation *Commit Work* ist für den Programmierer der Datenbankanwendung sehr praktisch. Er braucht sich nämlich um nichts mehr zu kümmern. Das Datenbankmanagementsystem übernimmt die Aufgaben der Nebenläufigkeitskontrolle und kann auch im Falle einer Fehlersituation entweder alle Zwischenergebnisse zurücksetzen oder aber alle Operationen zu Ende führen. Alle Operationen zwischen zwei *Commit*-Aufrufen werden als eine Transaktion behandelt[2]. Wenn der Programmierer eine Transaktion abbrechen muss, so ruft er *Rollback Work* auf und keine der Operationen wird ausgeführt.

Wiederanlaufstrategien und Logging-Mechanismen: Ein System, das auch bei Fehlersituationen wie dem Ausfall eines Rechners während der Bearbeitung zu-

[2] Es gibt DBMS, bei denen eine explizite *beginTransaction*-Operation unterstützt wird. Der SQL-Standard sieht dies nicht vor. Die Transaktion liegt hier zwischen zwei *Commit*-Aufrufen.

rechtkommen muss, benötigt Wiederanlaufstrategien (Recovery). Für einen Wiederanlauf sind Informationen erforderlich, die Auskunft über den Stand der vor dem Ausfall ausgeführten Operationen geben. Nur wenn Informationen während der Bearbeitung protokolliert werden und zwar so, dass sie einen Ausfall überleben, kann eine Wiederanlaufstrategie die nötigen Vorkehrungen treffen. Die Informationen müssen also zeitnah und sicher in ein sog. *Logfile* protokolliert werden. Diesen Vorgang nennt man *Transaktionslogging* oder kurz *Logging*. Jede Zustandsänderung, die nach einem Ausfall nachvollziehbar sein muss, wird protokolliert.

Koordination: Die Komplexität der Transaktionsverarbeitung steigt mit der Verteilung der beteiligten Systeme enorm an. Die Komponenten können innerhalb eines Rechnersystems oder aber auch auf mehrere Rechnersysteme und sogar über WAN oder MAN verteilt sein. Zur Abstimmung der beteiligten Komponenten benötigt man Kommunikationsprotokolle, die in der Anwendungsschicht angesiedelt sind und als Koordinationsprotokolle bezeichnet werden. Das klassische Beispiel für ein Koordinationsprotokoll ist das Two-Phase-Commit- oder 2PC-Protokoll, das in verschiedenen Ausprägungen existiert und in diesem Kapitel ausführlich besprochen wird. Eine grundlegende Voraussetzung für ein funktionierendes 2PC-Protokoll ist - wie wir noch sehen werden - das Logging.

Transaktionsmodelle und Fehlerbehandlung: Idealtypische Transaktionen müssen in klassischen OLTP-Anwendungen schnell ausgeführt werden. Sie sind kurz und blockieren daher die Ressourcen nur für sehr kurze Zeit.[3] Das grundlegende und älteste Transaktionsmodell wird als *flaches Transaktionsmodell* bezeichnet. Die weiter unten beschriebenen *ACID-Transaktionen* sind hier das Mittel zur Erzielung von korrekten Operationen, wobei das Akronym ACID für vier wichtige Eigenschaften einer flachen Transaktion steht.

Im Laufe der Jahre wurden weitere Transaktionsmodelle entwickelt, die auch andere Anwendungsbereiche abdecken sollen. Weitere Konzepte sind *verschachtelte (nested)*, *verkettete (chained)* und langlebige *(long)* Transaktionen und ihre verschiedenen Sonderformen.

Während einer verteilten Transaktion können verschiedene Fehlersituationen auftreten, die von einem *transaktionsverarbeitenden System* (oder kurz: *Transaktionssystem*), also einem System, das Transaktionen unterstützt, erkannt und behoben werden müssen. Beispiele hierfür sind:

- Ausfall des Netzwerks oder eines Teils davon
- Ausfall eines an der Transaktion beteiligten Rechnerknotens
- Ausfall einer Anwendung, die an der Transaktion beteiligt ist

[3] Komplizierter wird es schon bei sog. Batchanwendungen, die viele Objekte bearbeiten und daher lange belegen müssen. Hier greift das klassische Transaktionskonzept nicht so optimal.

Von einem Transaktionssystem wird erwartet, dass derartige Fehlersituationen auf keinen Fall zu Inkonsistenzen führen. Noch nicht abgeschlossene Transaktionen müssen entweder zurückgesetzt oder erfolgreich zu Ende gebracht werden. Fehlersituationen, die durch verfälschte oder verlorengegangene Nachrichten verursacht werden könnten, müssen durch Nutzung geeigneter Kommunikationsprotokolle vermieden werden. Nach Ausfall eines Rechnerknotens während einer Transaktion kann es sein, dass er nach einer endlichen Zeit wieder aktiv wird. Es kann aber auch vorkommen, dass der Rechner nie mehr startet. Im ersten Fall muss ein Transaktionssystem nach dem erneuten Start des Knotens die Information über den Status der abgebrochenen Transaktion(en) ermitteln. Dies kann durch das Lesen des erwähnten permanent und dauerhaft geführten (persistenten) Logfiles oder durch Nachfragen bei anderen Knoten erfolgen. Wird ein Knoten nicht mehr oder erst nach längerer Zeit wieder aktiv, kann dies zu Blockierungen von Anwendungen und Ressourcen führen.

Ein weiteres Problem stellen *byzantinische Fehler* dar. Hier handelt es sich um Fehlersituationen, die aufgrund inkorrekt arbeitender Hard- oder Softwarekomponenten entstehen können. Beispielsweise kann ein Teilnehmer einer Transaktion während einer 2PC-Protokollabwicklung mit "ok" antworten, obwohl er die Transaktion nicht erfolgreich zu Ende führen kann[4]. Solche Fehlersituationen können zum Beispiel durch fehlerhafte Protokollimplementierungen oder Hardwarefehler entstehen. Um derartige Fehler zu erkennen, müssen sehr aufwändige Algorithmen implementiert werden. Der Aufwand ist in der Praxis bei sehr kritischen Anwendungen gerechtfertigt, die jederzeit korrekt arbeiten müssen (zum Beispiel weil sonst Menschenleben bedroht wären). Bei den meisten Transaktionsanwendungen im kommerziellen Bereich, also z.B. bei betrieblichen Informationssystemen geht man von funktionierender Hardware und Software aus und verzichtet auf die Implementierung komplizierter Algorithmen zur Erkennung solcher Fehlersituationen. Ein Ausfall einer Komponente bedeutet dann die sofortige Einstellung aller ihrer Aktivitäten, was wiederum mit angemessenem Aufwand durch andere Komponenten erkannt werden kann. Dieses Fehlermodell wird auch als *Fail-Stop-Modell* bezeichnet und ist die Basis für heutige Transaktionssysteme.

Wie schon erörtert, bieten Transaktionsmonitore und auch Datenbankmanagementsysteme eine Transaktionsunterstützung an. Es soll vorab erwähnt werden, dass auch heutige Objekt- und Komponentensysteme bzw. deren Implementierungen, die auch als Objektserver oder Application-Server bezeichnet werden, Transaktionen[5] unterstützen. Kommerziell verfügbare Lösungen basieren z.B. auf *JEE/EJB* und auf Microsofts *.NET Enterprise Services*.

---

[4] Wir werden den Ablauf des 2PC-Protokolls noch ausführlich erläutern.

[5] Meist allerdings nur flache Transaktionen.

Die angerissenen Aspekte sollen im Weiteren ausführlicher betrachtet werden. Zunächst wird der grundlegende Begriff der ACID-Transaktion näher erläutert.

4.2 Grundlagen

4.2.1 ACID-Eigenschaften

Wie bereits einführend erwähnt, sind Transaktionskonzepte vor allem bei Datenbanken (Bernstein 1987), bei Transaktionsmonitoren (Meyer-Wegener 1988)und bei Betriebssystemen (Tanenbaum 2007) seit längerem von hoher praktischer Bedeutung. Insbesondere im Datenbankbereich gibt es eine Fülle von Implementierungen. Nahezu jedes größere Anwendungssystem nutzt Transaktionen.

Der Begriff der klassischen Transaktion wird in der Literatur häufig über das Akronym "ACID" erläutert, das für vier Eigenschaften bzw. *Korrektheitskriterien* steht, die eine Transaktion erfüllen muss. ACID steht als Abkürzung für *atomicity, consistency, isolation* und *durability*. Eine Transaktion wird als *ACID-Transaktion* bezeichnet, wenn sie alle vier ACID-Eigenschaften erfüllt. Die vier Eigenschaften sollen im Folgenden erläutert werden.

Atomarität (Atomicity): Eine Transaktion ist entweder ganz oder gar nicht (atomar) auszuführen. Passiert während der Transaktion ein Fehler (zum Beispiel ein Systemausfall, ein Netzwerkzusammenbruch oder ein Programmabsturz), so müssen alle bis dahin ausgeführten Operationen wieder rückgängig gemacht werden.

Parallel ablaufende Transaktionen dürfen nichts davon merken. Bei erfolgreichem Transaktionsende (Commit) müssen alle Operationen vollständig ausgeführt werden. Für ein Zurücksetzen der veränderten Objekte in den Zustand vor Beginn einer Transaktion müssen während der Ausführung der Transaktion redundante Informationen über Änderungsoperationen auf einem dauerhaften Speicher mitprotokolliert werden. Der Protokolliervorgang wird auch als *Logging,* der Wiederanlauf nach einem Fehler als *Recovery* bezeichnet.

Konsistenz (Consistency): Ein Transaktionssystem muss sicherstellen, dass die Korrektheit der Datenobjekte jederzeit gewährleistet ist. Ein Datenobjekt, das in einem korrekten Zustand ist, wird als konsistentes Objekt bezeichnet. Eine Transaktion führt einen konsistenten Zustand der Daten in einen neuen konsistenten Zustand über. Eine umfangreiche Erläuterung des Konsistenzbegriffs findet sich in (Tanenbaum 2007).

Isolation: Parallel ausgeführte Transaktionen dürfen sich gegenseitig nicht beeinflussen, müssen also voneinander isoliert ablaufen (isolation, auch independence genannt). Teilresultate einer Transaktion dürfen für andere Transaktionen nicht sichtbar werden, weshalb die Bearbeitungsreihenfolge eine wesentliche Rolle spielt. Wenn alle Transaktionen seriell ausgeführt werden, können sie sich nicht beeinflussen. Deshalb wird auch das Prinzip der Serialisierbarkeit herangezogen, um

das Isolationskriterium zu erfüllen. Transaktionen müssen also serialisierbar sein. Man spricht hier auch von seriellen Schedules, wie weiter unten noch erläutert wird.

Dauerhaftigkeit (Durability): Werden innerhalb einer Transaktion wie bei einem Datenbanksystem persistente Daten bearbeitet, so müssen diese bei Transaktionsende dauerhaft (durable) auf einem nicht-flüchtigen Speicher abgelegt werden. Bei Auftreten eines Fehlers muss ein Wiederanlaufverfahren dafür sorgen, dass die Objekte wiederhergestellt werden. Für die Wiederherstellung oder Vollendung eines konsistenten Zustands dient auch das Logfile, in dem die aktuellen Zustände aller Transaktionen abgelegt werden.

4.2.2 Wichtige Transaktionsmodelle

In den letzten Jahren wurden unterschiedliche Transaktionsmodelle entwickelt. Neben dem klassischen, flachen Transaktionsmodell sind vor allem geschachtelte Transaktionen und deren Ableitungen sowie verkettete Transaktionen zu nennen. Eine ausführliche Einführung in weitere, zum Teil auch sehr theoretische Ansätze sowie ihre potenziellen Einsatzgebiete sind in (Elmagarmid 1992) nachzulesen.

Nachfolgend wird ein kurzer Abriss über die bekanntesten und für betriebliche Informationssysteme praxisrelevanteren Modelle gegeben.

Flache Transaktionen: Die klassischen, *flachen Transaktionen* (flat transactions) fassen eine Menge von Operationen zu einer unteilbaren Aktion zusammen und werden nicht mehr weiter unterteilt. Ein Fehler während der Ausführung bewirkt ein Zurücksetzen bis zum Transaktionsbeginn, was bei umfangreichen Aktionen sehr kostenintensiv ist. Dieses Transaktionsmodell ist die Basis für die meisten DBMS-Implementierungen. Die Korrektheit der Transaktionen ist durch die Erfüllung der ACID-Eigenschaften gegeben.

Beispiel: Typische Beispiele für flache Transaktionen sind in vielen Anwendungen zu finden. Ein Beispiel ist die Umbuchung eines Lagerbestandes von einem Lagerplatz auf einen anderen in einem klassischen Lagerverwaltungssystem. Ein Codeausschnitt zu diesem Beispiel ist nachfolgend in einer Java-Methode dargestellt:

```
void umbuchen(int artikelnummer, int anzahl,
            Lagerplatz quelle, Lagerplatz ziel) {
   …
   try (…) {
      …
      beginTransaction();
      quelle.abbuchen(artikelnummer, anzahl);
      ziel.zubuchen(artikelnummer, anzahl);
      commitTransaction();
   }
   catch() {
```

```
      rollbackTransaction();
      ….
   }
   finally() {
      ….
   }
   ….
}
```

Im Fehlerfall, also bei Auftreten einer Ausnahme, wird ein Rollback ausgeführt, während im Erfolgsfall die Transaktion zu Ende geführt wird. Die Dienstaufrufe *begin-*, *rollback-* und *commitTransaction* werden in betrieblichen Informationssystemen meist gegen ein DBMS ausgeführt.

Geschachtelte Transaktionen: *Geschachtelte Transaktionen* (nested transactions) sind Transaktionen, die aus Subtransaktionen bestehen, die wiederum aufgeteilt werden können (Weikum 1989, Vossen 1993, Elmagarmid 1992 und Ramamritham 1997). Hierdurch entsteht ein „Transaktionsbaum", dessen Blätter die eigentlichen Operationen entsprechend einer flachen Transaktion repräsentieren. Die oberste Transaktion wird als Top-Level-Transaktion bezeichnet, die darunterliegenden als Subtransaktionen (Subtransactions). Subtransaktionen können wiederum weitere Subtransaktionen aufrufen.

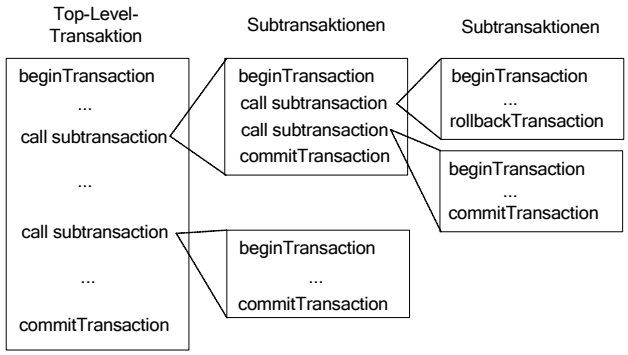


Abbildung 4-2: Transaktionsbaum einer geschachtelten Transaktion

Die Subtransaktionen können auch parallel ausgeführt werden. Jede Subtransaktion wird für sich atomar abgewickelt, Persistenz gilt jedoch nur für die gesamte Transaktion. Geschachtelte Transaktionen können insbesondere in verteilten Systemen zu einer Erhöhung der Parallelität (Intratransaktions-Parallelität) umfangreicher Transaktionen führen. Eine Transaktion, die in mehreren Knoten ausgeführt werden muss, kann mit geschachtelten Transaktionen nebenläufig abgewickelt werden. Zum anderen können geschachtelte Transaktionen robuster sein, da

ein Abbruch einer Subtransaktion nicht unbedingt zum Zurücksetzen der gesamten Transaktion führen muss, falls die übergeordnete Transaktion eine andere Aktion ausführen kann, die trotzdem zum Erfolg führt. In Abbildung 4-2 ist ein Beispiel für eine geschachtelte Transaktion aufgeführt.

Man unterscheidet *geschlossen geschachtelte Transaktionen* und *offen geschachtelte Transaktionen* sowie die Spezialfälle *Sagas* und *Contracts*[6]:

- Bei *geschlossen geschachtelten Transaktionen* werden die Zwischenergebnisse von Subtransaktionen erst für andere Transaktionen sichtbar, wenn die „Top-Level-Transaktion" erfolgreich beendet wurde.
- *Offen geschachtelte Transaktionen* erlauben die Freigabe von Zwischenergebnissen beendeter Subtransaktionen für nebenläufige Transaktionen und erhöhen damit den Grad der *Intertransaktions-Parallelität*. Die Folge davon können Verletzungen der Konsistenz sein.
- *Sagas* sind ein Spezialfall von *offen geschachtelten Transaktionen* mit einem maximal zweistufigen Transaktionsbaum. *Sagas* wurden für langlebige Transaktionen entwickelt wie sie z.B. für die Behandlung von Workflows diskutiert werden (Jablonski 1997). Sagas sind also eine Folge von einzelnen ACID-Transaktionen $T_1,...,T_n$, die gemeinsam eine Gesamttransaktion bilden. Bei Abbruch der Gesamttransaktion können daher die bis dahin abgeschlossenen Einzeltransaktionen T_i (mit $i \in 1,,..,n$) nicht zurückgesetzt werden, da sie bereits im Zustand „committed" sind. Um nun Inkonsistenzen durch die Liberalisierung der Isolationseigenschaft entgegenzuwirken, werden bei Sagas kompensierende Transaktionen eingeführt, die ein semantisches Zurücksetzen eines bereits freigegebenen Ergebnisses ausführen. Somit muss zu jeder Transaktion T_i ($i \in 1,\ldots,n$) eine geeignete Kompensationstransaktion C_i vorhanden sein. Für den Fall, dass die Gesamttransaktion abbricht, weil z.B. ihre Einzel-Transaktion T_j ($j \in 2,...,n$) nicht erfolgreich ist, kann T_j normal durch Abort bzw. Rollback zurückgesetzt werden. Die bereits abgeschlossenen Einzel-Transaktionen T_i ($i \in 1,\ldots,i\text{-}1$) müssen nun durch Ausführung ihrer Kompensationstransaktionen C_i in umgekehrter Reihenfolge zurückgesetzt werden (Niemann 2000). Die Aufweichung der Isolationseigenschaft kann natürlich wiederum zu kaskadierten Aborts führen.
- *Contracts (*Elmagarmid 1992) sind eine Erweiterung zu Sagas. Das Modell sieht ebenfalls einzelne ACID-Transaktionen vor, die in einem zusammengehörigen Kontrollfluss ausgeführt werden. Eine einzelne Transaktion hängt dabei immer von dem Ergebnis seiner Vorgängertransaktion ab. Auf Basis des Contracts-Modells lassen sich ganze Kontrollflüsse von aufeinanderfolgenden Transaktionen realisieren. Kompensationsstrategien sind wie

[6] Weitere Spezialfälle wie Mehrschichtentransaktionen sind u.a. in (Vossen 1993), (Elmagarmid 1992) und (ISO 1992b) nachzulesen.

bei Sagas notwendig, wenn eine Teiltransaktion nicht erfolgreich ist. Ein Kontrollfluss kann also z.B. vorsehen, bei einem Fehler innerhalb einer Einzeltransaktion zunächst eine Kompensationstransaktion auszuführen und dann im entsprechenden Zweig des Kontrollflusses weitermachen.

Geschachtelte Transaktionen sind sehr aufwändig zu realisieren. Dies gilt insbesondere dann, wenn alle ACID-Korrektheitskriterien erfüllt werden sollen. Welche Korrektheitskriterien sinnvoll sind, ist vom Anwendungsfall abhängig. In umfangreichen Workflows hat es z.B. keinen Sinn, bei Abbruch einer von mehreren hintereinander ausgeführten, aber logisch zusammengehörigen Transaktionen, alle vorher ausgeführten und erfolgreich beendeten Transaktionen zurückzusetzen. Können aber nicht alle vier ACID-Kriterien eingehalten werden, so spricht man nicht mehr von einer ACID-Transaktion.

Gerade in lang andauernden Transaktionen ist vor allem das Einhalten des Isolationskriteriums sehr hinderlich, da dadurch meist nebenläufige Transaktionen behindert werden. Weicht man aber ein Kriterium auf, sind auch die anderen Kriterien in Frage gestellt. Bei Verletzung des Isolationskriteriums kann also streng genommen auch die Konsistenz gefährdet sein. Hier kann man ebenfalls sog. Kompensationstransaktionen programmieren. Dies obliegt aber nach heutigem Kenntnisstand dem Anwendungsprogrammierer. Allerdings sind diesem Konzept Grenzen gesetzt, da je nach Transaktionsumfang Kompensationsstrategien schnell sehr komplex werden können.

Lange (langlebige) Transaktionen: Betrachten wir den stark vereinfachten Workflow einer Auftragsbearbeitung in einem Großhandelsunternehmen mit mehreren eigenen, abgeschlossenen Teiltransaktionen. In der Abbildung 4-3 sind fünf Arbeitsschritte skizziert, die in drei verschiedenen Systemen (ERP-, Versand- und Logistiksystem) insgesamt fünf Transaktionen ausführen. Es hätte in diesem Fall keinen Sinn, über den logisch zusammengehörigen Gesamtvorgang eine übergreifende Transaktion zu setzen, die alle ACID-Kriterien erfüllt. Stattdessen führt man einzelne Teiltransaktionen aus, um den Vorgang abzuwickeln.

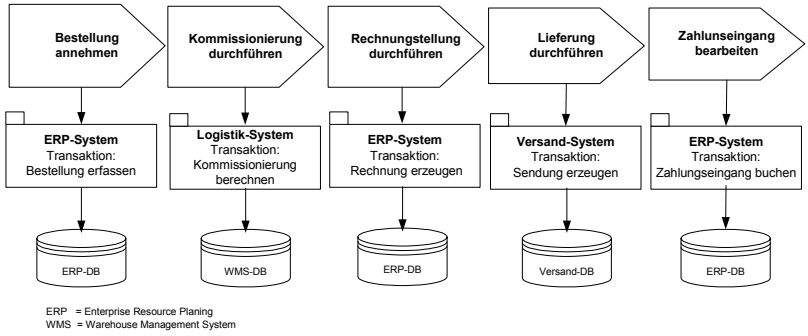


Abbildung 4-3: Beispiel einer langen Transaktion

In diesem Beispiel haben also die Teiltransaktionen nichts miteinander zu tun. Alle Änderungen in den beteiligten Datenbanksystemen werden völlig unabhängig voneinander durchgeführt. Müssen Änderungen in einem Folgeschritt zurückgesetzt werden, so wendet man dafür Kompensationstransaktionen an, die im entsprechenden Anwendungssystem unterstützt und damit in der Anwendung ausprogrammiert werden müssen. Lang andauernde (langlebige) Transaktionen für die Workflow-Bearbeitung sind demnach typische Anwendungsfälle für Sagas oder Contracts.

Verkettete Transaktionen und Savepoints: Klassische *Verkettete Transaktionen* (auch *chained Transactions* genannt) ermöglichen die sequentielle Ausführung mehrerer flacher Transaktionen, wobei beim Übergang der Transaktionskontext weitergereicht wird. Jede Transaktion innerhalb der Kette ist eine eigene ACID-Transaktion und wird beim Übergang zur nächsten vollständig beendet. Man befindet sich also immer in einer Transaktion. Durch die Verkettung kann ein Leistungsgewinn erzielt werden. Der Übergang von einer Transaktion zur nächsten, also die Operationen *commit* und *begin*, werden in einer Operation (*chainTransaction*) ausgeführt. Heutige Transaktionssysteme unterstützen meist verkettete Transaktionen, bieten aber auch die Möglichkeit, die automatische Verkettung abzuschalten. Auch in Zusammenhang mit Sagas spricht man von einer Verkettung von Transaktionen.

Schließlich unterstützen einige kommerzielle Datenbankmanagementsysteme als Ergänzung zu flachen Transaktionen sog. *Savepoints*. Dies sind Sicherungspunkte innerhalb einer Transaktion, auf die wieder aufgesetzt werden kann, ohne die gesamte Transaktion zurückzusetzen. Sie werden durch explizite Anweisungen markiert, sind aber nicht persistent. Ein Abbruch der Transaktion bedeutet, dass auch die mit Savepoints markierten Zwischenergebnisse wieder zurückgesetzt werden.

Queued Transactions: Das *Queued Transaction Processing Modell* teilt eine längere Transaktion in mehrere Schritte auf. Im Gegensatz zu einer typischen Client-Server-Verarbeitung wird eine Transaktion nicht direkt zwischen Client und Server ausgeführt. Der Client legt seinen Request vielmehr in einer ersten Teiltransaktion in eine Queue (Warteschlange). Der Request wird dann in einer zweiten Transaktion vom Server ausgelesen und bearbeitet. Der Server stellt schließlich das Ergebnis in eine weitere Queue, die dann wiederum in einer dritten Transaktion vom Client ausgelesen wird (Bernstein 1997).

Queued Transactions ermöglichen damit eine Entkopplung von Client und Server. Beide können ausfallen und nach einem Neustart die Bearbeitung fortsetzen. Die einzelnen Transaktionen unterliegen für sich den ACID-Eigenschaften.

Queued Transactions benötigen ein Message-Queueing-System, das die Queues persistent verwaltet und die Einzeltransaktionen ausführt. Dieses Message-Queueing-System stellt ein eigenes Transaktionssystem dar. Benötigt der Server

für die Bearbeitung des Requests neben der Queue noch weitere Ressourcen (wie z.B. Datenbanken) ist allerdings eine Transaktionskoordination notwendig. Heute ist dieses Konzept eines der Standardkonzepte der Transaktionsverarbeitung. Nahezu jedes Middlewareprodukt unterstützt auch Queued Transactions. Im Java-Umfeld gibt es auch einen offenen Standard für die Nutzung von Message Queues, der als JMS (Java Message Service) bezeichnet wird und den heute die meisten Hersteller von Application-Server- und Message-Queueing-Middleware unterstützen (siehe Kapitel 2).

4.2.3 Anatomie eines Transaktionssystems

Die erforderlichen Mechanismen zur Realisierung eines Transaktionssystems bzw. zur Unterstützung der ACID-Eigenschaften können den Anwendungen in Form von Diensten zur Verfügung gestellt werden, die wir als *transaktionsunterstützende Dienste* bezeichnen.

In einem Transaktionssystem, das ACID-Transaktionen unterstützt, findet man üblicherweise die folgenden transaktionsunterstützendem Dienste bzw. Services:

- Transaction Service (TS)
- Logging Service (LS)
- Concurrency Control Service (CS)
- Recovery Service (RS)

Der TS-Dienst verwaltet die Transaktionskontexte und wickelt ein Commit-Protokoll zwischen allen an einer Transaktion beteiligten Komponenten ab. Er stellt Dienstprimitive zum Starten (*begin*), Beenden (*commit*) und Zurücksetzen (*rollback*) einer Transaktion bereit. Der CS-Dienst dient der Synchronisation nebenläufiger Zugriffe auf Objekte und stellt entsprechende Dienstprimitive bereit. Der LS-Dienst stellt Dienstprimitive bereit, die es ermöglichen, Logginginformation persistent abzuspeichern und diese wieder aufzufinden. Der RS-Dienst ermöglicht das Zurücksetzen aktiver Transaktionen bei einer Rollback-Anforderung und den Restart nach einem Ausfall einer Komponente. Er verwendet den LS-Dienst, um die Zustände der Transaktionen vor einem Ausfall zu ermitteln. Neben diesen Diensten werden für die Zugriffe auf die Datenobjekte noch Datenzugriffsdienste (Data-Access-Service, DS) und bei Bedarf Dienste zur persistenten Speicherung von Datenobjekten (Persistency Service, PS) benötigt.

In Transaktionssystemen arbeiten üblicherweise Applikationen, Koordinatoren und Ressourcenmanager zusammen. Alle kommunizieren über einen einheitlichen Protokollstack, der neben den klassischen Transportprotokollen auch die darüberliegenden Transaktionsprotokolle enthält.

Die beteiligten Komponenten benötigen unterschiedliche Dienste oder stellen diese bereit. Wie in Abbildung 4-4 zu sehen ist, benötigt ein Koordinator einen RS- und einen LS-Dienst, um Logging- und Recovery durchzuführen. Die Diensteimplementierungen müssen nicht unbedingt, wie in der Abbildung dargestellt, im Ko-

ordinator enthalten sein. Sie können auch von außerhalb bezogen werden. Der Koordinator stellt selbst einen TS-Dienst bereit, der die Transaktionskontexte verwaltet und die klassischen Operationen (*begin*, *commit*,...) zur Verfügung stellt. Ein Ressourcenmanager benötigt dagegen noch einen CS-Dienst und evtl. Dienste zur Datenhaltung sowie zur persistenten Datenspeicherung (DS und PS).

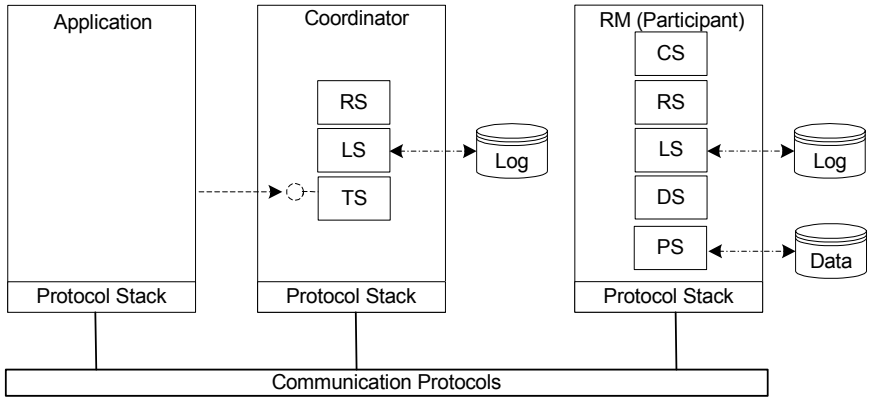


Abbildung 4-4: Typische Softwarekomponenten transaktionaler Systeme

Konkrete Implementierungen stellen die Dienste in Programmiersprachen über APIs, zur Verfügung. Welche Dienstaufrufe die Anwendungen tatsächlich nutzen und welche implizit vom Transaktionssystem verwendet werden, hängt von der jeweiligen Implementierung ab. Umso schmaler das von den Anwendungsentwicklern zu nutzende API ist, umso leichter sind Transaktionsanwendungen zu implementieren. In relationalen DBMS stellt zum Beispiel SQL ein API dar, das dem Anwendungsprogrammierer die Logging-, Recovery- und auch die Concurrency-Control-Services weitgehend versteckt, da diese implizit vom DBMS verwendet werden.

Die wesentlichen Dienste, die für eine Realisierung eines Transaktionssystems erforderlich sind, werden wir im Folgenden näher betrachten.

4.2.4 Concurrency Control

Durch nebenläufig ausgeführte Transaktionen kann es ohne Einsatz geeigneter Synchronisationstechniken zu Fehlersituationen kommen. Man unterscheidet als mögliche Fehlerfälle (Anomalien) u.a. *lost-update*, *dirty-read*, *unrepeatable-read* und *Phantoms*:

- Ein *lost-update*-Problem kann entstehen, wenn eine Transaktion T_1 eine Änderungsoperation (Write-Operation) auf ein Objekt ausführt, welches in einer parallelen Transaktion T_2 ebenfalls verändert wird. Dadurch kann das Ergebnis der Operation aus T_1 verlorengehen.

- Mit *dirty-read* wird eine Fehlersituation verstanden, die auftreten kann, wenn eine Transaktion T_1 eine Write-Operation auf ein Objekt ausführt und dann abbricht. Noch vor Abbruch von T_1 kann nämlich eine Transaktion T_2 das Objekt mit einer Leseoperation (read- oder get-Operation) anschauen und somit einen Wert lesen, der nicht korrekt ist, weil T_1 zurückgesetzt werden muss.
- Unter *unrepeatable-read* ist zu verstehen, dass innerhalb einer Transaktion die Wiederholung eines Lesevorgangs mit gleichem Ergebnis zu einem späteren Zeitpunkt nicht möglich ist. Dieser Fall tritt ein, wenn T_2 eine Read-Operation auf ein Objekt ausführt, dann T_1 eine Write-Operation auf das gleiche Objekt durchführt und schließlich T_2 wieder eine Read-Operation auf dieses Objekt. Die erste Read-Operation von T_2 ist somit nicht wiederholbar, weil das Objekt von einer parallel ablaufenden Transaktion T_1 zwischenzeitlich verändert wurde.
- Unter dem *Phantomproblem* versteht man eine spezielle Form für ein nicht wiederholbares Lesen. Wird z.B. von einem Prozess eine Anweisung *select * from <tabelle>* durchgeführt und ein nebenläufiger Prozess erzeugt anschließend ein Tupel in der Datenbanktabelle *<tabelle>*, das auf das Selektionskriterium gepasst hätte, kann die Selektionsanweisung nicht mit dem gleichen Ergebnis wiederholt werden. Das neue Tupel wird als Phantom bezeichnet.

Diese Fehlerfälle können nur vorkommen, wenn die Isolationseigenschaft einer Transaktion nicht erfüllt wird. Die Sequenz an Read- und Write-Operationen parallel ausgeführter Transaktionen wird auch *Schedule* genannt. Im ANSI-SQL-Standard wurden verschiedene Stufen der Isolierung definiert, die von einer Anwendung mit einer SQL-Anweisung (*set transaction isolation level*) einstellbar sind. Die Einstellung *Serializable* ist die strengste Stufe und fordert Serialisierbarkeit. Bei Anwendung von *Serializable* ist sogar das Phantom-Problem ausgeschlossen[7]. Die Nutzung von *Serializable* ist aber in vielen Anwendungsszenarien zu streng und muss deshalb oft abgemildert werden. Man wendet es daher eher selten an. Die weiteren definierten Isolationslevel aus dem SQL-Standard[8] sind *Read Uncommitted*, *Read Committed*, *Repeatable Read* und *Unrepeatable Read*. Bei rein sequentiell ablaufenden Transaktionen kommt es zu keinen Problemen, da es keine Nebenläufigkeit gibt. Anomalien können nur bei parallel oder quasi parallel ausgeführten Transaktionen auftreten.

[7] Die korrekte Implementierung vorausgesetzt.

[8] Isolationsstufen sind im SQL-Standard für relationale Datenbankmanagementsysteme beschrieben.

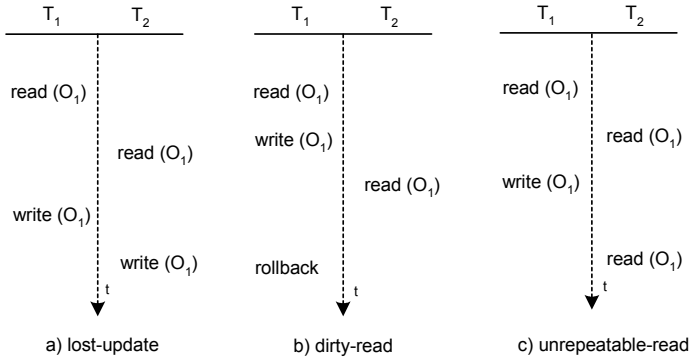


Abbildung 4-5: Mögliche Anomalien

Die grundlegenden Probleme der Nebenläufigkeitskontrolle wurden in den 80er Jahren insbesondere im Umfeld der Datenbankforschung eingehend untersucht. Die wesentlichsten Erkenntnisse wurden in dieser Zeit gewonnen und in der Serialisisierbarkeitstheorie festgehalten. Hierzu werden einige Begriffe definiert:

Serieller Schedule: Ein serieller Schedule ist ein Schedule, in dem die Transaktionen $T_1,...,T_n$ nacheinander ausgeführt werden, d.h. zunächst werden alle Aktionen von T_1 ausgeführt, dann alle Aktionen der Transaktionen T_2 usw.

Serialisierbarkeit eines Schedule: Ein Schedule S heißt serialisierbar genau dann, wenn es zu S einen äquivalenten seriellen Schedule S' gibt, der, angewandt auf denselben Ausgangszustand der Ressourcen, zu denselben Ausgaben und zu demselben Endzustand wie S führt.

Korrektheit paralleler Transaktionen: Eine nebenläufige und überlappte Ausführung mehrerer Transaktionen ist dann und nur dann korrekt, wenn es mindestens eine serielle Ausführungsreihenfolge für diese Transaktionen gibt, die auf den selben Ausgangszustand angewandt auch zum selben Endzustand der Ressourcen (z.B. Datenbanken) führt.

Als grundlegende Forderung gilt also, dass Operationen nebenläufiger Transaktionen serialisierbar sein müssen, das heißt die nebenläufigen Transaktionen müssen die Ergebnisse liefern, die auch ein serielles Ausführen der Transaktionen liefern würde. Ein sog. Transaktions-Scheduler übernimmt diese Aufgabe in Datenbankmanagementsystemen. Dies ist ein Programm oder eine Softwarekomponente, die die Ausführung nebenläufiger Transaktionen kontrolliert und steuert.

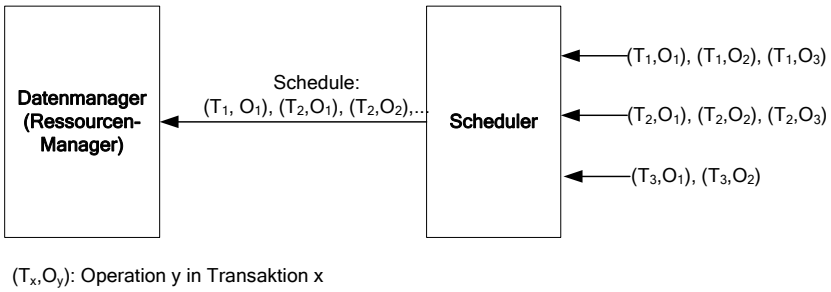


Abbildung 4-6: Verarbeitung von Transaktionen durch den Scheduler nach (Vossen 1993)

In Abbildung 4-6 sieht man beispielsweise, wie ein Scheduler die Operationen von drei nebenläufigen Transaktionen so serialisiert, dass sie bei der Ausführung konfliktfrei abgewickelt werden können. Die Operationen werden serialisiert an die in der Abbildung als Datenmanager bezeichnete Komponente zur Verarbeitung übergeben. Hierfür muss der Scheduler Regeln für die Reihenfolgebildung implementieren. Beispielsweise führen nebenläufige Leseoperationen in der Regel nicht zu Schwierigkeiten, sondern vor allem Änderungsoperationen. Die *Serialisierbarkeitstheorie* beschreibt unterschiedliche Arten von Serialisierbarkeit.

Zur Serialisierung der Zugriffe werden spezielle Verfahren wie zum Beispiel das Sperren *(Locking)* der Ressourcen, auf die parallel zugegriffen wird, eingesetzt. Im Datenbankbereich wurden einige Verfahren zur Synchronisation nebenläufiger Transaktionen entwickelt. Beispiele hierfür sind Sperr-, Zeitmarken- und optimistische Verfahren. Die ersten beiden Verfahren werden auch als pessimistisch bezeichnet, da von vornherein mit Konflikten gerechnet wird. Alle pessimistischen Verfahren definieren bestimmte Regeln zur Behandlung von Konfliktsituationen. Bei jeder Operation wird sofort entschieden, ob sie ausgeführt werden darf oder nicht. Dagegen gehen optimistische Verfahren davon aus, dass Konflikte selten auftreten und führen Operationen einer Transaktion sofort aus, ohne zunächst zu prüfen, ob Konflikte mit anderen Transaktionen vorliegen. Die wichtigsten Verfahren sind ausführlich in (Gray 1993) beschrieben und werden im Folgenden kurz erläutert.

Two-Phase-Locking

In praktischen Implementierungen hat sich überwiegend das *Two-Phase-Locking (2PL)* durchgesetzt. Beim 2PL werden nach Beginn einer Transaktion in einer ersten Phase alle benötigten Sperren besorgt und erst dann in einer zweiten Phase wieder freigegeben, wenn keine weiteren Sperren mehr benötigt werden. Die erste Phase wird auch als Zunahmephase bezeichnet, die zweite als Abnahmephase.

Konservatives 2PL: Man spricht von *konservativem 2PL (*Abbildung 4-7), wenn jede Transaktion zunächst alle benötigten Sperren in einer sog. Zunahmephase anfordert und danach in einer absteigenden Phase nach und nach wieder frei gibt.

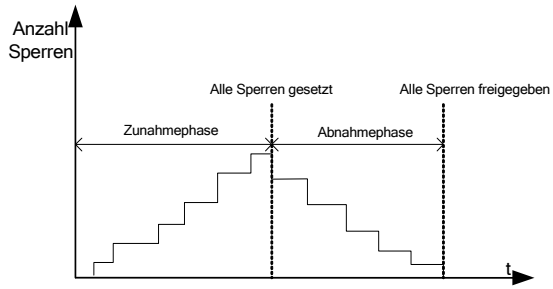


Abbildung 4-7: Phasen beim konservativem Two-Phase-Locking

Sobald der Scheduler mit der Freigabe der Sperren begonnen hat, darf keine weitere Sperre mehr für die Transaktion gesetzt werden. Hier vergeht eine bestimmte Zeit, bis alle Sperren freigegeben sind, in der bereits freigegebene Objekte schon wieder von anderen Transaktionen gesperrt werden können.

Strenges 2PL: Von *strengem* oder *striktem 2PL* wird dagegen gesprochen, wenn alle Sperren am Ende der Transaktion (nach der letzten Operation) auf einmal, logisch zusammengehörend, freigegeben werden. Die Freigabe dauert natürlich auch eine bestimmte Zeit, allerdings kann in dieser Zeit keine andere Transaktion die beteiligten Objekte sperren.

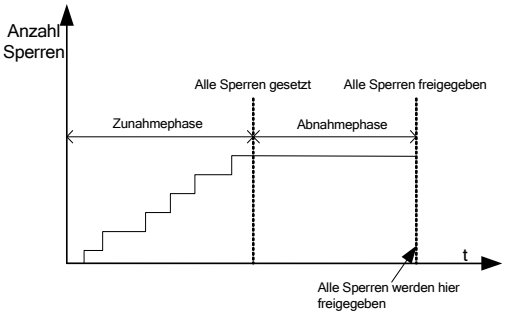


Abbildung 4-8: Phasen beim strengen Two-Phase-Locking

Das strenge 2PL-Verfahren hat den Vorteil, dass eine Transaktion nie ein Objekt erhält, das gerade durch eine andere Transaktion gesperrt ist und das sie damit eigentlich nicht sehen dürfte. Damit vermeidet man sog. *kaskadenförmige (kaskadier-*

te) Abbrüche, in denen bereits festgeschriebene Transaktionen noch einmal rückgängig gemacht werden müssten. Aus diesem Grund wird in Datenbanktransaktionen gängiger Datenbankmanagementsysteme das strenge 2PL verwendet.

Bei beiden Varianten des 2PL-Verfahrens sind Deadlocks möglich, wenn nämlich zwei Prozesse versuchen, dieselben Objekte zu sperren. Sperrt die erste Transaktion zunächst das erste Objekt und die zweite Transaktion zunächst das zweite Objekt und danach umgekehrt, ist ein Deadlock möglich.

Die Tabelle 6-1 zeigt die Lock-Kompatibilitätsmatrix für das Read-Write-Modell. Eine Transaktion möchte eine Read- oder Write-Operation ausführen. In den Spalten wird angezeigt, ob bereits ein Lock durch eine andere Transaktion gesetzt ist. „Ok" gibt nun an, dass die anfordernde Transaktion einen Lock setzen darf. „Wait" zeigt an, dass die Transaktion, die den Lock anfordert, solange warten muss, bis das Objekt nicht mehr von einer anderen Transaktion gesperrt ist.

Tabelle 6-1: Lock-Kompatibilitäsmatrix

	Durch beliebige andere Transaktion bereits gesetzter Lock		
Lock-Anforderung	**Read-Lock**	**Write-Lock**	**Kein Lock**
Read	Ok	Wait	Ok
Write	Wait	Wait	Ok

Optimierungen des Two-Phase-Locking

Nachteilig an Sperrverfahren sind der hohe Overhead zur Verwaltung der Sperren und die Möglichkeit von Deadlocks. Einige Optimierungen wie das Zwei-Versionen-Locking und hierarchische Locks sind in (Gray 1993) beschrieben.

Im *Zwei-Versionen-Locking* werden exklusive Locks bis zum Commit verzögert. Leseoperationen werden nur auf „committed" Objekte zugelassen. Neben den bereits bekannten Read/Write-Locks wird hier ein *Commit-Lock* benötigt, der für alle geänderten Objekte einer Transaktion gesetzt wird, wenn die Commit-Bearbeitung durchgeführt wird. Bei der Vergabe von Locks sind folgende Konfliktregeln zu beachten:

- *Read-Locks* werden immer gewährt, außer das zu sperrende Objekt wird gerade durch einen Commit-Lock gesperrt. In diesem Fall muss gewartet werden, bis der Commit-Lock freigegeben ist.
- *Write-Locks* werden gewährt, wenn das zu sperrende Objekt nicht bereits durch einen Write- oder Commit-Lock gesperrt ist.
- Beim Commit wird versucht, alle Write-Locks in Commit-Locks umzuwandeln. Ist ein Objekt durch einen Read-Lock gesperrt, muss die Commit-Bearbeitung verzögert werden, bis die Transaktion, die den Read-Lock hält, beendet ist.

Vorteil des Verfahrens ist, dass Transaktionen nur in der relativ kurzen Commit-Bearbeitung blockiert werden können. Andererseits können Read-Locks die Commit-Bearbeitung behindern.

Hierarchische Lockverfahren sind insbesondere bei Transaktionslasten mit einer Mischung aus vielen kurzen und wenig langen Transaktionen sinnvoll und zwar bei hierarchisch angeordneten Objekten wie zum Beispiel Datenbanken mit Tabellen und Sätzen. Lang andauernde Transaktionen können ihre Absicht, in Zukunft Objekte zu lesen oder zu verändern durch spezielle Locks bekanntgeben. Neben Read- und Write-Locks werden zusätzlich *Intention-Read-* und *Intention-Write-Locks* unterschieden.

Intention-Locks werden gesetzt, wenn eine Transaktion beabsichtigt, in Zukunft ein Objekt zu lesen oder zu ändern. Beabsichtigt zum Beispiel eine Transaktion eine ganze Tabelle oder viele Sätze aus einer Tabelle zu lesen, so setzt sie einen Intention-Read-Lock auf die Tabelle. Dadurch werden nebenläufige Transaktionen an der Ausführung von Write-Operationen auf die Tabelle gehindert und die Transaktion wird nicht blockiert.

Zeitmarkenverfahren

Beim Zeitmarkenverfahren wird jeder Transaktion vom Transaktionsmanager ein *netzweit eindeutiger Zeitstempel* zugeordnet, der beim Zugriff am Objekt hinterlegt wird. Nebenläufige Transaktionen werden durch die Zeitstempel zeitlich geordnet. Im einfachsten Zeitmarkenverfahren werden keine temporären Kopien von Objekten unterstützt. Greift eine Transaktion auf ein Objekt lesend oder schreibend zu, wird das Originalobjekt mit dem Zeitstempel der Transaktion markiert. Der Zugriff ist nur erlaubt, wenn folgende Bedingungen erfüllt werden:

- Eine Write-Operation auf ein Objekt ist nur zulässig, wenn die letzte Read- und die letzte Write-Operation auf das Objekt von einer älteren Transaktion ausgeführt wurden. Erfolgte der letzte Zugriff durch eine jüngere Transaktion, wird die Operation abgewiesen.
- Eine Read-Operation auf ein Objekt ist nur zulässig, wenn die letzte Write-Operation auf das Objekt durch eine ältere Transaktion ausgeführt wurde.

Ist die jeweilige Bedingung nicht erfüllt, steht eine Transaktion in Konflikt mit anderen Transaktionen und wird abgebrochen. Die Verwendung des Originalobjekts hat zur Folge, dass zu einer Zeit nur eine Transaktion Zugriff auf ein Objekt haben kann.

Erweiterungen dieses Verfahrens sehen *temporäre Kopien* der Objekte vor, die nebenläufige Zugriffe mehrerer Transaktionen nach bestimmten Regeln erlauben. Nebenläufige Zugriffe werden erlaubt, wenn sie nicht in Konflikt zueinander stehen, was ebenfalls über den Zeitstempel erkannt wird.

Optimistische Verfahren

In optimistischen Verfahren wird während der Ausführung von Operationen nicht auf Konfliktsituationen geachtet. Optimistische Verfahren prüfen erst am Ende einer Transaktion (bei der Commit-Behandlung), ob Konflikte mit nebenläufigen Transaktionen auftreten. Eine Transaktion wird in *drei Phasen* ausgeführt:

- In einer *Lesephase* werden alle Objekte, die verändert oder gelesen werden in einem temporären Speicher abgelegt. Jede Transaktion hat einen eigenen Speicherbereich. Da nur Objekte im Zustand *committed* verwendet werden, sind Anomalien wie *dirty read* nicht möglich.
- *In* einer *Validierungsphase* (bei Ausführung eines Commit-Aufrufes) wird über eine festgelegte Strategie geprüft, ob Konflikte mit anderen Transaktionen auftreten. Wenn ja, wird entschieden, dass die zu beendende oder eine andere Transaktion zurückgesetzt wird.
- Schließlich wird in einer *Schreibphase* ein dauerhaftes Speichern aller temporären Write-Operationen ausgeführt.

Die Validierung nutzt Read/Write-Konfliktregeln und setzt eine zeitlich aufsteigende, eindeutige Nummerierung der Transaktionen voraus. Eine Transaktion beendet ihre Lesephase erst, nachdem alle Transaktionen mit niedrigerer Nummer ihre Lesephase beendet haben. Validierungsverfahren wie Rückwärts- und Vorwärtslesen sind möglich.

Nachteil der optimistischen Verfahren ist, dass sie zum „Verhungern" von Transaktionen führen können, wenn diese immer wieder aufgrund von Konflikten zurückgesetzt werden. Erweiterte Verfahren zur Vermeidung dieser Problematik sind in der angegebenen Literatur ausgeführt.

Schlägt eine Transaktion einmal fehl, so kann beispielsweise in der Validierungsphase ein Sperren aller Objekte dieser Transaktion erfolgen, um diese in der Wiederholung in jedem Fall sicher durchführen zu können. Da der Aufwand, eine Transaktion vollständig neu auszuführen, beträchtlich ist, ist dieses Verfahren nur sinnvoll, wenn wenige Konflikte auftreten können. Die Validierung kann über einen zentralen Knoten oder verteilt vorgenommen werden. Deadlocksituationen können bei diesem Verfahren nicht auftreten.

Deadlocks

Ähnlich wie in lokaler Umgebung kann es durch das Verwenden von Sperren in verteilten Transaktionen zu Deadlocksituationen kommen, die entweder vermieden oder erkannt und aufgelöst werden müssen. Deadlocks können bekanntlich dadurch vermieden werden, dass eine der vier Bedingungen, die für eine Deadlock-Situation erfüllt sein müssen, durchbrochen wird. Diese vier Bedingungen sind:

- Mutual Exclusion (gegenseitiger Ausschluss) für die benötigten Betriebsmittel

- Verteilte Prozesse belegen Betriebsmittel und fordern weitere an.
- Kein Entzug eines Betriebsmittels ist möglich.
- Zwei oder mehrere verteilte Prozesse warten in einer verketteten Liste (in einer Schleife, circular waiting) auf weitere Betriebsmittel.

Eine Möglichkeit zur Deadlock-Vermeidung ist es, alle Sperren bei Transaktionsbeginn anzufordern und dann erst die Operationen auszuführen, wenn alle Sperren gesetzt sind. Können eine oder mehrere Sperren nicht gesetzt werden, werden alle anderen auch wieder freigegeben und die Transaktion wird nicht ausgeführt. Dieses Verfahren wird aber selten eingesetzt.

Zur Deadlock-Erkennung und deren Auflösung wird in der Praxis häufig ein einfaches aber robustes Verfahren, das auf Zeitüberwachung beruht, verwendet. Wenn eine Transaktion ein Objekt nicht bearbeiten kann, weil es von einer anderen Transaktion gerade gesperrt wird, wird maximal eine bestimmte Zeit (Timeout-Zeit) gewartet. Nach Ablauf der Timeout-Zeit wird die Transaktion zurückgesetzt. Problematisch an diesem Verfahren ist die richtige Wahl der Wartezeit, da sie unmittelbare Auswirkung auf den Durchsatz eines Systems hat. Bei diesem Verfahren kann es vorkommen, dass Transaktionen abgebrochen werden, die nicht zwangsläufig zu einem Deadlock führen. Hier handelt es sich um einen sog. *unpräzisen* Deadlock-Erkennungs-Algorithmus.

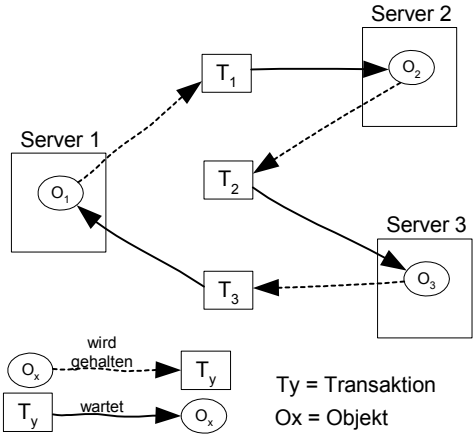


Abbildung 4-9: Deadlock in einem verteilten System

Präzise Deadlock-Erkennungs-Algorithmen erkennen dagegen echte Deadlocks, sind aber für verteilte Anwendungen in praktischen Implementierungen kaum eingesetzt. Eine präzise Erkennung von Deadlocks ist zum Beispiel über die Verwaltung von Wait-for-Graphen möglich. Über diese Graphen wird ermittelt, welche Objekte von welchen Transaktionen gesperrt werden und welche Transaktionen auf welche Objekte warten. Ist im Graphen ein Zyklus vorhanden, so besteht

ein Deadlock. Es gibt verschiedene Techniken, Wait-for-Graphen in verteilten Systemen zur Deadlock-Erkennung zu verwenden. Eine Möglichkeit ist, einen Knoten als *Deadlock-Detector* einzusetzen, der von den anderen Knoten die lokalen Wait-for-Graphen empfängt. Aus allen lokalen Wait-for-Graphen kann der Deadlock-Detector Zyklen ermitteln und bei Bedarf Maßnahmen zur Deadlock-Auflösung einleiten (zum Beispiel einen Abbruch einer Transaktion). Nachteile der Methode sind schlechte Verfügbarkeit und hohe Netzbelastung.

Betrachten wir den Wait-for-Graphen aus Abbildung 4-9 mit drei nebenläufigen Transaktionen und drei verschiedenen Knoten (Serversystemen), auf denen Objekte verwaltet werden. Hier ist im Graphen eine Schleife erkennbar, die einen Deadlock anzeigt. Die Transaktion T_1 wartet nämlich auf das Objekt O_2 während es O_1 sperrt (vereinfacht: ohne Rücksicht auf die Art der Sperre), die Transaktion T_2 wartet auf das Objekt O_3 während es O_2 hält und die Transaktion T_3 wartet auf das Objekt O_1 während es O_3 sperrt.

Besser, aber schwieriger zu implementieren sind Methoden für eine verteilte Deadlock-Erkennung, in der mehrere Knoten involviert sind. Ein Beispiel hierfür ist der sog. *Edge-Chasing- oder Path-Pushing-Algorithmus*. In diesem Verfahren wird kein globaler Wait-for-Graph aufgebaut, sondern die beteiligten Komponenten tauschen Informationen über die ihnen bekannten Kanten aus (sog. Probe-Nachrichten).

4.2.5 Logging und Recovery

In diesem Abschnitt werden die beiden eng zusammenhängenden Funktionen Logging und Recovery, die für ein Transaktionssystem essentiell sind, erläutert.

Grundlegendes

Mit Recovery ist der Wiederanlauf nach einem Fehler gemeint. Alle Transaktionen, die zum Zeitpunkt eines Fehlers schon abgeschlossen waren, müssen auch einen Fehlerfall „überleben“. Dies trifft in Abbildung 4-10 auf die Transaktionen T_1, T_2 und T_3 zu. Transaktionen, die noch aktiv waren, müssen vollständig zurückgesetzt werden (siehe die Transaktionen T_4 und T_5). Ein Rücksetzmechanismus muss also sicherstellen, dass die Statusveränderungen der noch nicht abgeschlossenen Transaktionen im Gesamtsystem nicht wirksam werden.

Als Logging wird das Verfahren bezeichnet, das in Transaktionssystemen zur Verwaltung redundanter Information über Änderungsoperationen verwendet wird. Diese Redundanz wird bei der Bearbeitung von Fehlerfällen zum Wiederherstellen eines konsistenten Zustands benötigt. Die Softwarekomponente, die diese Aufgabe übernimmt, wird als *Loggingmanager* bezeichnet und muss in jedem Knoten eines verteilten Systems vorhanden sein. Die redundante Information wird in einem Log auf einem nicht-flüchtigen Speicher mitprotokolliert, so dass sie nach einem Sys-

temausfall noch vorhanden ist. Zugriffe auf das Log sind nur über die Dienste des Loggingmanagers möglich.

Im DTP-Modell der X/Open (Distributed Transaction Model) muss jeder Ressourcenmanager sein privates Log verwalten (DTP 1991). In (Gray 1993) wird ein Vorschlag gemacht, der in jedem Knoten einen Loggingmanager, auf den der Transaktionsmanager und alle Ressourcenmanager gleichermaßen zugreifen, bevorzugt.

Das Log wird sequentiell mit Logsätzen gefüllt. Jeder Logsatz wird eindeutig identifiziert (zum Beispiel durch eine Log-Sequence-Number oder LSN). Wann ein Logsatz von welcher Komponente geschrieben werden muss und welche Information dieser Satz enthält, hängt davon ab, welche Recovery-Technik verwendet wird. Die zeitliche Reihenfolge, in der Operationen ausgeführt werden, muss sich auch im Log widerspiegeln. Falls also eine Transaktion T_1 ein Objekt O_1 verändert, bevor eine Transaktion T_2 eine Änderung am Objekt O_2 vornimmt, muss diese Reihenfolge auch in den zugehörigen Logsätzen gelten, was eben durch eine sequentielle Verwaltung des Logs erreicht wird.

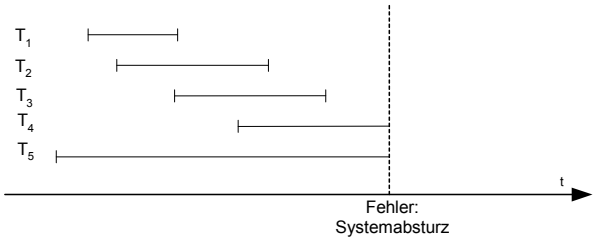


Abbildung 4-10: Szenario eines Systemfehlers nach (Vossen 1993)

Der *Recoverymanager* ist die Softwarekomponente, die auf einem Knoten bei einem Transaktionsabbruch oder bei einem Restart eines Systems aufgerufen wird, um die Konsistenz wiederherzustellen. Er braucht die Dienste des Loggingmanagers, um seine Aufgaben zu erledigen, damit er anhand der geschriebenen Logsätze die Zustände der zuletzt bearbeiteten Transaktionen nachvollziehen kann.

Ein Problem beim Recovery bilden *reale* beziehungsweise *persistente* Operationen. Dies sind Operationen, die bei der Ausführung sofort wirksam werden und nicht mehr zurückzusetzen sind, wie zum Beispiel das Verändern eines Hardwareschalters. Reale Operationen dürfen vom zuständigen Ressourcenverwalter erst ausgeführt werden, wenn die Commit-Entscheidung getroffen ist.

Zur Kommunikation der beteiligten Komponenten werden Protokolle benötigt, welche die Möglichkeit bieten, bei Fehlern eine Koordination über das tatsächliche Ergebnis einer Transaktion durchzuführen. Gray (Gray 1993) unterscheidet hierzu *lokale* und *verteilte Recovery-Protokolle*, wobei die lokalen Protokolle für die Kommunikation der Komponenten innerhalb eines Knoten eingesetzt werden, wäh-

rend die verteilten Protokolle zur Kommunikation über ein Netzwerk verwendet werden.

Um eine Änderungsoperation im Fehlerfall zurücksetzen zu können, muss vor deren Ausführung ein Logsatz mit dem alten Wert des Objekts (*Before-Image*) geschrieben werden. Diese Technik ist auch als *WAL* (*write ahead log*) bekannt. Das kontinuierliche Schreiben der Logginginformation auf einen dauerhaften Hintergrundspeicher belastet die Performance eines Systems, weshalb verschiedene *Ersetzungs- und Einbringstrategien* zur Optimierung entwickelt wurden.

Einbringstrategien legen den Zeitpunkt der Schreibzugriffe für Daten, die bereits *committed* sind, auf den nicht-flüchtigen Speicher fest. Hierzu gehören:

- Eine bekannte Strategie wird als *Noforce-Strategie* bezeichnet. Hier wird am Ende einer Transaktion kein Schreiben der Änderungsoperationen auf einen nicht-flüchtigen Speicher gefordert. Die Änderungen werden asynchron gesichert, was aber im Fehlerfall Nachteile beim Recovery hat.
- Bei der sog. *Force*[9]*-Strategie* wird am Ende einer Transaktion das Schreiben der Änderungsoperationen auf einen nicht-flüchtigen Speicher gefordert.

Ersetzungsstrategien legen fest, wann Daten, die noch nicht „committed" sind, auf nicht-flüchtigen Speicher geschrieben werden dürfen. Man unterscheidet:

- Die *Steal*[10]*-Strategie* erlaubt es, dass Daten bereits geschrieben werden dürfen, wenn sie noch nicht *committed* sind.
- Im Gegensatz dazu dürfen bei der *Nosteal-Strategie* Daten nicht geschrieben werden, wenn sie noch nicht *committed* sind.

Die unterschiedlichen Optimierungen für das „Einbringen" und „Ersetzen" der Daten erfordern im Recovery-Fall bestimmte Strategien, die im nächsten Absatz diskutiert werden.

Undo/Redo-Verfahren

Die lokalen Recovery-Protokolle unterscheiden sich durch die Maßnahmen, die im Recovery-Fall zu ergreifen sind, um bereits ausgeführte Aktionen rückgängig zu machen oder erneut auszuführen. Sie sind auch unter der Bezeichnung *Undo- und Redo-Protokolle* bekannt:

- Ein Redo-Protokoll ist erforderlich, wenn eine Transaktion beendet werden kann, noch bevor alle Operationen, die bereits committed sind, dauerhaft ausgeführt wurden. Dies ist die Folge einer *Noforce*-Einbringstrategie. Das Schreiben der Änderungen wird also bei Transaktionsende nicht erzwungen.

---

9 Engl.: to force = erzwingen.

10 Engl.: to steal = entwenden.

Nach einem Fehler kann es dann aber passieren, dass einige Operationen erneut auszuführen sind.

– Ein Undo-Mechanismus muss unterstützt werden, wenn in einer Transaktion bereits Operationen ausgeführt (auf nicht-flüchtigen Speicher geschrieben), also persistent gemacht werden, bevor die Transaktion in den Committed-Zustand übergeht. Dies kann bei Verwendung der *Steal*-Ersetzungsstrategie vorkommen. Im Fehlerfall ist es hier möglich, dass eine Transaktion zurückgesetzt wird und daher noch Änderungen von Objekten wirksam bleiben, die beim Restart zurückgenommen werden müssen.

Muss ein Recovery-Protokoll eine Undo- und eine Redo-Logik unterstützen, um im Fehlerfall wieder einen konsistenten Zustand herzustellen, so wird es als *Undo/Redo-Protokoll* bezeichnet. Ist zur Konsistenzerhaltung weder eine Undo- noch eine Redo-Logik notwendig, wird das Recovery-Protokoll als *No-Undo/No-Redo-Protokoll* bezeichnet. Dementsprechend können noch die Protokolle *Undo/Redo* und *No-Undo/Redo* auftreten.

Die Recovery-Performance wird durch sehr große Logs beeinträchtigt. Da das Log mit der Zeit stark anwachsen kann, muss ein Mechanismus existieren, der dessen Größe begrenzt. Aus diesem Grund unterstützen DBMS häufig *Sicherungspunkte* oder *Checkpoints*. Dies sind spezielle Logsätze, die bei einem Restart die Suche im Log nach offenen oder erneut durchzuführenden Transaktionen begrenzen. Eine ausführliche Einführung in die Verwendung von Checkpoints ist in (Gray 1993) zu finden.

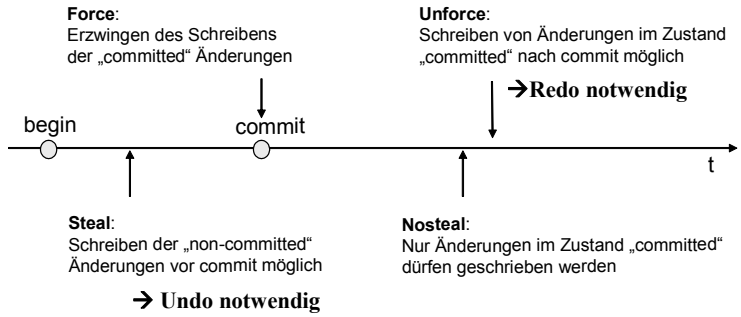


Abbildung 4-11: Steal- und Nosteal – Undo und Redo

In einem DBMS wird der Teil des Logs, der nicht mehr für aktuell ablaufende Transaktionen benötigt wird, zyklisch auf ein anderes Speichermedium (zum Beispiel andere Platte oder Magnetband) archiviert. Dieser ausgelagerte Teil wird nur noch benötigt, wenn durch einen Plattencrash die Datenbank oder Teile davon zerstört werden. In einem Restartverfahren können die bereits ausgeführten Transaktionen aus dem Archiv nachgefahren (redo) werden. Das direkt zugreifbare Log wird für schnelle Wiederanlaufverfahren nach einem Systemcrash, nach

Absturz einer Anwendung oder bei einem von einer Anwendung geforderten Rollback einer Transaktion verwendet. Abbildung 4-11 zeigt nochmals die Zusammenhänge zwischen den Ersetzungs- und Einbringstrategien und den Recovery-Verfahren.

Abschließend zu dieser kurzen Betrachtung der Logging- und Recovery-Verfahren kann festgehalten werden, dass diese eng mit dem Transaktionsmechanismus zusammenspielen. Insbesondere in DBMS hat man hier eine Vielzahl von Optimierungen entwickelt, die in Kombination eingesetzt werden.

4.3 Koordinationsprotokolle und verteiltes Recovery

Bei einer verteilten Transaktion wird angenommen, dass diese von einem Knoten beziehungsweise einem Prozess dieses Knotens angestoßen wird, der auch als Koordinator (superior, coordinator) die gesamte verteilte Transaktion kontrolliert. Alle Knoten, die an der Transaktion teilnehmen, werden als Teilnehmer (subordinate, participant) bezeichnet. Für die Koordination werden als Freigabe- bzw. Koordinationsprotokolle (auch Commit-Protokolle genannt) unter anderem das *One-Phase-*, das *Two-Phase-* und das *Three-Phase-Commit-Protokoll (2PC und 3PC)* verwendet. Die Abstimmung zwischen Koordinator und Teilnehmern geschieht dabei in einer oder in mehreren Phasen.

4.3.1 Two-Phase-Commit-Protokoll

Beim 2PC-Protokoll sendet der Koordinator in der ersten Phase (Prepare-Phase) einen Request (Prepare-Request) an alle Teilnehmer einer Transaktion, damit diese sich auf das Transaktionsende vorbereiten. Die Teilnehmer antworten (votieren) mit "ready" oder "not-ready", je nachdem, ob sie die Transaktion erfolgreich zu Ende führen können oder nicht. Votiert ein Teilnehmer mit "not-ready", setzt er die innerhalb der Transaktion ausgeführten Operationen selbstständig zurück. Nur wenn alle Teilnehmer mit "ready" antworten, wird vom Koordinator die zweite Phase eingeleitet, um die Transaktion erfolgreich zu beenden. Andernfalls setzt der Koordinator die Transaktion zurück, indem er allen Teilnehmern, die mit "ready" antworteten, einen Abort-Request sendet.

Wenn ein Teilnehmer "ready" meldet, muss er in der Lage sein, die Transaktion aus seiner Sicht zu beenden oder zurückzusetzen, je nachdem wie sich der Koordinator entscheidet. Eine einmal getroffene Entscheidung darf weder vom Koordinator noch von einem Teilnehmer zurückgenommen werden.

In der zweiten Phase sendet der Koordinator einen Commit-Request, um die Transaktion abzuschließen. Die Teilnehmer nehmen den Request entgegen, führen ihn aus und beantworten ihn entsprechend. Bei der Abwicklung des Protokolls sind zu jeder Zeit Fehler (etwa durch Nachrichtenverluste) vorstellbar. Diese Fehler können dazu führen, dass ein Teilnehmer oder Koordinator eine Nachricht, auf

die er wartet, nicht oder nicht sofort erhält. Deshalb sind Timer zur Begrenzung der Wartezeit erforderlich. Ein Knoten soll nicht durch andere Knoten blockiert werden. Im Falle eines Systemabsturzes muss ein Teilnehmer oder der Koordinator über ein Log verfügen können, aus dem er den Zustand der Transaktion erfahren kann. Daher ist es notwendig, dass bestimmte Informationen während der Koordinierung protokolliert werden.

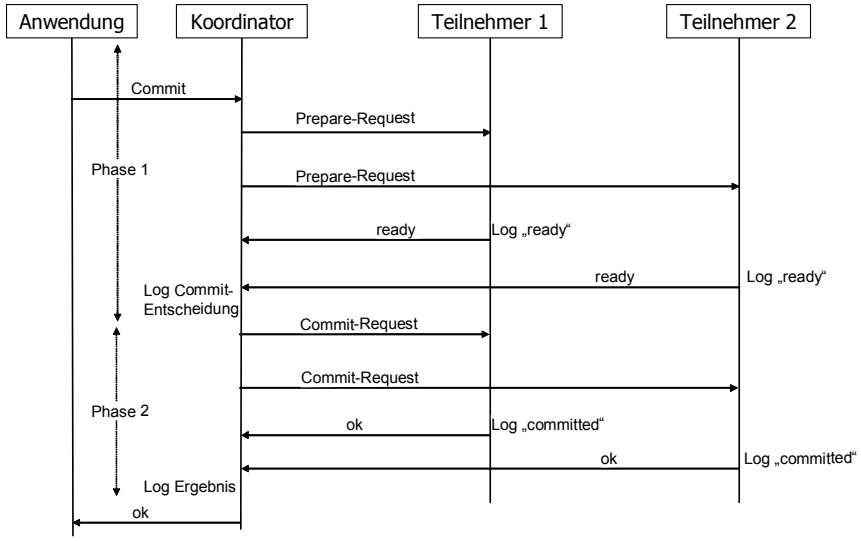


Abbildung 4-12: Ordnungsgemäßer 2PC-Protokollablauf mit zwei Teilnehmern

Der positive Ablauf einer 2PC-Koordination mit zwei Transaktionsteilnehmern ist im Sequenzdiagramm in Abbildung 4-12 dargestellt. Die betroffene Anwendungskomponente beendet mit einem Commit-Aufruf aktiv die Transaktion und ist bis zum Ende der Koordination blockiert. Der zuständige Koordinator führt das 2PC-Protokoll mit den beiden Teilnehmern aus. Für die Koordination sind insgesamt acht Nachrichten erforderlich.

In Abbildung 4-13 ist der Fehlerfall dargestellt, dass Teilnehmer 2 seine Operationen nicht erfolgreich zu Ende führen kann. Er liefert als Ergebnis der Phase 1 „not ready“, worauf sich der Koordinator für den Abbruch der Transaktion entscheidet und dies dem Teilnehmer 1 mitteilt. Das Schreiben der Logging-Informationen ist in den beiden Abbildungen nur angedeutet und auch von der Implementierung und entsprechenden Optimierungen abhängig, das Grundprinzip wird aber damit beschrieben.

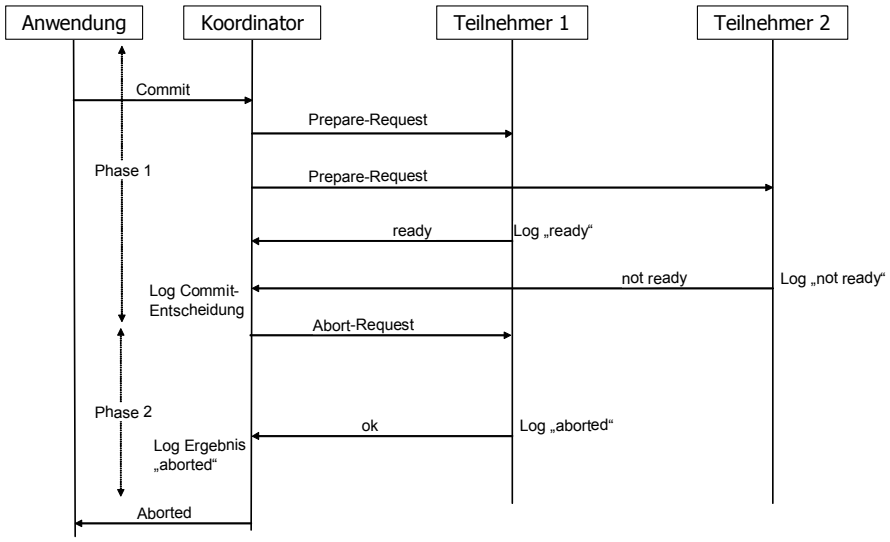


Abbildung 4-13: Fehlerhafter 2PC-Protokollablauf

Es ist nun möglich, dass zu jeder Zeit ein System- oder Netzausfall vorkommen kann und das Transaktionssystem darauf mit einer Wiederanlaufstrategie reagieren muss. Diese wird durch den Koordinator initiiert. Kommt z.B. der Koordinator nicht mehr zum Laufen oder er ist vom Netzwerk abgeschnitten oder aber es ist ein Teilnehmer von den anderen abgeschnitten, so befindet sich mindestens ein Teilnehmer in einem unsicheren Zustand und kann auch mögliche Sperren, die er für nicht abgeschlossene Transaktionen gesetzt hat, nicht freigeben.

Es ist auch möglich, dass ein Teilnehmer bei Ausfall einer Netzverbindung eine heuristische Entscheidung über die Beendigung der Transaktion trifft, wie dies etwa im von der OSI genormten CCR Service Element vorgesehen ist[11]. Ist die Verbindung zum Koordinator wieder da, muss diesem aber die Entscheidung mitgeteilt werden. Hat dieser bereits anders entschieden, liegt eine Inkonsistenz vor.

Im 2PC-Protokoll besteht die Möglichkeit, dass Transaktionen, die nach einer Ready-Antwort auf einen Commit-Request der Phase 2 warten, nicht zu Ende gebracht werden können und damit Datenobjekte etwa durch Sperren blockieren. Dies ist dann der Fall, wenn der Koordinator nach der Phase 1 ausgefallen ist. Das 2PC-Protokoll wird daher auch als blockierendes Commit-Protokoll bezeichnet.

[11] Zu OSI CCR siehe weiter unten.

Der Zustand eines Teilnehmers, in dem er blockiert ist und auf die Entscheidung des Koordinators wartet, wird auch als unsicherer Zustand bezeichnet.

4.3.2 Three-Phase-Commit-Protokoll

Im 2PC-Protokoll können sich nebenläufige Transaktionen blockieren. Diesen Mangel versucht das relativ selten implementierte 3PC-Protokoll (Three-Phase-Commit) zu beheben. Es ist eine echte Erweiterung zum 2PC-Protokoll, in der die Phase 1 identisch zum 2PC ist. Im 3PC-Protokoll wird aber nach der ersten Phase eine Phase zur Entscheidungsvorbereitung eingeschoben.

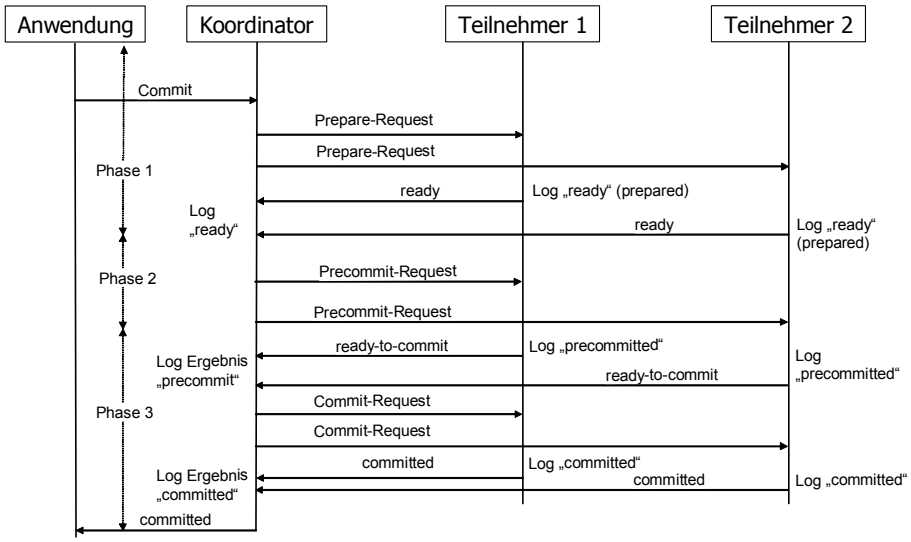


Abbildung 4-14: Ordnungsgemäßer 3PC-Protokollablauf

Die Entscheidungsphase wird ihrerseits wieder in zwei Phasen aufgeteilt:

- Nach einer erfolgreich verlaufenden Phase 1 haben alle Teilnehmer dem Koordinator mit „ready“ geantwortet. Sie befinden sich nun in einem unsicheren Zustand und warten auf weitere Anweisungen des Koordinators.
- Daraufhin sendet der Koordinator einen „Precommit-Request“ an alle Teilnehmer. Wenn ein Teilnehmer diese Nachricht erhält, weiß er, dass alle anderen Teilnehmer mit „ready“ votiert haben. Er ist also nicht unsicher.
- Der Koordinator entscheidet aber noch nicht auf „Commit“, weiß aber, dass er dies tun wird, wenn er nicht abstürzt.
- Jeder Teilnehmer sendet eine Antwort „ready-to-commit“ an den Koordinator.

– Wenn der Koordinator alle Antworten erhalten hat, verschickt er den Commit-Request, den die Teilnehmer ihrerseits beantworten.

Bei Ausfall des Koordinators während der Commit-Phase (Phase 2 und 3) wird ein Terminierungsprotokoll angewendet, in dem die verbleibenden Teilnehmer einen neuen Koordinator wählen. Auf dieses soll hier nicht weiter eingegangen werden. Weitere Erläuterungen zum 3PC-Protokoll finden sich in (Vossen 1993) und (Bernstein 1987).

4.3.3 One-Phase-Commit-Protokoll

Beim Ein-Phasen-Commit-Protokoll (1PC) wird die Prepare-Phase weggelassen, wobei es hier auch wieder mehrere Varianten gibt. Die einfachste Variante, die auch im DTP-Modell unterstützt wird, ist in Abbildung 4-15 dargestellt. Der Koordinator schickt einfach anstelle eines Prepare-Request sofort einen Commit-Request, der vom Teilnehmer beantwortet wird. Der Teilnehmer schließt daraufhin die Transaktion vollständig ab und schreibt im Erfolgsfall die geänderten Daten auf einen persistenten Speicher.

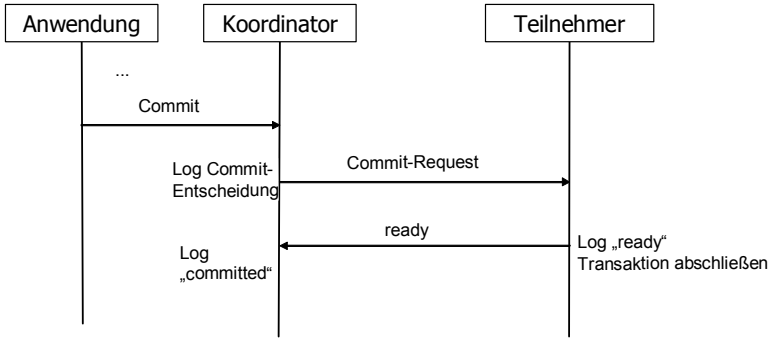


Abbildung 4-15: Klassischer, fehlerfreier 1PC-Protokollablauf

Eine andere Variante des 1PC-Protokolls soll ebenfalls kurz vorgestellt werden: Bereits mit der letzten Operation einer Transaktion kann der Teilnehmer auch aufgefordert werden, schon vorab in den Zustand *prepared* zu wechseln. Der Teilnehmer votiert daraufhin entweder „ready“ oder „abort“. Für die eigentliche Commit-Behandlung wird somit ebenfalls nur noch eine Phase benötigt.

Das Transaktionssystem muss beim 1PC-Protokoll bei der Ausführung von Operationen die Möglichkeit bieten, zu jeder Zeit eine Commit-Behandlung einleiten können, es ist also eine entsprechende Schnittstelle für das Anwendungsprogramm vorzusehen.

Beim 1PC-Protokoll reduziert sich in verteilter Umgebung entsprechend auch die Anzahl der zu sendenden Nachrichten, die für die Koordinierung erforderlich sind. Bei einer großen Anzahl an Transaktionen ist dies durchaus ein wichtiger Leistungsaspekt. Allerdings ist dieses Commit-Protokoll nur dann wirklich sinnvoll (und dann aber auch nützlich), wenn an der Transaktion lediglich ein Transaktionsteilnehmer beteiligt ist.

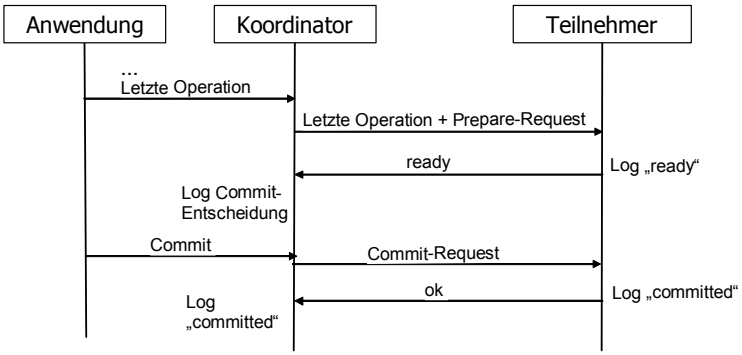


Abbildung 4-16: Ein anderer 1PC-Protokollablauf

4.3.4 Vergleich und Optimierungsvarianten

Das 2PC-Protokoll ist zweifelsohne das bedeutendste Commit-Protokoll. Nach (Ramamritham 1997) und (Stamos 1993) gibt es mehrere Variationen dieses Protokolltyps mit unterschiedlichsten Optimierungen:

- 2PC mit *presumed abort* optimiert zum Beispiel die Anzahl der Schreibzugriffe auf das Log. Der Koordinator schreibt nur Commit-Records in die Log-Datei und keine Abort-Records. Der Koordinator nimmt bei diesem Protokoll bei einem Wiederanlauf nach einem Absturz an, dass Transaktionen, für die kein Commit-Satz im Log gefunden werden kann, zurückzusetzen sind.
- Weitere Optimierungen sind *presumed nothing*, *presumed commit* und *early prepare*. Bei *presumed commit* werden im Gegensatz zu *presumed abort* nur Abort-Records geschrieben. Das Verhalten im Recovery-Fall ist hier genau umgekehrt. Für eine detaillierte Beschreibung dieser Spezialfälle, insbesondere der Fehlerbehandlung wird auf die angegebene Literatur verwiesen.

Betrachtet man die verschiedenen Commit-Protokolle im Vergleich, so kann man neben dem Aufwand für das Schreiben des Logs auch das Nachrichtenaufkommen für die Commit-Behandlung bei den einzelnen Varianten betrachten.

Die Tabelle 4-2 gibt hier eine Übersicht über das Nachrichtenaufkommen bei den „Standardformen“ der 1PC-, 2PC- und 3PC-Protokolle. In der Tabelle wird mit n

die Anzahl der beteiligten Knoten angegeben, wobei einer davon der Koordinator ist.

Tabelle 4-2: Nachrichtenaufkommen bei Commit-Protokollen

Commit-Protokoll	Allgemeine Formel	Beispiel 1 (n=2)	Beispiel 2 (n=10)	Beispiel 3 (n=11)
1PC	2 * (n-1)	2	18	20
2PC	4 * (n-1)	4	36	40
3PC	6 * (n-1)	6	54	60

Optimierungen sind in der Tabelle nicht berücksichtigt. Eine Optimierung wäre beispielsweise, dass rein lesende Teilnehmer in der Phase 2 nicht mehr kommunizieren müssen, da sie ohnehin keine Veränderungen vorgenommen haben.

4.3.5 Zustandsautomaten für Commit-Protokolle

Commit-Protokolle sind im Allgemeinen recht komplexe Kommunikationsprotokolle, die im ISO/OSI-Referenzmodell der Schicht 7 und im TCP/IP-Referenzmodell der Schicht 5 zuzuordnen sind. Kommunikationsprotokolle werden häufig als erweiterte, kommunizierende, endliche Automaten, sog. CEFSM (Communicating Extended Finite State Machine) beschrieben.[12]

Sowohl der Koordinator als auch ein Teilnehmer einer Transaktion kann in einem Zustandsautomaten beschrieben werden. Wir wollen an dieser Stelle die Zustandsautomaten für das 2PC-Protokoll exemplarisch und vereinfacht darstellen. In Abbildung 4-17 sind jeweils ein Koordinator- und ein Teilnehmerknoten dargestellt, die mit einer 2PC-Protokollinstanz ausgestattet sind. Diese Protokollinstanzen nutzen ein Transsportsystem (heute meist TCP oder UDP) zur Kommunikation. In den Knoten sind auch Softwarekomponenten, die den eigentlichen Koordinator sowie den Teilnehmer repräsentieren. Beide nutzen jeweils den im Bild bezeichneten 2PC-SAP (Service Access Point), um die Dienste der 2PC-Instanzen zu verwenden.

Die beiden 2PC-Instanzen tauschen 2PC-Nachrichten (PDUs) aus. Dies sind u.a. eine *2PC-Prepare-Request-PDU* und eine *2PC-Commit-Request-PDU* mit den entsprechenden Antworten. In den Nachrichten werden meist neben der eigenen Identifikation der Transaktionskontext bzw. ein global eindeutiger Transaktions-Identifier übertragen, der an den Transaktionskontext gebunden ist.

---

[12] Ein CEFSM erhält von seiner Umgebung Eingangssignale wie ankommende Nachrichten von Partnerautomaten oder Dienstanforderungen höherer Schichten, die er in Aktionen bearbeitet. Die Beschreibung kann u.a. in SDL (Specification and Description Language) erfolgen.

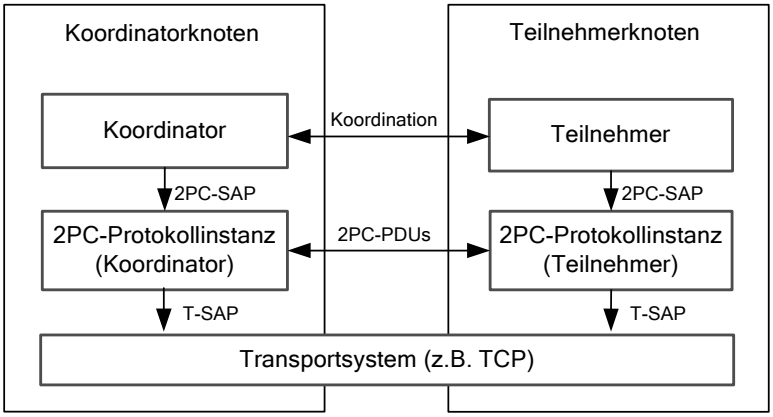


Abbildung 4-17: Einordnung des 2PC-Protokolls in den Kommunikationsstack

Alle beteiligten Instanzen können mit Zustandsautomaten modelliert werden. Jeder Automat ist natürlich mit einem Zeitgeber (Timer) ausgestattet, um gewisse Timer-Events zu erhalten. Man benötigt derartige Mechanismen, um auf fehlende Nachrichten nach einer bestimmten Zeit reagieren zu können. Dieser Aspekt wird aber hier vernachlässigt. Exemplarisch sollen in diesem Abschnitt die vereinfachten Automaten der beiden 2PC-Instanzen dargestellt werden. Die beiden Automaten sind nur ein Beispiel für eine 2PC-Implementierung für die Abwicklung flacher Transaktionen ohne Optimierungen.

Der Koordinatorautomat unseres Beispiels aus Abbildung 4-18 hat sieben Zustände. Im Zustand *Init* ist die Transaktion erzeugt, der Transaktionskontext ist propagiert und es werden Operationen (im Bild vernachlässigt) ausgetauscht. Im positiven Fall ruft die Anwendung, welcher der Transaktionskontext zugeordnet ist, einen Commit-Dienst am 2PC-SAP auf. Daraufhin wird an alle Teilnehmer eine *Prepare-Request-PDU* gesendet und es wird in den Zustand *Preparing* gewechselt. In dem Zustand verbleibt der Automat so lange, bis alle *Prepare-Response-PDUs* der Teilnehmer eingesammelt wurden. Kommen diese nicht alle in einer bestimmten Zeitspanne an, oder sendet ein Teilnehmer „not ready“, wird die Transaktion abgebrochen, also in den Zustand *Aborting* versetzt.

Würde nun eine Antwort nicht gleich kommen, so wäre ein erneutes Senden der *Prepare-Request-PDU* nach einer definierten Zeit sinnvoll. Das könnte man auch noch mehrmals wiederholen, bevor die Transaktion negativ beendet würde. Ähnliches gilt für die Commit-Behandlung. Man sieht, dass die Fehlerbehandlung den Zustandsautomaten aufbläht.

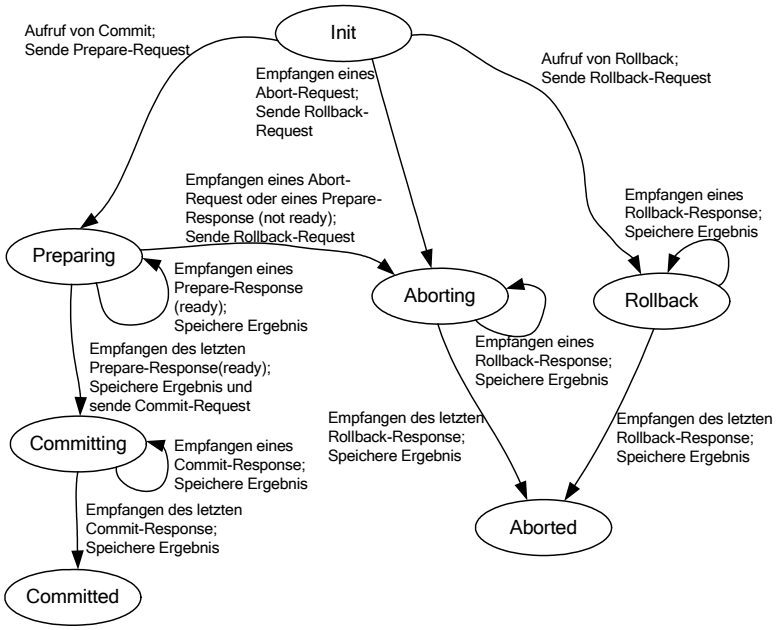


Abbildung 4-18 Vereinfachter 2PC-Koordinatorautomat

Bricht die Anwendung von sich aus die Transaktion mit einem *Rollback*-Aufruf ab, so wird an alle Teilnehmer eine *Rollback-Request-PDU* gesendet und es werden die Antworten im Zustand Rollback eingesammelt.

Weiterhin kann es sein, dass ein Teilnehmer vorzeitig eine Transaktion abbricht und dies dem Koordinator mitteilt. In diesem Fall wird der Zustand *Aborting* eingenommen. Alle anderen Teilnehmer werden ebenfalls abgebrochen. Sowohl vom Zustand *Rollback* als auch vom Zustand *Aborting* wird anschließend in den Zustand *Aborted* gewechselt.

Der Koordinator muss für jede einzelne Transaktion, die er erzeugt hat, einen derartigen Zustandsautomaten verwalten.

Der Teilnehmerautomat ist etwas einfacher. Im Zustand *Init* werden alle ankommenden Operationen auf die nebenläufig genutzten Ressourcen bearbeitet. Sobald eine *Prepare-Request-PDU* empfangen wird, wird eine *Prepare-Response-PDU* gesendet und es wird der Zustand *Prepared* eingenommen. In diesem Zustand kann der Teilnehmer nicht mehr alleine entscheiden, ob er die Transaktion zu Ende führt oder abbricht. Er ist unsicher über den Ausgang der Transaktion. Das weitere Vorgehen entscheidet der Koordinator.[13]

[13] Ausgenommen sind heuristische Entscheidungen.

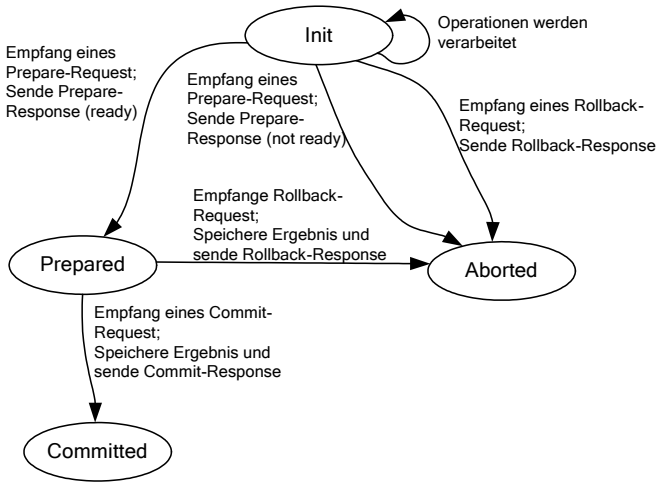


Abbildung 4-19: Vereinfachter 2PC-Teilnehmerautomat

Vom Zustand *Prepared* kommt der Teilnehmer über den Empfang einer *Commit-Request-PDU* in den Zustand *Committed*. Auf diese PDU reagiert er mit einer *Commit-Response-PDU*. Falls der Teilnehmer einen *Prepare-Request* nicht positiv beantworten kann, sendet er eine *Prepare-Response-PDU (not ready)* und geht in den Zustand *Aborted*.

Ankommende *Rollback-Request-PDUs* quittiert der Teilnehmer, protokolliert sie und geht direkt in den Zustand *Aborted*. Der Teilnehmer muss natürlich nach dem *Commit-* oder einem *Abort-Response* alle evtl. gesetzten Sperren freigeben.

Es soll noch erwähnt werden, dass alle wesentlichen Zustandsänderungen, die ein Recovery-Vorgang benötigt, persistent geloggt werden müssen. Dies gilt sowohl für den Teilnehmer als auch für den Koordinator. Dabei ist die unterstützte Recovery-Strategie von Bedeutung. Nicht dargestellt wurde der Recovery-Fall, in dem die erneut gestartete Komponente seinen Zustand aus dem Logfile und dem Netzwerk erfragen muss, um dann Transaktionen, die noch im Zustand P*repared*, *Preparing* oder *Committing* sind, weiter bearbeiten zu können. Eine vollständige Beschreibung der Zustandsautomaten eines 2PC-Protokolls wird sehr komplex.

Der Teilnehmer muss für jede nebenläufige Transaktion, die auf seine Ressourcen zugreift, einen Zustandsautomaten der dargestellten Form verwalten. Die Summe der Zustände aller kooperierenden Zustandsautomaten zu einer Transaktion stellt den aktuellen Gesamtzustand der Transaktion dar.

4.4 Standards der Transaktionsverarbeitung

4.4.1 OSI-Transaktionsprotokolle

In diesem Abschnitt wird ein kurzer Überblick über die von der ISO definierten OSI-Transaktionsstandards gegeben. Dies ist zum einen OSI CCRSE und zum anderen OSI TPSE, das auch kurz als OSI TP (Transaction Processing)[14] bezeichnet wird. Um diese in der Schicht 7 des ISO/OSI-Referenzmodells angesiedelten Protokolle und Dienste zu verstehen, wird vorab eine Einführung in die Strukturierung der ISO/OSI-Anwendungsschicht gegeben. Dieser OSI-Standard ist in (Kerner 1993) ausführlich beschrieben.

Die OSI-Anwendungsschicht

Im Rahmen der ISO-Standardisierungsbemühungen wurden internationale Standards für das Transaktionsmanagement kommunizierender Anwendungsprozesse entwickelt. Es handelt sich dabei um die Service Elemente CCRSE und TPSE mit den dazugehörigen Protokollen, die beide der Schicht sieben des ISO/OSI-Referenzmodells zuzuordnen sind. Zunächst sollen die Grundlagen der OSI-Anwendungsschicht dargestellt werden, um anschließend einen Überblick über die beiden Standards geben zu können.

OSI-Standards sind für *offene Systeme* entwickelt. Darunter versteht man kein konkretes Rechnersystem, sondern ein Modell eines Systems, dessen Komponenten den OSI-Standards gehorchen. Ein *reales offenes System* besteht aus Rechnern, Software, Peripherie, Übertragungsmedien und einem Satz von OSI-Standards für die Kommunikation mit anderen realen offenen Systemen. Das OSI-Architekturmodell normt nur die Interaktionen, nicht aber die lokale Implementierung in einem realen System. "Offenheit" wird also nur durch die genormten Kommunikationsmechanismen unterstützt. Für jede der Schichten 1 bis 6 des ISO/OSI-Referenzmodells ist ein bestimmtes *Dienstelement* (*Service Element*) zur Nutzung durch die darüberliegende Schicht definiert. Ein Dienstelement besteht aus einem oder mehreren *Diensten* (*Services*), die wiederum als *Dienstprimitive* (*Service Primitives*) definiert werden. Zur Nachrichtenübertragung sind für jede Schicht Protokolle definiert. Ein aktives, funktionsfähiges Element einer Schicht wird als *Entity* (oder Instanz[15]) bezeichnet, der tatsächlich ablaufende Code einer Entity als *Invocation*.

---

[14] OSI-TP ist sehr stark mit dem SNA-Protokoll LU6.2 von IBM vergleichbar.

[15] Der engl. Begriff *Entity* kann auch im Deutschen mit Instanz übersetzt werden.

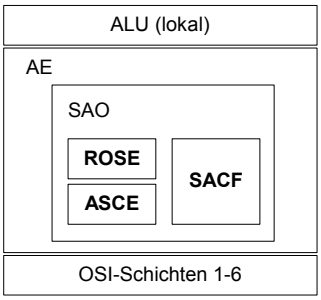


Abbildung 4-20: Beispiel für einen Anwendungsprozess mit einem SAO

Im OSI-Sprachgebrauch besteht eine verteilte Anwendung aus Anwendungsprozessen, die miteinander kommunizieren. Ein Anwendungsprozess verwendet die Dienste seines lokalen offenen Systems inkl. der Schichten 1 bis 6. Er enthält einen lokalen Teil, der außerhalb der Normierung liegt (auch *Application Layer User* oder ALU genannt) und einen Teil, der Bestandteil des offenen Systems ist (*Application Entity*, AE). Da die Schicht sieben durch die Vielfalt der möglichen Anwendungen sehr komplex ist, kann sie nicht durch einen einzigen Dienst repräsentiert werden. Stattdessen werden mehrere Basiselemente, die *Application Service Elements* (ASE) definiert. Standardisierte ASEs sind zum Beispiel FTAM, ACSE, ROSE, CMISE, CCRSE und TPSE. Eine Anwendung wählt einen Satz von ASEs zur Erfüllung ihrer Aufgaben aus. Zusammen mit einer *Single Association Control Function* (SACF), die für die Koordination verantwortlich ist, bildet ein Satz von ASEs ein *Single Association Object* (SAO). Abbildung 4-20 zeigt einen Anwendungsprozess mit einem SAO bestehend aus ROSE und ACSE.

Zwei AEs können miteinander kommunizieren, wenn sie das gleiche SAO verwenden. Eine Kommunikationsbeziehung der Schicht 7 wird als *Assoziation* bezeichnet. Assoziationen werden durch ACSE (Association Control Service Element) verwaltet, das von den meisten anderen ASEs verwendet wird. Ein AE kann beliebig viele SAOs verwenden, wobei ein SAO wiederum aus ASEs und SAOs bestehen kann. Die Koordination mehrerer SAOs regelt eine übergeordnete CF. Eine konkrete Ausprägung beziehungsweise ein tatsächlich ablaufender Code eines AE heißt *Application Entity Invocation* (AEI).

OSI CCRSE

OSI-CCRSE (*Commitment, Concurrency and Recovery Service Element*) wurde in den 80er Jahren spezifiziert und bietet Dienste und entsprechende Protokolle zur Koordination verteilter Transaktionen. CCRSE ist ein ASE und verwendet als Kommunikationsbasis ACSE und damit einen vollständigen OSI-Stack. Eine Transaktion ist dabei als Menge von Aktionen beschrieben, die unter Einhaltung der ACID-Eigenschaften ausgeführt werden. CCRSE dient der Koordination einfacher Asso-

ziationen. Für die Abwicklung mehrerer Assoziationen durch einen Anwendungsprozess ist eine übergreifende CF mit der Bezeichnung CCRCE (*Commitment, Concurrency and Recovery Coordination Element*) erforderlich, für die keine Normierung vorliegt. CCRSE verwendet für die Funktionseinheiten (Anwendungsprozesse), die sich an einer Transaktion beteiligen auch die Bezeichnung *Knoten* (*node*). Um Verwirrungen zu vermeiden, verwenden wir hier jedoch den Begriff Anwendungsprozess. CCRSE bietet ein 2PC-Protokoll (Variante *presumed abort*) zur Koordinierung verteilter Transaktionen an, das die Anwendungsprozesse verwenden können und fordert, dass die über CCRSE kooperierenden Prozesse die ACID-Eigenschaften einhalten sollen. Durch das 2PC-Protokoll unterstützt CCRSE nur die Atomaritätseigenschaft, nicht aber die anderen drei Eigenschaften einer ACID-Transaktion, der Begriff CCR ist also etwas irreführend.

Mechanismen zur Behandlung nebenläufiger Zugriffe auf Objekte, zur Konsistenzerhaltung und zur persistenten Speicherung sind nicht genormt und unterliegen dem lokalen Teil eines Anwendungsprozesses (also dem ALU). Der Implementierer einer verteilten Transaktionsanwendung muss also selbst geeignete Locking-, Logging- und Recovery-Mechanismen entwickeln, um die ACID-Eigenschaften erfüllen zu können. Die Regeln für die Kommunikation und das Verhalten der Partner ist in CCRSE definiert.

Der Prozess, der eine verteilte Transaktion initiiert, wird als *Superior* bezeichnet und der ausführende Prozess als *Subordinate*. Der prinzipielle Ablauf einer Transaktion zwischen zwei Anwendungsprozessen sieht wie folgt aus (vgl. Abbildung 4-21):

- Der Superior startet eine Transaktion mit dem Dienst-Aufruf C-BEGIN, und sendet anschließend zusammengehörende Operationen an den Subordinate. Die Transaktion wird global eindeutig über einen *Atomic Action Identifier* verwaltet, so dass parallel ablaufende Transaktionen unterschieden werden können. Für die eigentlichen Operationen werden keine Dienste angeboten. Hierfür sind die Dienste einer anderen ASE zu verwenden.
- Bei Transaktionsende leitet er in der Phase 1 die Commit-Behandlung (über C-PREPARE) ein. Der Subordinate bestätigt seine Bereitschaft zum Commit (C-READY) oder teilt dem Superior über C-REFUSE mit, dass er die Transaktion nicht erfolgreich beenden kann. Die Commit-Behandlung kann auch vom Subordinate durch direkten Aufruf von C-READY initiiert werden.
- In der zweiten Phase werden alle Subordinates der Transaktion vom zuständigen Superior über C-COMMIT aufgefordert, die Transaktion dauerhaft zu machen.
- Der Subordinate bestätigt dies über C-DONE. In Fehlerfällen kann die Transaktion über entsprechende Dienste (C-ROLLBACK, C-RESTART) zurückgesetzt werden.

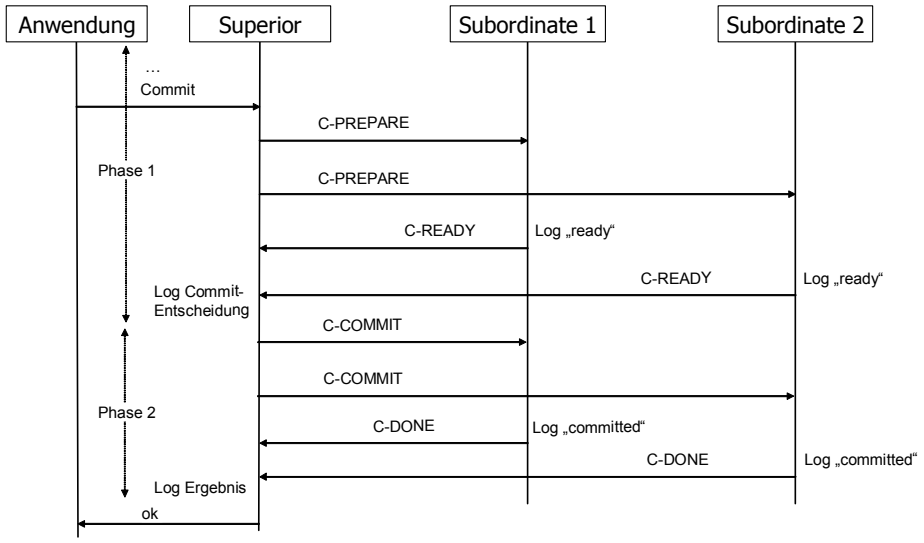


Abbildung 4-21: Typischer Ablauf der CCRSE-Kommunikation

CCRSE unterstützt heuristische Entscheidungen von unsicheren Subordinates. Der Subordinate darf jedoch die Transaktion erst abschließen, wenn der Superior von der Entscheidung unterrichtet wurde. Da der Superior zu einer anderen Entscheidung kommen kann, sind durch heuristische Entscheidungen Inkonsistenzen möglich. Deshalb schreibt CCRSE vor, dass die Entscheidungen vom Anwender zu treffen sind.

Mit CCRCE kann ein Superior mehrere Assoziationen gleichzeitig bedienen, also mehrere Subordinates in eine Transaktion einbeziehen. Die Subordinates können wiederum ihrerseits als Superiors andere Subordinates mit einbeziehen, wodurch ein dynamischer *Aktionsbaum* entsteht. Die Knoten des Baumes sind die Anwendungsprozesse mit den entsprechenden Dienstelementen. Die Kanten werden als *Transaktionszweige* (Transaction Branches) bezeichnet und stellen die Aktionen beziehungsweise Teiltransaktionen dar. Jede Operation geht aber von der Wurzel des Baumes aus, also vom ursprünglichen Superior. CCRCE ist verantwortlich für die Koordination der beteiligten Anwendungsprozesse, wie etwa für das Übermitteln der C-BEGINs zu allen Subordinates und die Bearbeitung der C-READY-Antworten. Abbildung 4-22 zeigt einen Aktionsbaum, bei dem ein Anwendungsprozess als Superior auftritt, der mit zwei weiteren Anwendungsprozessen (Subordinates) eine Transaktion ausführt. Neben CCRSE werden die Service Elemente ROSE und ACSE verwendet.

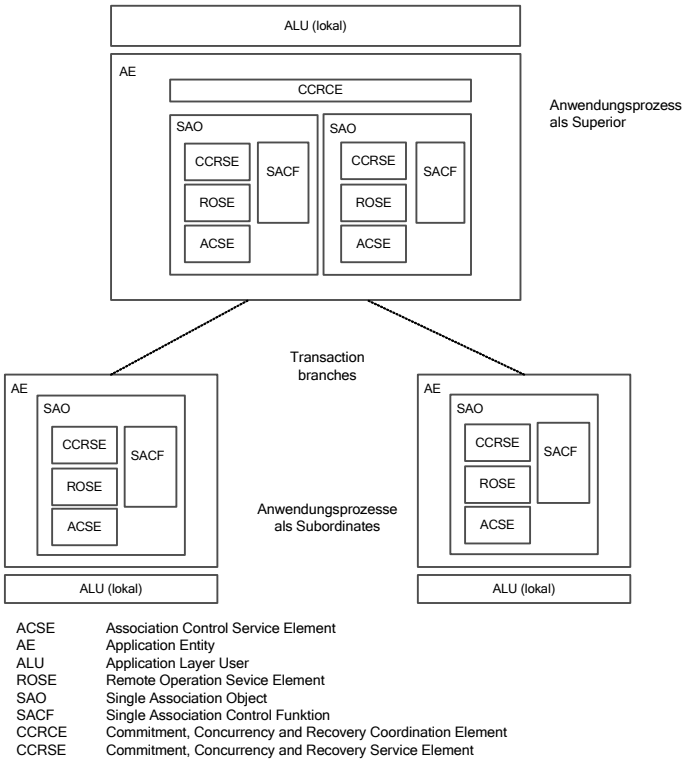


Abbildung 4-22: Aktionsbaum mit einem Superior und Subordinates

OSI TPSE

TPSE (ISO 1992a und ISO 1992b) beinhaltet die Dienste von CCRSE und CCRCE und führt als wesentliche Erweiterung den Begriff des *Dialogs* ein. Das Transaktionsprotokoll wird mit OSI TP bezeichnet. Ein Dialog ist eine über einer Assoziation liegende Kommunikationsbeziehung zwischen zwei TPSUIs (Transaction Processing Service User Invocation), also zweier TPSE-Benutzer. TPSE bietet mehrere Dienste, die zu sieben Funktionseinheiten zusammengefasst sind ("Dialogue", "Shared Control", "Polarized Control", "Handshake", "Commit", "Chained Transactions", "Unchained Transactions"). Für die Benutzung der Funktionseinheiten gelten verschiedene Regeln. Beispielsweise muss die Funktionseinheit "Dialogue" immer verwendet werden, während "Shared Control" und "Polarised Control" sich gegenseitig ausschließen. Die wesentlichen Dienste sind in den Funktionseinheiten "Dialogue" und "Commit" enthalten.

TPSE verwendet zur Definition einer Transaktion die Begriffe von CCRSE. Ein Superior ist verantwortlich für die Transaktion, die Subordinates führen sie aus

und können wiederum andere Subordinates in die Transaktion einbeziehen. Die beteiligten Anwendungsprozesse beziehungsweise deren TPSUIs bilden einen *Dialogbaum*. Die Knoten des Baumes bilden die TPSUIs und die Kanten stellen die Dialogbeziehungen dar.

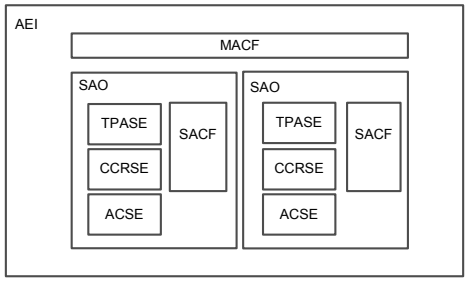
Während der Ausführung einer Transaktion bildet sich dynamisch ein Transaktionsbaum, bei dem die Knoten wiederum die TPSUIs und die Kanten die Teiltransaktionen (Transaction Branches) darstellen. Bei Verwendung der Funktionseinheit "Chained Transactions" wird zum Dialog implizit ein neuer Transaktionszweig im Transaktionsbaum erzeugt.

Dialog- und Transaktionsbaum können also durchaus voneinander abweichen. Über einen Dialogbaum können auch gleichzeitig mehrere Transaktionsbäume entstehen. Eine TPSUI kann zu einer Zeit nur an einer Transaktion beteiligt sein. Die TPSUI, welche die Transaktion einleitet, vergibt eine global eindeutige Transaktionsidentifikation (entspricht dem *Atomic Action Identifier* aus CCRSE) und ist auch für die Beendigung der Transaktion verantwortlich. In der Regel sind Dialogbäume wesentlich langlebiger als Transaktionsbäume.

TPSE bietet wie CCRSE nur Unterstützung zur Erzielung der Atomaritätseigenschaft, nicht jedoch für die anderen ACID-Eigenschaften. Die Erfüllung dieser Transaktionsanforderungen liegt außerhalb der OSI-Sicht. Für den Recovery-Fall gilt in Phase 1 wie bei CCRSE *presumed abort.* Bei der Fehlerbehandlung sind auch heuristische Entscheidungen der Subordinates zulässig. In Phase 2 ist ein zusätzlicher Mechanismus zu Fehlerbehebung möglich. Bei Ausfall einer Assoziation stellt TPSE einen *TP-Channel* als weitere Assoziation zur Fehlerbehebung zur Verfügung. Über diese Assoziation ist TPSE im Gegensatz zu CCRSE in der Lage, nach einem Assoziationsabbruch ein vollständiges Recovery durchzuführen.

Einem TPSU werden die TP-Dienste vom TPSP (*Transaction Processing Service Provider*) geliefert. Der TPSP ist als globaler Service-Provider zu sehen, der für jede TPSUI eine TPPM (Transaction Processing Protocol Machine) bereitstellt und für jede AEI eine CPM (*Channel Protocol Machine*). Eine AEI entspricht in der Regel einer oder mehreren TPSUIs, die jeweils ein oder mehrere SAOs (*Single Association Object)* enthält. Ein SAO ist für eine Assoziation mit einer Partner-TPSUI verantwortlich und besteht seinerseits wiederum aus einer CF (SACF, *Single Association Control Function*) und mehreren ASEs (ACSE, TPASE, CCRSE, ...). Zur Koordinierung mehrerer SAOs enthält eine AE eine übergeordnete CF (MACF, *Multiple Association Control Function*).

Abbildung 4-23 zeigt eine Beispiel-TPSUI innerhalb einer AEI mit zwei SAOs, wobei die TPPM und die CPM nicht dargestellt werden.



ACSE	Association Control Service Element
AEI	Application Enity Invocation
CCRSE	Commitment, Concurrendy and Recovery Service Element
MACF	Multiple Association Control Function
SACF	Single Association Control Function
SAO	Single Association Object
TPASE	Transaction Processing Application Service Element

Abbildung 4-23: TPSUI mit zwei SAOs für zwei Dialoge

4.4.2 Das DTP-Modell

In diesem Abschnitt wird das DTP-Modell einführend beschrieben. Dieses Modell repräsentiert den wohl derzeit wichtigsten Standard für die verteilte Transaktionsverarbeitung. Eine ausführliche Beschreibung des Modells und seiner Standard-APIs findet sich in (DTP 1991).

Überblick

Das DTP-Modell (Distributed Transaction Model) wurde für die Entwicklung lokaler und verteilter Transaktionsanwendungen konzipiert und wird heute vorwiegend zur Integration heterogener Datenbanksysteme eingesetzt. Alle gängigen DBMS unterstützen das DTP-Modell und bieten entsprechende Schnittstellen an.

Um das Modell in einen Standard zu überführen, hat die X/Open[16] ihr Referenzmodell mit konkreten Schnittstellen ausgestattet. Der Standard beschreibt jedoch nicht, wie die einzelnen logischen Komponenten des Referenzmodells auf ein konkretes Transaktionsverarbeitungssystem umgesetzt werden. Das DTP-Modell beinhaltet folgende Teilkonzepte:

- *Transaktionsmodelle*: Das DTP-Modell unterstützt nur das Transaktionsmodell der *Flat Transactions*, keine *Nested Transactions*. Weiterhin kann der Transaktionsinitiator vor dem Beginn jeder Transaktion wählen, ob er das Modell der *Chained Transactions* oder aber sog. *Unchained Transactions* verwenden will.
- *Interoperabilität*: Das DTP-Modell unterstützt verteilte Transaktionen auf Basis des Protokolls OSI TP.

[16] X/Open wurde in Open Group umbenannt.

– *Zuordnung von Anwendungsprogrammen und Transaktionen*: Das DTP-Modell erlaubt jedem Thread eines Anwendungsprogramms des Transaktionsinitiators max. eine aktive Transaktion zu einem Zeitpunkt. Eine Implementation des Modells kann aber entweder ein oder mehrere aktive Transaktionen je Prozess erlauben.

Architektur und Standardschnittstellen

Die im DTP-Modell beschriebene logische Architektur eines Transaktionssystems kennt vier verschiedene Komponenten:

– Anwendungsprogramme (AP), die Transaktionen abwickeln
– Ressourcenmanager (RM) zur Bearbeitung von Objekten. Beispielsweise ist im DTP-Modell ein DBMS als Ressourcenmanager zur Verwaltung von Datenbankobjekten zu sehen.
– Transaktionsmanager (TM) zur Verwaltung der Transaktionskontexte lokaler und verteilter Transaktionen
– Spezielle Ressourcenmanager zur Kommunikation zwischen den an einer Transaktion beteiligten Knoten (Communication Resource Manager, CRM). Bei verteilten Transaktionen ist der CRM der Stellvertreter für alle entfernten RMs, die an der Transaktion beteiligt sind.

DBMS als Ressourcenmanager können an einer *globalen Transaktion* beteiligt sein, die von einem dem XA-Standard entsprechenden Transaktionsmanager koordiniert wird.

Die Implementierung der Komponenten wird im Modell nicht festgelegt. Die Kommunikation der CRMs untereinander erfolgt über das von der ISO genormte 2PC-Transaktionsprotokoll OSI TP, was auf jedem Knoten einen kompletten OSI-Stack voraussetzt.

Die Schnittstellen zwischen den Komponenten sind in Schnittstellenspezifikationen genormt.[17]:

– Die TX-Schnittstelle zwischen AP und TM
– Die XA-Schnittstelle zwischen TM und RM als Herzstück des DTP-Modells
– Die XA+-Schnittstelle zwischen TM und CRM als Erweiterung der XA-Schnittstelle
– Die XAP-TP-Schnittstelle zwischen CRM und OSI TP
– Drei Schnittstellen zur Nutzung verschiedener Kommunikationsparadigmen zwischen AP und CRM, und zwar für Remote Procedure Calls (TxRPC), Peer-to-Peer-Kommunikation (CPI-C) und für die Client-Server-Kommunikation (XATMI-Schnittstelle)

[17] Die Schnittstellen sind in (DTP 1991, DTP 1992, DTP 1993a, DTP 1993b, DTP 1993c, DTP 1993d, DTP 1993e, DTP 1993f) beschrieben.

Ein AP, ein TM und ein oder mehrere RMs werden zu einer *Instanz* (*Instance)* zusammengefasst. Eine verteilte Anwendung besteht aus zwei oder mehreren Instanzen, wobei jeder Instanz noch ein CRM hinzugefügt wird. Eine Instanz ist somit eine konkrete Implementierung des Modells und enthält ein konkretes AP, einen konkreten TM und einen oder mehrere konkrete RMs (siehe Abbildung 4-24).

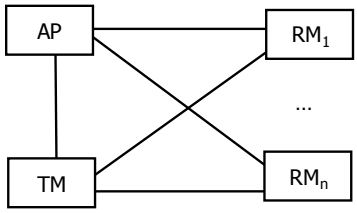


Abbildung 4-24: Instanz im DTP-Modell

Mehrere Instanzen, die den gleichen TM verwenden, werden zu einer *TM-Domain* (*Transaktionsdomäne*) zusammengefasst. Es wird sowohl die Weitergabe von Transaktionsgrenzen an fremde *Transaktionsdomänen* als auch die Übernahme von Transaktionsgrenzen aus fremden Transaktionsdomänen unterstützt.

Der Zugriff auf die RMs erfolgt über die von diesen zur Verfügung gestellten APIs, zum Beispiel über SQL für den Zugriff auf relationale Datenbanken. X/Open DTP unterstützt für jeden RM-Typ eine beliebige Anzahl von Instanzen. Der TM weist jeder Instanz eines RMs einen eindeutigen Identifikator zu und gibt mit jeden Aufruf an der XA-Schnittstelle diesen im Parameter *rmid* mit.

Die Schnittstellenspezifikationen beschreiben die Funktionsaufrufe meist in der Sprache C oder COBOL und geben detaillierte Implementierungshinweise. Auch die Einbringung eigener CRMs zur Unterstützung spezieller Kommunikationsprotokolle widerspricht nicht dem DTP-Modell.

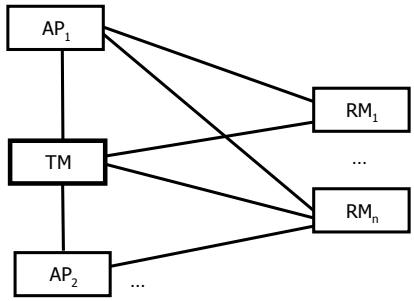


Abbildung 4-25: Transaktionsdomäne im DTP-Modell

Für das Verständnis des Zusammenspiels der Komponenten ist der Begriff des *Thread-of-Control* (kurz: Thread) von besonderer Bedeutung. Bei der Abwicklung einer Transaktion müssen AP, TM, CRMs und RMs das gleiche Verständnis über den Transaktionskontext haben. Dieses gemeinsame Verständnis wird auch logisch als *Thread-of-Control* bezeichnet. Über die Implementierung eines *Thread-of-Control* macht das DTP-Modell keine Aussage. Am einfachsten ist natürlich eine Implementierung in einem Betriebssystemprozess.

Weiterhin ist der Begriff des *Transaction Branches* für das Verständnis wichtig. Ein Transaction Branch kann innerhalb einer globalen Transaktion erzeugt werden und ist Teil einer globalen Transaktion. Er hat einen Transaction Branch Identifier, der wiederum die Identifikation für die übergreifende globale Transaktion beinhaltet. In Abbildung 4-26 ist eine globale Transaktion mit drei Transaction Branches aus Sicht der Anwendng AP_1 dargestellt.

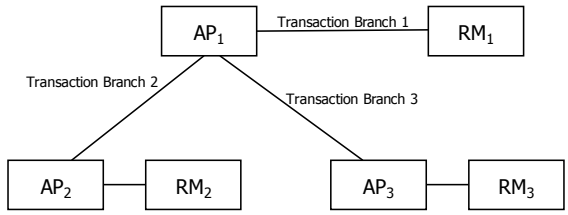


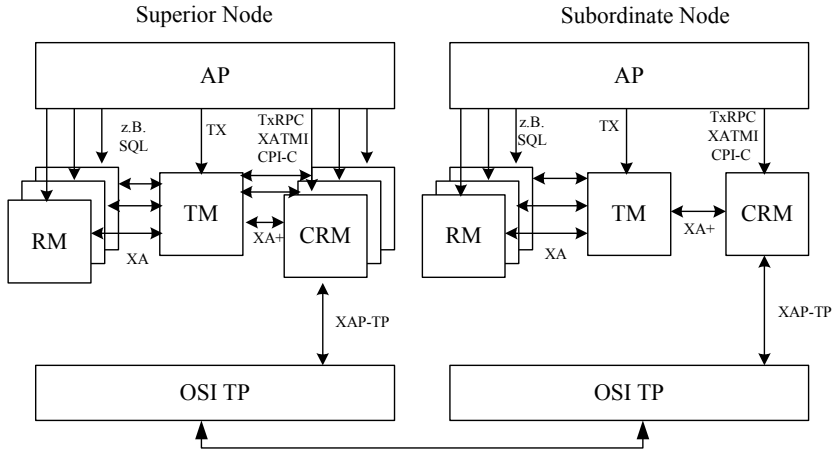
Abbildung 4-26: Beispiel eines Transaction Branches im DTP-Modell

Abbildung 4-27 zeigt die Komponenten und Schnittstellen des DTP-Modells im Überblick, wobei die Pfeile uni- bzw. bidirektionale Dienstschnittstellen darstellen. Im Bild sind zwei an einer Transaktion beteiligte Knoten ersichtlich. Der Knoten, in dem die Transaktion initiiert wird (Transaktionsinitiator), wird auch *Superior Node* genannt, der an der Transaktion teilnehmende Knoten dagegen *Subordinate Node*.

Werden verteilte RMs in der Transaktion verwendet, so müssen auch CRMs und entfernte APs involviert werden. Der CRM arbeitet als lokaler Stellvertreter und beauftragt die entfernten TMs mit der Commit-Bearbeitung für ihre Knoten. Der lokale TM führt die Ergebnisse der lokalen RMs und der entfernten TMs zusammen und entscheidet über den Ausgang der Transaktion. Ein entfernter RM kann nicht direkt, sondern nur über ein auf dem Knoten liegendes AP verwendet werden.

Im DTP-Modell sind keine Standarddienste für Logging, Concurrency Control und Recovery beschrieben. Wenn ein DBMS als RM fungiert, stellt das kein großes Problem dar, da diese Dienste ohnehin im DBMS integriert sind. Wird aber ein RM neu entwickelt, müssen auch diese Dienste mit implementiert werden. Ein RM

muss selbst für die Nebenläufigkeitskontrolle beim Zugriff auf seine Objekte sorgen, sofern er konkurrierende Zugriffe erlaubt. Weiterhin muss er die persistente Speicherung der Objekte selbst organisieren und er ist für das Logging und das Recovery verantwortlich. Eine spezielle Logging- und Recovery-Strategie wird im DTP-Modell nicht bevorzugt, sie muss aber das 2PC-Protokoll ausreichend unterstützen.



AP	Application Program
CPI-C	Common Programming Interface for Communication
CRM	Communication Resource Manager
OSI TP	Open Systems Interconnection Transaction Processing
RM	Resource Manager
SQL	Structured Query Language
TM	Transaction Manager
TxRPC	Transactional Remote Procedure Call
TX	Interface zwischen AP und TM
XA	Interface zwischen RM und TM
XA+	Interface zwischen CRM und TM
XAP-TP	X/Open Transaction Processing API
XATMI	Application/Transaction Management Interface

Abbildung 4-27: Komponenten und Schnittstellen des DTP-Modells

Eine Besonderheit des 2PC-Protokolls ist die Möglichkeit eines RM, selbstständig eine heuristische Entscheidung über den Ausgang einer Transaktion zu treffen. Dies ist zum Beispiel dann sinnvoll, wenn die Kommunikation mit dem TM für längere Zeit unterbrochen wird. In diesem Fall kann der RM selbstständig eine Transaktion beenden, ohne das Ergebnis zu kennen. Die Entscheidung muss aber aufbewahrt und nach Wiederaufnahme der Verbindung mit dem TM abgestimmt werden.

Die TX-Schnittstelle

Die TX-Schnittstelle wird einem AP zur Verfügung gestellt, um Transaktionen abzuwickeln. Alle Aufrufe an der TX-Schnittstelle eines AP werden blockierend ausgeführt, das AP wartet also auf das Ergebnis. Die Operationen der TX-Schnittstelle werden in der Regel über eine Bibliothek bereitgestellt. Die TX-Schnittstelle wurde in C definiert. Wie man an dem folgenden Codeausschnitt erkennen kann, enthält TX im Wesentlichen Operationen, die es einem AP ermöglichen, Transaktionsklammern zu setzen.

```
…
struct tx_info_t {
    XID xid;
    COMMIT_RETURN when_return;
    TRANSACTION_CONTROL transaction_control;
    TRANSACTION_TIMEOUT transaction_timeout;
    TRANSACTION_STATE   transaction_state;
};

typedef struct tx_info_t TXINFO;
extern int tx_begin(void);
extern int tx_close(void);
extern int tx_commit(void);
extern int tx_info(TXINFO *);
extern int tx_open(void);
extern int tx_rollback(void);
extern int tx_set_commit_return (COMMIT_RETURN);
extern int tx_set_transaction_control (TRANSACTION_CONTROL);
extern int tx_set_transaction_timeout (TRANSACTION_TIMEOUT);
…
```

Für jede Transaktion werden über den Datentypen *TXINFO* Informationen wie eine globale Identifikation (*XID*) und der Transaktionsstatus verwaltet. Diese Informationen können über die Funktion *tx_info* abgefragt werden. Ein AP kann auch über die Funktion *tx_set_tranasction_timeout* eine Zeitbegrenzung für eine Transaktion vorgeben.

Die XA-Schnittstelle

Da das XA-Interface im DTP-Modell eine besonderere Bedeutung hat, soll es hier kurz erläutert werden. Weitere Informationen zu den anderen Schnittstellen (TxRPC, XATMI, CPI-C, XA+, XAP-TP) sind der angegebenen Literatur zu entnehmen.

Ein RM stellt gemäß XA-Schnittstellenspezifikation eine externe Variable des Typs *xa_switch_t* als Schnittstelle für den TM zur Transaktionskoordination bereit. Die

Struktur sieht vereinfacht wie folgt aus (C-Notation) und enthält im Wesentlichen Zeiger auf die Funktionen der XA-Schnittstellen:

```
struct xa_switch_t {
  …
  int (*xa_open_entry)(char *, int, long);
  int (*xa_close_entry)(char *, int, long);
  int (*xa_start_entry)(XID *, int, long);
  int (*xa_end_entry)(XID *, int, long);
  int (*xa_rollback_entry)(XID *, int, long);
  int (*xa_prepare_entry)(XID *, int, long);
  int (*xa_commit_entry)(XID *, int, long);
  int (*xa_recover_entry)(XID *, long, int, long);
  int (*xa_forget_entry)(XID *, int, long);
  int (*xa_complete_entry)(int *, int *, int, long);
};
```

Die definierten Funktionen des XA-Interface werden typischerweise blockierend aufgerufen, es gibt aber auch die Möglichkeit der nicht-blockierenden (asynchronen) Ausführung. Ein Transaktionsmanager nutzt die XA-Routinen der RMs bei Bedarf nach vorgegebenen Regeln:

- Die Routine *xa_open* wird verwendet, um eine Verbindung zum RM aufzubauen und *xa_close* wird zum Abbau einer Verbindung aufgerufen. Die Verbindung bezieht sich dabei auf den Thread. Nach dem Start fordert z.B. ein TP-Monitor den Transaktionsmanager auf, alle Ressourcenmanager (RMs) zu öffnen, die in einer Gruppe von Anwendungsservern definiert sind. Der Transaktionsmanager übergibt die *xa_open*-Aufrufe an die Ressourcenmanager, damit diese für die DTP-Verarbeitung initialisiert werden können. Im Rahmen dieser Startprozedur führt der Transaktionsmanager eine Resynchronisation (resync) durch, um alle unbestätigten Transaktionen aufzulösen.
- Die Routinen *xa_start* und *xa_end* werden benutzt, um einem RM mitzuteilen, dass ein Thread seine Arbeit für eine Transaktion, gemäß DTP-Modell für einen sog. *Transaction Branch* (Transaktionsverzweigung) als Teil einer globalen Transaktion, beginnt bzw. beendet. Ein *Transaction Branch* wird innerhalb eines *Thread-of-Control* ausgeführt, den alle Beteiligten kennen müssen. Ein *Transaction Branch* (wir bleiben im Folgenden vereinfachend bei dem Begriff *Transaktion*) wird durch eine global eindeutige Identifikation, die sog. *XID* gekennzeichnet.
- Die Routinen *xa_prepare*, *xa_rollback* und *xa_commit* werden für die Transaktionskoordinierung (2PC) benötigt. Der Transaktionsmanager kann eine einphasige (1PC-Protokoll) anstelle der zwei-phasigen Festschreibung durchführen, wenn nur ein Ressourcenmanager beteiligt ist oder wenn ein Ressourcenmanager antwortet, dass seine Transaktionsverzweigung im reinen Lesemodus arbeitet.

- Die Routine *xa_recover* ermöglicht es einem TM, während der Initialisierungsphase vom RM eine Liste der Transaktionen zu erhalten, die von diesem heuristisch beendet wurden.
- Durch *xa_forget* kann der RM veranlasst werden, diese Transaktionen zu vergessen, d.h. er muss sich keine Informationen mehr für diese aufbewahren.
- Zur Durchführung der Resynchronisation (im Recovery-Fall) setzt der Transaktionsmanager einen Aufruf *xa_recover* an jeden Ressourcenmanager mindestens einmal ab, um unbestätigte Transaktionen zu identifizieren. Der Transaktionsmanager vergleicht die Antworten mit den Informationen im eigenen Protokoll, um zu bestimmen, ob die Ressourcenmanager angewiesen werden sollen, eine *xa_commit*-Operation oder eine *xa_rollback*-Operation für diese Transaktionen durchzuführen. Wenn ein Ressourcenmanager seine Entscheidung für eine unbestätigte Transaktion aufgrund einer heuristischen Operation durch den Administrator bereits festgeschrieben oder rückgängig gemacht hat, setzt der Transaktionsmanager einen Aufruf *xa_forget* an diesen Ressourcenmanager ab, um die Resynchronisation abzuschließen.
- Die Routine *xa_complete* dient schließlich dem Zweck, asynchrone Aufrufe der oben angegebenen Dienste zu beenden.

Dem RM stehen seinerseits zwei Routinen zur Verfügung, die im Transaktionsmanager implementiert und zugänglich gemacht werden müssen:

- Die Routine *ax_reg* dient zum dynamischen Registrieren einer Transaktion beim TM. Sie wird benutzt, wenn ein RM dynamisch bekanntgibt, dass ein Thread einer Anwendung eine Operation an ihn geschickt hat. Beim Aufruf wird die globale Transaktions-Id übergeben. Der Transaktionsmanager teilt daraufhin dem RM im Returnwert mit, ob eine Beteiligung des RM an dieser Transaktion über den TM koordiniert wird oder nicht. Wird eine XID übergeben, so muss der RM an der koordinierten Transaktion teilnehmen. Bei jeder neuen Transaktion muss sich der RM erneut mit ax_reg registrieren. Im anderen Fall, also wenn der TM gerade keine globale Transaktion kennt (NULLXID wird zurückgegeben), führt der RM die Operationen außerhalb einer vom TM koordinierten Transaktion aus.
- Die Routine *ax_unreg* dient dazu, einem TM mitzuteilen, dass Aktionen eines RM, die sich außerhalb einer bekannten Transaktion befanden, abgeschlossen sind. Erst wenn der RM nach der Teilnahme an einer „externen“ Transaktion die Routine *ax_unreg* aufruft, kann ihn der TM wieder in eine Transaktion mit einbeziehen. Bis dahin lässt er das involvierte AP keine neue globale Transaktion starten und es darf auch kein anderer RM an der laufenden globalen Transaktion teilnehmen. Dies könnte zu Inkonsistenzen führen.

Abbildung 4-28 zeigt die Einbettung der XA-Routinen in den Transaktionsmanager und in einen Ressourcenmanager.

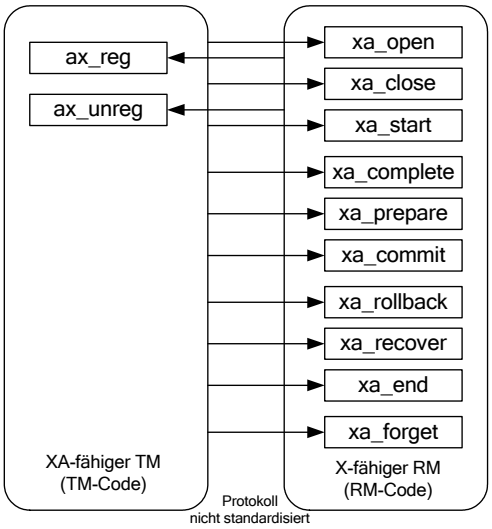


Abbildung 4-28: Zusammenspiel zwischen TM und RM über XA

In der Regel involviert ein TM alle über *xa_open* angemeldeten RMs in eine Transaktion. ax-Aufrufe durch den RM sind dann sinnvoll, wenn ein RM nur selten benutzt wird. Durch Nutzung dieser Funktion kann nämlich vermieden werden, dass ein RM in eine Transaktion verwickelt wird, für die er gar nichts tun muss. Die Festlegung der Nutzung von *ax_reg* und *ax_unreg* wird in einer Variablen des Typs *xa_switch_t* im Attribut *flags* getroffen. Die dynamische Registrierung ist flexibler für den RM und wird von einigen DBMS genutzt (z.B. von IBM DB2). Ein RM registriert sich beim TM nur dann mit Hilfe eines *ax_reg*-Aufrufs, wenn der RM eine Anforderung für seine Ressource empfängt.

Für die statische Registrierung ist es erforderlich, dass der Transaktionsmanager (für jede Transaktion = Transaction Branch) die Folge von Aufrufen *xa_start*, xa_begin,..., *xa_end* an alle registrierten Ressourcenmanager absetzt. Dies gilt unabhängig davon, ob der einzelne Ressourcenmanager innerhalb der Transaktion verwendet wird oder nicht. Dieses Vorgehen ist nicht sehr effizient, wenn nicht ohnehin jeder Ressourcenmanager an jeder Transaktion beteiligt ist. Der Grad der Ineffizienz ist proportional zur Anzahl der definierten Ressourcenmanager.

Die Bereitstellung einer Bibliothek durch jeden Hersteller eines RM erlaubt es, dass das Transaktionsprotokoll, über das TM und RM kommunizieren, vom RM-Hersteller entwickelt wird und keinem Standard entsprechen muss.

Bei der Programmierung einer Anwendung (AP) muss beachtet werden, dass der Verbindungsaufbau und das Setzen der Transaktionsgrenzen implizit erfolgt. Es dürfen daher keine RM-Dienste für diese Zwecke (z.B. *dbconnect* und *begin* bzw. *commit work*) verwendet werden. Für den Zugriff auf die Daten eines RM benutzt die Anwendung ansonsten die üblichen Schnittstellen, die dieser bereitstellt (z.B. Embedded SQL oder Oracle Call Interface).

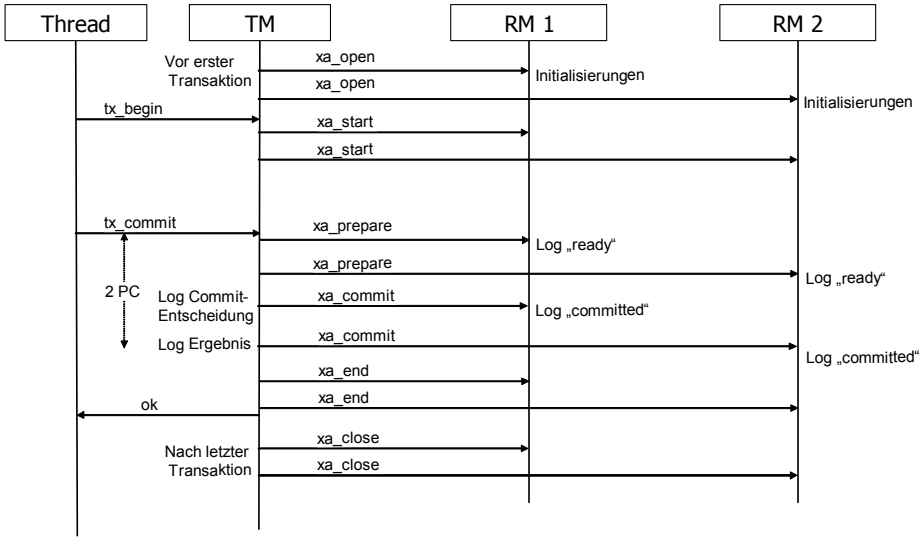


Abbildung 4-29: Kommunikation zwischen TM und RM (statisches Registrieren)

Werden von einem Objektserver mehrere RMs benutzt, dürfen die XA-Libraries der verschiedenen Hersteller natürlich nicht miteinander kollidieren, d.h. es dürfen z.B. Symbole innerhalb der verschiedenen Bibliotheken nicht doppelt vergeben sein. Einheitliche Regelungen zur Vergabe externer Namen für Variable wurden in der Spezifikation nicht eingeführt und sind daher den Implementierern überlassen.

In Abbildung 4-29 ist der Ablauf der Kommunikation zwischen TM und RM bei statischer Registrierung dargestellt. Der TM initialisiert die RMs mit dem Aufruf *xa_open* und beendet die Bearbeitung mit *xa_close*. Dazwischen können beliebig viele Transaktionen ausgeführt werden. Eine (globale) Transaktion, an der ein oder mehrere lokale RMs beteiligt sind, wird im Prinzip wie folgt abgewickelt:

– Ein AP setzt einen Funktionsaufruf *tx_begin* an den TM ab, der daraufhin eine globale Transaktions-Id (XID) generiert und alle dem AP hinzugebundenen RMs mit Hilfe der XA-Schnittstelle (*xa_start*) über den Beginn einer Transaktion informiert, also den Transaktionskontext propagiert.

- Das AP führt nun über die spezifischen RM-Schnittstellen seine Operationen aus.
- Um die Transaktion zu beenden, ruft das AP die Funktion *tx_commit* im Erfolgsfall bzw. *tx_rollback* zum Zurücksetzen der Transaktion auf. Bei *tx_commit* wickelt der TM ein 2PC-Protokoll mit allen RMs ab und übergibt dem AP das Ergebnis.

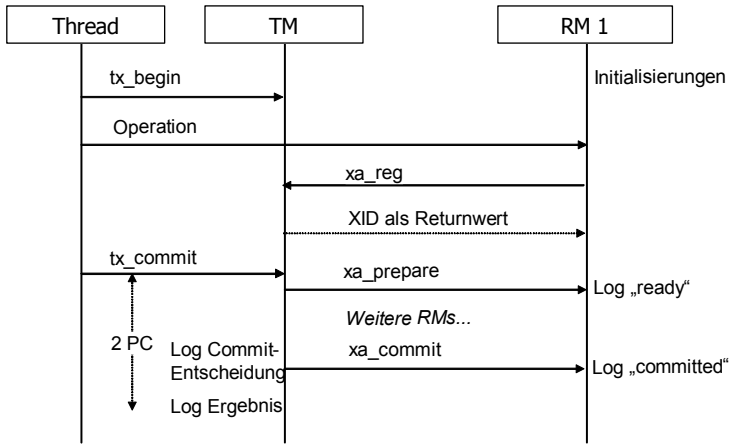


Abbildung 4-30: Dynamisches Registrieren an einer Transaktion

Der Ablauf einer dynamischen Transaktionsteilnahme unter Verwendung der Routinen *ax_reg* ist in der Abbildung 4-30 skizziert. Wie die Abbildung zeigt, wird der RM in einer Transaktion erst aktiv, wenn er von einem Anwendungsthread eine Operation empfängt. Er erkundigt sich daraufhin beim TM, ob die Transaktionskoordination über ihn erfolgt oder nicht. Im Beispiel koordiniert der TM die Transaktion, da eine gültige XID an den RM übergeben wird.

Abbildung 4-31 zeigt ein Szenario, an dem ein RM (RM 1) außerhalb einer bekannten globalen Transaktion für einen Thread-of-Control aktiv ist. Der TM sperrt diese Transaktion für andere RMs und lässt das AP an keiner weiteren Transaktion mehr teilnehmen, solange bis der betroffene RM sich mit *ax_unreg* zurückmeldet.

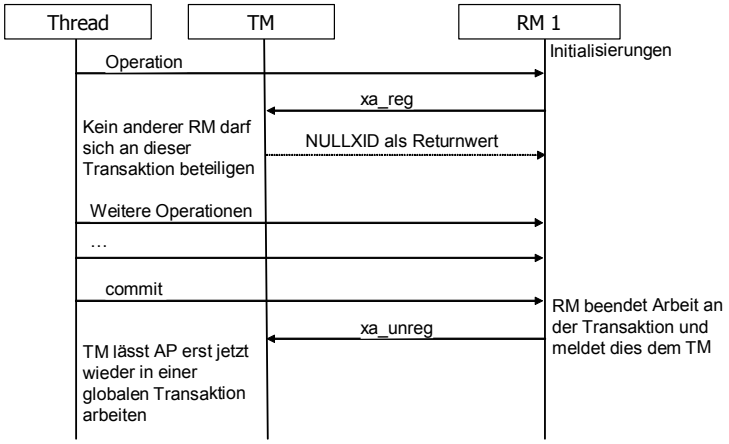


Abbildung 4-31: RM arbeitet außerhalb einer globalen Transaktion

Unterstützung verteilter Transaktionen

Da die Ausführung verteilter Transaktionen im DTP-Modell recht komplex ist und mehrere Komponenten involviert sind, soll sie in diesem Abschnitt nochmals kurz erläutert werden. Für die Durchführung von verteilten Transaktionen müssen die TM mehrerer Transaktionsdomänen kooperieren. Dies bedeutet, dass der übergeordnete TM die ihm untergeordneten TM und RM benutzt. Was die Transaktionssteuerung betrifft, dienen die in den verschiedenen Transaktionsdomänen beteiligten CRM im Wesentlichen der Weiterleitung der Aufrufe der übergeordneten TM an die untergeordneten RM.

Der untergeordnete TM steuert die in seiner Transaktionsdomäne gelegenen Komponenten entsprechend dem Fortschritt des 2PC-Protokolls der übergeordneten Transaktion (Greulich 1997). *Prepare-* und *Commit-*Aufrufe durch den übergeordneten TM werden an den untergeordneten TM weitergeleitet. Auf diese Weise pflanzt sich der jeweilige Aufruf vom TM, mit dem die verteilte Transaktion begonnen wurde (= Wurzel des Abhängigkeitsbaums in Abbildung 4-32 links) bis zu den verschiedenen RM (= Blätter des Abhängigkeitsbaums) fort. Nach Empfang der Antwort der adressierten RMs wird das Transaktionsergebnis dem übergeordneten TM zurückgereicht. Auf diese Weise wird in jedem untergeordneten TM eine Teilentscheidung und vom initiierenden TM zuletzt die Gesamtentscheidung über den Transaktionsausgang gefällt. Das Gesamtergebnis erhält schließlich der Transaktionsinitiator. Falls bei einem RM eine heuristische Entscheidung getroffen wurde, muss noch ein *Forget-*Aufruf an alle beteiligten RM propagiert werden.

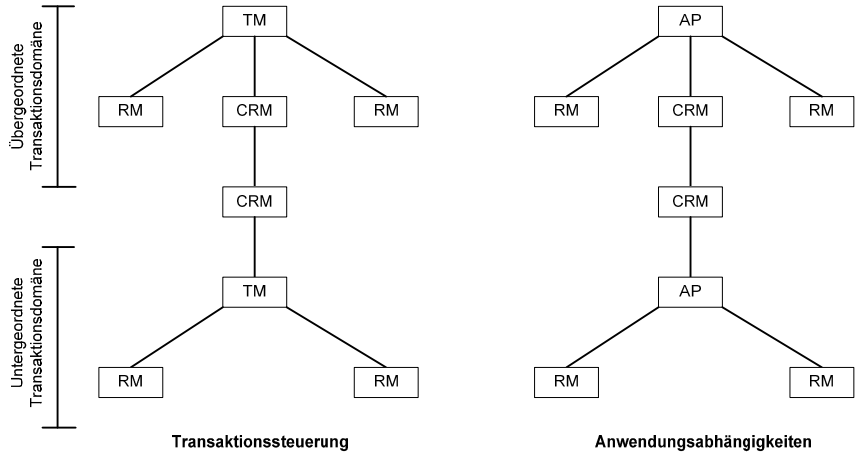


Abbildung 4-32: Globale Transaktion im DTP-Modell nach (Greulich 1997)

Im Falle eines Systemausfalls während der Koordination sorgt der TM für die Fortsetzung der Verarbeitung. Nachdem sich alle vom Systemausfall betroffenen Komponenten selbstständig wiederhergestellt haben, erkundigt sich der TM bei allen ihm bekannten RM/CRM nach nicht abgeschlossenen Transaktionen. Dies tut er, indem er mit Hilfe der Funktion *xa_recover(...)* eine Liste aller Transaktionen bzw. XIDs der betroffenen Transaktionen anfordert. Im Anschluss daran steuert er mit Hilfe des 2PC-Protokolls den Ausgang dieser Transaktionen.

Da dieser Vorgang bei tiefen Transaktionsbäumen relativ komplex werden kann, sind in der Praxis meist nur sehr flache Hierarchien vorzufinden. Der überwiegende Einsatz des DTP-Modells ist die Anbindung eines DBMS an einen Transaktionsmonitor oder Transaktionsmanager. Ein CRM wird in der Praxis nur selten eingesetzt.

4.4.3 OMG/CORBA-Transaktionsmodell

Überblick

Der Object Transaction Service (OTS) ist einer der im Kontext der OMA spezifizierten CORBA-Services. Er bietet CORBA-Objekten gemäß OMA die Möglichkeit, auf festgelegte Weise verteilte Transaktionen durchzuführen.

Die OMG war sich bewusst, dass OTS darauf angewiesen ist, mit anderen Transaktionsverarbeitungssystemen zusammenzuarbeiten. Deswegen hat sie die Architektur von OTS verträglich mit dem DTP-Modell gemacht. OTS liegen daher im Kern die gleichen Konzepte zugrunde.

Die OMG hat den Herstellern für die Implementierung der Funktionalität von OTS gewisse Freiräume gelassen. Manche Teile der Funktionalität sind zwingend vorgeschrieben und manche optional. Einige Beispiele hierzu sind:

- Die Spezifikation von OTS schreibt für jede Implementierung zwingend ein Verfahren vor, das es erlaubt, verteilte Transaktionen unter Beteiligung mehrerer OTS-*Transaktionsdomänen*[18] durchzuführen. Eine *Transaktionsdomäne* von OTS kann sich über mehrere Systeme erstrecken. Deswegen können und dürfen die verwendeten Verfahren keinerlei Annahmen über die Verteilung der beteiligten Objekte, wie beispielsweise über gemeinsame Adressräume, machen.
- Die Spezifikation sieht für OTS sowohl das Transaktionsmodell der *Flat Transactions* als auch das der *Nested Transactions* vor. Die Unterstützung von *Nested Transactions* ist jedoch optional.
- OTS erlaubt die flexible Kontrolle darüber, ob ein aufgerufenes Objekt an einer Transaktion beteiligt werden soll oder nicht. Jede Implementierung muss die beiden in der Spezifikation beschriebenen Verfahren zur Kontrolle und Propagierung von Transaktionen unterstützen. Dies sind die *direkte* und die *indirekte Kontextpropagierung*[19].
- Die Spezifikation von OTS erlaubt Implementierungen mit einem *Ausführungspfad*[20], aber auch mit optional mehreren *Ausführungspfaden* je Objekt der transaktionalen Anwendung.

OTS unterstützt also flache und geschachtelte Transaktionen, wobei letztere für eine Implementierung optional sind. Für eine Transaktion gelten die ACID-Eigenschaften in vollem Umfang. Das Commit-Protokoll ist ein 2PC-Protokoll, das heuristische Entscheidungen von Teilnehmern unterstützt. Nur die Dienstschnittstelle des 2PC-Protokolls, nicht die Implementierung derselben ist im Standard vorgegeben. Ein 2PC-Protokoll wurde durch die OMG noch nicht formal spezifiziert. Dies bedeutet, dass die OTS-Implementierungen verschiedener Hersteller derzeit nicht unbedingt miteinander kommunizieren können.

Architektur des Transaction Service (OTS)

Die Spezifikation der Schnittstellen von OTS besteht aus zwei Teilen bzw. IDL-Modulen: Im Modul *CosTransactions* sind die für transaktionalen Anwendungen wichtigen Datentypen, Exceptions und Schnittstellen festgelegt. Im Modul *CosTS-*

---

[18] Der Begriff entspricht der Verwendung im DTP-Modell.

[19] Direkte und indirekte Kontextpropagierung deutet grundsätzlich an, ob der Kontext direkt durch das Anwendungsprogramm oder indirekt durch OTS propagiert wird. Die Kontextpropagierungsvarianten werden noch weiter unten in diesem Abschnitt erörtert.

[20] Der Begriff des Ausführungspfades ist im Wesentlichen ein Synonym für den Begriff „Thread-of-Control“.

Portability werden die für die Kooperation von OTS und ORB notwendigen Schnittstellen beschrieben. Zu den im Modul *CosTransactions* definierten Datentypen gehören u.a:

- Datentypen, die der Kooperation mit der transaktionalen Anwendung dienen.
- Datentypen, die der Unterstützung der Interoperabilität mit anderen OTS-Implementationen dienen. Diese Gruppe umfasst die Definition der Datentypen *otid_t*, *TransIdentity* und *PropagationContext*.

Exceptions werden bei OTS dazu verwendet, Fehler zu melden. Die von OTS definierten Exceptions können in drei Gruppen eingeteilt werden: Standard-Exceptions, heuristische Exceptions und sonstige Exceptions.

Der Transaction Service kennt folgende Komponenten:

- Transactional Client (TC)
- Transactional Objects (TO)
- Recoverable Objects (RO)
- Transactional Servers
- Recoverable Servers

Der TC ist eine Client-Anwendung, die innerhalb von Transaktionen Methoden von TOs beziehungsweise ROs aufruft. Ein Transactional Server verwaltet ein oder mehrere TOs, wobei ein TO einfach ein Objekt ist, das an einer Transaktion beteiligt ist. TOs verwenden ihrerseits ROs mit ACID-Eigenschaften, die von Recoverable Servern verwaltet werden. Eine Transaktion kann viele Methoden beliebiger TOs und ROs aufrufen. Die Commit-Bearbeitung führt der Transaction Service in der Regel nach Aufforderung durch.

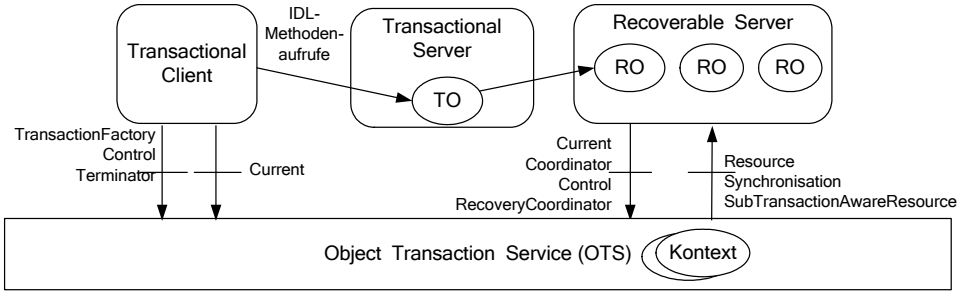


Abbildung 4-33: Komponenten des CORBA Transaction Service

Abbildung 4-33 zeigt die typische Anwendung des Transaction Service mit einem TC, der innerhalb von Transaktionen mit einem Recoverable Server kommuniziert. Der Transaktionskontext wird implizit von OTS mit den Application Requests an den Recoverable Server übertragen. Recoverable und Transactional Servers sind

nur abstrakte Komponenten und können in konkreten Implementierungen von Transaktionsanwendungen auch zusammenfallen. Der Transaktionskontext ist an den Thread-of-Control gebunden, der sie initiiert. OTS bietet die Möglichkeit, den Transaktionskontext implizit an die beteiligten Objekte weiterzureichen. Ein Objekt kann sich aber den Kontext einer Transaktion auch explizit besorgen und die Koordinatorrolle übernehmen.

OTS bietet seine Dienste als Methoden über definierte, in IDL spezifizierte Interfaces an, von denen die wichtigsten *TransactionFactory*, *Current*, *Resource* und *Coordinator* sind. Die im OTS unterstützten Schnittstellen sind alle in IDL spezifiziert und sollen nachfolgend nur kurz erläutert werden (Greulich 1997):

***Die* Current-Schnittstelle:** Das Current-Interface (auch als Current-Pseudo-Object bezeichnet) ist einfacher zu bedienen als das weiter unten erwähnte *TransactionFactory*-Interface und setzt auf diesem auf. Es erlaubt TCs, sich explizit als Koordinator an Transaktionen zu beteiligen und stellt dazu die notwendigen Dienstprimitive bereit (*begin, commit, rollback*). Die Vereinfachung besteht darin, dass die Anwendung bei Verwendung der Current-Schnittstelle vom direkten Kontextmanagement entlastet wird. Dies wird dadurch erreicht, dass die Implementierung eine Assoziation zwischen dem aufrufenden Ausführungspfad und einer Subtransaktion bzw. Transaktion herstellt. Alle Aufrufe von Operationen der *Current*-Schnittstelle beziehen sich implizit auf die aktuell mit dem Ausführungspfad assoziierte Transaktion. Damit ein Ausführungspfad die Beteiligung von Objekten an Transaktionen flexibel kontrollieren kann, unterstützt die *Current*-Schnittstelle auch Operationen, die es erlauben, die Zuordnung zwischen Ausführungspfad und Transaktion zu kontrollieren. Beispielsweise kann ein Ausführungspfad die Assoziation mit seiner Subtransaktion bzw. Transaktion zeitweise suspendieren.

Die Control-Schnittstelle: Die Control-Schnittstelle wird von Objekten unterstützt, die eine bestimmte Subtransaktion repräsentieren. Entsprechend bieten die von der Control-Schnittstelle bereitgestellten Operationen Zugriff auf die Objekte, welche die zugehörige Transaktion steuern (*Coordinator*- und *Terminator*-Objekt).

Die TransactionFactory-Schnittstelle: Die *TransactionFactory*-Schnittstelle stellt folgende Operationen bereit:

- Beginnen von Top-Level-Transaktion: Die dafür bereitgestellte Operation *create* wird von Transactional Clients dazu benutzt, Top-Level-Transaktionen zu erzeugen.
- Erzeugen von Repräsentationen von bereits begonnenen Transaktionen: Die für diese Aufgabe bereitgestellte Operation *recreate* dient dazu, die innerhalb der Transaktionsdomäne bereits existierende Repräsentation, eventuell auch einer importierten Transaktion, zu erfragen.

Beide Operationen liefern als Ergebnis ein *Control*-Objekt als Repräsentation für die entsprechende Transaktion.

Die Terminator-Schnittstelle*:* Diese Schnittstelle dient dazu, eine bestimmte Transaktion zu beenden oder abzubrechen. Deswegen wird jedes Objekt, das die *Terminator*-Schnittstelle unterstützt, implizit mit einer bestimmten Transaktion bzw. Subtransaktion assoziiert. Die Terminator-Schnittstelle wird zwar typischerweise von Transactional Clients verwendet, kann aber prinzipiell von jedem an der Transaktion beteiligten Objekt benutzt werden.

Die *Coordinator*-Schnittstelle: Über diese Schnittstelle registriert ein *Recoverable Server* die ROs als Ressourcen innerhalb einer Transaktion.

Die Coordinator-Schnittstelle: Diese Schnittstelle definiert die Operationen, die von einem *Commit*-Koordinator einer Transaktion unterstützt werden müssen und stellt folgende Operationen bereit:

- Operationen zum Erfragen des Zustands der Transaktion und ihrer Beziehung zu anderen Transaktionen
- Operationen für die dynamische Registrierung
- Eine Operation zur Registrierung von *Synchronization*-Objekten
- Operationen zum Erzeugen von untergeordneten Transaktionen
- Eine Operation zur Einflussnahme auf den Zustand der Transaktion

Die *Ressource*-Schnittstelle: Diese Schnittstelle wird von einem RO verwendet, um das Commit-Protokoll mit dem Transaction Service abzuwickeln. Es stellt unter anderem die Operationen *prepare*, *rollback* und *commit* bereit, die vom Transaction Service aufgerufen werden. Da jedes Objekt, das direkt daran teilnehmen will, eine eigene *Ressource*-Schnittstelle bereitstellen muss, kann ein *Ressource-Objekt* als Repräsentant von Daten betrachtet werden, die als Teil der entsprechenden Transaktion manipuliert werden.

Die RecoveryCoordinator-Schnittstelle*:* Jedes Objekt mit *RecoveryCoordinator*-Schnittstelle symbolisiert den Zustand eines bestimmten, an einer Subtransaktion beteiligten *Ressource*-Objekts. Die *RecoveryCoordinator*-Schnittstelle stellt nur die Operation *replay_completion* bereit. Sie dient dazu, in besonderen Recovery-Fällen die erneute Steuerung des Transaktionsausgangs an das mit der Operation mitgelieferte Ressource-Objekt anzufordern.

Die Synchronization-Schnittstelle: OTS unterstützt ein Verfahren, welches Transactional Objects erlaubt, unmittelbar vor und nach einem Transaktionsabschluss informiert zu werden. Ein Objekt, das dieses Protokoll-Feature nutzen will, muss die Synchronization-Schnittstelle bereitstellen. Sie umfasst nur zwei Operationen, die von OTS zwecks der Benachrichtigung unmittelbar vor und nach dem Abschluss oder Abbruch der Transaktion aufgerufen werden.

Die SubTransactionAwareRessource-Schnittstelle: Recoverable Objects, die geschachtelte Transaktionen unterstützen, können sich vom Abschluss von Subtransaktionen informieren lassen. Wenn ein Objekt dies möchte, dann kann es statt der *Ressource*- die davon abgeleitete *SubTransactionAwareResource*-Schnittstelle bereitstellen. Sie umfasst, neben den Operationen der *Ressource*-Schnittstelle, zwei

Operationen, die von OTS zwecks der Benachrichtigung vom Abschluss von Subtransaktionen aufgerufen werden.

Die TransactionalObject-Schnittstelle: Diese Schnittstelle stellt keinerlei Operationen bereit. Ein Transactional Object, dessen Schnittstelle von der Schnittstelle *TransactionalObject* abgeleitet ist, zeigt dem ORB an, dass er die Transaktion, die mit dem aufrufenden Ausführungspfad assoziiert ist, behandeln muss.

Im Folgenden werden die vereinfachten IDL-Spezifikationen der Interfaces *Current*, *Resource* und *Coordinator* dargestellt (OMG 2001):

```
interface Current {
         void begin();
         void commit();
         void rollback();
         void rollback_only();
         Status get_status();
         string get_transaction_name();
         void set_timeout(in unsigned long seconds);
         Control get_control();
         Control suspend();
         void resume(in Control which);
};
interface Resource {
         Vote prepare();
         void rollback();
         void commit();
         void commit_one_phase();
         void forget();
};
interface Coordinator {
         Status get_status();
         ...
         RecoveryCoordinator register_resource (in Resource r);
         ...
         void rollback_only();
         ...
};
```

Ablauf einer OTS-Transaktion

Eine typische OTS-Transaktion wird üblicherweise wie folgt abgewickelt:

– Ein TC, der auch als *Originator* bezeichnet wird, initiiert eine Transaktion über ein Objekt vom Typ *Current*, indem er die Methode *begin* aufruft.

- Anschließend setzt der TC beliebige Requests an TOs beziehungsweise direkt an ROs ab, wobei der Transaktionskontext implizit weitergereicht (propagiert) wird. Die ROs besorgen sich eine Referenz auf das Coordinator-Objekt mit der Methode *Current::get_control* und registrieren sich über die Methode *Coordinator::register_resource* beim Transaction Service. ROs müssen ihre Zustandsänderungen während der Transaktion selbst protokollieren und sind auch für die Synchronisierung paralleler Zugriffe verantwortlich.
- Nach Ausführung aller Requests ruft der TC die Methode *commit* des Current-Interface auf. Daraufhin wickelt der Transaction Service das Commit-Protokoll mit allen an der Transaktion registrierten ROs über das Resource-Interface ab. Ein Abbruch der Transaktion ist über die Methode *rollback* durch den TC jederzeit möglich, worauf der Transaction Service die Rollback-Methode aller registrierten ROs aufruft. Auch ein Recoverable Server kann eine Transaktion abbrechen (*rollback_only*).

Wird die Verbindung eines ROs zum Transaction Service in der unsicheren Phase (nach dem Votieren in Phase 1) unterbrochen, kann ein RO auch eine heuristische Entscheidung über den Ausgang der Transaktion treffen. Diese muss aber dauerhaft gespeichert werden, bis der Transaction Service das RO über die Operation *forget* auffordert, die Transaktion endgültig zu vergessen.

OTS-Transaktionskontext und -transfer

Das Kontextmanagement dient der Erzeugung, Löschung, der Aufbewahrung des Transaktionskontexts sowie der Zuordnung des Transaktionskontexts zu einem Aufruf eines *Transactional* Objects. Mit der Zuordnung kann der Aufrufer eines Transactional Objects die Beteiligung an der Transaktion und damit den Transaktionsinhalt kontrollieren. Die Spezifikation von OTS beschreibt entsprechend den beiden verschiedenen Formen der Repräsentation einer Transaktion zwei verschiedene Arten des Kontextmanagements und zwar das bereits genannte *direkte* und das *indirekte Kontextmanagement:*

- Beim direkten Kontextmanagement wird die Transaktion als *Control-Objekt* verwaltet. Die Verantwortung dafür liegt bei der transaktionalen Anwendung: Sie muss eigenverantwortlich für die Erzeugung, Vernichtung, Aufbewahrung und Zuordnung des *Control*-Objekts sorgen. Der Transactional Client erhält das *Control*-Objekt einer Transaktion, indem er ihre Erzeugung mit *TransactionFactory.create* anfordert.
- Beim indirekten Kontextmanagement wird die Transaktion als Datentyp *PropagationContext* verwaltet. Die Verantwortung für seine Erzeugung, Aufbewahrung, Löschung und Zuordnung liegt nicht bei der transaktionalen Anwendung, sondern bei OTS. OTS ordnet jedem Ausführungspfad eines Clients oder Objekts implizit einen Transaktionskontext zu. Die Verwaltung des Transaktionskontexts ist für die transaktionale Anwendung transparent.

Der Transactional Client beginnt eine Transaktion, indem er die Operation *Current.begin* aufruft.

Bei OTS ist der Transaktionskontext als IDL-Datentyp *PropagationContext* spezifiziert. Der Transaktionskontext wird von OTS für jede transaktionale Anwendung verwaltet und enthält als wichtigste Teile die mit der Transaktion assoziierten *Coordinator*- und *Terminator*-Objekte. Davon zu unterscheiden ist die Repräsentation der Transaktion als Control-Objekt. Da sowohl der Datentyp *PropagationContext* als auch das *Control*-Objekt die gleichen essentiellen Daten beinhalten, sind beide Formen für die Verwaltung von Transaktionen geeignet.

Man spricht bei OTS von *Kontexttransfer,* wenn *OTS* das aufgerufene *Transactional*-Object mit dem Transaktionskontext versorgt. Dies erfolgt bei OTS zur Laufzeit, da die an der Transaktion beteiligten Objekte nicht schon vor Beginn der Transaktion bekannt sind. Der Transaktionskontext oder das entsprechende Control-Objekt muss mit jedem Aufruf einer Operation eines Transactional Objects mitpropagiert werden. Zwei Arten des Transfers werden unterschieden:

- Beim *expliziten Kontexttransfer* muss der Transactional Client den Transfer des Transaktionskontexts explizit veranlassen. Ob ein Transactional Object das Verfahren des expliziten Kontexttransfers erfordert, kann an der Definition seiner Schnittstelle in IDL-Syntax abgelesen werden: Wenn die Methoden der spezifischen Schnittstelle eines Transactional Objects als letzten Eingabeparameter ein Control-Objekt erwarten, so erfordert seine Verwendung in einer Transaktion den expliziten Kontexttransfer. Das bedeutet für die transaktionale Anwendung, dass sie bei jedem Aufruf dieses Transactional Objects den letzten Parameter mit dem passenden Control-Objekt der Transaktion versorgen muss, in deren Rahmen es verwendet werden soll. Deswegen wird der explizite Kontexttransfer meist in Verbindung mit dem direkten Kontextmanagement verwendet.
- *Impliziter Kontexttransfer* bedeutet, dass sich der Transactional Client nicht um den Transfer des Transaktionskontexts zu den aufgerufenen Transactional Objects kümmern muss. Diese Aufgabe wird vom ORB übernommen, indem er den mit dem aufrufenden Ausführungspfad assoziierten Transaktionskontext implizit mit den Aufrufdaten propagiert. Insbesondere läuft dies für die transaktionale Anwendung transparent ab.

Registrierung von Recoverable Objects

OTS kennt nur das Verfahren der dynamischen Registrierung[21]. Ein *Recoverable Object* muss für jede Transaktion bzw. Subtransaktion an der es teilnimmt, genau ein Objekt mit einer *Ressource*-Schnittstelle beim entsprechenden *Coordinator*-Objekt registrieren. Das entsprechende *Coordinator*-Objekt kann das *Recoverable-*

[21] Das DTP-Modell kann auch die statische Registrierung zur Initialisierungszeit einer Anwendung.

Objekt aus dem mit dem Aufruf propagierten Transaktionskontext ermitteln. Ein regis-triertes Objekt mit *Ressource*-Schnittstelle nimmt am 2PC-Protokoll zur Steuerung des Transaktionsendes teil. *Transactional Objects*, die zugleich *Recoverable Objects* sind, führen die dynamische Registrierung durch, indem sie beim *Coordinator*-Objekt der entsprechenden Transaktion ihr *Ressource*-Objekt registrieren.

Transaktionsende und Recovery bei OTS

Das Transaktionsende wird bei OTS über eine optimierte Variante des 2PC-Protokolls durchgeführt. Zusätzlich unterstützt OTS auch heuristische Entscheidungen der Objekte, die eine *Ressource*-Schnittstelle besitzen. Das 2PC-Protokoll führt OTS mit jedem registrierten Objekt durch. Hierzu ruft OTS die von der *Ressource*-Schnittstelle definierten Operationen entsprechend dem Fortschritt des 2PC-Protokolls auf. Der erfolgreiche Ablauf einer Transaktionskoordinierung ist mit zwei *Recoverable Objects* in Abbildung 4-34 dargestellt.

Falls die *Ressource*-Schnittstelle nicht direkt vom *Recoverable Object* unterstützt wird, so obliegt es dem *Ressource*-Objekt, die einzelnen Aufrufe an das entsprechende *Recoverable Object* weiterzuleiten. Etwaige heuristische Entscheidungen melden Objekte mit *Ressource*-Schnittstelle ihrem Aufrufer mit Hilfe der im Modul *CosTransactions* definierten heuristischen Exceptions.

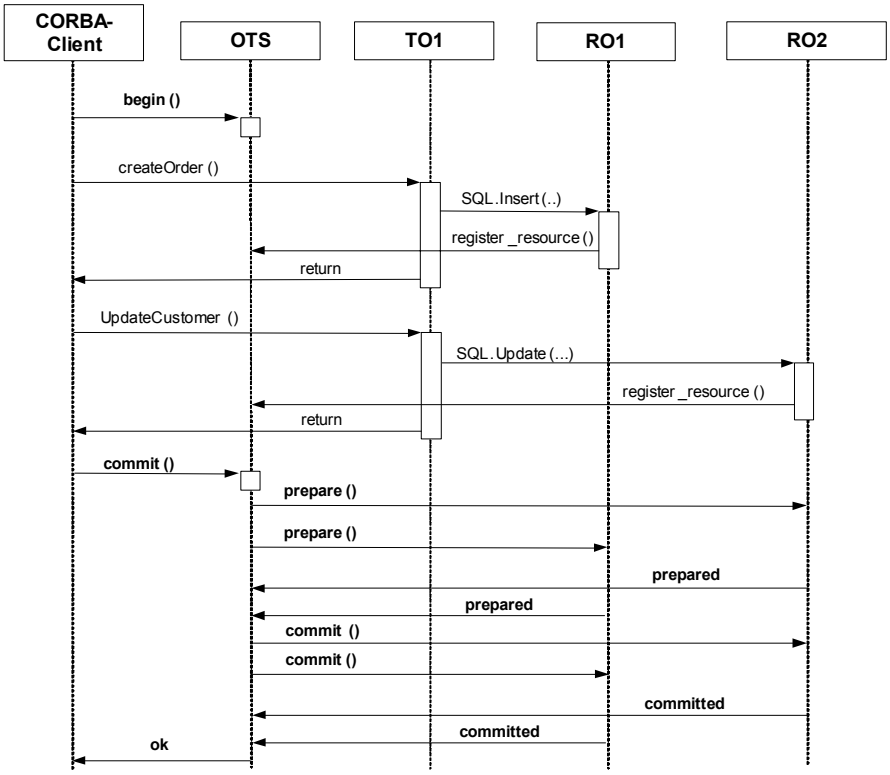


Abbildung 4-34: Erfolgreicher 2PC-Verlauf bei OTS

Bei einem Systemausfall wird versucht, die betroffene Transaktion abzuschließen. Zunächst stellen alle vom Systemausfall betroffenen Objekte so weit wie möglich ihren alten Zustand (vor dem Systemausfall) wieder her. Ein Recoverable Object, das die begonnenen Transaktionen nicht selbstständig abschließen kann, fordert mit Hilfe des ihm bei der Registrierung zugeordneten RecoveryCoordinator-Objekts die erneute Durchführung des Transaktionsabschlusses an (replay_completion(...)). Dabei übergibt es die Objektreferenz an das (neue) Ressource-Objekt, mit dem der Transaktionsabschluss durchgeführt werden soll. Damit das Recoverable Object auch nach dem Systemausfall das passende RecoveryCoordinator-Objekt erreichen kann, muss seine Objektreferenz persistent sein. Das bedeutet, dass das *RecoveryCoordinator*-Objekt nach erfolgter Wiederherstellung mit derselben Objektreferenz wie vor dem Systemausfall zu erreichen ist.

4.5 Transaktionsunterstützung in Middlewaretechnologien

In Datenbank-lastigen Client-Server-Anwendungen, die in einer zweischichtigen Architektur arbeiten (Client greift direkt auf den Datenbankserver zu) hat der Client in der Regel die Transaktionsklammerung im Griff. Er bestimmt, wann eine Datenbanktransaktion beginnt und endet. Bei dreischichtigen Architekturen mit Anwendungsservern für die Geschäftslogik hat er keinen Einfluss darauf, da er nur Requests absetzen kann. Dies führt zu einer nicht unerheblichen Verschlechterung der Transaktionsprogrammierung und soll deshalb kurz erwähnt werden.

Eine Transaktionssteuerung wird dem Client bzw. dem Programmierer nur ermöglicht, wenn er - etwa von einer Middleware an der Programmierschnittstelle - entsprechende Dienste für die Transaktionsklammerung (*begin*, *commit*) angeboten bekommt. Weiterhin impliziert dies in jedem Fall einen *stateful* Server, der im Sessionzustand auch noch Informationen über den Stand der Transaktion (Transaktionskontext) verwalten muss. Er könnte dann einen *begin*-Request absetzen, mehrere operative Requests und anschließend den *commit*-Request, um eine komplette Transaktion auszuführen. Der Server merkt sich die Zustände und nutzt auch die Isolationseigenschaften einer Datenbank für evtl. Sperren aus. Es gibt aber Ausfallsituationen, die dazu führen, dass ein Client nicht mehr weiß, ob nun seine Transaktion zu Ende gebracht wurde oder nicht. Beispielsweise braucht nur die Antwort-Nachricht auf den *commit*-Request verloren gehen. Zudem ist der Clientrechner meist kein zuverlässiges System und kann seinen Zustand daher auch jederzeit verlieren. Was in diesem Fall der Server machen muss, ist nicht gerade trivial.

Will man hier eine brauchbare Transaktionssteuerung, sollte man die Transaktionen nicht vom Client steuern lassen. Eine entsprechende Middleware kann diese Aufgabe übernehmen. In jedem Fall muss im Server ein Sessionzustand verwaltet werden, wenn man Transaktionen über einen Request hinaus ausführen möchte. Auf diese Thematik ist wird im Weiteren mit Bezug zu konkreten Middeware-Plattformen eingegangen.

4.5.1 Transaktionsunterstützung bei JEE/EJB

Die Java Enterprise Edition[22] wird von Sun Microsystems spezifiziert und definiert einen Satz von APIs für die Entwicklung verteilter Anwendungen im Java-Umfeld. Ein wesentlicher Bestandteil von JEE ist die Enterprise-Java-Beans-Spezifikation (EJB) als Basis für verteilte Komponentensysteme. Die Transaktionsunterstützung wird in JEE über die API-Standards JTA und JTS festgelegt, die vor allem im EJB-Standard angewendet werden, Diese „Standards“ werden in diesem Abschnitt betrachtet.

[22] Wir gehen hier von JEE V5.0 aus, siehe (Krafzig 2005).

Überblick

Die EJB-Architektur erlaubt die Nutzung von lokalen und von globalen Transaktionen. Für lokale Transaktionen werden die Schnittstellen des jeweiligen Ressourcenmanagers benutzt (z.B. SQL-Befehle über JDBC). Für globale Transaktionen benötigt man einen Transaktionsmanager, der Bestandteil der EJB-Application-Server-Infrastruktur sein sollte. Über diesen können dann Operationen auf mehrere Ressourenmanager in einer Transaktion geklammert werden. Hier könnte sich zum Beispiel der CORBA Transaction Service (OTS) aber auch ein Transaktionsmonitor wie CICS als Basismechanismus anbieten, um einem EJB-Application-Server die nötigen Transaktionsdienste zur Verfügung zu stellen. Lokale Transaktionen müssen laut Standard vom EJB-Server nicht über einen TM ausgeführt werden. Sie können auch direkt mit dem entsprechenden RM abgewickelt werden (vgl. Abbildung 4-35). Dies ist insbesondere dann sinnvoll, wenn es nur einen an der Transaktion beteiligten RM gibt.

Der Anwendungsprogrammierer muss sich nicht um die Transaktionsprotokolle kümmern. Diese werden vom EJB-Server-/Container-Hersteller bereitgestellt. Er übernimmt auch die Propagierung des Transaktionskontextes zwischen den beteiligten Komponenten.

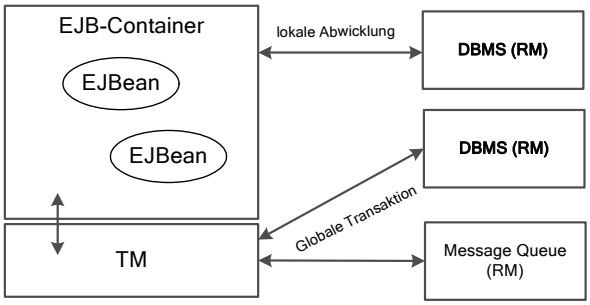


Abbildung 4-35: Abwicklung lokaler und globaler Transaktionen

Für die Unterstützung von Transaktionen ist innerhalb der EJB-Architektur die Java Transaction API (JTA) als grundlegende Schnittstelle bzw. als Satz von Schnittstellen vorgesehen (Sun 1999, Sun 1999b, Sun 2003, Sun 2007). Als Transaktionsmodell werden flache (im Gegensatz zu CORBA/OTS keine geschachtelten) Transaktionen unterstützt. JTS (Java Transaction Service) ist für Hersteller von EJB-Application-Servern gedacht, um die erforderliche Infrastruktur für die Transaktionsbehandlung zu implementieren, die EJB-Spezifikation fordert allerdings nicht, dass ein EJB-Container JTS einsetzen muss (Sun 1999b).

Die EJB-Spezifikation beschreibt drei Möglichkeiten Transaktionsgrenzen festzulegen: Der Client (*client-managed* demarcation), der Container (*container-managed* demarcation) oder die Bean (*bean-managed* demarcation) selbst setzt diese über

entsprechende Aufrufe der Methoden *begin, commit* bzw. *rollback* an einem JTA-Interface ab. Für eine Session-Bean muss festgelegt werden, ob sie entweder *container-managed* oder *bean-managed* Transactions unterstützt, beides ist nicht möglich. Entity-Beans können nur *container-managed* Transactions unterstützen.

Beispielszenario 1: Betrachten wir eine Transaktion, die ein Client durch Aufruf einer Methode der Bean B_1 initiiert, die wiederum eine Methode der Bean B_2 aufruft. Bean B_1 ruft Änderungsoperationen in zwei Datenbanken auf ($DBMS_1$ und $DBMS_2$), Bean B_2 verändert Daten in der Datenbank $DBMS_3$. Der EJB-Container hat die Aufgabe, die drei DBMS in die Transaktion zu involvieren, also den Transaktionskontext über einen Transaktionsmanager, der in der Infrastruktur verfügbar sein muss, zu propagieren.

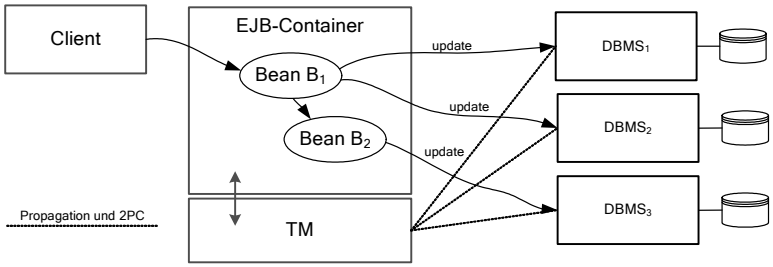


Abbildung 4-36: EJB-Transaktion über mehrere Beans in einem EJB-Container

Am Ende der Transaktion muss er die Commit-Behandlung über den Transaktionsmanager durchführen. Die Abbildung 4-36 verdeutlicht das Szenario.

Der EJB-Container führt ein 2PC-Protokoll mit den RMs aus, wenn die Transaktion beendet werden soll. Je nachdem, welche Komponente die Transaktionsgrenzen setzt (Client, EJBean oder Container) wird die Transaktion entweder aktiv vom Client bzw. der Enterprise-Bean beendet oder aber der Container leitet implizit am Ende der Requestbearbeitung und vor dem Senden des Ergebnisses an den Client die Commit-Behandlung ein. Im letzteren Fall passiert praktisch das ganze Transaktions-Handling „behind the scenes", ohne dass der Client- und der Bean-Programmierer dies explizit mitbekommen.

Beispielszenario 2: In diesem Szenario ruft ein Client auch eine Methode einer Bean B_1, die wiederum eine Methode der Bean B_2 aufruft. Bean B_2 befindet sich aber in einem anderen EJB-Container eines zweiten EJB-Servers. Beide Methoden verändern Daten in unterschiedlichen Datenbanken. Am Ende der Transaktion ist eine Abstimmung zwischen den Transaktionsmanagern der beiden EJB-Server notwendig, die auch für die Propagierung des Transaktionskontextes sorgen müssen.

Im EJB-Standard bzw. der zuständigen JTS-Spezifikation ist offen gelassen, über welches Protokoll die verteilten Transaktionsmanager untereinander zum Zwecke

der Kontextpropagierung und Transaktionskoordination kommunizieren. In der JEE-Spezifikation wird hierfür IIOP empfohlen.

Man setzt hier eine funktionsfähige Umgebung wie sie in CORBA/OTS beschrieben ist, voraus. Dies ist aber in der Praxis eine nicht unerhebliche Lücke, da EJB-Server unterschiedlicher Hersteller sich nur schwer auf ein gemeinsames Protokoll einigen können. Meist funktioniert eine über Rechnergrenzen hinweg verteilte Transaktion daher (wenn überhaupt) nur in einer homogenen EJB-Server-Landschaft eines Herstellers.

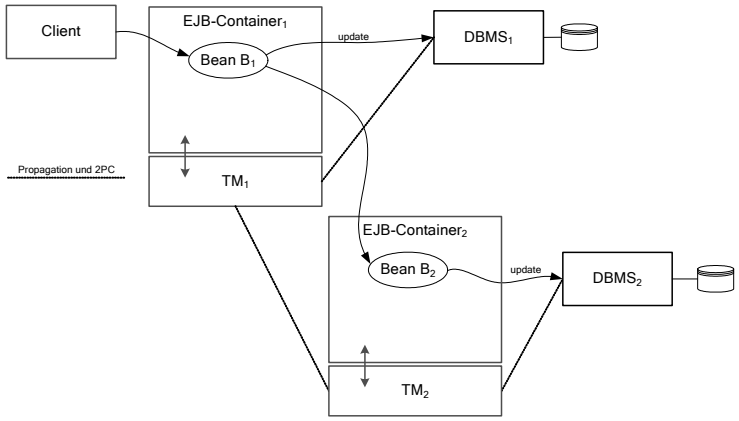


Abbildung 4-37: EJB-Transaktion über mehrere EJB-Container

Folgende Regeln gelten für die verschiedenen Arten von Enterprise-Beans (siehe Abbildung 4-38):

- Session-Beans (stateful und stateless) und message-driven Beans dürfen sowohl *bean-* als auch *container-managed* sein. Für jede Bean muss aber eine Entscheidung getroffen werden, die für alle Methoden gilt. Eine Mischung ist nicht zulässig.
- Entity-Beans können ausschließlich über die Variante *container-managed* in eine Transaktion eingebunden werden.
- Bei zustandsbehafteten Session-Beans ist es möglich, dass mehrere Methodenaufrufe in einer Transaktion ausgeführt werden können. Die Session-Bean bleibt solange im Zustand „Bereit in TA“ bis die Transaktion zu Ende ist und darf auch nicht deaktiviert werden.

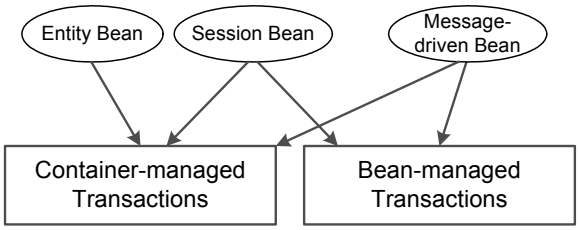


Abbildung 4-38: Möglichkeiten der Transaktionskontrolle bei EJB

Betrachten wir den etwas vereinfachten Zustandsautomaten einer zustandsbehafteten Session-Bean in Abbildung 4-39, so ist zu erkennen, dass eine Session-Bean, die sich in einer Transaktion befindet, keinen Zustandsübergang zum Zustand „passiviert“ hat.

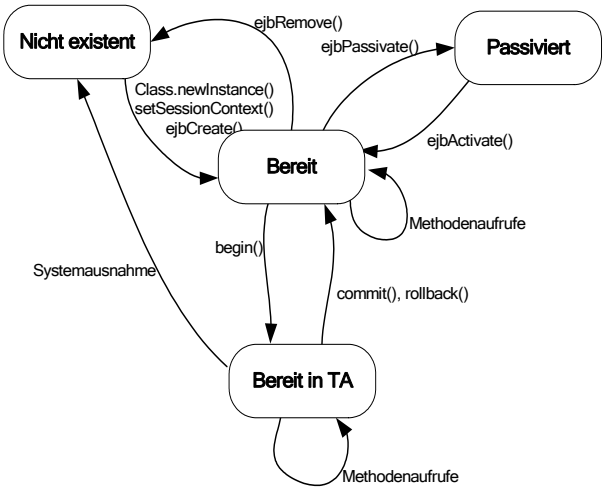


Abbildung 4-39: Zustandsautomat einer zustandsbehafteten Session-Bean

Nach dieser Einführung soll im Weiteren auf die Standardschnittstellen JTA und JTS eingegangen werden. Anschließend werden die verschiedenen Möglichkeiten der Transaktionsunterstützung bei EJB näher besprochen.

4.5.2 Java Transaction API (JTA)

Das Java Transaction API (JTA) liefert höherwertige Transaktionsdienste, die aus drei Teilen bestehen (Sun 1999a):

- Eine Schnittstelle für Anwendungsprogrammierer, mit denen diese Transaktionsgrenzen setzen können. Sie enthält die klassischen Methoden *begin*, *commit* und *rollback*.
- Ein Java-Mapping für das standardisierte XA-Protokoll der Open Group für die Teilnahme an globalen Transaktionen, die von einem Transaktionsmanager koordiniert werden.
- Eine Schnittstelle für einen Application-Server, um mit einem Transaktionsmanager zu kommunizieren. Diese Schnittstelle ist für die Transaktionsabwicklung durch den Application-Server (bzw. EJB-Container) notwendig und wird für *container-managed* Transactions verwendet.

JTA lehnt sich stark an den XA-Standard an. Das JTA-Transaktionsmodell sieht die gleichen Komponenten wie das XA-Modell vor (AP, TM, RM, CRM) und ergänzt die Rolle des Application-Servers. JTA wird vom Hersteller eines EJB-Produktes implementiert. JTA definiert aber keine Kommunikationsmechanismen für die Progagierung von Transaktionen und für die verteilte Transaktionskoordination. Hier wird auf den JTS-Standard verwiesen.

Wie in Abbildung 4-40 zu erkennen ist, sind in JTA im Wesentlichen drei Schnittstellen beschrieben. Dies sind die Interfaces *UserTransaction*, *TransactionManager* und *XAResource*. Über das Interface *UserTransaction* kommuniziert eine Anwendung (AP) mit dem Transaktionsmanager. Das Interface *TransactionManager* wird für die Kontextpropagierung und Transaktionskoordination zwischen Application-Server und Transaktionsmanager verwendet. Über das Interface *XAResource* wird das Transaktionshandling zwischen Ressourcenmanagern und einem Transaktionsmanager (Registrierung[23] von RMs, Kontextpropagierung und Koordination) geregelt.

Die Implementierung des Transaktionsmanagers sowie die Kommunikationsmechanismen werden vorausgesetzt. Wie unschwer zu erkennen ist, setzt das JTA-Modell auf die XA-Konzepte auf. In Richtung RM wird gemäß dem JEE-Standard entweder über JDBC (DBMS-Interface) oder JMS (Interface zu Message-Systemen) kommuniziert. Ein AP entspricht entweder einer serverseitigen Enterprise-Bean, was der Standardfall ist, oder aber einer Clientanwendung, welche die Transaktionsgrenzen über ein höherwertiges Interface selbst definiert. Der Protokollstack zur Kommunikation mit anderen Instanzen ist nicht näher festgelegt.

[23] Das hier nicht weiter diskutierte Subinterface *Transaction* stellt für die Bekanntmachung der Beteilung an einer Transaktion hier z.B. die Methode *enlist()* zur Verfügung, die vom RM aufgerufen wird.

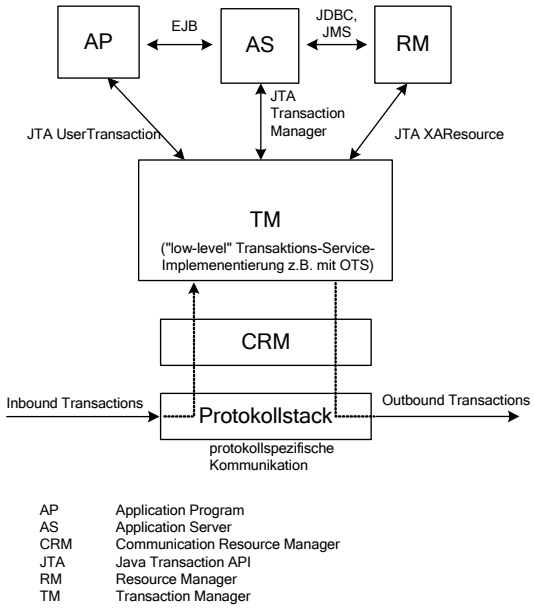


Abbildung 4-40: Komponenten und Schnittstellen des JTA- und des JTS-Modells

Client-managed Transactions: Bei der Variante *client-managed* benutzt der EJB-Client zur Begrenzung der Transaktionen das JTA-Interface *javax.transaction.-UserTransaction.* Anzumerken ist, dass die Verfügbarkeit der Schnittstelle in der EJB-Spezifikation nicht obligatorisch geregelt ist. Der Standard schreibt die Unterstützung also nicht zwingend vor.

Der Client-Programmierer setzt in diesem Fall die Transaktionsgrenzen so wie er es für richtig hält und kann damit - wie dies das Szenario aus Abbildung 4-41 zeigt - auch Methoden-übergreifende Transaktionen formulieren.

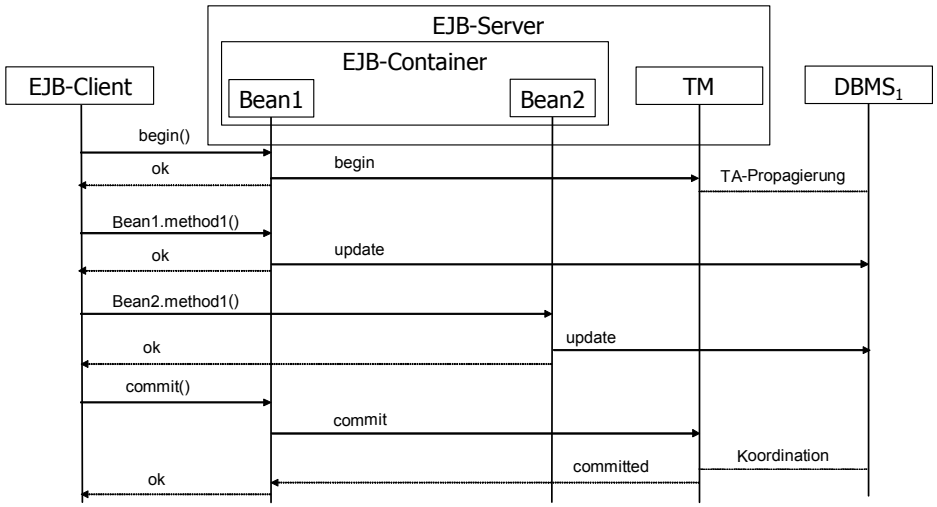


Abbildung 4-41: Client-managed Transaction in EJB-Umgebung

Im Szenario führt der Client eine Transaktion erfolgreich aus, in der zwei EJBeans beteiligt sind, die in einem EJB-Container liegen. Der in der Infrastruktur des EJB-Servers vorhandene TM koordiniert die Transaktion. In diesem Fall werden von beiden Beans Änderungen in einer Datenbank ausgeführt. In der Abbildung ist der Sachverhalt etwas vereinfacht dargestellt. Auch das verwendete Commit-Protokoll ist hier nur angedeutet. In diesem Fall könnte der TM eine 1PC-Koordinierung anstoßen, da nur ein DBMS beteiligt ist.

Anzumerken ist, dass der Client-Entwickler für das Setzen der Transaktionsgrenzen alleine verantwortlich ist. Bei komplexeren Anwendungen kann dies eine mögliche Fehlerquelle sein. Das Verhalten des Systems bei einem Ausfall des Clients während der Commit-Behandlung ist nicht eindeutig spezifiziert. So stellt sich z.B. die Frage, wie ein Client nach einem Restart erkennen kann, ob die Transaktion erfolgreich beendet wurde oder etwa zurückgesetzt werden musste.

Bean-managed Transactions: Bei der Variante *bean-managed* Transactions muss der Bean-Programmierer die Transaktionsgrenzen selbst programmieren. Hierzu nutzt er die entsprechenden JTA-Schnittstellen. Die Transaktionsgrenzen dürfen nur über das Interface *UserTransactions* gesetzt werden und nicht direkt über die Schnittstellen zu den RM (etwa über JDBC).

Im Folgenden ist ein Beispielcode (Pseudocode) für eine Methoden-übergreifende Transaktion über drei Methoden die *bean-managed* gesteuert wird, dargestellt:

```
public class MySessionEJB implements SessionBean
{
    EJBContext ejbContext;
```

```
    ...
    public void m1() {
     InitialContext initContext = new InitialContext() ;
     UserTrans = ejbContext.getUserTransactions();
     userTrans.begin();
     // Arbeit verrichten, Datenbankzugriffe ausführen
    }
    public void m2() {
     // Arbeit verrichten, Datenbankzugriffe ausführen
    }
    public void m3() {
     userTrans = ejbContext.getUserTransactions();
     // Arbeit verrichten, Datenbankzugriffe ausführen
     UserTrans.commit();
    }
}
```

Der Aufruf der Methoden müsste in der Reihenfolge *m1*, *m2* und *m3* ausgeführt werden. In Methode *m1* wird die Transaktion über das entsprechende JTA-Interface initiiert und in Methode *m3* wird diese beendet. Interessant ist hier die Frage, was der Container mit der Transaktion macht, wenn sich z.B. nach Aufruf der Methode *m2* der Client beendet oder abstürzt.

Container-managed Transactions: Bei der Variante *container-managed* sind die Transaktionsgrenzen weder durch den Client- noch durch den Bean-Programmierer festzulegen. Lediglich beim Deployment der EJBean ist das Transaktionsverhalten jeder einzelnen Methode anzugeben.

Der EJB-Container entscheidet bei jedem Methodenaufruf einer Bean anhand des zugewiesenen Transaktionsattributs, wie sich die Bean während der Transaktion verhält. Man bezeichnet diese Variante auch als deklarativ. In einem Transaktionsattribut kann für jede Methode im Deployment-Descriptor separat festgelegt werden, welche Transaktionsunterstützung gewünscht wird. Es werden folgende Transaktionsatribute unterstützt:

- „NotSupported“
- „Required“
- „Supports“
- „RequiresNew“
- „Mandatory“
- „Never“

Die Bedeutung dieser Werte kann der EJB-Spezifikation entnommen werden und wird hier kurz beschrieben:

- Wird für eine Methode der Wert „NotSupported“ angegeben, so muss der EJB-Container bei Aufruf dieser Methode die Transaktion suspendieren und nach der Ausführung der Methode wieder reaktivieren.
- Wird „Required“ angegeben, so muss eine Transaktion vorhanden sein. Ist beim Aufruf einer Required-Methode kein Transaktionskontext vorhanden, wird er vom Container angelegt. Ist ein Kontext vorhanden, so läuft die aufgerufene Methode in diesem Kontext ab.
- Bei Angabe von „Supports“ muss keine Transaktion da sein, eine vorhandene Transaktion ist aber nicht weiter störend und wird unterstützt.
- „RequiresNew“ erfordert eine neue Transaktion, ein bestehender Transaktionskontext wird also nicht verwendet. Für die Ausführung der Methode wird vom EJB-Container immer ein neuer Transaktionskontext angelegt.
- Ist „Mandatory“ angegeben, muss bereits beim Aufruf der Methode ein Transaktionskontext angelegt sein. Vom Container wird allerdings kein neuer Transaktionskontext erzeugt, wenn zum Aufrufzeitpunkt keiner angelegt ist.
- Wird „Never“ verwendet, so darf kein Transaktionskontext vorhanden sein. Falls zum Aufrufzeitpunkt doch ein Transaktionskontext vorhanden ist, wird vom EJB-Container eine Exception geworfen.

Als Standardvariante wird „Required“ empfohlen und zwar, wenn möglich, durchgängig für die gesamte EJB-Anwendung.

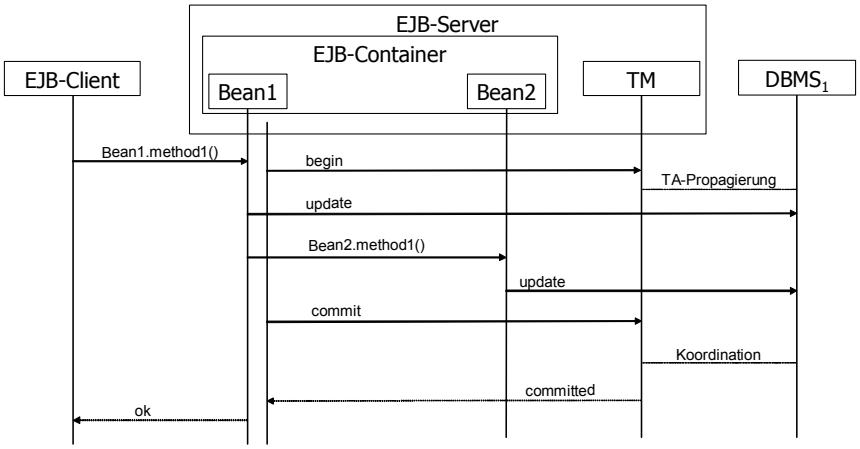


Abbildung 4-42: Container-managed Transaction in EJB-Umgebung

In Abbildung 4-42 ist ein Szenario für eine Transaktion dargestellt, die durch den Methodenaufruf auf *Bean1* initiiert wird. *Bean1* nutzt ihrerseits *Bean2* und beide Beans verändern Daten in einer Datenbank. Wir nehmen an, die beiden Methoden haben als Transaktionsattribut „Required“ gesetzt, d.h. der Container erzeugt eine

Transaktion beim ersten Methodenaufruf und beendet diese bevor das Ergebnis an den Client zurückgegeben wird.

Die Propagierung des Transaktionskontextes sowie die Koordination mit dem DBMS werden vom EJB-Container initiiert und durch den im EJB-Server vorhandenen TM ausgeführt. Laut EJB-Spezifikation ist die Transaktion noch vor der Übergabe des Methodenergebnisses an den Client zu beenden. Der Client sieht die Transaktionsgrenzen nicht, der Programmierer muss sich aber über die Semantik der Methoden im Klaren sein.

Bei Nutzung einer Methode einer zustandslosen Session-Bean innerhalb einer Transaktion muss die Transaktion beendet werden, bevor das Ergebnis zum Client gesendet wird, da die Session-Bean anschließend nicht mehr verfügbar ist.

4.5.3 Java Transaction Services (JTS)

JTS spezifiziert die Implementierung eines Transaktionsmanagers, der extern die JTA-Interfaces als Dienst bereitstellt und diese intern auf CORBA/OTS 1.1 abbildet (siehe hierzu Abbildung 4-43). JTS definiert also eine JTA-Implementierung auf Basis von OTS. Es ist ein Binding für CORBA/OTS 1.1 spezifiziert. JTS ist daher an Implementierer von Transaktionsmanagern und EJB-Application-Server gerichtet und wird von den Herstellern der EJB-Infrastrukturen benutzt, um ihre Produkte zu implementieren und interoperabel zu machen.

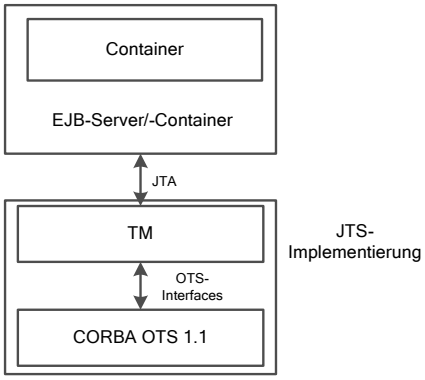


Abbildung 4-43: JTS als Interface zwischen einem EJB-Server und einem Transaktionsmanager

Als Transaktionsmodell werden flache Transaktionen (OTS-Terminologie: Top-Level-Transaktionen) unterstützt, jedoch keine geschachtelten Transaktionen. CORBA/OTS wird also nicht vollständig ausgenützt

Die Schnittstellen und die architektonischen Zusammenhänge der JTS-Spezifikation sind in der Abbildung 4-44 dargestellt. Ein JTS-basierter TM bildet die

Transaktionen in Richtung Transaktions-Infrastruktur auf CORBA/OTS ab. Hierfür implementiert er die Java-Packages *org.omg.CosTransactions* und *org.omg.CosTSPortability*. Er bindet verschiedene RM entweder über die in JTA verfügbare XARessource-Schnittstelle oder optional über sog. native Anbindungen direkt auf Basis von XA bzw. CORBA/OTS.

Ein JTS-unterstützender Transaktionsmanager hat folgende Aufgaben:

- Er verwaltet die Transaktionskontexte aller über ihn initiierten globalen Transaktionen. Er erlaubt es auch, dass mehrere Java-Threads einer JMV oder verschiedener JVMs zu einer Zeit in dieselbe Transaktion verwickelt sind und Operationen für diese ausführen.
- Er kommuniziert mit Ressourcenmanagern zum Zwecke der Kontext-Propagierung und der Transaktionskoordination über das XA-Protokoll der Open Group. Die Anbindung eines RM über JTA geschieht üblicherweise über Ressource-Adapter, welche das Mapping von JTA zum proprietären RM übernehmen. Java-Anwendungen werden in der Regel über einen Application-Server in die Transaktion involviert (*bean-* oder *container-managed*).
- Er kommuniziert mit anderen Transaktionsmanagern standardmäßig über die Dienste von CORBA/OTS. Hier wird IIOP als Transportprotokoll für die Transaktionspropagierung empfohlen. Eine weitere Festlegung wird in der JTS-Spezifikation aber nicht vorgenommen.

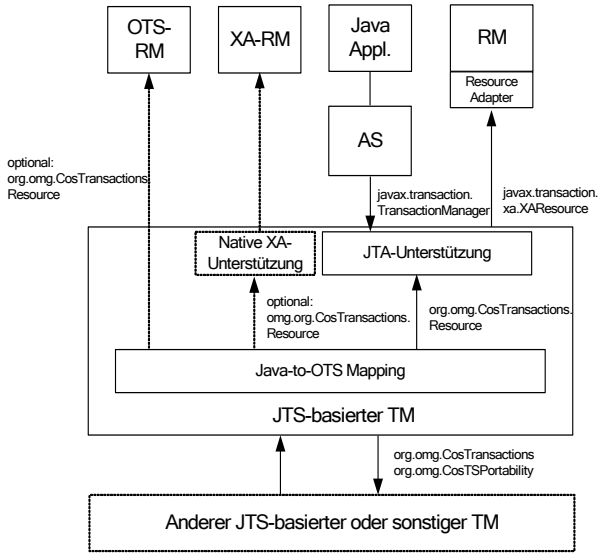


Abbildung 4-44: Architektur einer TM-Implementierung auf Basis von JTA

Der TM unterstützt die JTA-Schnittstellen (*TransactionManager*, *Transaction* und *UserTransaction*), um einem Application-Server oder einer „stand-alone" Anwen-

dung die Möglichkeit zu geben, Transaktionen auszuführen und Transaktionsgrenzen zu setzen.

Damit sich ein CORBA ORB beim JTS TM bekannt machen kann, implementiert dieser ein Interface namens *javax.jts.TransactionService* mit der einzigen Methode *identifyORB*. Ein ORB nutzt das Interface üblicherweise während der Initialisierungsphase. Ein Vorschlag für den Ablauf der Initialisierung ist in der JTS-Spezifikation (Sun 1999b) beschrieben.

4.5.4 Transaktionsverarbeitung bei EJB 3.0

Bei EJB 3 hat sich im Vergleich zu EJB 2.x bezüglich des Transaktions-Handlings konzeptionell nichts verändert. Man kann nach wie vor deklaratives CMT oder BMT mit expliziter Programmierung der Transaktionsgrenzen einsetzen und die JEE-Schnittstelle JTA dient weiterhin als Basis.

Bei Einsatz von CMT können in EJB 3 Methoden oder ganze Bean-Klassen auch über Annotationen mit dem Transaktionsattribut versehen werden. Weiterhin ist auch eine Angabe der Transaktionsattribute über einen DD möglich. Die Transaktionsattribute haben sich nicht verändert. Als Notation wird „@TransactionAttribute“ bereitgestellt. Die Nutzung von Annotationen soll an einem Beispiel verdeutlicht werden:

```
...
@Stateless
@TransactionAttribute(NOT_SUPPORTED)
public class ArticleMgrSessionBean
     implements ArticleMgrRemoteI {
   public void createArticle(…) {…}
   public vector findAllArticles() {…}

@TransactionAttribute(REQUIRED)
   public void deleteArticle(…) {…}
   …
}
```

In diesem Beispiel gilt für die gesamte EJBean *ArticleMgrSessionBean*, dass keine Transaktionen unterstützt werden (NOT_SUPPORTED). Für die Methode *deleteArticle* wird diese Einstellung allerdings über die Angabe des Transaktionsattributs „REQUIRED“ überdefiniert. Dies bedeutet, dass diese Methode immer eine Transaktion fordert, die bei jedem Aufruf - sofern noch nicht im Kontext - vom EJB-Container angelegt wird. Über einen Deployment-Descriptor wäre eine derartige Attributierung wie folgt möglich:

```
<ejb-jar>
     …
   <description>Artikelmanager</description>
```

```
   <display-name>Artikelmanagement-Application
   </display-name>
   <enterprise-beans>
   <session>
  <display-name>ArticleMgrSessionBean</display-name>
  <description> Bean für die Artikelverwaltung</description>
  <ejb-name>ArticleMgrSessionBean</ejb-name>
  <ejb-class>ArticleManagement.ArticleSessionBean
  </ejb-class>
  <home>ArticleManagement.ArticleMgrSessionBeanHome
  </home>
  <remote>
      ArticleManagement.ArticleMgrSessionBeanRemote
  </remote>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>
  </session>
  …
    </enterprise-beans>
    …
    <assembly-descriptor>
      <container-transaction>
         <method>
            <ejb-name>ArticleMgrSessionBean</ejb-name>
                 <method-name> * </method-name>
         </method>
         <trans-attribute>NotSupported</trans-attribute>
      </container-transaction>
         <container-transaction>
         <method>
                 <ejb-name>ArticleMgrSessionBean</ejb-name>
                 <method-name>deleteArticle</method-name>
         </method>
         <trans-attribute>Required</trans-attribute>
      </container-transaction>
         …
   </assembly-descriptor>
</ejb-jar>
```

In diesem Beispiel wird gezeigt, wie man alle Methoden einer EJBean mit dem gleichen Transaktionsattribut ausstatten und eine Ausnahme definieren kann. Ohne Angabe eines Transaktionsattributs gilt „REQUIRED“ als Standardeinstellung. Bei message-driven Beans (sog. MDBs) kann man übrigens nur „REQUIRED“ oder „NOT_SUPPORTED“ verwenden, andere Einstellungen ergeben kei-

nen Sinn, da eine MDB-Transaktion von Client, der sie initiiert entkoppelt ist. Dies gilt auch für EJB 2.1.

4.5.5 Transaktionsunterstützung bei JMS

Wir haben in Kapitel 2 bereits die Zuverlässigkeitsmaßnahmen erläutert, die JMS ermöglicht. JMS erlaubt durch die Einstellung eines Transaktionsmodus die Definition von atomaren, lokalen Transaktionen. Ein JMS-Client kann die gesamte Nachrichtenbearbeitung (Senden und Empfangen) innerhalb einer JMS-Session in atomare Transaktionen gliedern. Diese Transaktionsdefinition erhebt aber nicht den Anspruch einer ACID-Transaktion, da die anderen Korrektheitskriterien (außer „A“) nicht festgelegt sind.

JMS-Provider können auch optional (nicht im Standard vorgeschrieben) an einer verteilten Transaktion teilnehmen. Hierzu wird vorgegeben, dass das JTA-API, und zwar speziell das *XAResource*-API, unterstützt wird. Die Spezifikation schreibt dies aber für eine JMS-Implementierung nicht zwingend vor. Schließlich ist es möglich, dass ein JMS-Client eine verteilte Transaktion direkt steuert.

4.5.6 Transaktionsunterstützung bei .NET

Die Transaktionskonzepte von .NET (.NET Framework) und JEE ähneln sich stark. In diesem Fall hat vermutlich die Java-Gemeinde vom Microsoft Transaction Manager (MTS) profitiert, den es schon einige Zeit gibt.

.NET unterstützt verteilte Transaktionen im Komponentenmodell .NET *Enterprise Services*. Jede Komponente, die Transaktionen unterstützen soll, ist als sog. *Serviced Component* zu implementieren, indem sie von der Basisklasse *ServicedComponent* abgeleitet wird.

Alle *Serviced Components* werden in Containern verwaltet, die heute noch übergangsweise auf die COM+-Infrastruktur abgebildet werden. Sie werden in einem Katalog, dem sog. COM+-Katalog mit all ihren Metadaten registriert und damit bekanntgemacht. Dies übernimmt das Laufzeitsystem, also die CLR.

Transaktionen sind also in die *.NET Enterprise Services* (Namensraum *System.EnterpriseServices*) integriert. Ein Teil der Enterprise Services ist ein klassischer Transaktionskoordinator für verteilte Transaktionen namens *MS DTC* (Microsoft Distributed Transaction Coordinator).

Ähnlich wie bei JEE/EJB werden auch im .NET-Komponentenmodell automatische Transaktionen unterstützt, die vollständig vom Container abgewickelt werden. Wenn also eine Clientanwendung die Dienste einer Komponente nutzt, merkt sie in der Regel nicht, ob dahinter eine Transaktion abläuft oder nicht. Das Setzen der Transaktionsgrenzen übernimmt der Container. Die Komponente muss lediglich mit Metadaten beschrieben werden, in denen ein Transaktionsattribut angegeben ist. Dies sieht im Coding einer Komponente beispielsweise wie folgt aus (hier in C#), wobei *MyComponent* ein willkürlicher Name einer Komponente ist:

```
[Transaction (TransactionOption.Required)]
public class MyComponent : ServiceComponent, IMyComponent
{
    // Implementierung der Methoden der Schnittstelle
    // IMyComponent
    public void m1() {…}
    public void m2() {…}
}
```

Die Metainformation wird mit *TransactionsOption* angegeben. Die eigene Komponente erbt von der Klasse *ServiceComponent* und implementiert die eigene Schnittstelle *IMyComponent*.

Als Transaktionsattribute sind ähnlich wie bei JEE/EJB folgende Möglichkeiten gegeben:

- „NotSupported“
- „Required“
- „Supported“
- „RequiresNew“
- „Disabled“
- „Suppress“

Die Transaktionsattribute gelten im Gegensatz zu JEE/EJB für die ganze Komponente. Die Attribute haben folgende Bedeutung (Vasters 2002):

- Wird für eine Komponente „NotSupported“ angegeben, so kann diese keine Transaktionen unterstützen. Erfolgt der Aufruf einer Methode der Komponente in einem Transaktionskontext, so muss der Container bei Aufruf dieser Methode einen neuen Kontext ohne Transaktion erzeugen. Dies ist die Standardeinstellung für Komponenten.
- Bei Angabe von „Supported“ unterstützt die Komponente auch die Abwicklung von Transaktionen. Falls im aktuellen Kontext keine Transaktion angelegt ist, wird die aufgerufene Methode ohne Transaktionsunterstützung ausgeführt.
- Wird „Required“ angegeben, so muss für eine Transaktion wie bei JEE/EJB ein Transaktionskontext vorhanden sein. Ist beim Aufruf einer Required-Methode kein Transaktionskontext vorhanden, wird einer vom Container angelegt.
- „RequiresNew“ erfordert zwingend eine neue Transaktion, ein bestehender Transaktionskontext wird also nicht verwendet. Für die Ausführung der Methode wird vom Container ein neuer Transaktionskontext angelegt.
- Wird „Disabled“ verwendet, wird ein aktuell vorhandener Transaktionskontext ignoriert und in einem neuen Kontext ohne Transaktionsunterstützung ausgeführt.

– Verwendet man „Suppress“[24] wird nie eine Transaktion benutzt, unabhängig davon, ob bereits ein Transaktionskontext vorhanden ist oder nicht.

Beispiel: Betrachten wir den Ablauf einer Transaktion an einem Beispiel, in dem ein .NET-Client eine Methode einer Komponente *Komp1* aufruft, die als Transaktionsattribut „Required“ gesetzt hat (Abbildung 4-45).

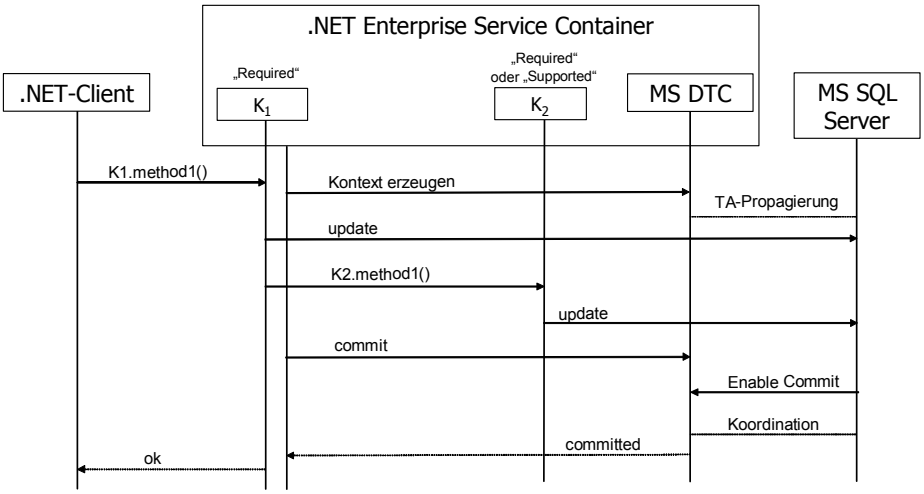


Abbildung 4-45: Verteilte Transaction in .NET-Umgebung

Beim Aufruf wird (sofern noch nicht vorhanden) über den DTC ein Transaktionskontext angelegt. Die Methode führt eine Update-Operation an einen MS SQL-Server aus, der seinerseits den aktuellen Transaktionskontext direkt vom DTC anfordert. Im Beispiel ruft die Methode aus K_1 eine weitere Methode der Komponente K_2 auf. Diese führt wieder eine Update-Operation mit demselben SQL Server aus. Da das Transaktionsverhalten von K_2 mit Transaktionsattribut „Required“ oder „Supported“ festgelegt ist, wird derselbe Transaktionskontext verwendet. Der Transaktionskontext ist dem SQL Server bereits bekannt. Der SQL Server meldet dem DTC, dass er in der Lage ist, die Transaktion erfolgreich zu beenden.

Bevor der Container das Ergebnis des Methodenaufrufs an den Client zurück sendet, kommuniziert er mit dem DTC, um die Transaktion zu Ende zu führen. Dieser wickelt die Koordination ohne weitere Kommunikation mit den Komponenten ab.

Welches Commit-Protokoll zwischen dem DTC und dem SQL Server verwendet wird und wie zwischen den beteiligten Systemen die Transaktionspropagierung stattfindet, ist nicht offen gelegt.

[24] Engl.: to suppress = ausblenden.

Es ist bei .NET Enterprise Services auch möglich, dass ein Client mehrere Methodenaufrufe innerhalb eines Transaktionskontextes ausführt. In diesem Fall muss der Komponentenentwickler aber das Transaktionsende explizit programmieren, in dem er innerhalb des Kontextes eine Methode *setComplete* aufruft. Dazu muss aber auch die Standardvorgehensweise der automatischen Beendigung nach dem Methodenaufruf deaktiviert werden. Dies geschieht über das Setzen eines Flags im Kontext (*EnableCommit = false;).* Aber auch bei .NET sollte man auf diese Variante nach Möglichkeit verzichten. Bei Fehlern während der Transaktionsausführung führt der Container automatisch einen Rollback durch.

Abschließend soll noch erwähnt werden, dass in neueren .NET-Versionen (seit .NET 2.0) ein sog. *Lightweight Transaction Manager* (LTM) eingeführt wurde. LTM unterstützt lokale und globale Transaktionen. Zur Optimierung der Leistung kann ein RM (z.B. SQL Server 2005) ankommende Transaktionen zunächst lokal verwalten. Eine lokale Transaktion kann zu einer globalen Transaktion erweitert werden, was aber nur notwendig wird, wenn weitere RMs involviert werden. Ist dies nicht notwendig erfolgt der Abschluss der Transaktion mit einem 1PC-Protokoll. Wird es notwendig, übernimmt der MS DTC die Transaktionskoordination. Dieser Mechanismus wird als *Promotable Single Phase Enlistment* (PSPE) bezeichnet.

4.5.7 Transaktionen für Webanwendungen

In Webanwendungen sendet man vom WWW-Browser für den Aufruf eines serverseitigen Dienstes einen HTTP-Request, der üblicherweise eine Verarbeitung im Webserver anstößt. Der Webserver bearbeitet den Request und sendet als Ergebnis einen HTTP-Response zum WWW-Browser. Diesen synchronen Vorgang kann man mit Hilfe von Ajax-Technologien[25] auch asynchron durchführen. Für die Bearbeitung des Requests kann man serverseitig beliebige Mechanismen verwenden, je nachdem in welchem Technologieumfeld man sich bewegt.

In der Praxis hat es sich bewährt, je HTTP-Request eine Transaktion durchzuführen (siehe hierzu Transaction-Per-Request-Pattern in Bauer 2002). Die Transaktion beginnt nach der Ankunft des Requests und wird kurz vor dem Absenden der Response-Nachricht abgeschlossen. Eine Transaktion, deren Grenzen eine Request/Response-Bearbeitung überschreitet, wird aufgrund der Unkontrollierbarkeit der Clients als nicht besonders sinnvoll erachtet.

Im Server können im Rahmen der Transaktion durchaus mehrere RMs beansprucht werden. Ob eine gemeinsame Transaktionskoordinierung durchgeführt wird, hängt von der eingesetzten Middleware (JEE/EJB, .NET,…) ab.

---

[25] Ajax ist ein Akronym für Asynchronous JavaScript and XML und dient als Unterstützung für die Programmierung interaktiver Webanwendungen.

4.5.8 Transaktionen für Webservice-basierte Anwendungen

Auch für serviceorientierte Architekturen insbesondere für Webservices wurde das Konzept der Transaktionen in den letzten Jahren intensiv diskutiert. Einige Vorschläge zur Nutzung wurden bereits erarbeitet, allerdings hat sich noch kein Standard in diesem Umfeld herauskristallisiert. Man benötigt dann übergreifende Transaktionen, wenn mehrere Webservices zur Erreichung eines gemeinsamen Ziels eingesetzt werden sollen, also gemeinsam atomar auszuführen sind.

Serviceorientierte Architekturen innerhalb von Unternehmen (wir bezeichnen diese als „geschlossene" Architekturen) sind aus Transaktionssicht heute im Wesentlichen nichts anderes als die bereits diskutierten Transaktionen in verteilter Umgebung. Betrachtet man aber „offene" Architekturen, bei denen eine Anwendung innerhalb einer Transaktion Webservices unterschiedlichster Anbieter über das Internet einbinden möchte, ergibt sich eine weit komplexere Situation. Nur letztere sind für unsere weitere Betrachtung interessant, da geschlossene Architekturen prinzipiell nichts Neues im Vergleich zu JEE/EJB usw. mit sich bringen.

In den letzten Jahren wurden einige Vorschläge zur Anwendung des Transaktionskonzepts in diesem Umfeld gemacht. Im Wesentlichen werden in den Standardisierungsvorschlägen heute zwei Ansätze diskutiert:

- Atomare Transaktionen mit allen ACID-Eigenschaften
- Langlebige Transaktionen mit Aufweichung der klassischen ACID-Eigenschaften

Die Standardisierungsbemühungen werden überwiegend durch das OASIS-Konsortium (WWW-036) zusammengeführt. Für serviceorientierte Umgebungen existieren mehrere Spezifikationen, die alle als Grundlage das bekannte Koordinator-Teilnehmer-Modell favorisieren:

- *WS-Coordination* mit den Koordinationstypen *WS-Atomic-Transaction* und WS-Business-Activity (WS-C, WS-AT und WS-BA), siehe hierzu (OASIS 2007a, OASIS 2007a und (OASIS 2007c)
- Business Process Transaction (BTP), siehe (OASIS 2002)
- Web Services Composite Application Framework (WS-CAF), siehe (Sun 2005)

Es lässt sich grundsätzlich festhalten, dass in diesen Ansätzen keine neuen Konzepte vorgeschlagen werden, sondern die bereits bekannten Konzepte der ACID-Transaktion und der langlebigen Transaktionen auf Webservices angewendet werden. Eine Transaktion klammert in diesem Fall den Aufruf mehrerer logisch zusammenhängender Webservice-Operationen.

Da Webservices auch evtl. von unterschiedlichen Providern bereitgestellt werden, ist für eine derartige Transaktion möglicherweise eine Koordination der beteiligten Provider erforderlich. Jeder Webservice-Provider sorgt für sich, dass die auszuführenden Operationen den Ansprüchen einer Transaktionslogik genügen. Die ein-

zelnen Ansätze sollen kurz skizziert werden. Wir beginnen mit dem *WS-Coordination*-Ansatz von OASIS.

WS-Coordination: *WS-Coordination* stellt Basisdienste bereit, die die Verwaltung von Transaktionskontexten und die Aktivierung von Transaktionen sowie die Registrierung von Teilnehmern unterstützen. Die Spezifikation ist so allgemein gehalten, dass verschiedene Koordinationstypen möglich sind. In der Abbildung 4-46 ist das Zusammenspiel der Komponenten skizziert. Die Anwendung AP_1 erzeugt zunächst einen Transaktionskontext bei ihrem Koordinator A und sendet anschließend eine Nachricht, also den Aufruf eines Webservices, an die Anwendung AP_2. AP_2 macht ihrerseits die Transaktion nun bei ihrem Koordinator B bekannt und registriert diese, wobei ein Koordinationsprotokoll festgelegt wird. Dieses wird schließlich zwischen den Registration-Services beider Koordinatoren ausgetauscht. Anschließend nutzen sie das abgestimmte Protokoll. Im Beispiel sind bereits zwei Koordinatoren im Einsatz, die bei der Commit-Bearbeitung zusammenarbeiten müssen.

Als Koordinationstypen sind *WS-AT* und *WS-BA* definiert. Beide setzen auf WS-C auf:

- *WS-AT* stellt sog. *Atomic Transactions* für klassische, kurze ACID-Transaktionen bereit. Die Koordination erfolgt über ein definiertes 2PC-Protokoll, wobei zwischen einem volatilen (für Cache-Resourcen) und einem dauerhaften (für persistente Ressourcen) 2PC unterschieden wird.
- *WS-BA* stellt sog. Business Activities für langlebige Transaktionen zur Verfügung. Durch die Aufweichung der Isolationseigenschaft benötigt man wie bei Sagas (offen geschachtelte Transaktionen) Kompensationsoperationen. Auch bei WS-BA sind verschiedene Completion-Protokolle spezifiziert, um Transaktionen zu Ende zu führen.

Alle WS-C/AT/BA-Nachrichten sind als XML-Nachrichten spezifiziert. Die Protokolle sind in Form von Zustandsautomaten definiert.

BTP ist ebenfalls einen Initiative von OASIS für lose gekoppelte Anwendungen. In BTP werden als Transaktionsmodelle „*Atomic Business Transactions*“ und „*Cohesive Business Transactions*“ definiert. Erstere sind verteilte Transaktionen nach dem ACID-Prinzip, letztere sind offen geschachtelte Transaktionen. Der Koordinator erzeugt die Transaktion und ruft innerhalb dieser mehrere Services in Subtransaktionen auf. Wenn eine Subtransaktion nicht erfolgreich verläuft, kann der Koordinator über die weitere Ausführung bzw. den Abbruch entscheiden. Dies muss vom Entwickler der Haupttransaktion bei der Programmierung festgelegt werden. Die ACID-Korrektheitskriterien können hier nicht erfüllt werden.

WS-CAF gliedert sich schließlich in die Rahmenwerke WS-Context (WS-CTX), WS-Coordination-Framework (WS-CF) und WS-Transaction-Management (WS-TMX). Auch bei diesem Ansatz sind sowohl ACID-Transaktionen als auch langlebige Transaktionen mit Kompensationsmechanismen spezifiziert.

Aus heutiger Sicht hat wohl WS-C/AT/BA die besten Aussichten auf eine allgemeine Anerkennung. Da die Diskussion aber noch in vollem Gange ist, soll auf eine weitere Erläuterung der derzeit vorhandenen Ansätze verzichtet werden.

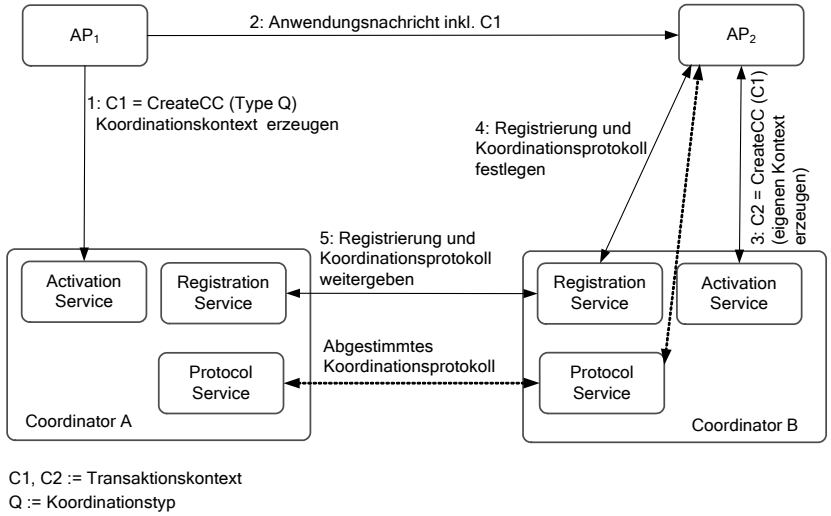


Abbildung 4-46: Zusammenspiel bei WS-Coordination

Es sei noch angemerkt, dass Webservice-Transaktionen mit ACID-Eigenschaften, die mehrere Koordinatoren einbeziehen, in der Praxis sehr schwer zu organisieren sind. Auch der Aufbau von Infrastrukturen für langlebige Transaktionen und die Entwicklung von Kompensationstransaktionen sind ein komplexes Unterfangen.

Da die an einer Transaktion beteiligten Webservices möglicherweise auf völlig eigenständig agierenden Transaktionsplattformen ablaufen, die nur schwer in Einklang zu bringen sind, ist der praktische Einsatz der genannten Ansätze sehr problematisch. Man muss sich die Frage stellen, ob die Umsetzung einer der genannten Ansätze überhaupt praxisrelevant ist. Aktuell gibt es noch keine verbreitete Implementierung hierfür.

4.6 Zusammenfassung, praktische Ansätze und Ausblick

Es lässt sich festhalten, dass heutige verteilte Transaktionssysteme bzw. Transaktionsanwendungen meist flache Transaktionen und gelegentlich offen geschachelte Transaktionen unterstützen. Die vorherrschenden Transaktionsstandards kümmern sich entweder um das Commit-Protokoll (OSI TP), wobei 2PC und 1PC üblich sind, oder um die Schnittstellen zwischen den beteiligten Partnern (CORBA/OTS, JTA, JTS). Bei den Webservice-Transaktionsansätzen werden beide Aspekte betrachtet. Aufgrund der Schwäche im 2PC-Modell versucht man häufig,

Transaktionen, die über viele Knoten verteilt sind, im praktischen Einsatz möglichst zu vermeiden.

Heute üblich Ressourcenmanager sind vor allem Datenbankmanagementsysteme (Oracle, MS SQL Server, IBM DB2, …) und Message-Queueing-Systeme (Websphere MQ, …). Allgemein anerkannte Standards für die Nebenläufigkeitskontrolle, das Logging und das Recovery sind heute so gut wie nicht verfügbar. Als Concurrency-Control-Mechanismus wird in der Regel ein pessimistisches, meist strenges 2PL-Sperrkonzept eingesetzt, das heute in vielen Varianten implementiert wird. Logging- und Recovery-Funktionen implementiert jeder Hersteller für sich, natürlich im Einklang mit dem Koordinator-Teilnehmer-Modell. Auch die heute verfügbaren Transaktionsmanager (CORBA-basiert, EJB-basiert, …) implementieren ihre eigenen Logging- und Recovery-Funktionen.

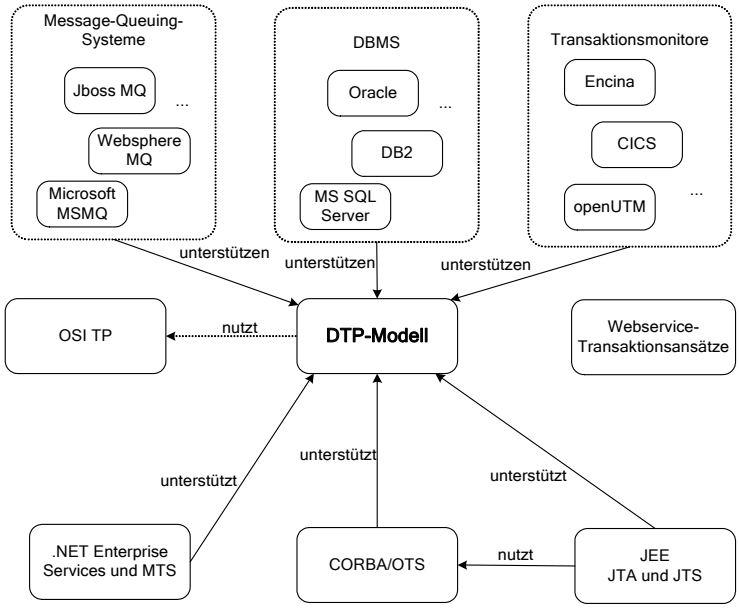


Abbildung 4- 47:Abhängigkeiten der verschiedenen Transaktionsansätze

Als zentrales und wichtigstes Transaktionsmodell hat sich in der Praxis das DTP-Modell der Open Group durchgesetzt. Die meisten anderen Ansätze unterstützen die im DTP-Modell beschriebenen Schnittstellen. Die am weitesten verbreitete DTP-Schnittstelle ist das XA-Interface zwischen einem Ressourcenmanager und einem Transaktionsmanager. XA wird von praxisrelevanten Datenbankmanagementsystemen, Application-Servern und Transaktionsmonitoren unterstützt. Auch die OMG legt in ihrer OTS-Spezifikation die Zusammenarbeit mit DTP-basierten Lösungen fest. Da die meisten Datenbankmanagementsysteme schon XA unter-

stützen, ist eine XA-Anbindung dieser Ressourcenmanager an OTS-Implementierungen eine naheliegende Lösung. In Abbildung 4- 47 sind die Beziehungen der verschiedenen Ansätze mit dem DTP-Modell als zentralem Modell grafisch dargestellt.

Bei Implementierungen von Plattformen für verteilte Transaktionssysteme findet man auch häufig CORBA/OTS. Dieser Standard gilt auch als Basis für die Implementierung verteilter Transaktionssysteme im JEE/EJB-Umfeld. Praktische Anwendungssysteme, die unterschiedliche EJB-Application-Server verwenden und über mehrere heterogene Container verteilte Transaktionen einsetzen, sind aber in der Praxis selten anzutreffen. Für das transaktionale Zusammenspiel von Containern unterschiedlicher Hersteller gibt es in der EJB-Spezifikation auch nur die Empfehlung und nicht die Vorgabe IIOP als Protokoll für die Kontextpropagierung zu verwenden. Dies macht ein Zusammenspiel unterschiedlicher Container schwierig.

Die Transaktionsansätze für Webservice-basierte Transaktionen nutzen das DTP-Modell bzw. die XA-Schnittstelle natürlich implizit durch die Verwendung von Datenbanken.

Praktische Ansätze und Ausblick

In der heutigen Praxis werden überwiegend flache Transaktionen genutzt, die aber in den meisten Fällen völlig ausreichen. Geschachtelte Transaktionen gelten als sehr aufwändig in der Implementierung, vor allem bei der Commit-Bearbeitung.

Das Zusammenspiel der beteiligten Komponenten ist insbesondere bei heterogenen Produkten recht komplex und fehleranfällig. Kompliziert wird das Ganze bei Fehlersituationen wie z.B. dem Ausfall eines Koordinators während der Commit-Bearbeitung (z.B. nach der Phase 1). Beispielsweise hat man immer wieder Probleme bei XA-basierten verteilten Transaktionen, wenn das Zusammenspiel zwischen TM und RM in kritischen Situationen nicht funktioniert und dadurch Transaktionen und damit auch Sperren „hängen“ bleiben. Es ist heute sicher schwierig, nahezu fehlerfreie Recovery-Implementierungen vorzufinden.

Aufgrund der Komplexität der Implementierung verteilter Transaktionsmiddleware sind für die Anwendungsentwicklung eher pragmatische Ansätze anzustreben, die nur die notwendigsten Features verteilter Transaktionen ausnutzen. Es sind heute u.a. folgende Ansätze bei der Nutzung verteilter Transaktionen üblich:

- Verteilte Transaktionen sind nur dann zu nutzen, wenn man sie unbedingt benötigt und dann so einfach wie möglich zu gestalten (also flach und nicht geschachtelt).
- ACID-Transaktionen sind grundsätzlich nur für kurze Transaktionen sinnvoll einsetzbar. Zusammenhängende Operationen, die über Organisationsgrenzen hinausgehen, sind nicht als ACID-Transaktionen implementierbar. Hier müssen seriell ausgeführte Transaktionen implementiert werden. Als

vorteilhafte Lösung hat sich die Aufgliederung einer langen Transaktion in kleinere Transaktionsschritte erwiesen. Die Zusammensetzung und die Realisierung evtl. notwendiger Kompensationsoperationen bleiben aber dem Anwendungsentwickler überlassen.

- Transaktionen beginnen üblicherweise im Server, da das Setzen von Transaktionsgrenzen durch einen Client, der im Wesentlichen der Präsentation dient, zu unsicher ist. Er kann nicht die Rolle eines Transaktionskoordinators übernehmen.
- Ein pragmatischer Ansatz sind „service-orientierte Transaktionen" (Transaction-per-Request-Pattern). Je Methodenaufruf gibt es nur eine in sich abgeschlossene Transaktion, die im Server abläuft. Wenn der Methodenaufruf eines Clients abgearbeitet ist, sollte auch die Transaktion abgeschlossen werden. Hier bleibt immer noch das Problem, dass der Client ggf. aufgrund eines Netzwerkausfalls das Methodenergebnis (Transaktionsergebnis) nicht mitbekommt. In dialogorientierten Anwendungen ist für eine evtl. Abklärung des Ergebnisses ein entsprechender Dialog vorzusehen. Das Isolationsprinzip von Datenbanken kann bei diesem Ansatz ebenfalls nur teilweise genutzt werden. Nach jeder Methode werden die Objektsperren freigegeben. Objekte, die methodenübergreifend bearbeitet werden, sind zunächst „optimistisch" durch die Anwendung zu „sperren" und vor einer Änderungsoperation nochmals im Server auf ihren Ursprungswert hin zu überprüfen. Bei Abweichungen kann die Änderungsoperation nicht ausgeführt werden und es ist eine Ausnahmebehandlung durch die Anwendung erforderlich.

Die genannten praktischen Ansätze gelten auch für Webservice-orientierte Transaktionen. Eine Operation eines Webservice sollte höchstens eine ACID-Transaktion enthalten. Verteilte Transaktionsmanager, die als Koordinatoren in unternehmensübergreifenden Anwendungen zusammenarbeiten, sind im Moment noch nicht praxisrelevant. Andere Ansätze müssen sich erst noch bewähren.

Ein sehr verbreiteter Ansatz ist auch der Einsatz von Queued Transactions, sofern das Interaktionsverhalten einer Anwendung dies zulässt.

Als pragmatischer Ansatz zur Implementierung von langlebigen Transaktionen kann auch ein abgewandeltes Queued-Transactions-Modell verwendet werden, bei dem die Queues in Datenbanktabellen simuliert werden. Man teilt die langen Transaktionen in kurze ACID-Transaktionen auf und schreibt jeweils die Zwischenergebnisse persistent in Datenbanktabellen. Die nächste ACID-Transaktion in der Bearbeitungskette bearbeitet die Zwischenergebnisse weiter. Hier gibt man bewusst die Isolationseigenschaft auf und muss sich in der Anwendungsentwicklung um eine geeignete Strategie zur Lösung von Konsistenzproblemen kümmern. In vielen Fällen ist dies aber möglich.

Insbesondere die Aufweichung der Isolationseigenschaft lässt sich bei Client-Server-Anwendungen in der Praxis nur schwer vermeiden, wenn man robuste

Anwendungen entwickeln möchte. Wie man aus den genannten Ansätzen erkennen kann, liegt die Entscheidung für den richtigen Einsatz von Transaktionen und für Kompensationsstrategien auch heute noch beim Anwendungsarchitekten bzw. beim Softwareentwickler.

Das Transaktionskonzept wird sicher ein wichtiges Konzept für verteilte Informationssysteme bleiben. Serviceorientierte Architekturen auf Basis von Webservices werden wohl die nächsten Diskussionen bestimmen und man muss sich überlegen, welche Transaktionsansätze hierfür wirklich gewinnbringend sind. Hier gibt es sicher große konzeptionelle Unterschiede zwischen dem Einsatz von Webservices in geschlossenen und durch ein Unternehmen kontrollierbaren Umgebungen sowie dem globalen, unternehmensübergreifenden Einsatz von SOA. Es bleibt abzuwarten, ob sich ein Standardisierungsansatz durchsetzen wird und welche Bedeutung er für die praktische Anwendungsentwicklung erlangt.

4.7 Übungsaufgaben

1. Welche Rolle spielt im DTP-Modell der Transaktionsmanager und welche Schnittstelle bietet er in Richtung Anwendungsprogramm (AP) an?
2. Wozu dient der CRM im DTP-Modell?
3. Wie startet ein AP eine Transaktion?
4. Was ist im DTP-Modell ein Thread of Control bzw. ein Ausführungspfad?
5. Erläutern Sie die beiden Phasen des 2PC-Protokolls!
6. Was bedeutet im 2PC-Protokoll ein „unsicherer Zustand“, wer kann ihn einnehmen und in welcher Situation tritt er auf?
7. Wie wird üblicherweise eine XA-Unterstützung durch den Hersteller eines DBMS realisiert?
8. Was passiert bei einer 2PC-Koordinierung zwischen einem Koordinator und zwei Teilnehmern, wenn ein Teilnehmer während der Phase 1 „ready“ an den Koordinator meldet, der zweite allerdings „not ready“. Erläutern Sie den Ablauf anhand eines Sequenzdiagramms!
9. Sollte ein PC-basierter Client als Transaktionskoordinator dienen? Begründen Sie Ihre Entscheidung!
10. Warum ist ein persistenter Logging-Mechanismus wie z.B. nach dem WAL-Prinzip für die Realisierung von (verteilten) ACID-Transaktionen notwendig?
11. Wozu muss ein Teilnehmer an einer Transaktion, der in der Phase 1 einer 2PC-Koordination mit „ready“ antwortet, in der Lage sein?
12. Welche Aufgabe hat JTA in einer JEE/EJB-Umgebung und welche Schnittstellen stellt JTA zur Verfügung?
13. Welches Transaktionsmodell unterstützt EJB?
14. Erläutern Sie den Unterschied zwischen bean-managed, container-managed und client-managed Transaktionen.

15. Wie kann ein EJB-Client nach einem Ausfall während einer Commit-Bearbeitung erfahren, ob die gerade ausgeführte Transaktion erfolgreich zu Ende geführt wurde oder nicht?
16. Können ein EJB-Container und ein .NET-Enterprise-Services-Container eine gemeinsame Transaktion ausführen und diese koordinieren? Begründen Sie Ihre Entscheidung!
17. Warum funktioniert das starre ACID-Konzept für Transaktionen mit Webservices nicht? Welches Transaktionsmodell würden Sie für eine Webservice-basierte Anwendung, die drei Webservice-Operationen unterschiedlicher Diensteprovider in einer zusammenhängenden (atomaren) Aktionsfolge nutzen muss, vorschlagen?
18. Was sind im BTP-Transaktionsstandard *Cohesive Business Transactions*?

5 Architekturen verteilter betrieblicher Anwendungen

Es ist eine der schwierigsten Aufgaben in der Softwareentwicklung, eine Architektur zu entwerfen, die den Anforderungen genügt und auch eine Weiterentwicklung des Systems gestattet. Neben den technischen Idealen sind hier natürlich andere Aspekte wie die vorhandene Entwicklungszeit, die vorgegebenen Werkzeuge und das Budget wichtige Einflussfaktoren. Architekturen können aus verschiedenen Blickwinkeln betrachtet werden. Fachliche Aspekte einer Anwendungsdomäne spielen eine ebenso große Rolle wie technische Fragestellungen. Dieses Kapitel soll die Problematik aus technischer Sicht diskutieren und stellt die IT-Architektur in den Vordergrund.

Die überwiegende Anzahl an neueren verteilten Systemen nutzt heute vorwiegend das Client-Server-Modell. Zwar kann man auch in betrieblichen Informationssystemen nicht jeden Anwendungstyp auf das Modell abbilden, aber die meisten Anwendungssysteme werden so gestaltet. Deshalb wird dieses Modell oft bei der Architekturdiskussion in den Mittelpunkt der Betrachtung gerückt. Aber auch andere Architekturvarianten bzw. Architekturstile sind möglich und für bestimmte Anwendungen sinnvoll.

Nach einer grundsätzlichen Einordnung des Begriffs „IT-Architektur“ in die Unternehmensarchitektur versuchen wir in diesem Kapitel, den Architekturbegriff mangels einer anerkannten Definition über mehrere Definitionsversuche aus technischer Sicht zu erläutern und gehen anschließend auf Architekturstile und Qualitätsmerkmale von Architekturen ein. Danach diskutieren wir Schichtenarchitekturen und wie man die Schichten im Netzwerk verteilen kann. Die Datenzugriffsarchitektur, eine spezielle Teilarchitektur, wird ebenfalls betrachtet, da diese insbesondere bei betrieblichen Informationssystemen von besonderer Bedeutung ist. Ebenso werden clientseitige Architekturaspekte insbesondere bezogen auf die Anbindung von Clientsystemen an die Serverseite sowie serviceorientierte Ansätze betrachtet.

Unser Hauptaugenmerk liegt auf der Verteilung von Softwarebausteinen in verteilten betrieblichen Informationssystemen. IT-Architekturen betrachten neben einer Bausteinzerlegung auch die für eine Umsetzung vorhandenen Konzepte und Technologien für verteilte Anwendungssysteme. Ohne das Wissen über Konzepte und Technologien fällt es schwer, geeignete Architekturen zu entwerfen. Das Grundwissen aus den vorangegangenen Kapiteln ist daher für die weitere Diskussion sehr hilfreich.

Zielsetzung des Kapitels

Der Studierende soll verschiedene Aspekte der Gestaltung von IT-Architekturen für verteilte Systeme unter Einsatz von Technologien und Konzepten verteilter Systeme kennenlernen und auf konkrete Problemstellungen anwenden können. Dabei soll er vor allem in Softwareschichten denken können.

Wichtige Begriffe

Architekturstile, IT-Architektur, Software-Architektur, Schichtenarchitektur, Service-orientierte Architektur, SOA, Middleware, Application Server, Container, Peer-to-Peer-Architekturen, Object-Relational-Mapping (ORM), JEE-Architektur, JDBC, Java Persistence API (JPA), Entity-Manager, Container-managed Persistency (CMP), Bean-managed-Persistency (BMP), Persistency Unit, Persistenzkontext, Datenzugriffsarchitektur, clientseitige Architektur.

5.1 Begriffe und Definitionen

Architektur ist in der Informationstechnik ein sehr häufig und unterschiedlich verwendeter Begriff. Betrachtet man die Gesamtarchitektur eines Unternehmens, so kann man verschiedene Architekturtypen erkennen, die zum eine fachliche und zum anderen eine technisch Sicht liefern. Keller unterscheidet in der sog. Architektur-Modellpyramide verschiedene Architekturtypen eines Unternehmens (Keller 2007), die im Rahmen der Diskussion um Unternehmensarchitekturen (Enterprise Architektur) eine Rolle spielen (Abbildung 5-1):

- *Geschäftsarchitektur*: Hierunter wird die Summe aller Geschäftsprozesse eines Unternehmens verstanden. Die verantwortlichen Instanzen (Owner) dieser Architektursicht sind die Fach- oder Organisationsabteilungen. Diese Architektursicht gibt Auskunft über die Prozesse, die durch IT unterstützt werden sollen.
- *Fach- und Informationsarchitektur:* Diese beiden Architekturtypen beschreiben die fachlichen Aspekte und die fachlichen Informationsobjekte, die von den Anwendungssystemen verwaltet werden. Die Beschreibung erfolgt hier rein fachlich, technische Aspekte werden nicht betrachtet. Anwendungssysteme und deren Zusammenspiel und die wichtigsten fachlichen Services der Systeme werden beschrieben. Die Owner dieser Architektursichten sind die Fachabteilungen.
- *IT-Architektur* bzw. *Anwendungsarchitektur*: Dieser Architekturtyp gibt die technische Sicht auf ein Anwendungssystem wieder. Eine Komponentenzerlegung wird hier beschrieben. Die eingesetzten technischen Konzepte, eine evtl. Schichtenaufteilung sowie auch der Einsatz bestimmter Technologien werden betrachtet. Owner dieses Architekturtyps ist die IT-Abteilung.
- *Systemarchitektur und Infrastrukturarchitekur:* Die Systemarchitektur beschreibt die Zuordnung der Softwarekomponenten zu physikalischen oder

virtuellen Hardwaresystemen. Man bezeichnet diese Architektur auch als IT-Basisinfrastruktur. Die Infrastrukturarchitektur gibt einen Überblick über alle Basissysteme wie Netzwerke, Serverrechner und systemnahe Softwarekomponenten (Betriebssysteme) eines Unternehmens sowie deren Zusammenspiel. In Erweiterung zur reinen Softwarebetrachtung in der Anwendungsarchitektur bezieht man in die *Systemarchitektur* auch die Hardwarekomponenten in die Architektur mit ein, es wird also das vollständige System und das Zusammenspiel aller Bausteine (Hardware, Software) betrachtet. Owner dieser Architektursichten ist die IT-Abteilung.

Wie schon die Definitionsversuche zeigen, sind die verschiedenen Architekturtypen nicht scharf voneinander trennbar, geben aber grundlegende Sichten, die für eine unternehmensweite Betrachtung sinnvoll und nützlich sind, wider. Unser Hauptaugenmerk liegt auf der Anwendungs- bzw. IT-Architektur. Auch die Fach- und die Informationsarchitektur als Schnittstellen zwischen den rein fachlichen Geschäftsmodellen und der technischen Umsetzung sind für die Schaffung ganzheitlicher Lösungen wichtig.

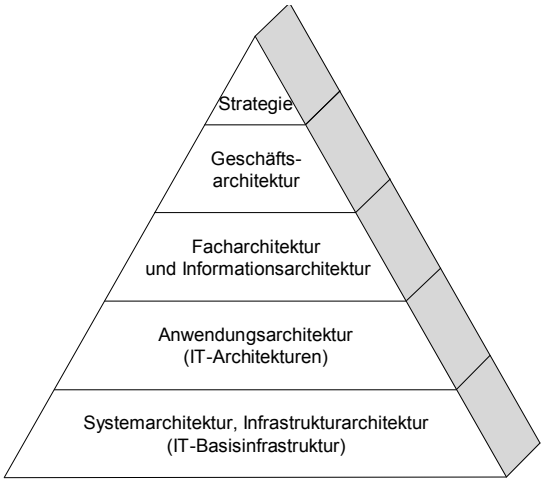


Abbildung 5-1: Architekturtypen für Enterprise Architekturen angelehnt an Keller 2007

Wir wollen in diesem Kapitel die Begriffe *IT-Architektur* bzw. *Anwendungsarchitektur* weiter erläutern und sprechen in Weiteren von Softwarearchitektur oder nur von Architektur. Beziehen wir die Hardware- und Software gleichermaßen in die Betrachtung ein, so sprechen wir von Systemarchitektur. Informatiker haben mehrere Definitionsversuche für den Begriff der Architektur unternommen, aber es hat sich keine allgemein anerkannte Definition etabliert. Daher werden vorab einige Definitionen aus verschiedenen Sichtweisen diskutiert.

Architektur als Aufbau eines Softwaresystems

Nach Kuhrmann (Kuhrmann 2004) beschreibt eine Architektur den Aufbau eines Softwaresystems einschließlich aller Softwarebauelemente und deren Beziehungen (siehe Abbildung 5-2) untereinander. Ein Softwarebauelement ist dabei ein möglichst in sich abgeschlossenes und evtl. (aber nicht zwingend) auch ein wiederverwendbares Softwarefragment, das seine Funktionalität durch eine oder mehrere Schnittstellen beschreibt. Ein Softwarebauelement kann im objektorientierten Sinne z.B. über eine Objektklasse realisiert sein, im prozeduralen Sinne über ein Modul und in der komponentenorientierten Softwareentwicklung über eine Komponente. Sehr wichtig für eine gute Architektur sind die Schnittstellen der Softwarebauelemente. Typische Architekturen verteilter Anwendungen werden weiter unten noch diskutiert.

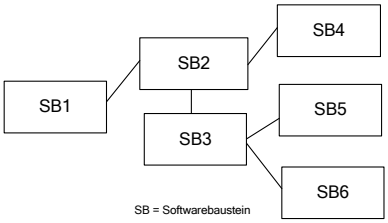


Abbildung 5-2: IT-Architektur beschreibt Bauelemente und Beziehungen

Architektur als statische und dynamische Struktur eines Softwaresystems

Andresen definiert Software-Architektur in (Andresen 2003) ähnlich und bezieht die Dynamik eines Systems mit ein (siehe Abbildung 5-3): „*Software-Architektur ist die Identifikation, Spezifikation und Dokumentation sowohl der statischen Struktur als auch der dynamischen Interaktion eines Software-Systems, welches sich aus Komponenten und Systemen zusammensetzt. Dabei werden sowohl die Eigenschaften der Komponenten und Systeme als auch ihre Abhängigkeiten und Kommunikationsarten mittels spezifischer Sichten beschrieben und modelliert. Software-Architektur betrifft alle Artefakte der Software-Entwicklung*.“

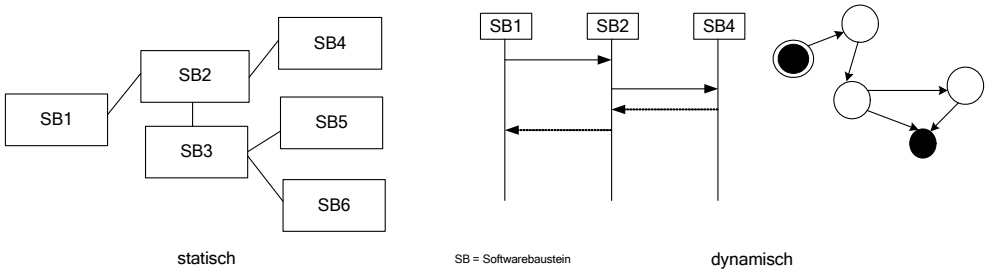


Abbildung 5-3: IT-Architektur beschreibt die statische und dynamische Struktur

Architektur als Bauplan eines Softwaresystems

Dustdar bezeichnet Software-Architektur als Bauplan für ein Softwaresystem, der die wesentlichen Elemente und Teile der Software und die wichtigsten Interaktionen der Teile beschreibt. Er räumt auch ein, dass eine einzige Darstellung für die Software-Architektur nicht möglich ist. Vielmehr sind bestimmte Aspekte wie die Kommunikation, Schichten, Module und Subsysteme in eigenen Plänen zu beschreiben (Dustdar 2003). Man muss zudem unterscheiden, ob die Architektur nur für ein bestimmtes individuelles Einzelsystem entwickelt wird, für eine Reihe gleichartiger Systeme oder gar als Referenzarchitektur für eine ganze Generation ähnlicher Software-Produkte innerhalb einer Systemfamilie dienen soll.

Architektur definiert sich über Schnittstellen

Auch Siedersleben diskutiert den Begriff der Softwarearchitektur ausführlich und stellt dabei die Schnittstellen in den Vordergrund der Betrachtung. Eine Softwarearchitektur definiert sich über die Schnittstellen (siehe Abbildung 5-4): „*Die Schnittstelle ist eine der wichtigsten Ideen des Ingenieurwesens überhaupt, aber in der Informatik hat sich der Begriff nur mühsam und bis heute noch nicht vollständig durchgesetzt.*“ *Siedersleben (Siedersleben 2004) bemängelt zum Thema Schnittstellen aber, dass es „... bis heute keine allgemein akzeptierte Art der Dokumentation für deren Semantik ...*“ gibt.

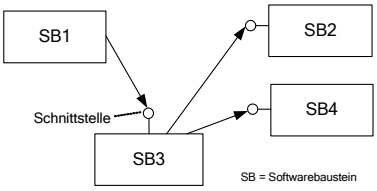


Abbildung 5-4: Schnittstellen im Mittelpunkt der IT-Architektur

Kern einer Architektur

Ein wichtiger Aspekt ist nach Siedersleben auch, dass ein Anwendungssystem um einen *Anwendungskern* herum gebaut werden sollte (Abbildung 5-5) und natürlich sollte man anwendungsspezifische Bausteine von den technischen Bausteinen weitgehend trennen. Siedersleben beschreibt dies in seiner *Qualitätssicherungsarchitektur* namens *Quasar* ausführlich. Softwarebausteine einer Architektur werden im sog. Blutgruppenmodell in fünf Gruppen eingeteilt, die als „Blutgruppen O, A, AT, R und T bezeichnet werden. T-Bausteine sind z.B. Bausteine, die von der Technik und nicht von der Anwendung abhängen, A-Bausteine werden durch die Anwendung bestimmt, sind aber von der Technik unabhängig, O-Bausteine sind sowohl von der Technik als auch von der Anwendung unabhängig, AT-Bausteine sind technik- und applikationslastig und daher nicht optimal für eine Wiederverwen-

dung geeignet, R-Bausteine transformieren fachliche Objekte in externe Repräsentationen und wieder zurück.

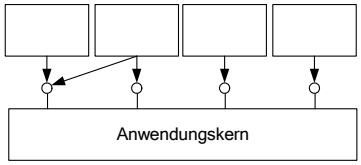


Abbildung 5-5: Anwendungskern als Mittelpunkt der IT-Architektur

Architekturen für Produktlinien und Referenzarchitekturen

Es soll noch erwähnt werden, dass die Architektur für ein individuelles Einzelsystem anders aussieht als eine Architektur für eine Produktlinie oder gar eine Referenzarchitektur. Nach Dustdar ist eine Produktlinie eine Menge von Software-intensiven Systemen, die sich eine gemeinsame Menge von Features und die speziellen Anforderungen von Markt-Segmenten teilt oder die aus einer gemeinsamen Menge von Komponenten in einer vorgeschriebenen Art und Weise entwickelt wird (Dustdar 2003). Software-Produktlinien und deren Entwicklung sind auch in (Böckle 2004) ausführlich erläutert.

Eine Referenzarchitektur zielt dagegen auf den Anwendungsbereich – die sogenannte Fachdomäne – ab und hat als Ziel der Modellierung eine Architektur, nach deren Vorgabe oder Schablone eine ganze Menge von Software-Systemen in der entsprechenden Fachdomäne erstellt werden können.

Architektursichten

Man sollte eine Architektur aus mehreren Sichten (Views) betrachten. Eine mögliche Vorgabe oder Empfehlung, wie man Architekturen dokumentiert, findet man im IEEE-Standard IEEE STD 1471-2000. Im Standard sind vor allem Systeme im Fokus, bei denen Software einen wesentlichen Einfluss auf das Design, die Konstruktion, den Einsatz und auf sonstige Belange ausübt.

In diesem Vorschlag werden verschiedene Begriffe wie *View* (Sicht), *Stakeholder* (am System Beteiligte), *Concerns* (Aspekte) usw. erläutert. Eine Sicht repräsentiert dabei ein Softwaresystem aus der Perspektive einer verwandten Menge von Aspekten. Jede Sicht stellt ganz spezifische Informationen bereit. Die einzelnen Sichten können sich dabei durchaus überschneiden, sind also nicht zwingend orthogonal. Welche Sichten dies sind, wird im Standard nicht vorgegeben.

Nach Dustdar unterscheidet man beispielsweise folgende Sichten (Dustdar 2003):

- *Konzeptionelle Sicht*: In dieser Sicht betrachtet man das System noch unabhängig von Implementierungsentscheidungen. Man skizziert die Entitäten oder Objekte des Problemraums und deren Beziehungen zueinander.

- *Modulsicht*: Diese Sicht zeigt die Struktur eines Softwaresystems in Form von Modulen (Subsystemen, Bausteinen) und deren Beziehungen zueinander. Eine Schichtenorganisation der Module kann hier erfolgen.
- *Prozess-Sicht*: Diese Sicht bildet die Sicht auf ausführbare Einheiten wie Prozesse und Threads eines Betriebssystems ab. Es wird also skizziert, in welchen Ausführungsinstanzen die Entitäten und Objekte ablaufen.
- *Physische Sicht*: Diese Sicht bildet die Softwareelemente auf die Hardware ab. Sie konzentriert sich auch auf nicht-funktionale Anforderungen wie das Leistungsverhalten, Anforderungen an die Fehlertoleranz, die Verfügbarkeit und die Skalierbarkeit. Die Elemente dieser Sicht sind vorwiegend physische Komponenten.

Wie die Beschreibung zeigt, ist eine vollständige Trennung der Sichten nicht gegeben. Sie überschneiden sich zum Teil und sind auch nicht präzise definiert.

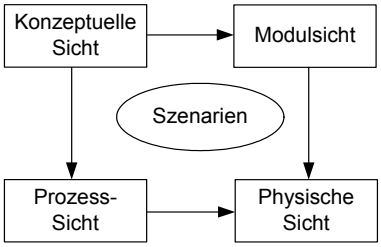


Abbildung 5-6: Architektursichten nach (Kruchten 1995)

Im sog. „4+1"-Sichtenmodell von Kruchten (Kruchten 1995) wird diesen Sichten noch eine Sicht auf wichtige Szenarien des Systems hinzugefügt. Als Szenarien sind hier Use-Cases oder Sequenzen von Interaktionen zwischen Objekten oder Prozessen zu verstehen. Diese Sichten sind in Abbildung 5-6 nochmals skizziert. Die Szenarien stehen im Mittelpunkt und werden in alle anderen Sichten mit einbezogen.

Eine andere Sichtendarstellung wird zum Beispiel von Starke vorgeschlagen und unterscheidet Laufzeitsichten, Bausteinsichten, Verteilungssichten und Kontextsichten. Diese vier Sichten werden ggf. um eine Datensicht ergänzt (Starke 2005):

- In den *Bausteinsichten* werden alle Bausteine des Systems beschrieben. Als Bausteine versteht man hier u.a. Klassen, Prozeduren, Pakete, Komponenten (nach UML-Definition) und Subsysteme. Die Sichten bilden die Funktionalität des Systems auf Softwarebausteine ab, behandeln also eine Zerlegung des Systems in einzelne Bausteine.
- In den *Laufzeitsichten* wird die Zusammenarbeit der einzelnen Bausteine skizziert, wobei sowohl Daten- als auch Kontrollflüsse beschrieben werden. Diese Sichten dienen also vorwiegend der Beschreibung dynamischer Aspekte des Systems.

- Die *Verteilungssichten* skizzieren die Ablaufumgebung des Systems und beziehen die Hardwarekomponenten mit ein. In dieser Sicht wird also dargestellt, auf welchen Rechnersystemen bestimmte Softwarebausteine ablaufen sollen und wie diese Rechnersysteme über Netzwerke miteinander verbunden sind. Man bezeichnet diese Sichten daher auch als *Infrastruktursichten*. Sie sind auch für die Betreiber und Systemadministratoren von Bedeutung.
- Die *Kontextsichten* sind Sichten höherer Abstraktionsebenen, die die anderen Sichten im Zusammenhang betrachten.
- Die *Datensichten* stellen nach Starke zwar keine eigenständige Architektursicht dar, können aber bei sehr datenzentrierten Anwendungen nützlich sein. Dies ist bei betrieblichen Informationssystemen in der Regel der Fall. In diesen Sichten kann beschrieben werden, welche Daten in welchen Softwarebausteinen benötigt werden und wo diese Daten gespeichert werden.

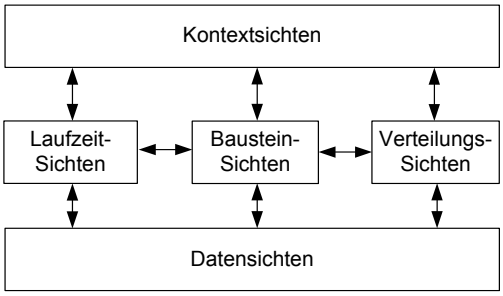


Abbildung 5-7: Architektursichten nach (Starke 2005)

Notationen zur Beschreibung der Architektur

Bei der Beschreibung von Architektursichten ist man generell sehr frei. Es gibt keine feste Notationsvorgabe. Neben in der Praxis sehr selten angewendeten Architekturbeschreibungssprachen (ADLs) wie Acme von der Carnegie Mellon University (Mehnert 2006) verwendet man in erster Linie grafische Notationen, wobei sich in den letzten Jahren UML als Beschreibungssprache durchgesetzt hat (Jeckle 2004). Die verschiedenen UML-Notationen ermöglichen die Darstellung mehrerer Sichten auf die Architektur (Komponentensicht, Verteilung, statisch, dynamisch,...). Ein UML-Verteilungsdiagramm skizziert beispielsweise wohl eher eine Sicht auf die Systemarchitektur oder die Prozesssicht, während ein UML-Paketdiagramm die Modulsicht im Auge hat. Allerdings gibt es für den Einsatz der UML-Diagrammtypen keine starren Regeln. Wichtig ist, dass die erzeugte Grafik den gewünschten Effekt, nämlich die Darstellung einer bestimmten Sicht auf das System, gut unterstützt.

Die Tabelle 3-1 zeigt Nutzungsmöglichkeiten von UML- oder sonstigen Diagrammen bei der Beschreibung von Architektursichten nach Dustdar, Kruchten und Starke. Wie man anhand der verwendbaren Notationen erkennen kann gibt es zwischen den beiden Sichtenmodellen große Parallelitäten. So entspricht die konzep-

tionelle Sicht gemeinsam mit der Modulsicht aus (Dustdar 2003) in etwa den Bausteinsichten aus (Starke 2005), die Laufzeitsicht und vor allem die Kontextsicht kann mit der Sicht auf die Szenarien nach dem „4+1"-Modell von Kruchten verglichen werden, die physische Sicht und die Prozesssicht können auf die Verteilungssichten abgebildet werden. Lediglich für die Datensicht aus (Starke 2005) gibt es in (Dustdar 2003) keine echte Entsprechung.

Tabelle 5-1: Diagramme zur Beschreibung verschiedener Sichten auf eine Architektur

Sichtenbezeichnung	**Mögliche Diagramme**
Konzeptionelle Sicht	UML-Paketdiagramme, UML-Klassendiagramme, einfache Blockdiagramme
Modulsicht	UML-Komponentendiagramme, einfache Blockdiagramme
Prozesssicht	UML-Komponentendiagramme
Physische Sicht	UML-Verteilungsdiagramme (UML Deployment-Diagramme)
Szenarien	UML-Use-Case-Diagramme, UML-Aktivitätsdiagramme, UML-Kollaborationsdiagramme bzw. UML-Kommunikationsdiagramme
Bausteinsicht	UML-Paketdiagramme, UML-Klassendiagramme, UML-Komponentendiagramme, einfache Blockdiagramme
Laufzeitsicht	UML-Use-Case-Diagramme, UML-Aktivitätsdiagramme, UML-Kollaborationsdiagramme bzw. UML-Kommunikationsdiagramme
Verteilungssicht	UML-Verteilungsdiagramme (UML Deployment-Diagramme)
Kontextsicht	UML-Use-Case-Diagramme, UML-Aktivitätsdiagramme, UML-Kollaborationsdiagramme bzw. UML-Kommunikationsdiagramme und auch die Diagrammtypen der anderen Sichten
Datensicht	Entity-Relationship-Diagramme, UML-Klassendiagramme

Man kann eine Architektur am Anfang einer Projektphase schon so exakt wie möglich definieren oder aber eine schrittweise Vorgehensweise wählen. Die erste Variante wird in der sog. Masterplan-Schule des bekannten Software Engineering Institute (SEI) der Carnegie Mellon University empfohlen. Die zweite, eher iterative,

Variante folgt dem Prinzip der kleinen Schritte und entstand im Umfeld der agilen Softwareentwicklung.

Üblicherweise wird man bei der Beschreibung der Architektur in der Praxis eher den Weg der Verfeinerung gehen und die verschiedenen Sichten parallel entwickeln. Die Diagramme werden dann von Iteration zu Iteration immer genauer. Wichtig ist aber, dass man eine erste grobe Sicht auf die Architektur schon relativ früh im Projekt erstellt. Einige nützliche Hinweise, wie man lesbare und übersichtliche Architekturdiagramme erstellt, sind in (Koning 2002) zu finden.

Resümee

Man kann aus den Definitionsversuchen ableiten, dass das Thema Architektur sehr umfassend ist. Man befasst sich auch schon sehr lange mit Architekuren, das Thema ist also nicht neu. Wichtige Erkenntnisse stammen bereits aus den 70er und 80er Jahren (Parnas 1984). System- und Softwarearchitekten müssen über sehr viel Erfahrung in Softwareentwicklungsprojekten verfügen. Sie müssen Architekturentscheidungen treffen können, bei denen implizit viele Regeln angewendet werden müssen. Architekturentscheidungen sind sehr wichtige Entscheidungen in einem Entwicklungsprojekt, die zum Teil umfangreiche Auswirkungen haben. Einer von vielen Aspekten ist der Aspekt der Verteilung von Softwarebausteinen. Wir betrachten im Folgenden einige wichtige Architekturaspekte insbesondere mit Blick auf verteilte, betriebliche Informationssysteme.

5.2 Architekturstile und Architekturqualität

5.2.1 Architekturstile

Im Laufe der Zeit haben sich einige Architekturstile für IT-Systeme entwickelt. Allerdings ist eine exakte Definition auch hier schwierig. Schwierigkeit bereitet auch die Unterscheidung eines Architekturstils von der angewendeten Technologie, da die Architektur oft auch durch die Technologie beeinflusst wird. Architekturelle Stile bestimmen aber eine Art von grundlegender, fundamentaler Struktur für ein System. Ein Architekturstil verfügt über bestimmte Eigenschaften und ist die Basis oder Vorlage, also der grobe Bauplan für ein konkretes Softwaresystem. In (Shaw 1996) ist ein Katalog von möglichen Architekturstilen zu finden. Man unterscheidet hier u.a. folgende Stile (siehe auch Abbildung 5-8:

- *Datenzentrierte Architekturen*: Hauptziel dieses Architekturstils ist die Verwaltung von Daten und der Zugriff auf die Daten. Hierunter fallen sog. *Repositories* und *Blackboards*. Repositories sind passive Datenbehälter, die heute vorwiegend in Datenbanken verwaltet werden. Ein Blackboard sendet zudem Benachrichtigungen aktiv an interessierte Clients, wenn sich Daten verändern. Die Abgrenzung einer Blackboard-Architektur von einer *Unabhängigen-Komponenten-Architektur* (siehe unten) kann schwierig sein. Wenn

die Clientprogramme in eigenen Prozessen ablaufen, handelt es sich eher um Letzteres.

- *Datenfluss-Architekturen*: Dieser Architekturstil entspricht dem klassischen Batch-Stil für die Jobverarbeitung in Großrechnern oder dem Pipes-and-Filters-Stil, wie er in Unix-Programmen zu finden ist.
- *Call-and-Return-Architekturen*: Hierunter versteht man den klassischen RPC, Hauptprogramm-Unterprogramm-Strukturen, objektorientierte Kommunikation und Schichten-Architekturen.
- *Unabhängige Komponenten-Architekturen*: Eigenständige Peers (Objekte oder Prozesse) kommunizieren hier über Nachrichten und zwar unabhängig miteinander.

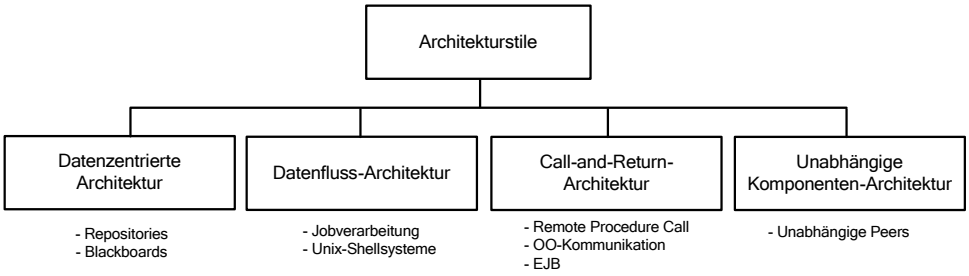


Abbildung 5-8: Architekturstile nach (Dustdar 2003)

Alle genannten Architekturstile sind in heutigen betrieblichen Informationssystemen zu finden. Architekturstile sollten aber nicht als Dogma verstanden werden, sondern dienen eher zur groben Einordnung einer Architektur. In der Regel ist die Architektur eines Anwendungssystems kaum genau einem Stil zuzuordnen. Aus der Kombination und der Integration der Stile entstehen wiederum neue heterogene Stile:

Beispiel 1: Ein web-basiertes Shop-System nutzt meistens mehrere Stile und zwar den datenzentrierten Stil, den Call-and-Return-Stil und evtl. auch den Datenfluss-Architekturstil.

Beispiel 2: COBOL- oder PL/1-basierte CICS-Hostanwendungen für Banken sind meist datenzentriert. Im Zuge der Client-Server-Erweiterung kommt hier aber immer mehr der Call-and-Return-Stil dazu. Meist verfügen diese Anwendungen über eine Dialog- und eine Batchschnittstelle.

Beispiel 3: Moderne Client-Server-Anwendungen im Bankenumfeld verfügen meist über einen Call-and-Return-Stil, aber auch Datenzentriertheit ist üblich genau so wie der klassische Datenfluss für die Nachtverarbeitung.

5.2.2 Qualitätseigenschaften

Um messen bzw. evaluieren zu können, ob eine Architektur gut oder schlecht ist, benötigt man entsprechende Kriterien bzw. Metriken oder Qualitätsattribute. Dustdar versucht, hierzu einige Regeln für die Bewertung zu definieren, von denen die wichtigsten genannt werden sollen (Dustdar 2003):

- Prinzipiell sollte die Software-Architektur ein Grundgerüst für eine spätere Implementierung festlegen, das für eine inkrementelle Entwicklung des Systems verwendbar ist.
- Die Software-Architektur muss in einer verständlichen Form und gut dokumentiert sein, so dass sie alle Beteiligten verstehen können.
- Eine Software-Architektur sollte quantitativ und qualitativ überprüfbar sein. Quantitative Prüfung bedeutet z.B. die Prüfung des maximal möglichen Durchsatzes oder der Antwortzeiten. Eine qualitative Prüfung berücksichtigt Kriterien wie die Portabilität und die Änderungsfreundlichkeit. Es sollte zudem möglich sein, über die Architekturbeschreibung mögliche Ressourcenengpässe aufzudecken.
- Hinsichtlich der Struktur einer Architektur sollten wohlbekannte Konzepte des Software Engineering eingehalten werden. Das System sollte in Bausteine (Module) zerlegt werden, deren Zuständigkeiten den Prinzipien des *Information Hiding* folgen und damit eine *Kapselung* von Daten ermöglichen. Weiterhin sollte der wichtige Aspekt der *Trennung von Zuständigkeiten* (*Separation of Concerns*) beachtet werden, um eine möglichst *lose Kopplung* bei der Entwicklung der einzelnen Module zu erreichen. Infrastruktur-Spezifika sollten in Modulen gekapselt sein. Schließlich sollte eine Architektur möglichst nicht von einer bestimmten Version eines Basisprodukts oder eines Werkzeugs abhängen, über eine hohe Änderungsfreundlichkeit verfügen und nur möglichst einfache und wenige *Interaktionsmuster* zur Kommunikation zwischen den Bausteinen nutzen.

Die letzten Aspekte betreffen vor allem die im Software Engineering seit langem bekannten technischen Grundprinzipen *Information Hiding* und *Kapselung, Separatio-of-Concerns* und *lose Kopplung*, *Unabhängigkeit* von Basisprodukten, *Änderungsfreundlichkeit und Einfachheit* (so einfach wie möglich). Diese Aspekte hängen zum Teil voneinander ab.

Das Prinzip des *Separation-of-Concerns* soll in Abbildung 5-9 nochmals verdeutlicht werden. Wie in dem skizzierten Beispiel ersichtlich ist, nutzt der Customer-Management-Softwarenbaustein den Order-Management-Softwarenbaustein über eine dedizierte Schnittstelle. Interne Änderungen in den Softwarenbausteinen bleiben verborgen.

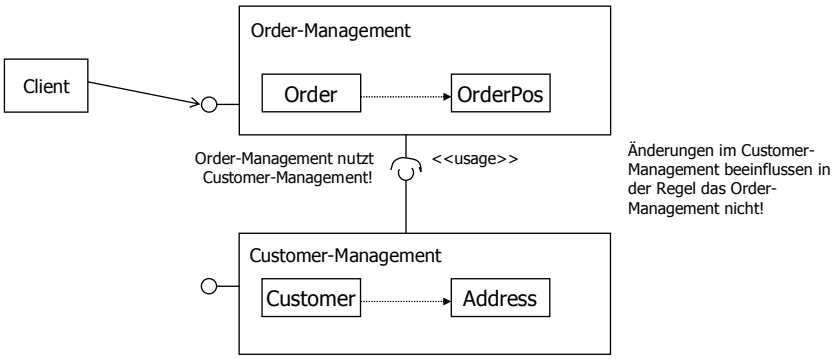


Abbildung 5-9: Separation of Concerns

Das Prinzip der hohen *Kohäsion* und der minimalen Kopplung von Softwarebausteinen soll in Abbildung 5-10 verdeutlicht werden. Die internen Bausteine *Order* und *OrderPos* des Order-Management-Bausteins sowie die internen Bausteine des Customer-Management-Bausteins *Customer* und *Address* gehören zusammen, verfügen also über eine hohe Kohäsion, während zwischen den „großen" Bausteinen nur eine geringe Kopplung vorliegt.

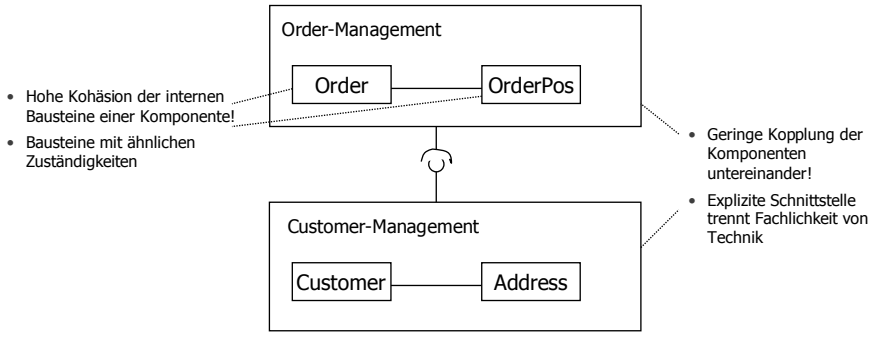


Abbildung 5-10: Kohäsion und Kopplung

Für die Umsetzung der Kapselung werden häufig *Muster* verwendet. Muster (Patterns) dienen dazu, häufig auftretende Entwurfsprobleme und entsprechende Lösungsmöglichkeiten zu skizzeren. Man unterscheidet verschiedene Varianten von Mustern. Architekturmuster beziehen sich sehr abstrakt auf die Architektur von Softwaresystemen. Entwurfsmuster (Design Patterns) sind konkreter und beziehen

sich auf die Programmierung bestimmter Sachverhalte.[1] Typisch für die Kapselung von Geschäftslogik über Dienste ist das Muster *Fassade*, das im Folgenden kurz skizziert werden soll:

Facade-Pattern: Eine Fassade (Facade-Pattern, Gama 1995) stellt ein einheitliches Interface für eine Menge von Interfaces eines Teilsystems bereit. Fassaden liefern ein höherwertiges Interface für ein Teilsystem und ermöglichen bei richtiger Anwendung eine einfachere Nutzung. Damit wird auch die Komplexität eines Systems reduziert, da Abhängigkeiten zwischen Teilsystemen auf Fassaden beschränkt bleiben. Die Reduktion der Komplexität wird in den Abbildungen (Abbildung 5-1 und Abbildung 5-12) deutlich.

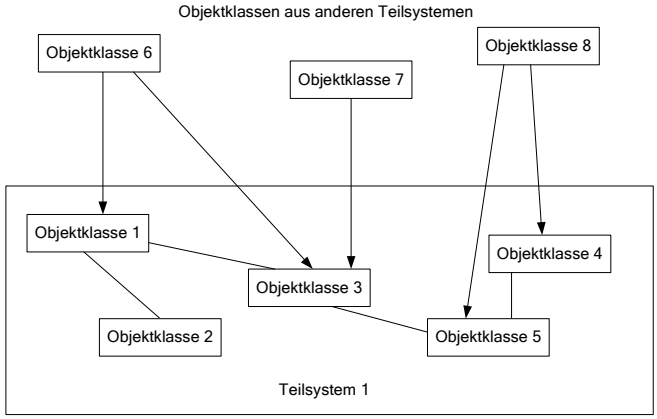


Abbildung 5-11: Nutzung eines Subsystem ohne Fassade

Fassaden werden auch in komponentenorientierten Systemen verwendet. Die Komponentenschnittstelle ist grobkörniger als die Objektschnittstelle der einzelnen Objektklassen eines Subsystems. Nur die Funktionen, die eine Komponente anbieten muss, werden durch die Fassade nach außen sichtbar, die internen Objekte werden verborgen. Die Fassade kennt und nutzt die internen Objektklassen zur Ausführung der Fassaden-Methoden.

[1] Eine Fülle von allgemeingültigen Design-Patterns ist in (Gamma 1995) beschrieben. Die Autoren werden auch als Gang of Four bezeichnet: Gamma, Helm, Johnson und Vlissides.

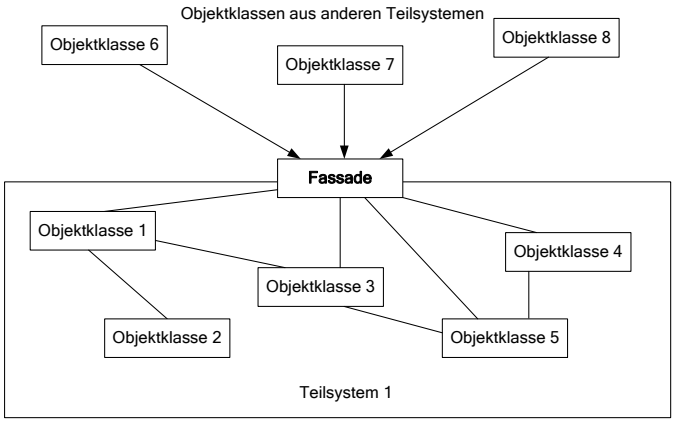


Abbildung 5-12: Nutzung eines Subsystem mit Fassade

Die konkrete Problemstellung sowie der Umfang der Aufgabenstellung sind für die zu realisierende Architektur entscheidend. Für kleine Softwareprogramme, die nur einmalig verwendet werden, wird man sicher andere Maßstäbe ansetzen als für große Systeme. Je größer allerdings ein System wird, umso wichtiger wird seine Struktur im Vergleich zu den verwendeten Algorithmen und Datenstrukturen.

Die Qualitätskriterien gelten im Prinzip sowohl für lokale als auch für verteilte Anwendungssysteme. Bei verteilten Systemen werden die Architekturen meist nur noch etwas komplexer. Nach (Dustdar 2003) unterscheidet man sichtbare und nicht beobachtbare Qualitätsattribute. „Sichtbar" bedeutet, dass das Kriterium bei der Ausführung des Systems erkennbar ist. Hierzu gehören:

- Funktionsfähigkeit
- Leistung, Skalierbarkeit
- Transaktionssicherheit
- Sicherheit (Authentifizierung, Autorisierung, Verschlüsselung)
- Verfügbarkeit
- Usability (Verwendbarkeit)

Nicht beobachtbare Qualitätsattribute sind dagegen zur Ausführungszeit nicht sichtbar. Zur Prüfung der Kriterien muss man das System im Detail betrachten. Folgende Kriterien sind von Bedeutung:

- Standardisierung und Strukturiertheit der Software (Prinzipien einhaltend)
- Portabilität
- Wiederverwendbarkeit
- Integrierbarkeit
- Testbarkeit

Weiterhin sollte nach Dustdar bei der Bewertung einer Architektur auf die Einhaltung von grundlegenden Prinizipien der Softwareentwicklung geachtet werden. Hierzu nennt er:

- Separation of Concerns
- Comprehension (intellektuelle Beherrschbarkeit)
- Korrektheit und Vollständigkeit
- Ersetzbarkeit
- Buildability
- Lose Kopplung: schmale Schnittstellen, wenig verschiedene Datenelemente an der Schnittstelle, keine Schichtensprünge wenn möglich
- Hohe Kohäsion (durch Kapselung erreichbar)
- Verringerung der Komplexität

Bei größeren Systemen, die ständig weiterentwickelt werden, ist auch die *Änderungsfreundlichkeit* (*Design-for-Change*) von großer Bedeutung. Wahrscheinliche Änderungen sollten also schon im Entwurf berücksichtigt werden. Die Design-for-Change-Anforderung wird bestimmt durch die Häufigkeit der erforderlichen Änderungen der zu entwickelnden Software.

Es soll noch festgehalten werden, dass eine gründliche Bewertung der Architekturqualität eines größeren Systems sehr aufwändig ist und auch die Projekthistorie mit einbeziehen muss. Es gibt oft Gründe im Projekt, die eine optimale Architektur behindern (Zeitaspekt, Kostenaspekt). Ein Architekt sollte also die Architekturentscheidungen gründlich dokumentieren, damit er diese später auch rechtfertigen kann.

5.3 Schichtenorientierte Architekturen

5.3.1 Schichtenarchitekturen

Nach dieser grundlegenden Architekturbetrachtung wollen wir uns nun schichtenorientierten Architekturmustern zuwenden. Insbesondere bei betrieblichen Informationssystemen werden oft mehrere logische Ebenen oder Schichten (Layer) unterschieden. Heute spricht man von mehrschichtigen Architekturen (Tanenbaum 2003). Ein Ziel ist es hierbei, für mehrere Anwendungssysteme oder auch nur Anwendungsbausteine eines Unternehmens möglichst dieselbe Business-Logik (Geschäftslogik) zu verwenden und damit die Widerverwendbarkeit zu fördrn. Darüberhinaus dient die Schichtung der Reduzierung der Komplexität. Natürlich muss man hier bedenken, dass eine komplexe Schichtenarchitektur bei kleinen Anwendungen Mehraufwand im Vergleich zu einer eher monolithischen Architektur bedeutet. Jedoch lohnt sich die Aufteilung auch schon bei Systemen mittlerer Größenordnung und insbesondere dann, wenn es nicht ein einmalig genutztes Programm werden soll, sondern weiterentwickelt werden muss.

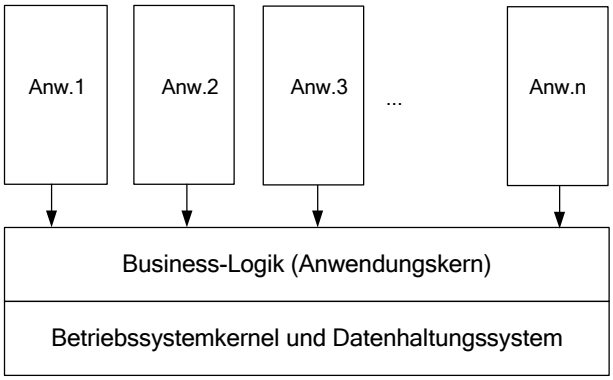


Abbildung 5-13: Typische Anwendungsarchitektur mit Middleware

In Abbildung 5-13 ist eine erste grobe Schichteneinteilung skizziert, die ein System in einen *Anwendungskern* und die (Client-)Anwendungen, die diesen nutzen, gliedert. Die einzelnen Anwendungsbausteine und der Anwendungskern werden natürlich wieder vertikal in mehrere Schichten bzw. horizontal in Bausteine zerlegt, wie dies exemplarisch in Abbildung 5-14 angedeutet ist. Die vertikalen Schichten dienen der Wiederverwendbarkeit und man kann diese beispielsweise nochmals in anwendungsspezifische und generische Basisfunktionalität einteilen.

Generische Basisfunktionalität beinhaltet eher technische Funktionen wie z.B. Trace- und Logging-Mechanismen, GUI-Basisfunktionen und Zugriffsfunktionen auf Ressourcen. Sie kann für verschiedene Anwendungssysteme eingesetzt werden und ist unabhängig von der Anwendungslogik zu sehen.

Applikationsspezifische Basisfunktionen sind Funktionen, die für eine spezielle Anwendung und ein Anwendungssystem relevant sind und nur dort sinnvoll nutzbar sind. Die Wiederverwendbarkeit ist also im Gegensatz zu generischen Basisfunktionen eingeschränkt. Typische applikationsspezifische Basisfunktionen sind z.B. spezielle Protokollierungsmechanismen, die nur in einem Anwendungssystem gebraucht werden oder anwendungsspezifische Kommunikationsfunktionen.

Bei den vertikalen Funktionen handelt es sich um Basisfunktionalität, die oft lokal in jeden Anwendungsprozess eingebunden werden. In Abbildung 5-14 sind diese Funktionen sowohl in den einzelnen Client-Anwendungen als auch im Anwendungskern zu finden. In der Abbildung ist nun auch eine Einteilung in drei grobe Schichten dargestellt. Dies sind die Präsentations-[2], die Applikations- und die Datenverwaltungsschicht. Der Anwendungskern ist damit nochmals in zwei Schichten unterteilt. Die Applikationsschicht kümmert sich um die Bereitstellung der eigentlichen Business-Logik, während sich die Datenverwaltungsschicht um das Le-

[2] Dies gilt für Anwendungen mit einem Benutzerinterface.

sen und Schreiben der Daten auf Datenbanken bzw. um den Zugriff auf andere Ressourcen in anderen Anwendungssystemen kümmert.

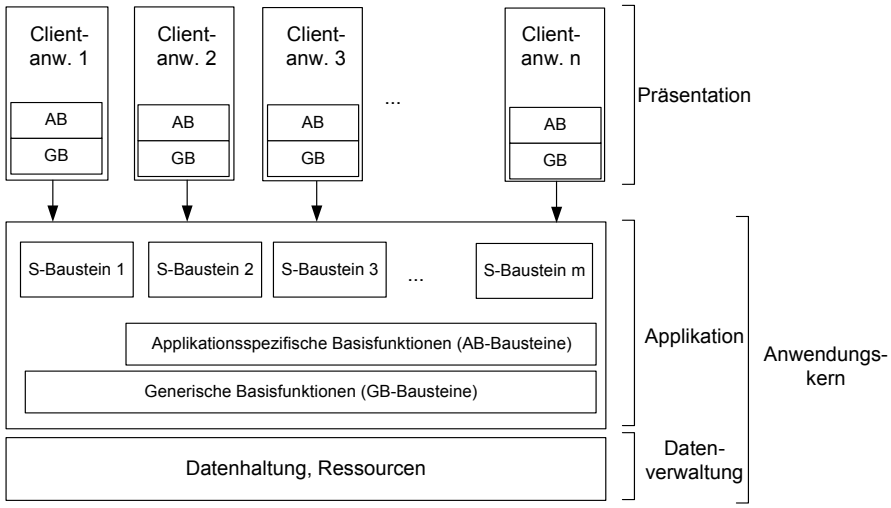


Abbildung 5-14: Verfeinerung der Business-Logik

Die angedeutete Zerlegung des Systems in einzelne Bausteine macht noch keine Aussage über die Verteilung. Die horizontalen Bausteine der Business-Logik und natürlich die einzelnen Client-Anwendungen sind Kandidaten für eine Verteilung. Applikationsspezifische und generische Basisfunktionen werden meist in die Client-Anwendungen und die Service-Bausteine eingebunden.

Diese Aufteilung in drei grobe Schichten und eine Separierung wiederverwendbarer Basisfunktionalität, die wiederum aus einzelnen Bausteinen besteht, findet man in der Praxis sehr häufig. Ausgehend von einer derartigen Zerlegung eines Systems in Schichten kann man eine Verteilung der Komponenten konzipieren.

5.3.2 Schichtenverteilung in Client-Server-Architekturen

Ein Softwarearchitekt entscheidet über die Verteilung der Softwarebausteine. Eine gute Schichtenarchitektur unterstützt eine mögliche Verteilung der Bausteine im Netz. Man kann sich über die Sinnhaftigkeit der Schichtenanordnung streiten. Sinnvoll ist es aber auch ohne Verteilung, jede logische Schicht so zu spezifizieren und auch zu implementieren, dass sie ohne große architektonische Anpassungen auf beliebigen Maschinen untergebracht werden kann. Die ersten Client-Server-Systeme hatten meist eine Zweiteilung (two-tiered) in Client- und Serverteil. Man spricht auch von vertikaler Verteilung, da die logisch übereinander liegenden

Softwareschichten auf Rechnersysteme verteilt werden. Es wird praktisch ein vertikaler Schnitt durch die Schichtenarchitektur gemacht.

Ein Schnitt durch die Architektur kann in vertikaler Richtung an verschiedenen Stellen gemacht werden. Folgende Verteilungsvarianten, die sich aus der unterschiedlichen Positionierung des Schnitts ergeben, sind möglich:

- Im Client, der meist auch auf einem Clientrechner platziert wird, liegt ein Teil der Benutzerschicht und zwar der terminalabhängige Teil. Im Serverteil liegen die Verarbeitungsbausteine sowie die Datenverwaltungsschicht. Diese Variante ist in Abbildung 5-15 (a) skizziert. Der Client übernimmt hier lediglich die Rolle eines Terminals. Die bekannteste Realisierung dieser Variante ist das X-Windows-System, das am MIT[3] entwickelt wurde und sich insbesondere in den 90er Jahren in der Unix-Welt stark verbreitete. Heute lebt diese Variante in veränderter Form bei Einsatz von Terminaldiensten bzw. Terminalservern wieder auf.
- Eine weitere Variante legt die gesamte Benutzerschnittstelle auf den Clientrechner (siehe Abbildung 5-15 (b)). Die Benutzerschnittstelle ist meist als grafische Oberfäche dargestellt. Auf der Serverseite ist die Applikations- und die Datenhaltungslogik untergebracht. Diese Variante wird auch als entfernte Präsentation bezeichnet (Dadam 1996). Man spricht in diesem Fall auch von einem *Thin-Client*.
- Variante (c) verschiebt die Applikationsschicht teilweise (Abbildung 5-15 (c)) auf den Server. Typisches Beispiel hierfür ist die Programmierung von Anwendungslogik in einer Datenbank mit sog. Stored Procedures. Dies schafft meist eine gewisse Abhängigkeit von der speziellen Syntax der vom Datenbankhersteller bereitgestellten Stored-Procedure-Sprache (Beispiel: Oracle PL/SQL ist kein Standard). Programmteile, die die Effizienz stark beeinflussen, aber auch gewisse Integritätsbedingungen, die durch sog. Trigger implementiert werden, können so realisiert werden.
- In der Variante (d) wird die Applikationsschicht ganz (Abbildung 5-15 (d)) auf den Clientrechner verlegt. Hier ist es typisch, dass der Anwendungsteil direkt über das Netzwerk mit SQL auf die entfernte Datenbank zugreift. Bei dieser Variante spricht man von einem *Fat-Client*.
- Bei der letzten Variante (siehe Abbildung 5-15 (e)) liegt die gesamte Benutzeroberfläche, die Anwendungslogik und auch ein Teil der Datenhaltung (meist die Datenbankzugriffsschicht) im Client. Beispielhaft für Anwendungen, die diese Variante nutzen, sind Fileserver-Systeme. Bei dieser Variante spricht man auch von einem *Fat-Client*.

Diese Schichtenaufteilung der Varianten (c) und (d) sind zwar heute bei größeren Neuentwicklungen nicht mehr so gebräuchlich, aber noch in vielen Anwendungen

[3] X-Windows wurde 1984 am MIT (Massachuesetts Institute of Technology) entwickelt.

vorzufinden. Sie haben den Vorteil, dass der Client direkt auf die Datenbank zugreifen kann und damit hat der Client-Entwickler auch die Transaktionssteuerung im Griff und kann die Isolationseigenschaften der Datenbank nutzen.

Es ist sinnvoll, neben den drei Schichten Präsentationsschicht[4] (Benutzeroberfläche), Applikationsschicht und Datenhaltungsschicht auch noch Zugriffsschnittstellen zu definieren. In größeren verteilten Anwendungssystemen verwendet man heute üblicherweise die Varianten (a), (b) und ggf. auch (c).

Erweitert wird die vertikale Verteilung der Schichten noch dadurch, dass die Datenverwaltung und die Applikationslogik auf verschiedenen Rechnersystemen installiert werden. Es gibt also oft einen eigenständigen Rechner, der als Datenbankserver dient. Weiterhin wird die Applikationslogik bei Bedarf auf mehrere Serverrechner verteilt, indem man dedizierte Bausteine der Business-Logik wie etwa einen Artikel-Manager oder einen Kunden-Manager definiert.

Günstige Kandidaten für die Verteilung sind aber auch die horizontalen Service-Bausteine des Anwendungskerns. Sie bilden in Client-Server-Anwendungen meist die Server, die bestimmte Services anbieten.

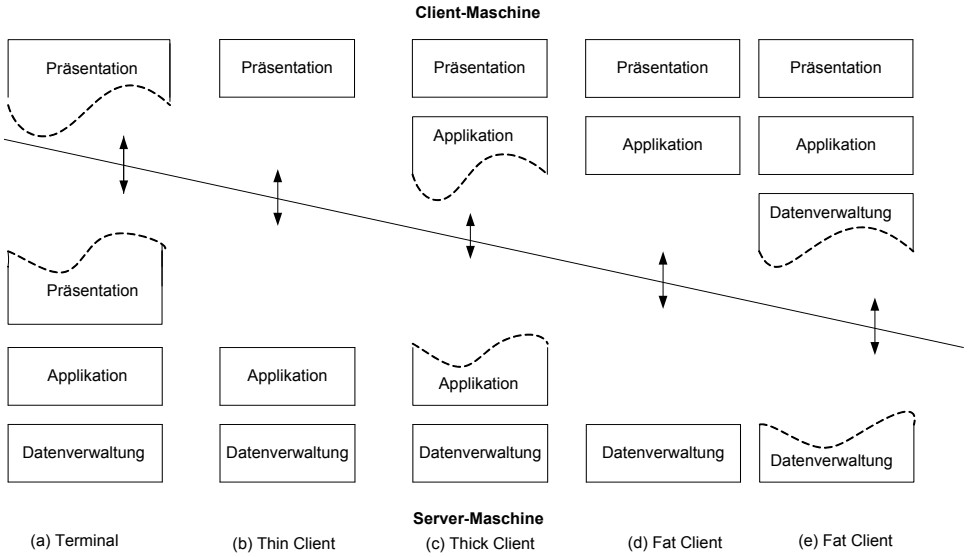


Abbildung 5-15: Client-Server-Architekturvarianten nach (Tanenbaum 2003)

Für größere Webanwendungen wird darüberhinaus noch ein eigener Webserverrechner erforderlich, der die HTTP-Requests der Web-Clients entgegennimmt. Die

[4] Wir sprechen hier nicht von klassischen Batchanwendungen, die heute immer noch sehr zahlreich in der Praxis vorhanden sind.

Zugriffe der Clients auf die Serverschicht (Applikationsschicht) werden meist über Middleware unterstützt.

Die Frage stellt sich natürlich, wann welche Verteilung sinnvoll ist. Ab wann rentieren sich z.B. mehrschichtige Architekturen und wann ist es besser die Schichten auf wenige Rechner zu konzentrieren? Diese Frage lässt sich nicht pauschal beantworten. Man muss sich immer den konkreten Fall genauer anschauen. Heutige Anwendungen vor allem im Web nutzen mehrstufige Architekturen. Im WWW wird häufig das Model-View-Controller-Modell (MVC) für die architektonische Gestaltung des sog. Web-Frontends verwendet. Der Controller ist damit die zentrale Komponente für die Ereignisbehandlung auf der Präsentationsseite, er läuft aber auf einem Webserver. Auf der „Backend-Seite“ werden für den Serverzugriff eine Anwendungsschicht sowie eine Datenhaltungsschicht benötigt.

5.3.3 Middleware im Schichtenmodell

Wie bereits erläutert, benötigen verteilte Systeme auf den beteiligten Rechnersystemen einheitliche Kommunikationsmechanismen, um Nachrichten austauschen zu können. Diese können direkt auf Basis eines Transportsystems oder auf einer darüberliegenden Middleware-Schicht aufgebaut sein.

Damit Rechnersysteme miteinander kommunizieren können, ist ein gemeinsam genutztes Transportsystem von Bedeutung. Dies deckt die Schichten 1 bis 4 gemäß OSI-Referenzmodell ab. Das von der Internet-Gemeinde entwickelte TCP/IP-Referenzmodell ist heute der Defacto-Standard in der Rechnerkommunikation. Es hat vier Schichten, wobei die Schicht 3 (Internet-Schicht) und die Schicht 4 die tragenden Schichten sind. In der Schicht 3 wird neben einigen Steuerungs- und Adressierungsprotokollen im Wesentlichen das Protokoll IP (Internet Protocol) benutzt. In Schicht 4 gibt es zwei Standardprotokolle. Das mächtigere, verbindungsorientierte TCP (Transmission Control Protocol) und das leichtgewichtige, verbindungslose UDP (User Datagram Protocol). TCP gab dem Referenzmodell seinen Namen (Mandl 2008b).

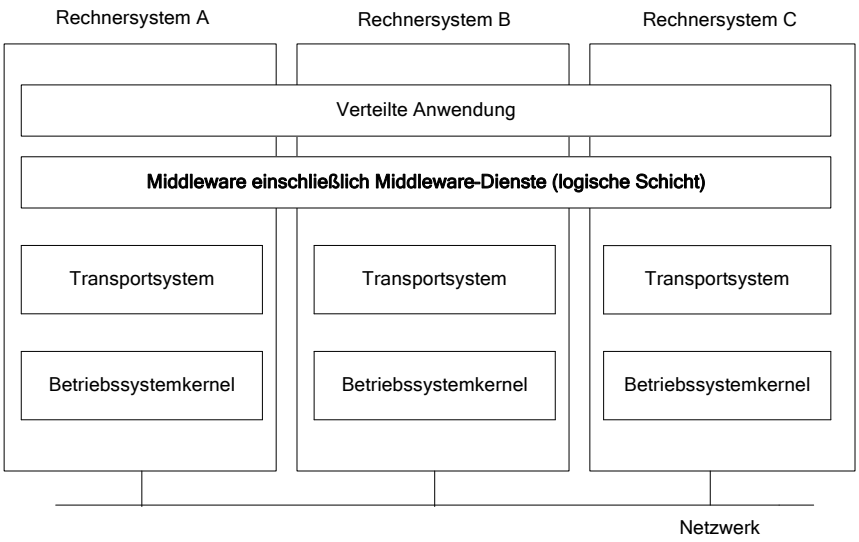


Abbildung 5-16: Einordnung von Middleware nach (Tanenbaum 2003)

Alle Datenkommunikationsspezialisten sehen die Notwendigkeit für ein Transportsystem, wenn auch die Funktionalität je nach Protokolltyp hier sehr unterschiedlich sein kann. Beispielsweise gibt es verbindungsorientierte (wie TCP) und verbindungslose (wie UDP) Protokolle mit recht unterschiedlichen Anforderungen. In den Schichten 1 und 2 legt sich das TCP/IP-Referenzmodell nicht fest. Hier wird die Anbindung an das Netzwerk gelöst und dies kann ein LAN-, ein WAN- oder ein MAN-Zugriff für jedes beliebige Netzwerk sein. Höhere Dienste für verteilte Anwendungssysteme verwenden in der Regel UDP oder TCP als Transportprotokoll. Wir interessieren uns hier vorwiegend für die Endsysteme. Zwischensysteme wie Router sind zwar für größere Netze unbedingt erforderlich, aus Sicht der verteilten Anwendung oder der Anwendungsentwickler aber transparent.

Die Zugriffsschnittstellen auf TCP/UDP-Dienste wie die Sockets-Schnittstelle bieten dem Anwendungsprogrammierer relativ wenig Komfort. Höhere Kommunikationsmechanismen brauchen daher zusätzliche Funktionalität, die von der Schichtenanordnung her über dem Transportsystem angesiedelt ist. Man spricht hier von Kommunikations-Middleware[5] (kurz Middleware). Die Einordnung der Middleware ist in Abbildung 5-16 skizziert. Eine Arbeitsdefinition für Middleware, so wie wir sie in unserem Zusammenhang der Kommunikation in verteilten Systemen verstehen, lautet wie folgt:

[5] Es gibt auch Middleware für andere Zwecke, z.B. für den Datenbankzugriff.

Definition Middleware: Middleware ist eine Softwareschicht, die den Anwendungen standardisierte, höhere Kommunikations- und sonstige Dienste bereitstellt und damit die transparente Kommunikation von Softwarebausteinen verteilter Systeme unterstützt. Sie liegt von der Schichtenanordnung zwischen den verteilten Anwendungen und dem Transportsystem und ist Bestandteil eines verteilten Systems. Middleware versteckt eine mögliche Heterogenität der an einem verteilten System beteiligten Rechnersysteme vor den eigentlichen Anwendungskomponenten, die die Geschäftslogik präsentieren und stellt den Anwendungen bzw. deren Entwicklern eine Programmierschnittstelle zur Verfügung.

Middleware befindet sich also zwischen den Softwarebausteinen der Anwendung und dem Transportsystem. Bausteine der Middleware sind auf jedem Rechnersystem, dass an einem verteilten System beteiligt ist, zu installieren.

Man erwartet von einer Middleware die Unterstützung einer oder mehrerer der bereits in Kapitel 2 diskutierten Kommunikationsparadigmen (RPC, Verteilte Objekte, verteilte Komponenten, Messaging). Middleware-Systeme, die sich auf Messaging spezialisieren, werden auch als *Message Oriented Middleware (MOM*) bezeichnet. Weiterhin sollte eine Middleware bestimmte Standarddienste bereitstellen. Es gibt zwar keine Norm hierfür, aber typisch sind neben den Kommunikationsdiensten je nach Ausprägung folgende Standarddienste:

- Verzeichnis- oder Namensdienste (Directory Service) zum Auffinden bestimmter Adressen von verteilten Bausteinen
- Transaktionsdienste (Transaction Service) zur Unterstützung verteilter Transaktionen. Dies ist schon ein „High-End"-Service, der nicht von jeder Middleware unterstützt wird.
- Persistenzdienst (Persistency Service) für den Zugriff auf dauerhaft gespeicherte Informationen z.B. in Datenbanken. Dieser Dienst wird nur von größeren Middleware-Lösungen, oft auch als separate Datenbankzugriffs-Middleware, bereitgestellt.
- Sicherheitsdienst (Security Service) für die Authentifizierung und Autorisierung der Benutzer sowie die Verschlüsselung von Daten

Weiterhin unterstützt Middleware oft auch das Deployment (die Inbetriebnahme) eines Systems, die Skalierbarkeit und die Lastverteilung und stellt eine komplette Infrastruktur für den Betrieb bereit.

Bezieht man Middleware in die Schichtenbetrachtung eines Referenzmodells mit ein, ergibt sich in etwa folgende Schichtenbildung gemäß OSI-Referenzmodell (siehe Abbildung 5-17). Die Middleware liegt also oberhalb der Transportschicht und unterhalb der Verarbeitungsschicht. Die Instanzen der Middleware sind auf allen Rechnersystemen vorhanden und kommunizieren über ein eigenes Middleware-Protokoll.

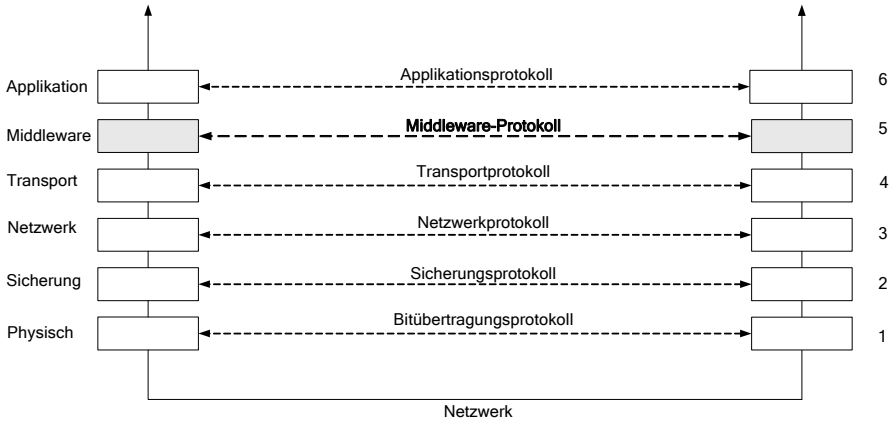


Abbildung 5-17: Referenzmodell einschließlich der Middleware-Schicht nach (Tanenbaum 2003)

Nach heutigem Stand der Technik sind Middleware-basierte verteilte Systeme für betriebliche Systeme State-of-the-Art. Einige Middleware-Produkte sind weitgehend Betriebssystem-unabhängig. Hier sind z.B. EJB-Application-Server einzuordnen. Andere sind auf spezielle Betriebssysteme fokussiert (siehe Microsoft .NET). Trotzdem sprechen alle Hersteller von der „Offenheit" ihrer Produkte. Ob die Produkte nun offen im Sinne offener Systeme sind oder nicht, kann den Marketing-Aussagen nicht so ohne weiteres entnommen werden und hängt auch von den Anforderungen und der IT-Landschaft des Anwenders ab. Man bezeichnet Middleware in der Regel als „offen", wenn sie nicht nur von einem Hersteller angeboten wird und sich an etablierte Standards hält (Beispiel: CORBA-basierte, EJB-basierte Middleware). Aber auch Microsoft bietet seine Middleware-Plattformen als offene Systeme an. Die Unabhängigkeit von einem Betriebssystem ist allerdings schon ein wichtiger Aspekt, der bei Microsoft-Produkten meist nicht gegeben ist. Bei EJB-basierter Middleware legt man sich allerdings auf die Programmiersprache (Java) fest, die bei .NET wiederum frei aus dem Pool der unterstützten Sprachen wählbar ist.

5.4 Service-orientierte Architekturen

5.4.1 Grundlegendes

Der Begriff *Serviceorientierte Architektur* (SOA), auch als dienstorientierte Architektur bezeichnet, wird in letzter Zeit viel diskutiert (Krafzig 2005). Eine genaue Begriffsdefinition fehlt noch. Wir sehen es als ein Architekturkonzept, an anderer Stelle wird es auch in den Bereich der Geschäftsprozesse eingeordnet, da die Geschäftsprozesse für ihre Umsetzung in einer IT-Umgebung Dienste nutzen, die

technisch von einer SOA-Infrastruktur bereitgestellt werden. Wesentlicher Vorteil der SOA ist die Trennung der Implementierung eines Service von seiner Implementierung. Dadurch wird ein Service unahängig von der Plattform und auch von der Technologie.

Ein Service stellt eine vorgegebene Funktionalität bereit, die über eine standardisierte Schnittstelle verwendet werden kann und in einem beliebigen Softwarebaustein (heute in der Regel in einer Softwarekomponente) implementiert wird. Die Implementierungen benötigen eine Ablaufumgebung (Service-Provider). Man kann sich das auch so vorstellen, dass Geschäftsprozesse durch eine Folge von Dienstaufrufen modelliert werden. Dies nennt man gelegentlich auch „Komposition von Services“. Die Komposition resultiert in einer Anwendung, die einen Geschäftsprozess realisiert. Die einzelnen Dienste sind idealerweise lose gekoppelt und hinterlassen nach der Abarbeitung keinen Zustand zurück.

Serviceorientierte Architekturen nutzen typischerweise das klassische Client-Server-Modell. Auch in serviceorientierten Architekturen muss ein Client die Diensteimplementierungen finden. Das allgemeine Konzept stellt ein Dienstregister bereit, das der Dienstelieferant mit Informationen zu den angebotenen Diensten befüllt. Ein Client holt sich zunächst aus dem Dienstregister die Informationen zum Dienst, muss also nicht wissen, wo er genau implementiert ist und kann mit diesen Informationen den Dienst aufrufen.

Das Konzept der serviceorientierten Architektur wurde auch insbesondere mit den bereits erläuterten Webservices wieder aufgegriffen. Ein Diensterbringer (Service Provider) stellt hier Webservices bereit, welche die Dienste darstellen. Ein Client ruft einen Webservice über ein standardisiertes Protokoll (hier SOAP mit XML für die Präsentation) auf und die Webservices sind in einer festgelegten Sprache (WSDL) beschrieben.

Bei der Implementierung eines Dienstes kapselt man einen zusammengehörigen Teil einer Geschäftslogik. Heute muss man oft davon ausgehen, dass Geschäftslogik bereits in einem bestehenden Anwendungssystem programmiert ist und abläuft, d.h. eine Umstellung auf eine SOA-Architektur verursacht zunächst zusätzlichen Aufwand. Dieser Aufwand muss den Vorteilen, die sich durch eine Kapselung der Geschäftslogik ergeben, gegenübergestellt werden. Wir betrachten daher hierzu im Folgenden einige SOA-Nutzungsvarianten.

5.4.1 Architekturvarianten

Serviceorientierte Architekturansätze gehorchen prinzipiell dem Client-Server-Modell und man kann sie für mehrere Aufgabenstellungen nutzen. Die einfachste Form der Nutzung ist in Abbildung 5-18 dargestellt. SOA wird in diesem Fall nur für eine Anwendung genutzt, um einen Anwendungskern zu kapseln. Diese Form der Nutzung stellt die Minimalnutzung dar und spiegelt nicht die Grundabsicht

von SOA wider. Die skizzierte SOA-Schicht ersetzt einen proprietären Dienstezugang und ist durchaus eine sinnvolle Anwendung des SOA-Konzepts.

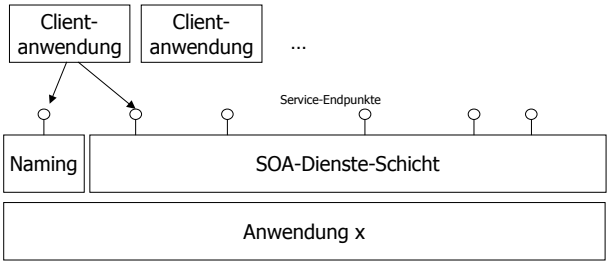


Abbildung 5-18: SOA-Architekturen, Variante 1

Eine fortgeschrittenere Variante zeigt Abbildung 5-19. Hier werden bereits mehrere Anwendungen durch eine SOA-Schicht gekapselt. Neue Client-Anwendungen können die bestehenden Anwendungssysteme über die Service-Schnittstelle nutzen. Um möglichst unabhängig von Programmiersprache und Technologie zu sein, dürfen an der Schnittstelle keine Sprachabhängigkeiten wie serialisierte Java-Objekte an der Schnittstelle verwendet werden.

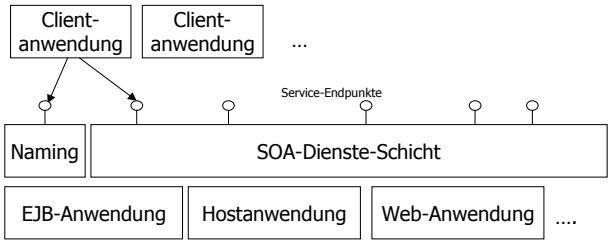


Abbildung 5-19: SOA-Architekturen, Variante 2

Die aufwändigste SOA-Variante ist in Abbildung 5-20 skizziert. Die Abbildung zeigt eine unternehmensübegreifende Architektur und stellt heute wohl noch Zukunftsmusik dar. Mehrere selbstständig agierende Dienste-Provider stellen für verschiedene Anwendungen unterschiedliche Dienste bereit. Für einen ordnungsgemäßen Betrieb einer derartigen Architektur sind Verträge zwischen den Service-Providern und den Nutzern notwendig, die die Qualität der Dienste (Quality of Service) regeln.

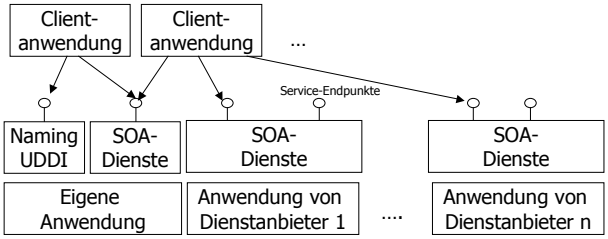


Abbildung 5-20: SOA-Architekturen, Variante 3

Diese Art von SOA-Architektur ist nicht für alle Anwendungen geeignet. Sehr unternehmenskritische Anwendungen bleiben sicher bis auf absehbare Zeit in der vollständigen Kontrolle eines Unternehmens. Welche Dienste auf diese Art und Weise tatsächlich angeboten werden können, ist noch weitgehend offen. Denkt man aber an die heutige Internet-Landschaft, so lassen sich durchaus schon einige Dienste finden, die von speziellen Providern angeboten werden können und vereinzelt auch schon angeboten werden. Meist handelt es sich aber heute um Auskunftsdienste, wie etwa Adressauskunft oder Bonitätsprüfung. Aber auch Payment-Gateways für verschiedene Bezahlsysteme im E-Commerce werden schon vielfach über Services, im speziellen über Webservice-Implementierungen, angeboten.

5.5 Spezielle Architekturen verteilter Systeme

5.5.1 Peer-to-Peer-Architekturen

Eine recht neue Architekturvariante, die als kombinierter Stil aus *unabhängigen Komponenten* und *Call-and-Return-Architektur* betrachtet werden kann, ist die Peer-to-Peer-Architektur. Die Verarbeitung wird im Gegensatz zum Client-/Server-Modell auf die Endgeräte (Peers) verlagert.

Ein Peer ist ein Netzknoten, der sowohl als Client als auch als Server agiert. Jeder Peer ist also ein potenzieller Server und stellt Dienste bereit. Die Peers sind nicht von vorneherein bekannt, wie dies etwa bei Servern im Client-Server-Umfeld der Fall ist. Ein Peer verwaltet auch eigene Ressourcen und erlaubt den Zugriff darauf. Da Peer-to-Peer-Architekturen durch die Anforderungen aus dem Internet entstanden sind, bezeichnet Dustdar diese Architekturvariante auch als „Internet auf Applikationsebene über dem Internet“ (Dustdar, 2003).

Bei Anwendung der Peer-to-Peer-Architektur ergeben sich aufgrund der Fexibilität und der Möglichkeiten der Fehlertoleranz einige Vorteile im Hinblick auf die Skalierbarkeit. Grundgedanke der Architekturvariante ist ohnehin die Skalierbarkeit eines Systems. Wenn also eine datenzentrierte Client-Server-Architektur etwa

aufgrund immenser Benutzerzahlen nicht mehr ausreichend skalierbar ist, könnte eine Peer-to-Peer-Architektur eine Alternative darstellen.

Schill unterscheidet drei Varianten von Peer-to-Peer-Systemen (Schill 2007):

- Pure (reine) Peer-to-Peer-Systeme, die über keinen zentralen Server verfügen, Beispiel: Filesharing-Dienst *Gnutella*
- Hybride Systeme, Beispiel: Filesharing-Dienst *Napster*
- Superpeer-Systeme, die über spezielle Peers, die sog. Superpeers verfügen, Beispiel: Filesharing-Dienst *KaZaa*

Auch bei Peer-to-Peer-Systemen müssen Dienstimplementierungen, die einzelne Peers bereitstellen, im Netzwerk gefunden werden. Bei reinen Peer-to-Peer-Systemen erfolgt die Suche nach Diensten nicht über eine zentrale Instanz (zentrales Verzeichnis), sondern beispielsweise über den sog. Flooded-Requests-Algorithmus, bei dem die Dienstsuche über einen Broadcast direkt an alle Nachbar-Peers erfolgt, die ihrerseits an ihre Nachbarn weiterverteilen usw.

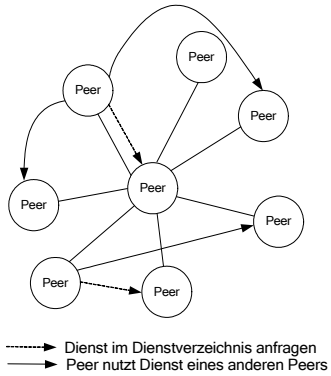


Abbildung 5-21: Reine (pur) Peer-to-Peer-Systeme

In ihrer Reinform (siehe Abbildung 5-21) verfügen Peer-to-Peer-Systeme über folgende Eigenschaften:

- Es gibt keine zentrale Koordination, die Peers koordinieren sich vielmehr verteilt.
- Es gibt keine zentrale Datenbasis, kein Peer kennt also alle Ressourcen (Daten bzw. Objekte) des Systems. Diese sind auf die Peers verteilt.
- Peers arbeiten autonom.
- Peers und Verbindungen zwischen den Peers müssen nicht zuverlässig sein.

Bei hybriden Peer-to-Peer-Systemen (siehe Abbildung 5-22) gibt es einen zentralen Verzeichnisserver für die Dienstsuche. Peers registrieren sich dort. Die zentrale Infrastruktur bietet dann nur begrenzte Möglichkeiten der Skalierung.

Die Variante mit den Superpeers bildet eine Mischform aus der reinen und der hybriden Variante. Superpeers (siehe Abbildung 5-23), bieten spezielle Informationen oder Dienste an, die nicht jedes Peer hat. Einige Superpeers können z.B. die Information über die Peers, welche Dienste anbieten, bereitstellen. Diese Variante ist etwas besser skalierbar als die hybride Form.

Für Peer-to-Peer-Architekturen gibt es vielfältige Anwendungsmöglichkeiten:

- Verteilte Daten- und Verzeichnisdienste
- Kollaborative Systeme
- Parallelisierung von rechenintensiven Operationen
- Parallelisierung der Bearbeitung großer Datenbestände, z.B. Parallelisierung von Batchverarbeitungen oder Nachtverarbeitungen
- Verteilte Nutzung von Ressourcen, die sonst nicht zugänglich wären (Musikdateien, Videos,…)
- Generell Systeme, die aus Skalierungssicht an ihre Grenzen stoßen, da sie zentral organisiert sind

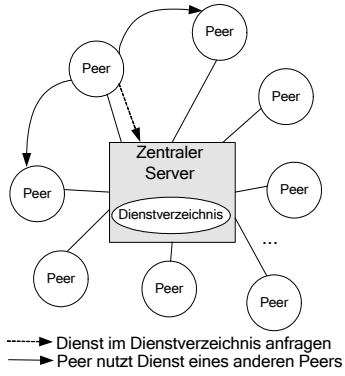


Abbildung 5-22: Hybride Peer-to-Peer-Systeme

Allerdings muss man bedenken, dass in betrieblichen Informationssystemen eine zentrale, persistente Datenhaltung oft unbedingt erforderlich ist. Die Aufgabenstellung muss also eine Verteilung der Ressourcen auf unabhängige Peers zulassen.

Es gibt auch schon Basissysteme zur Entwicklung von Peer-to-Peer-Systemen. Ein Beispiel hierfür ist P-Grid, eine offene Peer-to-Peer-Plattform, die im P-Grid-Projekt (WWW-029) entwickelt wird. Bekannte Peer-to-Peer-Systeme aus der Praxis sind Napster, Gnutella und Freenet.

Napster: Napster war das erste Peer-to-Peer-System im Internet. Aufgabe von Napster war die Bereitstellung einer Datenbank mit Musikdateien, die mit IP-Adressen von Peers, auf denen sie verfügbar waren, verküpft wurden. Die Suche erfolgte aber über einen Napster-Server. Insofern ist Napster nur ein hybrides Peer-to-Peer-System. Weitere Informationen sind in (WWW-030) zu finden.

Gnutella: Aufgabenstellung von Gnutella ist das Tauschen von Waren. Gnutella ist ein reines Peer-to-Peer-System. Es handelt sich um eine Tauschbörse, in der vor allem Dateien (z.B. Audiodateien) ausgetauscht werden. Gnutella verfügt über ein eigenes Protokoll. Die Software wurde im Jahre 2000 im Internet frei verfügbar gemacht. Eine Weiterentwicklung (Gnutella-2) des ursprünglichen Gnutella-Netzwerks nutzen heute weit mehr als zwei Millionen Nutzer. Weiterführende Erläuterungen zu Gnutella sind in (WWW-031) zu finden.

Freenet: Freenet ist ein System zur dezentralen Speicherung von Publikationen mit einer Suchinfrastruktur zum Auffinden von Inhalten über das Internet. Freenet ist ein reines Peer-to-Peer-System. Weitere Ausführungen zu Freenet sind in (WWW-032) zu finden.

Weitere Peer-to-Peer-Anwendungen wie *Chord*, *Pastry* und *Tapestry* sowie *P-Grid* können in (Mahlmann 2007) nachgelesen werden.

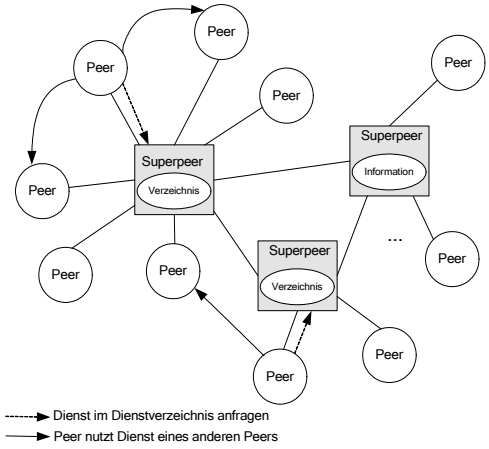


Abbildung 5-23: Superpeer Peer-to-Peer-Systeme

5.5.2 Push- und Event-basierte Architekturen

Push- und Event-Architekturen benötigt man in erster Linie zum aktiven Verteilen von Informationen. Während Push-Systeme eher im Gegensatz zum klassischen Pull-Verfahren für eine zeitgerechte Verteilung vorklassifizierter Informationen sorgen soll, sind Event-Architekturen dazu da, bestimmte Ereignsse oder Ereignismuster bei deren Auftreten an Interessierte zu kommunizieren. Beide Typen von Architekturen sind sich sehr ähnlich. Ein Vergleich der Eigenschaften dieser Spezialarchitekturen wird in (Dustdar 2003) versucht.

Ein Push-System besteht aus Konsumenten (Subscribern) und Produzenten (Pubishern), die über Informationskanäle miteinander verbunden sind. Der Verbindungsaufbau erfolgt über eine Subscriptionsphase. Push-Architekturen eignen sich

beispielsweise für Nachrichtenagenturen wie Reuters, die aktuelle Nachrichten verteilen.

BlackBerry-System: Das System von der Firma *Research in Motion* (RIM) dient der Verteilung von E-Mails an drahtlose Geräte über ein spezielles BlackBerry-Protokoll. Die Verteilung erfolgt über den BlackBerry Enterprise Server. Der Server muss an das Mailsystem eines Unternehmens angebunden werden, um die E-Mails entgegennehmen zu können. Der Server überwacht die eingehenden E-Mails und leitet diese an ein Routing-System, das *Mobile Routing Center* der Firma RIM weiter, welches dann die Verteilung an die BlackBerry-Handhelds vornimmt. Weitere Informationen finden sich unter (WWW-033).

Bei Event-basierten Systemen steht eher die Kommunikation lose gekoppelter, unabhängiger Komponenten im Vordergrund. Jeder Teilnehmer kann Events erzeugen und an die Infrastruktur des Event-Systems weiterleiten. Event-basierte Systeme eignen sich auch für die Verteilung von Börsenkursen (siehe Reuters-Informationssysteme).

5.5.3 Grid-Architekturen

Die Idee von Grids (Gitter, Netz) ist es, Computerleistung wie Strom (Power Grid = Stromnetz) aus der Steckdose nur über das Internet oder Intranet bereitzustellen. Der Grid-Computing-Ansatz ist, was die Verteilung anbelangt, mit dem Peer-to-Peer-Ansatz verwandt. Nach (Seidenfaden 2003) handelt es sich sogar um ein Anwendungsgebiet des Peer-to-Peer-Ansatzes. Massiv verteilte Ressourcen sollen für eine bestimmte Aufgabe, meist für aufwändige Berechnungen oder die Analyse von sehr großen Datenmengen, zur Verfügung gestellt werden. Beispielsweise werden PCs aus dem Homeoffice-Bereich in Forschungsprojekten über das Internet zu einem Grid verbunden, um aufwändige Berechnungen durchzuführen. Damit nutzt man die im Internet im Überfluss vorhandene Ressource „CPU“, die in den PCs ohnehin brach liegt, sinnvoll aus. Nach (Mahlmann 2007) ist der Hauptunterschied zu Peer-to-Peer-Architekturen, dass Grid-Architekturen zentral administriert werden. Die Koordination einer Berechnung im Grid erfolgt zentral durch einen Rechner.

Die Nutzung von Grids für betriebliche Probleme steht noch am Anfang und bedarf weiterer Forschungsaktivitäten. Aber auch in der Praxis sind bereits einige Ansätze sichtbar. Beispielsweise versuchen einige Großfirmen komplexe Batchabläufe, die meistens nachts ablaufen, durch Grid-Ansätze zu parallelisieren. Für die Zukunft erwartet man, dass komplexere Dienstleistungen (Grid-Services) innerhalb eines Grids angeboten werden.

Eine Standardisierungsbemühung für Grid-Architekturen ist die *Open Service Grid Architecture* (OSGA) des *Global Grid Forums*, die auf der sog. *Open Service Grid Infrastructure* (OSGI) basiert. Grids sind eine Variante des Architekturstils *unabhängiger Komponenten*. Man spricht von *Service-Grids*, wenn als Kommunikationsbasis

eine Webservice-Infrastruktur verwendet wird. Die Webservices werden hier von den am Grid beteiligten Rechnersystemen angeboten. Weitere Informationen zum Thema Grid-Computing sind in (Schill 2007) und (Mahlmann 2007) zu finden. Interessantes zu verschiedenen Grid-Forschungsprojekten kann man in (WWW-034) nachlesen.

5.6 Datenzugriffsarchitekturen

Bisher haben wir die grobe Schichtenaufteilung einer verteilten Anwendung kennengelernt. Wie die interne Architektur einer Schicht realisiert werden kann, wurde bisher weniger betrachtet. Die Frage nach einer guten Umsetzung hängt stark mit der Schnittstelle, die eine Schicht bereitstellt, zusammen. Aber auch der Aufwand für eine vollständige Kapselung muss bedacht werden. Wir wollen aufgrund ihrer Bedeutung für betriebliche Informationssysteme die Datenhaltungs- oder Persistenzschicht im Weiteren etwas näher betrachten.

5.6.1 Grundlegendes zum Datenbankzugriff

In datenzentrierten betrieblichen Informationssystemen hat man es meistens mit einer umfangreicheren Datenhaltung zu tun, die heute in der Regel über relationale Datenbanksysteme abgewickelt wird. Gerade beim Zugriff auf die auf einem externen Medium liegenden Daten entstehen bei größeren Anwendungen die meisten Probleme hinsichtlich der Leistung. Andererseits ist die Datenbankzugriffsschnittstelle möglichst gut in die Architektur eines Softwaresystems einzubauen. Bei der Schichtenbetrachtung haben wir uns schon mit der Einordnung der persistenten Datenhaltung beschäftigt und haben festgestellt, dass es sich insbesondere für größere Systeme lohnt, diese Funktionalität in einer eigenen Schicht zu kapseln. Nun gibt es aber verschiedene Möglichkeiten, dies zu realisieren. Im Weiteren wollen wir einige dieser Möglichkeiten betrachten und uns dann einigen Fallbeispielen für die Realisierung von Datenbankzugriffsschnittstellen widmen.

5.6.2 Object-Relational Mapping und deren Varianten

Relationale Datenbanken bieten als Zugriffssprache in der Regel SQL an. Dies ist eine deskriptive Sprache, die nicht ohne weiteres in einer prozeduralen oder objektorientierten Sprache verwendet werden kann. Die Datenorganisation erfolgt in relationalen Datenbanken in Form von Tabellen und Spalten. Da heutige Anwendungen meist objektorientiert programmiert werden, entsteht ein sog. *Impedance Mismatch*, der über einen Abbildungsmechanismus (siehe Abbildung 5-24) ausgeglichen wird. Dieser wird auch als Object-Relational-Mapping (ORM) bezeichnet.

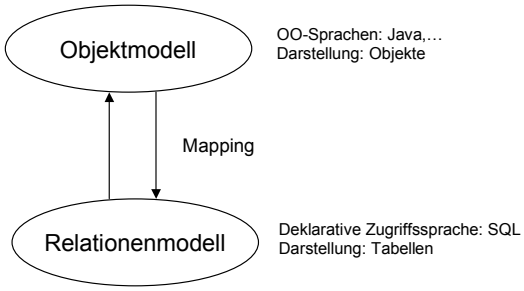


Abbildung 5-24: Mapping zwischen Objekt- und Relationenmodell

Es gibt mehrere Konzepte, die festlegen, wie man Objektklassen auf Tabellen bzw. Objekte auf Tupel in den Tabellen abbilden kann.

Mapping von Objektklassen auf Tabellen: Die einfachste Variante ist es, eine Objektklasse auf eine Tabelle abzubilden. Jedes Objektattribut wird eine Spalte in der Tabelle. Ein Objekt wird ein Tupel in der Tabelle. Die sprachabhängigen Objekt-Ids der Objekte können nicht direkt auf Primärschlüssel in der Datenbank abgebildet werden. Vielmehr ist eine persistente, eindeutige Objekt-Id als künstlicher Primärschlüssel sinnvoll. Dieser muss beim Mapping erzeugt werden.

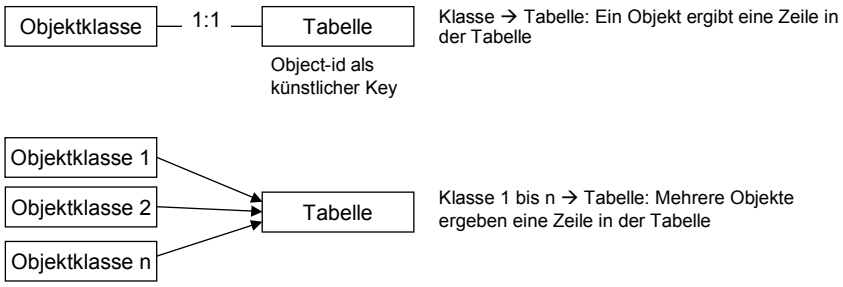


Abbildung 5-25: Mapping von Klassen auf Tabellen

Man kann aber auch mehrere Objektklassen auf eine Tabelle abbilden. Beispielsweise könnte eine Klasse *Kunde* und eine zugehörige Klasse *Adresse* auf eine Tabelle *Kunde* abgebildet werden, wenn ein Kunde nur eine oder wenige Adressen hat. Die Abbildungsvarianten sind in Abbildung 5-25 skizziert. Eine Verletzung von Normalisierungsregeln wird dabei wegen einer Leistungsoptimierung bewusst in Kauf genommen.

Mapping von Vererbungsbeziehungen: Vererbungsbeziehungen können im Relationenmodell nicht ohne weiteres dargestellt werden. Je nach Komplexität können

Klassen, die in einer Vererbungshierarchie stehen auf eine oder mehrere Tabellen abgebildet werden.

Man spricht von *Single-Table-Inheritence*, wenn alle Klassen einer Vererbungshierarchie einschließlich der abstrakten Klassen auf eine Tabelle abgebildet werden. Die Attribute aller Klassen werden dann zu Spalten dieser Tabelle. Bei *Class-Table-Inheritence* wird jede Objektklasse der Vererbungshierarchie auf eine Tabelle abgebildet. Bei *Concrete-Table-Inheritence* werden nur die konkreten und nicht die abstrakten Klassen auf eigene Tabellen abgebildet. Die verschiedenen Abbildungstypen sind in Abbildung 5-26 dargestellt.

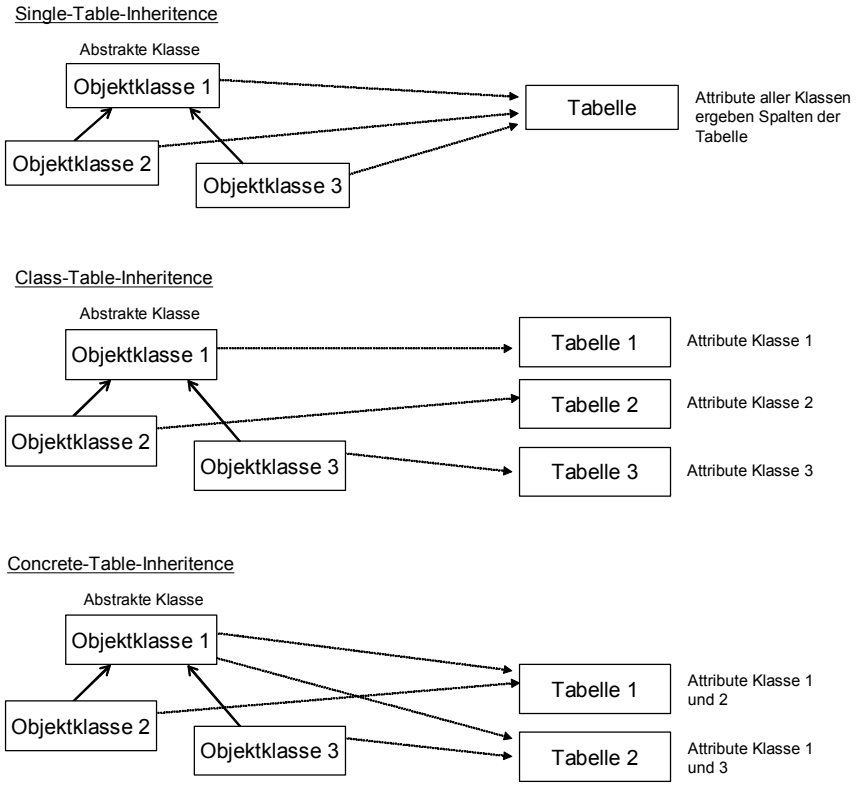


Abbildung 5-26: Mapping von Vererbungsbeziehungen

Mapping von Assoziationen: Assoziationen zwischen Objektklassen mit der Kardinalität 1:1 bildet man am besten über Fremdschlüssel ab. Dabei kann je nach Zugriffshäufigkeit in beiden Tabellen oder nur in einer ein Fremdschlüssel definiert werden. 1:n-Assoziationen kann man ebenfalls über einen Fremdschlüssel abbilden. Für m:n-Assoziationen benötigt man eine eigene Beziehungstabelle, de-

ren Primärschlüssel sich aus den Primärschlüsseln der in Beziehung stehenden Tabellen zusammensetzt. Die Schlüsselbestandteile sind gleichzeitig Fremdschlüssel, die auf die jeweiligen Tabellen verweisen.

In Abbildung 5-27 sind die drei Assoziationstypen und die entsprechenden Mapping-Möglichkeiten skizziert.

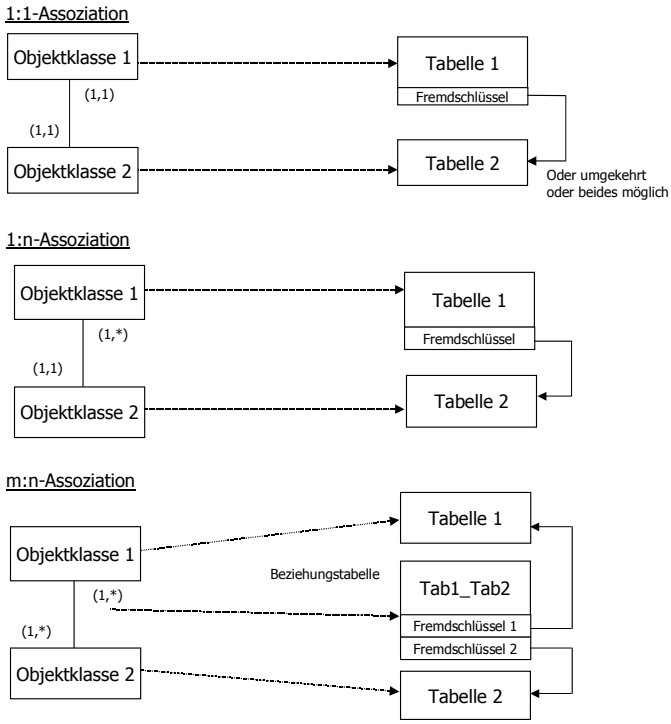


Abbildung 5-27: Mapping von Assoziationen

Die ORM-Schicht hat neben dem Mapping der Objekte auf Tabellen noch andere, wichtige Aufgaben zu erledigen:

- Der Datenbankzugriffsmechanismus muss sich zudem um die Transaktionssteuerung und die Nebenläufigkeitskontrolle kümmern. Bei der Transaktionssteuerung gibt es verschiedene Ansätze. Entweder gibt man dem Programmierer die Möglichkeit, die Transaktionsgrenzen selbst zu bestimmen, oder das Setzen von Transaktionsgrenzen wird vom Persistenzmechanismus automatisch durchgeführt. Die Nebenläufigkeitskontrolle wird üblicherweise über optimistisches oder pessimistisches Locking durchgeführt. Dazu gehört auch das Setzen des Isolationslevels.

- Eine weitere Aufgabe ist die Gewährleistung von möglichst guter Zugriffsleistung. Dies wird, je nach Ansatz, auch über spezielle Caching-Mechanismen erreicht.
- Weiterhin ist die ORM-Schicht für die Erzeugung der Primärschlüssel zuständig, die meist über einen fortlaufenden Zähler, die sog. Object-Id gebildet wird.
- Schließlich ist bei komplexen Objektnetzen zu entscheiden, wann welche Daten von der Datenbank gelesen werden sollen. Man unterscheidet hier üblicherweise ein *Lazy* oder ein *Eager Loading*. Bei *Lazy Loading* wird bei einem Zugriff auf ein Objekt erst einmal nur das Objekt gelesen und alle Objekte, die eine Assoziation mit diesem haben, erst bei Bedarf. Bei *Eager Loading* wird das gesamte Objektnetz eingelesen.

Welche Aufgaben die Datenzugriffsschnittstelle nun tatsächlich übernimmt, hängt von der Implementierung ab. ORM-Produkte unterstützen den Programmierer in der Regel bei den genannten Aufgaben. Im Folgenden diskutieren wir einige Möglichkeiten, den Zugriff auf die Datenbank zu kapseln und gehen dabei auch auf die zugehörigen Patterns ein, die in (Fowler 2006) ausführlich beschrieben werden. Zu den Patterns gehören *Active Records*, *Table Data Gateway* und *Data Access Object* (*DAO*), *Row Data Gateway* und *Table Mapper*.

Direkter Zugriff über die Fachklasse

Nach Starke ist die einfachste Variante des Datenbankzugriffs über die Business-Logik selbst realisierbar (Starke 2005). Eine Fachklasse (Business-Klasse, Klasse mit Geschäftslogik) kümmert sich hier eigenständig um den SQL-Zugriff auf die Datenbank (siehe Abbildung 5-28). Diese Vorgehensweise hat den Vorteil, dass sie recht einfach umzusetzen ist. Zudem kann jeder Entwickler einer Fachklasse sich direkt mit dem optimalen Zugriff auf die Daten, die er benötigt, beschäftigen. Eine Optimierung der Zugriffe liegt also in der Hand des Entwicklers einer Fachklasse.

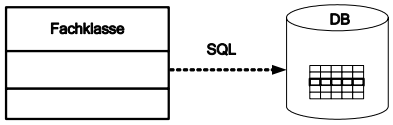


Abbildung 5-28: Datenbankzugriff über die Fachklasse

Der Zugriff auf die Daten erfolgt hier im Rahmen einer Client-Server-Architektur über die Business-Logik, eine gewisse Kapselung der Datenzugriffe über die Fachklasse ist also gegeben, allerdings ist von Nachteil, dass eine Änderung des Datenbankschemas immer auch eine Änderung der Fachklasse nach sich zieht. Dieser Aspekt wird viel diskutiert, da einige Praktiker der Meinung sind, dass eine Änderung im Schema meistens auch eine Änderung in der Business-Logik nach sich zieht und daher ohnehin auch diese und meistens sogar auch noch die Präsentationslogik angepasst werden muss.

In Java würde man den Zugriff auf die Datenbank über das JDBC-API (Java Database Connectivity) implementieren (Sun 2006a). Dies ist eine Möglichkeit, in Java SQL-Befehle an Datenbanken zu senden. Die API stellt Möglichkeiten zum Aufbau einer Verbindung und für alle notwendigen Zugriffe bereit. Queries werden in Form von Strings mit SQL-Anweisungen an die Datenbank-Engine übergeben. Bei dieser Zugriffsvariante muss man sich also in der Regel seine SQL-Anweisungen im Code als Strings aufbauen und diese über JDBC- oder sonstige Methoden an die SQL-Engine weiterleiten. In .NET würde man auf den recht ähnlichen ADO.NET-Mechanismus zurückgreifen.

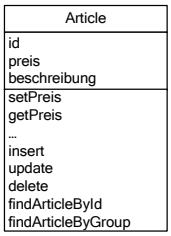


Abbildung 5-29: Beispiel für ein typisches Active Record

Dieser Ansatz kann mit dem Pattern *Active Record* (nach Fowler) umgesetzt werden. Objektklassen, die dieses Pattern nutzen, stellen an der Schnittstelle sowohl Datenzugriffs- als auch Business-Logik-Methoden zur Verfügung. Die Datenzugriffe werden gekapselt, aber nicht von der Business-Logik separiert. Man findet hier Methoden wie *insert*, *update*, delete, *find*, Setter- und Getter-Methoden für Attribute, aber auch Methoden der Geschäftslogik wie *findAllArticleById* oder *findArticleByGroup* (siehe hierzu die Beispielklasse in Abbildung 5-29).

Einführung einer Persistenzschicht über Datenklassen

Bei dieser Variante kapseln Datenklassen die Zugriffe auf die Datenbank und verbergen so die SQL-Anweisungen vor den Fachklassen. Man kann sich für jede Tabelle eine Datenklasse vorstellen, aber auch eine Datenklasse für den Zugriff auf mehrere Tabellen ist möglich und bei komplexeren Queries auch notwendig.

Nach Starke erreicht dieser Ansatz aber auch keine vollständige Trennung der Anwendungslogik von der Persistenzschicht, da ein Programmierer einer Datenklasse immer noch die Techniken des Datenbankzugriffs beherrschen muss. Von Vorteil ist aber, dass eine Änderung in der Datenklasse nicht unbedingt auf die Fachklasse ausstrahlt. Zudem hat der Entwickler auch hier die Optimierung der SQL-Zugriffe in der Hand. Diese Variante eignet sich recht gut, wenn die Datenbankzugriffe überwiegend auf einzelne Tabellen abzielen. Nicht so gut geeignet sind diese Datenklassen, wenn man sehr vernetzte Datenstrukturen hat. Die Vernetzung muss dann eigenständig durch entsprechende Zugriffsklassen, also spezielle Datenklassen ausprogrammiert werden.

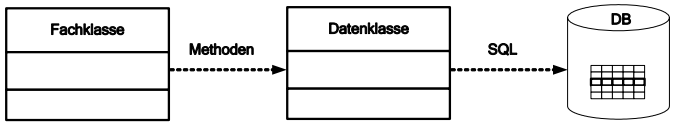


Abbildung 5-30: Datenbankzugriff über die eigene Datenklassen

Nach Fowler nutzt man für diese Variante typischerweise das *Table Data Gateway* Pattern (Fowler 2006). Ein *Table Data Gateway* kann als Okjektklasse implementiert werden, die die typischen Zugriffsmethoden wie *insert*, *delete*, *update* und evtl. mehrere *find*-Methoden zum Suchen über verschiedene Suchkriterien (*findByName*, *findByOrderNumber*, *findByDateAndName*,..) bereitstellt. Je Tabelle wird eine Gateway-Klasse verwendet und die Suchmethoden liefern Datensätze zurück, die dann in der Geschäftslogik bearbeitet werden können. Die Methoden der Klasse erzeugen den SQL-Code meist durch Zusammensetzen von SQL-Strings. Kompliziert wird es meistens, wenn sehr viele verschiedene Select-Operationen erforderlich sind. Die Entwicklung von Datenklassen kann also unterschiedlich komplex sein, je nachdem, welche Methoden an der Schnittstelle angeboten werden. Kommt wenig Business-Logik zum Datenbankzugriff hinzu, sind die Fachlassen auch meist nur Durchreicher der Parameter an die Methoden der Datenklasse. Auch hier würde man den Zugriff auf die Datenbank in Java über das JDBC-API implementieren. In .NET kann man auch sog. Datasets nutzen. In Abbildung 5-31 ist ein Beispiel für ein *Table Data Gateway* mit einigen Methoden skizziert.

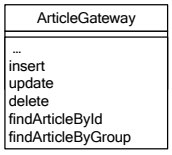


Abbildung 5-31: Beispiel für ein typisches Table Data Gateway

Eine weitere Alternative stellt nach Fowler das Pattern *Row Data Gateway* dar. Hier ist für jede einzelne Tabellenzeile ein Objekt als *Gateway* zur Datenbank vorgesehen. Ein *Row Data Gateway* verhält sich wie ein Objekt, das einen Datensatz repräsentiert. Die zugehörige Objektklasse enthält nur Datenzugriffsoperationen für den Zugriff auf die Tabelle und ggf. auch die *find*-Methoden, die aber auch in eigene *Finder-Klassen* ausgelagert werden können. An der Schnittstelle zum *Row Data Gateway* sollte im Gegensatz zum *Active Record* keine Business-Logik sichtbar sein. In Abbildung 5-31 ist ein Beispiel für ein *Row Data Gateway* mit einigen Attributen und Methoden skizziert.

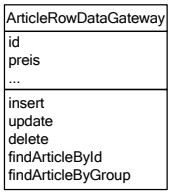


Abbildung 5-32: Beispiel für ein typisches Row Data Gateway

Diese Möglichkeit entspricht in etwa dem Einsatz des *DAO*-Patterns (*Data Access Object*) in Verbindung mit dem *DTO*-Pattern (*Data Transfer Object*). Diese Variante wird in (Starke 2005) als Mini-Persistenzschicht bezeichnet. Wie Abbildung 5-33 zeigt nutzt eine Fachklasse DAOs für den Zugriff auf die Datenbank. Die DAOs erzeugen bei Select-Operationen DTOs, die der Fachklasse zur weiteren Bearbeitung übergeben werden. Die DTOs sind eigenständige Objektklassen, die von der Datenbankstruktur entkoppelt sind, mit einfachen Methoden zur Bearbeitung.

Die Nutzung des DTO-Patterns entkoppelt den Zugriff auf die Datenbank weiter, indem von der Datenzugriffsschicht unabhängige Objekte (DTOs) erzeugt werden, die von den Datenstrukturen der DAOs entkoppelt sind. Die Änderungsanfälligkeit der Client-Anwendungen sinkt durch den Einsatz dieses Patterns. Allerdings sind DTOs eine zusätzliche Belastung für den Programmierer. Sie müssen befüllt werden, was den Programmieraufwand enorm erhöht und sie verschlechtern aufgrund ständiger Kopieraktivitäten die Leistungsfähigkeit eines Systems. Die weitere Entkopplung des Datenbankzugriffs steht also im Widerspruch zur Leistung eines Systems und zum Programmieraufwand. Auch hier kommt der Aspekt hinzu, dass sich bei einer Änderung des Datenbankschemas ohnehin möglicherweise eine Änderung in allen Schichten ergibt. Die technische Entkopplung des Datenzugriffs ist daher aber gegeben.

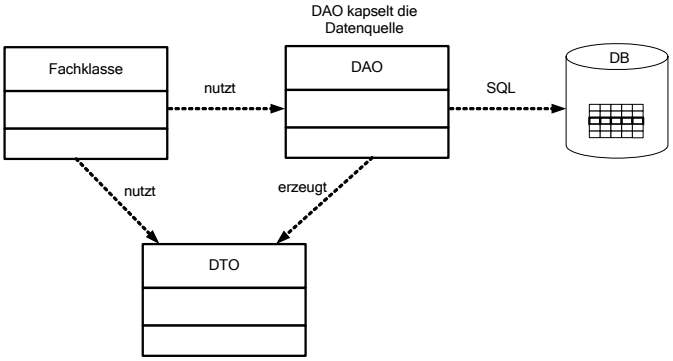


Abbildung 5-33: Datenbankzugriff über das DAO-Pattern

Der Einsatz von DTOs bringt zudem den Vorteil, dass bei entfernten Aufrufen auf die Fachklasse in verteilten Umgebungen größere Datenstrukturen übertragen

werden und nicht für jedes Attribut ein Remote-Zugriff erfolgen muss. Dies war die eigentliche Ursache für die Erfindung dieses Patterns. DTOs enthalten üblicherweise auch größere Datenmengen. Gerade für die Realisierung serviceorientierter Architekturen, bei denen unbedingt eine Entkopplung von der Datenhaltung notwendig ist, sind DTOs ein gutes Konzept, da sie sich relativ leicht auf eine serviceorientierte Schnittstelle abbilden lassen.

In Java implementiert man für ein DAO und auch für ein DTO jeweils eine eigene Klasse, wobei im DAO je nach Architektur entweder auf einzelne Tabellen oder auch auf mehrere Tabellen zugegriffen werden kann. Der Zugriff vom DAO auf die Datenbank könnte in Java wiederum über JDBC erfolgen. Obwohl Starke diese Variante des Datenbankzugriffs bereits als Persistenzschicht bezeichnet, ist der einzige markante Unterschied zum *Table Data Gateway* die Nutzung von DTOs, die die Leistungsfähigkeit von Remote-Zugriffen verbessern sollen.

Komplexere Persistenzschichten

Komplexere Persistenzschichten kümmern sich neben dem Mapping der Daten auf Objekte um weitere Aspekte des Datenzugriffs wie etwa um die Transaktionssteuerung, um eine komfortablere Abbildung von Vererbungsmechanismen und Assiziationen zwischen Objekten. Sie dienen im Wesentlichen dazu, die Datenbankzugriffe und das Object-Relational-Mapping für den Entwickler noch weiter zu vereinfachen. Heute werden von Herstellern einige ORM-Tools angeboten oder diese sind bereits in Application-Server-Produkte integriert.

In Java stehen mehrere Möglichkeiten zur Verfügung, eine komplexere Persistenzschicht zu nutzen. Hierzu gehören EJB Entity Beans, Hibernate JDO (Java Data Objects) sowie JPA (Java Persistence API). Im .NET-Umfeld gibt es z.B. Nhibernate für C#.NET. Wie man sieht gibt es auch hier nicht nur eine Lösung.

ORM-Tools stellen meist einen Satz von vordefinierten Interfaces und Klassen bereit, die ein Anwendungsprogrammierer für den Zugriff auf die Datenbank nutzen kann. Sie sind nicht integraler Bestandteil der Programmiersprache sondern werden als Zusatzpakete angeboten.

Neben den genannten Tools gibt es in der Praxis auch eine Fülle von eigenen ORM-Tools, die innerhalb von Projekten entwickelt wurden, als es noch keine brauchbaren Werkzeuge auf dem Markt gab. Da ORM-Tools dediziert die Aufgabe des Datenbankzugriffs übernehmen, könnte man sich vorstellen, dass sie einen sehr leistungsfähigen Zugriff auf die Daten gewährleisten. Dagegen spricht, dass die Ansätze generisch, ohne Rücksicht auf die konkreten Daten sind. Ein Nachteil des Einsatzes von ORM-Tools sind die eingeschränkten Möglichkeiten der Leistungsoptimierung bei komplexen Datenbankzugriffen.

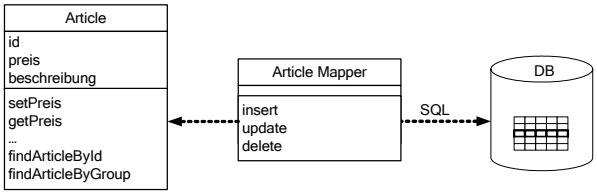


Abbildung 5-34: Beispiel für einen Data Mapper

Als ein Pattern für eine komplexere Persistenzschicht, das auch von ORM-Produkten verwendet wird, schlägt Fowler das *Data-Mapper*-Pattern vor (Fowler 2006). Ein *Data Mapper* wird hier als Softwareschicht bezeichnet, welche eine klare Trennung zwischen den Objekten der Programmiersprachen und den Datenbanktabellen herbeiführt. Geschäftslogik und Datenhaltung sollen also vollständig getrennt werden. Im Beispiel in Abbildung 5-34 ist ein Artikel-Objekt dargestellt, das über einen Article-Mapper mit Daten versorgt wird. Nur der Article-Mapper greift auf die Datenbank zu und versorgt die Fachklassen mit Daten.

Wir werden im Folgenden die ORM-Ansätze aus dem EJB-Umfeld in der älteren Version 2.x und in der neueren Version 3.0 etwas näher betrachten, allerdings sollen nur die wesentlichen Konzepte diskutiert werden. Weitere Informationen finden sich in (Backschat 2007) und (Burke 2006). In (Fowler 2006) sind auch einige Patterns erläutert, mit deren Hilfe eine eigene Persistenzschicht implementiert werden kann.

5.6.3 Fallbeispiel: BMP und CMP in EJB 2.x

Entity-Beans dienen bei EJB 2.x der Repräsentation persistenter Daten, die z.B. in einer Datenbank gespeichert werden. Die Lebensdauer einer Entity-Bean ist in der Regel länger als die Serverlaufzeit. Es handelt sich um Business-Entitäten, die vom EJB-Entwickler als Objekte angesprochen, aber meist in relationalen Datenbanken abgelegt werden. Ein Object-Relational-Mapping-Mechanismus zur Abbildung der Objekte auf Relationen wird durch den EJB-Container unterstützt. Übernimmt der Bean-Provider die Programmierung der Zugriffe auf die Datenhaltung selbst, so spricht man von *bean-managed* Persistency (BMP). Wickelt diese Aufgabe der Container ab, so spricht man von *container-managed* Persistency (CMP). Beide Varianten werden gemäß der EJB-Spezifikation unterstützt.

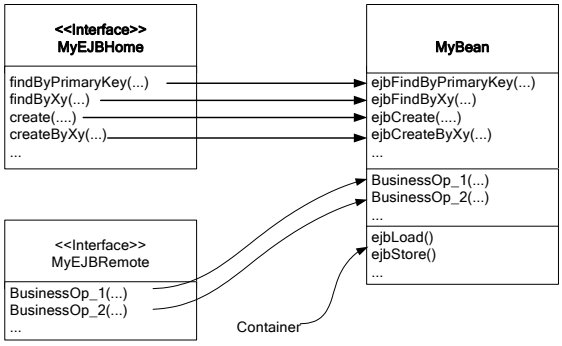


Abbildung 5-35: Aufruf der Methoden aus der eigentlichen Bean-Klasse

Ein Interface einer Entity-Bean hat neben den Methoden, die auch eine Session-Bean enthält, noch weitere Methoden, die von der eigentlichen Bean-Klasse implementiert werden müssen (siehe Abbildung 5-35). Dies sind neben beliebig vielen Methoden zum Erzeugen von Entity-Beans (*create*-Methode), Methoden zum Auffinden von Entity-Beans nach bestimmten Kriterien. Die Namen der "finder"-Methoden beginnen per Konvention mit dem Präfix „findBy", gefolgt von einer Bezeichnung für die Suchkriterien (*findBy<suchkriterium>;* Beispiele: *findByPrimaryKey* und *findByName*).

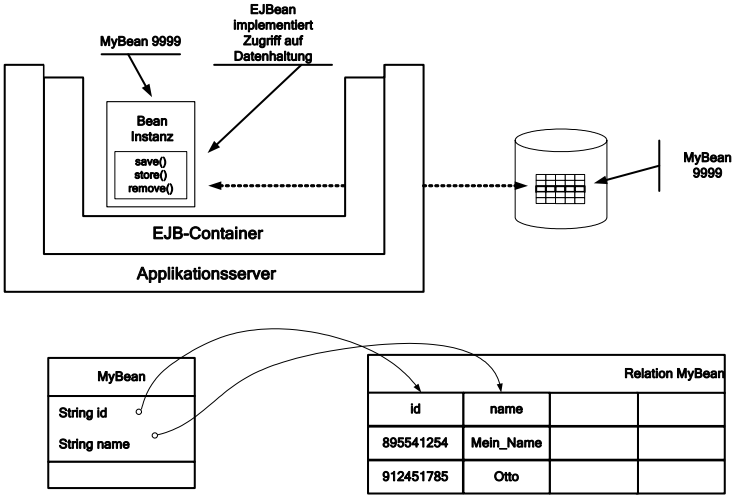


Abbildung 5-36: Bean-managed Persistency

Aufgabe des Bean-Providers ist es, die eigentliche Bean-Klasse mit den Methoden *ejbCreate*, *ejbRemove*, *ejbFindBy<suchkriterium>*, *ejbLoad* und *ejbStore* zu implementieren. Die letzten beiden benötigt der Container für das Laden der Daten aus dem

persistenten Speicher und das Speichern in diesen. Die Methode *create* aus dem sog. *HomeObject* ruft die Methode *ejbCreate* aus der eigentlichen *Bean-Klasse* und die Methode *findBy<suchkriterium>* ruft die Methode *ejbFindBy<suchkriterium>* auf. Auch die Methoden *setEntityContext* und *unsetEntityContext* sind obligatorisch zu implementieren. Diese beiden Methoden werden auch vom Container aufgerufen. Die Methode *setEntityContext* muss den Laufzeitkontext in der EJBean-Instanz in einer lokalen Variablen vom Typ *EntityContext* ablegen, *unsetEntityContext* setzt den Kontext auf „null". Über diese Variable kann die EJBean auf den für sie im Container verwalteten Kontext zugreifen. Werden die Entitäten in einer SQL-Datenbank gespeichert, so enthalten die genannten Methoden SQL-Anweisungen zum Anlegen (*ejbCreate*-Methode), Löschen (*ejbRemove*-Methode), Suchen (*ejbFindBy...*-Methoden und *ejbLoad*-Methode) und Aktualisieren (*ejbStore*-Methode) eines Tabelleneintrags. Wie wir noch sehen werden, gilt dies nur für *bean-managed* und nicht für *container-managed* Persistency, wo der Container alle Datenbankzugriffe übernimmt und das Mapping auf Tabellen und Tabellenfelder über den Deployment Descriptor erfolgt.

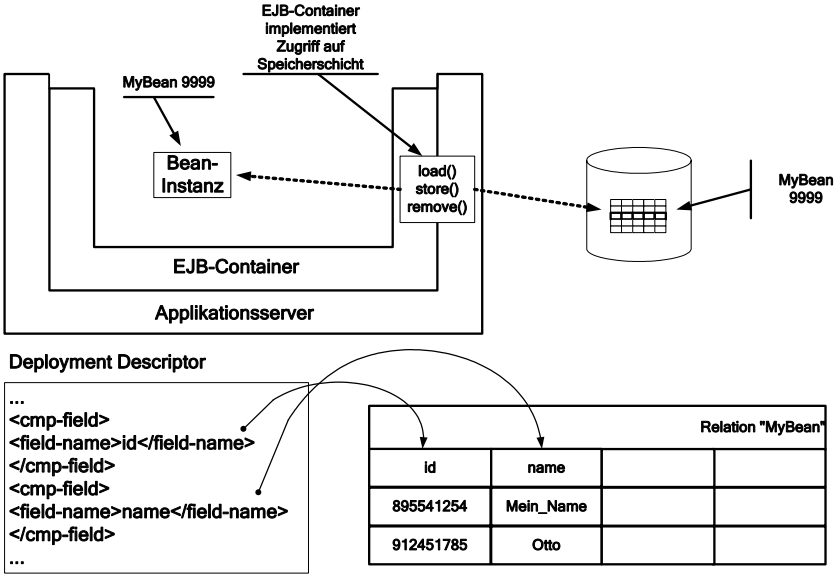


Abbildung 5-37: Container-managed Persistency

Bei BMP ist der Bean-Provider für die Entwicklung der Zugriffslogik auf die Persistenzschicht zuständig. Die Methoden *ejbStore*, *ejbLoad* usw. müssen programmiert werden. In Abbildung 5-36 ist die Arbeitsweise von BMP prinzipiell skizziert. In einer Bean-Instanz liegen beispielhaft die Attribute *id* und *name*. Diese werden über die EJBean-Methoden auf eine Tabelle abgebildet.

Bei CMP übernimmt der Container die Verwaltung der Datenhaltungsschicht (Persistenzschicht). Im Deployment Descriptor werden die Attribute der entsprechenden Datenbanktabellen sowie die Zuordnung zu Variablen in der EJBean konfiguriert. Die Methoden *ejbStore*, *ejbLoad* usw. sind nicht vom Bean-Provider zu programmieren. Bei CMP müssen alle Zugriffsroutinen auf die Attribute der Entity-Bean mit abstrakten, öffentlichen (public) get- und set-Methoden deklariert sein. Diese dürfen aber vom Bean-Provider nicht ausprogrammiert werden, da der Container den Code dafür zur Deployment-Zeit erzeugt. In Abbildung 5-37 sind die Zusammenhänge dargestellt.

Im Deployment Descriptor erfolgt die Mapping-Konfigurierung und dabei das Mapping der Variablennamen auf Attributnamen in der zugehörigen Relation. Die Speicherzugriffsoperationen liegen innerhalb des Containers und können aufgrund der Metainformation aus dem Deployment Descriptor abgeleitet werden.

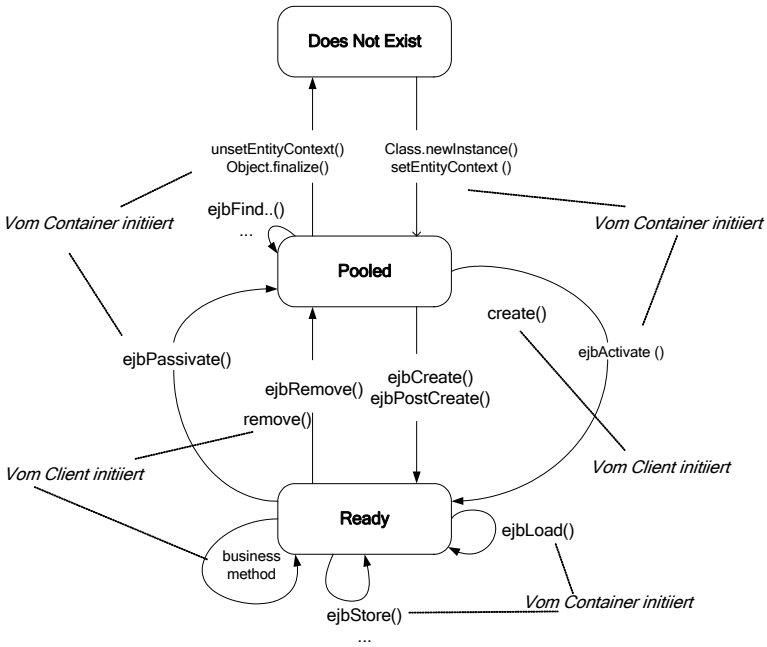


Abbildung 5-38: Zustandsdiagramm für eine Entity-Bean

Abbildung 5-38 enthält den Zustandsautomaten einer Entity-Bean. Alle Aktionen die vom EJB-Client bzw. vom EJB-Container initiiert werden, sind entsprechend markiert. Zunächst existiert die Entity-Bean nicht. Sie wird vom Container über eine eigene Strategie erzeugt und in einen Pool abgelegt. Durch Aufruf der Methode *create* über das Home-Interface wird die Instanz der Entity-Bean (gemeint ist die Instanz der eigentlichen Bean-Klasse) erzeugt. Bei den Zustandsübergängen

und auch im „Ready“-Zustand bei Veränderungen der Bean-Instanz, wird implizit durch den Container eine entsprechende Methode der Bean-Instanz aufgerufen.

Beispiel: Entity-Bean mit BMP: Ein Artikel kann als typisches Beispiel für eine Entity-Bean bezeichnet werden, deren Coding im Folgenden angedeutet werden soll. Das Remote-Interface könnte wie folgt aussehen:

```
package ArticleManagement;
…
import javax.ejb.*;
public interface Article extends EJBObject {
     public String getName()
           throws RemoteException;
     public void setName(String Name)
           throws RemoteException;
     public float getPrice()
           throws RemoteException;
     public void setPrice(float price)
           throws RemoteException;
…
}
```

Das Home-Interface des Artikelmanagers ist vereinfacht dargestellt und wird als *ArticleHome* bezeichnet. Es verfügt über eine Methode zum Erzeugen eines neuen Artikels und zum Suchen eines Artikels über den Primärschlüssel. Für den Primärschlüssel muss bei bean-managed Persistency entweder eine eigene Klasse bereitgestellt werden, die in unserem Fall *ArticlePK* heißt und serialisierbar sein muss. Der Primärschlüssel kann aber auch ein primitiver Java-Datentyp sein. Bei container-managed Persistency wird die Klasse vom Container bereitgestellt.

```
…
public interface ArticleHome extends EJBHome {
       public void create(int CatalogId, String name)
              throws CreateException, RemoteException;
       public Article findByPrimaryKey(ArticlePK pk)
              throws FinderException, RemoteException;
       …
}
public class ArticlePK implements java.io.Serializable {
       public int catalogId
}
```

Die eigentliche Bean-Klasse, die in unserem Beispiel mit *MyArticleBean* benannt ist, implementiert das Interface *javax.ejb.EntityBean* und sieht wie folgt aus:

```
…
public class MyArticleBean implements EntityBean {
```

```
    // Kontext
    private EntityContext entityContext = null;
    …
    public void ejbCreate(int CatalogId, String name) throws…
    {/* eigener Code */}
    public void ejbPostCreate(int CatalogId, String name)
    {/* eigener Code */}
    public Article ejbFindByPrimaryKey(ArticlePK pk) ) throws…
    {/* eigener Code */}
    // Business Methoden
    public String getName()throws…
    {/* eigener Code */} )
    public void setName(String Name) throws…
    {/* eigener Code */}
    public float getPrice() throws…
    {/* eigener Code */}
    public void setPrice(float price) throws…
    {/* eigener Code */}
    // Entity-Bean-Methoden
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
    public void ejbLoad(){ /* Eigene Implementierung */}
    public void ejbStore(){ /* Eigene Implementierung */}
    public void setEntityContext(EntityContext ctx){
          entityContext = ctx;
    }
    public void unsetEntityContext(){
          entityContext = null;
    }
    …
}
```

Alle Argumente und die Returnwerte von Methoden einer Entity-Bean müssen serialisierbar sein, da der Java-Objektserialisierungsmechanismus verwendet wird. Die *ejbStore*-Methode wird vom Container immer dann aufgerufen, wenn er es für richtig hält. Die EJBean-Instanz muss die Update-Operation in Richtung Datenbank selbst vornehmen. Nach einer Deaktivierung durch den Container ist bei der nächsten Nutzung durch eine Business-Methode wieder eine Aktivierung erforderlich, wobei vom Container anschließend ein Lesen aus der Datenbank mit *ejbLoad* initiiert wird. Wann eine Entity-Bean-Instanz tatsächlich deaktiviert wird, hängt von der Strategie des Containers ab und ist auch ganz dem Container-Hersteller überlassen. Die ejb*-Methoden, die sich mit der Datenhaltung beschäftigen, dienen u.a. der Synchronisation mit einer Datenbank. Bei Nutzung von *bean-*

managed Persistency sind die Zugriffe auf den dauerhaften Speichermechanismus z.B. über das *JDBC*-API zu programmieren.

Beispiel: Entity-Bean mit CMP: Bei Nutzung von CMP gibt es hinsichtlich der Implementierung einer Entity-Bean gravierende Unterschiede, da der Methodencode für Datenbankzugriffsmethoden und auch die „Getter-/Setter"-Methoden vom Container über die Informationen aus dem Deployment Descriptor erzeugt werden. Bei Nutzung von *container-managed* Persistency sind diese Methoden nicht auszuprogrammieren, da die Zuordnung der Daten auf persistente Elemente vom Container vorgenommen wird. Die erforderliche Metainformation steht dann im Deployment Descriptor.

CMP nutzt ein abstraktes Programmiermodell. Es ist daher eine abstrakte Bean-Klasse zu definieren, in der alle Zugriffsmethoden ebenso abstrakt deklariert werden müssen. Sie dürfen auch vom Bean-Provider nicht implementiert werden.

Die eigentliche Bean-Klasse, die in unserem Beispiel mit *MyArticleBean,* benannt ist, sieht mit CMP, etwas vereinfacht, wie folgt aus:

```
…
public abstract class MyArticleBean implements EntityBean {
     // Kontext
     private EntityContext entityContext = null;
     public void ejbCreate(int catalogId, String name) throws…
     {/* eigener Code: Zuweisen aller Attribute */}
     public void ejbPostCreate(int catalogId, String name)
     {/* eigener Code */}
     // Abstrakte Business-Methoden (Setter-/Getter)
     public abstract String getName() {}
     public abstract void setName(String Name) {}
     public abstract float getPrice() {}
     public abstract void setPrice(float price) {}
     // Entity-Bean-Methoden
     public void ejbActivate(){}
     public void ejbPassivate(){}
     public void ejbRemove(){}
     public void ejbLoad(){}
     public void ejbStore(){}
     public void setEntityContext(EntityContext ctx){
           entityContext = ctx;
     }
     public void unsetEntityContext(){
           entityContext = null;
     }
     …
}
```

In Abbildung 5-39 wird der Erzeugungs- und Entfernungsprozess für eine container-managed Entity-Bean-Instanz skizziert. Gleich nach der Erzeugung einer Entity-Bean-Instanz mit *create* bzw. *ejbCreate* werden die Felder der EJBean über die vom Container generierten *get*-Methoden ausgelesen und es wird eine entsprechende *Insert*-Operation in Richtung Datenbank abgesetzt. Bei Aufruf der *remove*-Methode durch den Client wird vom Container eine *delete*-Operation erzeugt und ausgeführt.

Für die Aktivierung und Passivierung einer Entity-Bean-Instanz nutzt der Container die Methoden *ejbPassivate* und *ejbActivate*. Üblicherweise wird vor der Passivierung einer Entity-Bean-Instanz und entsprechend nach einer Aktivierung durch den Container eine *update*- bzw. *select*-Operation ausgeführt, um den Datenbankeintrag zu aktualisieren bzw. auszulesen.

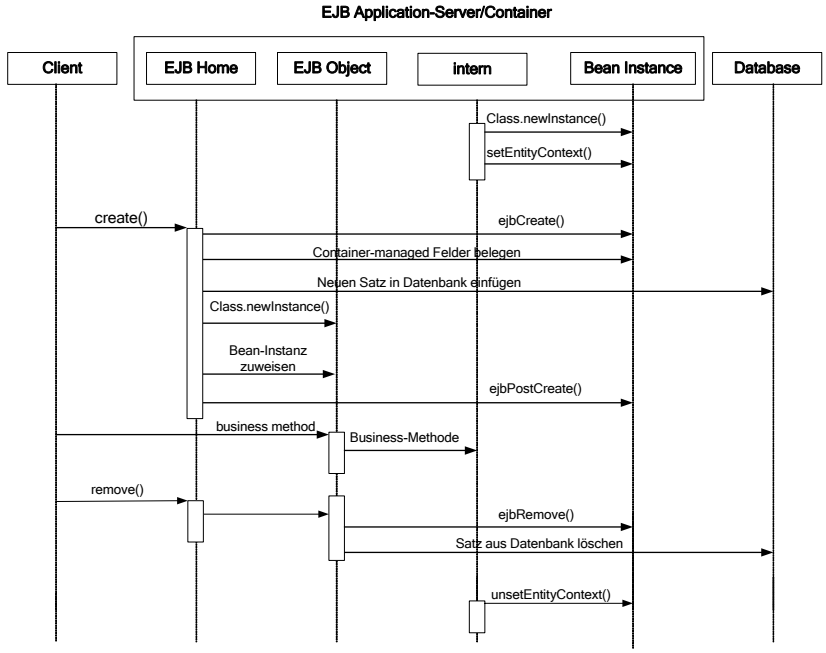


Abbildung 5-39: Erzeugen und Entfernen einer Entity-Bean bei CMP

Wie man insgesamt an dieser ORM-Methode erkennen kann, ist sie in der Anwendung noch relativ kompliziert. Wie wir im folgenden Abschnitt sehen werden, ist die weiterentwickelte Version der Entity-Beans in der EJB-Version 3.0 in der Handhabung einfacher.

5.6.4 Fallbeispiel: Java Persistence API und EJB 3.0

Ziel von Java Persistence API (JPAist es über der JDBC-API einen möglichst einfachen Object-Relational-Mapping-Mechanismus zu definieren. Entity Beans werden in EJB 3.0 auch als Entities (Entität) bezeichnet. Ein Entity wird in dieser Spezifikation als leichtgewichtiges persistentes Domänenobjekt bezeichnet. Der Lebenszyklus einer Entität wird über einen Softwarebaustein namens *Entity-Manager* verwaltet. Der Entity-Manager übernimmt auch das OR-Mapping.

Entity-Manager und Kontextverwaltung

Der Entity-Manager löst die aus der EJB 2.x-Spezifikation bekannten Home-Interfaces der Entity Beans ab und bietet zusätzliche Funktionalität. Er stellt die zentrale Instanz für alle Persistenzoperationen dar. Die Java Persistence API ist zwar im Zuge der EJB-3.0-Spezifikation entwickelt worden, ist aber im Prinzip eine eigenständige Java-API, die nicht unbedingt einen EJB-Container benötigt. Abbildung 5-40 zeigt die Einbettung eines Entity-Managers in einen Application-Server.

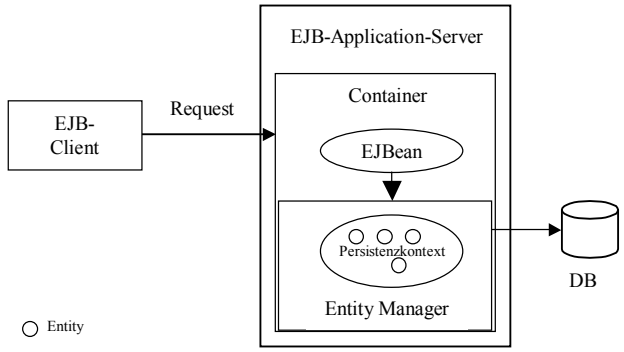


Abbildung 5-40: Einbettung des Entity-Managers in einen Application-Server

Ein Entity wird erst dann persistent, wenn man es einem Entity-Manager bekannt macht. Dazu wird es in einen Persistenzkontext eingetragen. Der Persistenzkontext verwaltet alle Entities, die automatisch in die Datenbank geschrieben werden müssen. Diese Entities werden als *managed* Entities bezeichnet. Das Schreiben auf die Datenbank hängt von der verwendeten Strategie ab. Wenn beispielsweise in einer Transaktion mehrere Entities verändert und wieder abgespeichert werden, besteht der Persistenzkontext während dieser Transaktion aus den bearbeiteten Entities. Für den Persistenzkontext gibt es zwei Einstellungen:

– *Transaction*: Der Persistenzkontext existiert, solange die Transaktion läuft. Wenn die Transaktion beendet ist, werden alle verwalteten Entities in den Status *detached* versetzt.

– *Extended*: Der Persistenzkontext bleibt ggf. über mehrere Transaktionen hinweg bestehen, bis er explizit aufgelöst wird. Der *Extended*-Modus kann nur im Zusammenhang mit *Bean Managed Transactions* und *Stateful Session Beans* benutzt werden.

Jeder Entity-Manager ist an eine Persistence-Unit gebunden, die über eine XML-Datei (persistence.xml) konfigurierbar ist. Dort werden Eigenschaften des Persistenzkontexts festgelegt.

Beispiel: Das folgende Codestück zeigt eine mögliche Definition der Persistence-Unit.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
   <persistence-unit name="articledb">
      <jta-data-source>java:/XAOracleDS</jta-data-source>
      <properties>
          <property name="…" value="…"/>
           <!-- Transaction settings for Container Managed Transactions -->…
      </properties>
   </persistence-unit>
</persistence>
```

Alle Entities, die zu einer Persistence-Unit gehören, können hier ebenfalls spezifiziert werden. Fehlt diese Angabe, so ermittelt der EJB-Container diese Information automatisch aus dem JAR-File der Anwendung und nimmt alle gefundenen Entities in die Persistence-Unit auf. Weiterhin werden die zu verwendenden Datenquellen (Data Sources) und der verwendete Transaktionsmanager angegeben. Die für einen Entity-Manager zur verwendende Persistence-Unit wird im Bean-Coding über eine Annotation bekanntgemacht:

```
@PersistenceContext(type=PersistenceContextType.TRANSACTION)
@PersistenceUnit(unitName="articledb")
private EntityManager manager; -...
```

Die *Entity-Manager*-Schnittstelle stellt die Dienste bereit, um Entities zu verwalten (*create*, *find*,…) und Queries oder sonstige Operationen auszuführen. Auch die Interaktion mit einem Transaktionsdienst über das *JTA API* wird über den Entity-Manager abgewickelt. Das Interface *Entity-Manager* soll im Folgenden gezeigt werden:

```
package javax.persistence;
public interface EntityManager {
   public void persist(Object entity);
   public <T> T merge(T entity);
   public void remove(Object entity);
   public <T> T find(Class<T> entityClass, Object primaryKey);
   public <T> T getReference(Class<T> entityClass, Object primaryKey);
```

```
    public void flush();
    public void refresh(Object entity);
    public boolean contains(Object entity);
    public Query createQuery(String ejbqlString);
    public Query createNamedQuery(String name);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString, Class result-Class);
    public Query createNativeQuery(String sqlString,
        String result-SetMapping);
    public void close();
    public boolean isOpen();
    public EntityTransaction getTransaction();
}
```

Eine Entity, die sich im Persistenzkontext eines Entity-Managers befindet, ist im Zustand *managed*, d.h. alle Änderungen an dieser Entity werden vom Entity-Manager mit der Datenbank synchronisiert. Es ist auch möglich, eine Entity aus dem Kontext des Entity-Managers zu lösen. Dies geschieht z.B., wenn die Bean an den Client gesendet wird. Der Zustand der Bean wird dann als *detached* bezeichnet. Die vom Client an der Bean vorgenommenen Änderungen sind dann wieder in die Datenbank zu übertragen. Dazu müssen diese erst wieder über die Methode *merge* an den Entity-Manager übergeben werden. Man spricht bei diesem Vorgang von einem *Merge*. Die wichtigsten Methoden des Entity-Managers sollen kurz erwähnt werden:

- *flush()*: Mit dieser Methode werden die Änderungen an den *managed* Entities in die Datenbank geschrieben. Normalerweise ist es nicht nötig, diese Methode explizit aufzurufen, da das Flushing vom Entity-Manager selbstständig verwaltet wird. Das Flush-Verhalten kann über die Methode *setFlushMode()* gesteuert werden. Dabei stehen zwei Einstellungen zur Verfügung:
  - *AUTO* (Default-Einstellung): Die Methode *flush* wird beim Transaktions-Commit und vor jeder Query durchgeführt, welche die bereits verwalteten (*managed*) Entities betrifft.
  - *COMMIT*: Flush wird hier nur beim Transaktions-Commit durchgeführt.
- *persist()*: Mit dieser Methode wird eine neu erstellte Entity in der Datenbank persistiert und die Entity in den aktuellen Persistenzkontext eingefügt.
- *merge()*: Mit dieser Methode kann eine Entity mit Status *detached* in den Persistenzkontext aufgenommen werden. Dabei werden eventuelle Änderungen zum Flush-Zeitpunkt mit der Datenbank synchronisiert.
- *refresh()*: Mit dieser Methode kann der Zustand einer Entity aktualisiert werden. Bei Enities im Status *managed* werden dadurch zwischenzeitlich erfolgte Änderungen von anderen Clients sichtbar. Bei Entities im Status *detached* werden damit etwaige Änderungen mit den aktuellen Werten aus der Datenbank überschrieben.

– *createQuery()*: EJB 3.0 stellt eine eigene Query-Sprache namens EJB QL zur Verfügung.[6] Die Syntax ist ähnlich zu SQL, nur dass die Anfragen auf Entities und ihren Properties durchgeführt werden anstatt auf Datenbanktabellen. Die EJB-QL-Anfragen werden vom Entity-Manager in native SQL-Anfragen übersetzt.

Wichtig ist, dass beim Aufruf von *persist()* noch kein Zustandsübergang in den Zustand *managed* durchgeführt wird. Dies liegt daran, dass das Einfügen von Objekten in die Datenbank erst zum Flush-Zeitpunkt erfolgt. Der Aufruf von *persist()* markiert ein Entity nur für eine später auszuführende SQL-Einfüge-Operation. Erst beim expliziten oder implizit durch den Container vorgenommenen Aufruf der Methode *flush()* wechselt die Entity in den Zustand *managed*.

Entities

Entity-Beans werden in EJB 3.0 über die Annotation „@Entity" bezeichnet. Das Mapping auf Tabellen einer relationalen Datenbank erfolgt über die Annotation „@Table". Zur Formulierung von Queries dient die gegenüber EJB 2.1 erweiterte Abfragesprache EJB QL. Queries können in EJB QL auch mit einem Namen versehen und im Coding symbolisch angesprochen werden.

Beispiel: Betrachten wir den bekannten Artikel als Entity-Bean mit einigen Zugriffsoperationen. Im Beispiel wird ein Entitätstyp mit den Namen *Article* definiert, der über zwei benannte Queries verfügt. Die Queries haben die Namen *findAllArticlesByKatalogId* und *findAllArticles*.

```
package ArticleManagement;
…
import javax.ejb.*;
import javax.persistence.*;

@Entity
    @NamedQueries(
    {
        @NamedQuery(name="findAllArticlesByGroup",
        queryString="from Article a where a.warengruppe = :warengruppe"),
            @NamedQuery(name="findAllArticles",
            queryString="from Article")
    }
)

@Table(name="ARTICLE")
public class Article implements java.io.Serializable {
```

[6] EJB QL hat große Ähnlichkeit mit der Hibernate Query Language (HQL).

```
    private static final long serialVersionUID = …;
    private long katalogId;
    private String beschreibung = new String();
    private float preis;
    private long warengruppe;

    public Article() {
        super();
    }
    …
    @id(generate = GenerateType.AUTO)
    @Column(name = "KATALOGID");
    public getKatalogId() {
        return catalogId;
    }
    public setKatalogId(long catalogId) {
        this.katalogId = katalogId;
    }

    @Column(name = "BESCHREIBUNG")
    public String getBeschreibung() {
        return beschreibung;
    }
    public String setBeschreibung( String beschreibung) {
        this.beschreibung =  beschreibung;
    }

    // @Column-Annotation für alle zu persistierenden Attribute
    …
    // Beziehungen sind ebenfalls über Annotionen möglich.
}
```

Über die Annotation „@Column“ werden die Tabellenspalten definiert. Jeder Spalte wird ein Attribut mit einer Get- und einer Set-Methode zugeordnet. Der Primärschlüssel wird mit der Annotation „@Id“ angegeben.

Die Entitäten können auch in Beziehung zu anderen Entitäten stehen. Alle möglichen Beziehungen (1:1, 1:n, n:m) zwischen Tabelleneinträgen können ebenfalls über Annotationen wie „@OneToMany“, „@ManyToMany“ usw. angegeben werden. Eine ausführliche Beschreibung dieser Annotationen ist in (Sun 2005b) und (Burke 2006) zu finden und soll an dieser Stelle nicht erfolgen.

Beispiel für die Nutzung einer Entity: Eine Entität vom Typ *Article* kann aus einer Session-Bean heraus beispielsweise wie folgt genutzt werden. Wir verwenden hierzu wieder unsere bekannte Session-Bean *ArticleMgrSessionBean:*

```
import java.util.collection;
import java.util.List;
…
@Stateless
public class ArticleMgrSessionBean implements … {
     @PersistenceContext(UnitName = "Artikelmanager")
     private EntityManager emgr; // Entity-Manager-Instanz
     private Article a;
     …
     public void createArticle(Article a) {
           this.a = a;
           // Schreiben in die Datenbank
           emgr.persist (Article.class, this.a);
     }
     // Suche über KatalogId
     public Article findArticleByKatalogId (long katalogId) {
          // Ohne Fehlerbehandlung
          return emgr.find("Article.class, katalogId);
     }
     // Suche aller Artikel
     public Article findAllArticles() {
          List<Articles> articleList =
          emgr.createNamedQuery("findAllArticles").getResultList();
     …
}
```

Im Beispiel wird ein Persistenzkontext mit der Bezeichnung *Artikelmanager* über die Annotation „@PersistenceContext“ erzeugt, über den man eine Referenz auf einen Entity-Manager besorgen kann. Weiterhin wird mit der Methode *createArticle* eine Entität vom Typ *Article* auf die Datenbank geschrieben (Methode *persist*). Zudem wird in der Methode *findArticleByKatalogId* gezeigt, wie ein Suchen über Primärschlüssel realisiert werden kann, und wie eine benannte Query benutzt werden kann (siehe Methode *findAllArticles*).

Da die Persistenzmechanismen in EJB 3.0 sehr umfangreich sind und eine detaillierte Einführung unseren Rahmen sprengen würde, soll an dieser Stelle bzgl. einer tiefergehenden Erläuterung auf die Literatur verwiesen werden (siehe (Backschat 2007) und (Burke 2006)). Abschließend soll noch der in EJB 3.0 implementierte Callback-Mechanismus erwähnt werden. Man kann als Entwickler Callback-Methoden in den Lebenszyklus von Entity-Beans einklinken. Diese Methoden werden auch als *Entity-Callbacks* bezeichnet. Als Callback-Ereignisse sind derzeit „@PrePersist“, „@PostPersist“, „@PostLoad“, „@PreUpdate“, „@PostUpdate“, „@PreRemove“ und „@PostRemove“ definiert. Die Namen der Annotationen sind auch selbsterklärend. Mit der Annotation „@PrePersist“ kann z.B. eine Methode

definiert werden, die im Lebenszyklus einer Entity-Bean vor einer dauerhaften Speicherung aufgerufen wird.

5.7 Clientzugriffsarchitekturen

Client-Anwendungen müssen an die Serversysteme angebunden werden, damit sie deren Dienste nutzen können. Wie macht man das üblicherweise und worauf sollte man dabei achten? Diese Frage wollen wir im Weiteren beantworten.

5.7.1 Grundlegendes

Wie bei den Technologiebetrachtungen bereits ausgeführt, werden Serversysteme meist über Proxies bzw. Stubs, die in der Regel auch automatisch erzeugt bzw. zur Übersetzungszeit über Tools generiert werden, angebunden. Diese Proxies verbergen Details der Kommunikation sowie des Marshallings. Eine gewisse Kapselung ist also automatisch gegeben, allerdings benötigt man noch technologieabhängige Zugriffe auf einen Naming-Service und auf die entfernten Softwarebausteine. Oft möchte man aber seine Clientsysteme vollständig von der Servertechnologie entkoppeln, um zu vermeiden, dass sich eine serverseitige Technologie-Veränderung auf die Clientprogramme auswirkt. In diesem Fall ist es sinnvoll, eine Schnittstelle zu den Serversystemen zu schaffen, die weitgehend technologieunabhängig ist. Hierzu muss man, wie in Abbildung 5-41 skizziert, auf der Clientseite einen Softwarebaustein ergänzen, der diese Technologie-Kapselung vornimmt. Dieser Baustein übernimmt dann implizit den Zugriff auf den Naming-Service und ggf. auch auf weitere Basisdienste und auf die Serverdienste. In Richtung der restlichen Clientbausteine stellt er eine technologieneutrale Schnittstelle zur Verfügung.

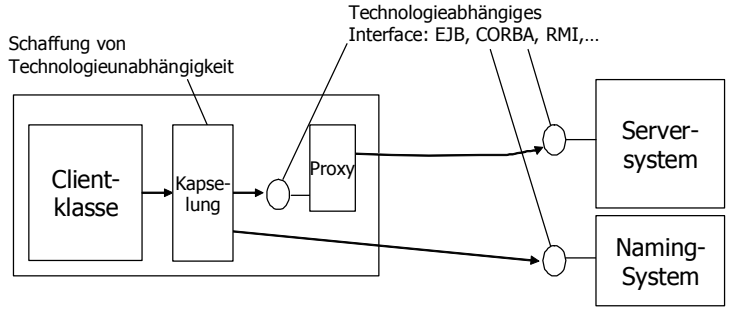


Abbildung 5-41: Kapselung eines Serversystems

Der Kapselungsgedanke kann für alle Kommunikationsparadigmen verwendet werden. Gelingt dies, so muss nicht bei jeder Technologie-Änderung (z.B. von Java-RMI nach EJB oder vom JMS auf ein nicht JMS-konformes Messaging-Produkt) in die Clientbausteine eingegriffen werden, es genügt vielmehr die Anpassung des Kapselungsbausteins. Allerdings muss beachtet werden, dass diese Art der Kapse-

lung eine erneute Schicht, die gepflegt werden muss, bedeutet. Die Kapselung ist also vor allem bei größeren Anwendungen sinnvoll, die durchaus eine oder mehrere Technologieveränderungen überdauern müssen. Ein Nachteil dieser Vorgehensweise ist sicherlich die schlechtere Leistung, die sich aufgrund der zusätzlichen Schicht ergibt.

Aus dem Software-Engineering sind für diese Art von Kapselung einige Vorgehensweisen bekannt. Wir wollen uns im Weiteren emplarisch die Vorschläge aus dem JEE-Patternkatalog speziell für Java-lastige Anwendungen näher ansehen. Die Grundkonzepte sind sprachunabhängig.

5.7.2 Fallbeispiel: JEE-Patterns für die Serverkapselung

Aus dem JEE-Umfeld sind auch einige Patterns hervorgegangen, die Probleme der verteilten Entwicklung mit Java adressieren. Insbesondere folgende Patterns aus dem JEE-Pattern-Katalog, die eine Trennung von Präsentations- und Business-Logik unterstützen sollen, werden wir hier kurz diskutieren (WWW-016):

- Das *Service-Locator-Pattern* zentralisiert z.B. das Auffinden von Objektreferenzen in verteilten Java-Umgebungen und versucht dabei Hersteller-spezifische Aspekte des sog. „Lookup-Codes“ des jeweiligen Naming-Service zu kapseln.
- Das *Session-Facade-Pattern* definiert als spezielle Form des Fassaden-Patterns einen höherwertigen Baustein einer Geschäftslogik. Mit dem Pattern kann man für Clients ein einziges Interface für Geschäftslogik-Bausteine festlegen.
- Das *Business-Delegate-Pattern* entkoppelt einen Geschäftslogik-Baustein von der technischen Implementierung. Damit kann der Lookup-Code und auch das Exception-Handling und weitere technische Details vor dem Nutzer verborgen werden. Die technische Serverzugriffsschnittstelle wird also gekapselt.

In Abbildung 5-42 sind die Zusammenhänge dargestellt. Auf der Clientseite werden für den Zugriff auf die Serverseite das Business-Delegate-Pattern und das Service-Locator-Pattern eingesetzt. Diese werden von den lokalen Clientbausteinen (im Bild als Front Controller bezeichnet) verwendet. Auf der Serverseite werden die Dienste über eine Fassade, die bei JEE als Session-Fassade bezeichnet wird, gekapselt. Die Technolologieabhängigkeit wird damit im Client weitgehend eliminiert.

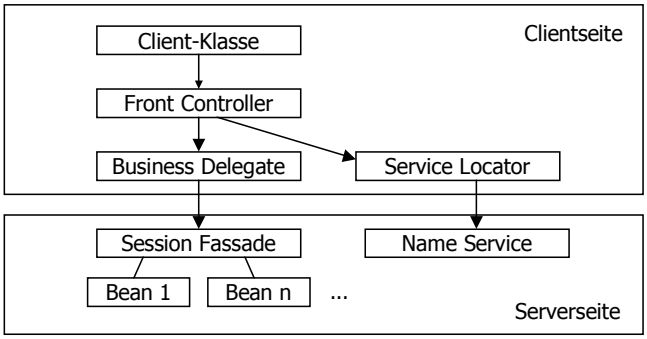


Abbildung 5-42: JEE-Patterns zur Kapselung des Servers

Es ist auch möglich, dass das Business-Delegate-Pattern den Zugang zum Naming-Service kapselt. Diese Variante ist in Abbildung 5-43 dargestellt.

Der Einsatz dieser Patterns sollte wohl bedacht werden, da er einen nicht unbeträchtlichen Zusatzaufwand für die Pflege der Zugriffsschicht nach sich zieht. Wenn man bedenkt, dass größere Anwendungen mehrere Hundert oder sogar Tausend Geschäftsmethoden nutzen, so bedeutet eine Anpassung an der Serverschnittstelle oft auch eine Anpassung am Business-Delegate-Interface. Dieser Aufwand muss im Vergleich zu den Technologieänderungen vertretbar sein, sonst bringt das Einziehen der Zugriffsschicht mehr Aufwand als er vermeidet. Das Prinzip des Design-for-Change muss für die Architektur eine große Rolle spielen, dann lohnt sich möglicherweise der Einsatz der Patterns.

An der Business-Delegate-Schnittstelle kann man z.B. auch gleich höherwertige Dienste anbieten, indem man eine Komposition vornimmt. Man kann ein Business-Delegate auch mit zusätzlichen Funktionen wie etwa das Logging der ausgehenden Serviceaufrufe zum Zwecke der Fehlerdiagnose anreichern. Bei kleineren Anwendungen kann man aber auch auf die Programmierung einer clientseitigen Zugriffsschicht verzichten.

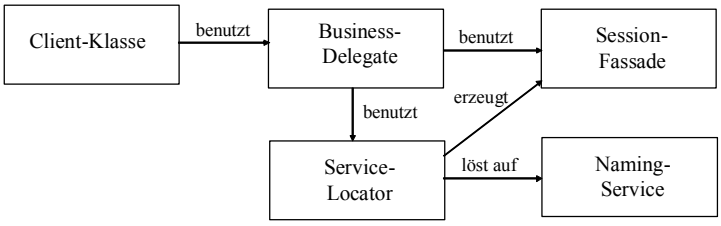


Abbildung 5-43: Einsatz von Business-Delegate aus (WWW-016)

5.8 Web-Architekturen

In vielen betrieblichen Anwendungen setzt man heute das WWW ein. Der Zugriff auf Dienste über den Browser als Präsentationswerkzeug ist ohne weitere Installation einer Clientsoftware für viele Anwendungen sogar unbedingt erforderlich. Denkt man beispielsweise an Online-Shopsysteme, so ist es heute üblich, dass man als Benutzerschnittstelle einen Browser nutzt.

5.8.1 Grundlegendes

Web-Architekturen sind im Prinzip Client-Server-Architekuren, bei denen die Clientseite üblicherweise etwas anders konstruiert wird, als bei klassischen Client-Server-Anwendungen. Für die Entwicklung der Client-Anwendungen benötigt man spezielle Programmiertechniken, die auch als Webtechniken bezeichnet werden. Hierzu gehören CGI (Common Gateway Interface), Java-Techniken wie Servlets und Java Server Pages (JSP), ASP.NET (Active Server Pages, Microsoft), Javascript, AJAX (Asynchronous JavaScript and XML) und PHP (Personal Home Page, offene Lösung), die zum Teil serverseitig und zum Teil clientseitig genutzt werden.

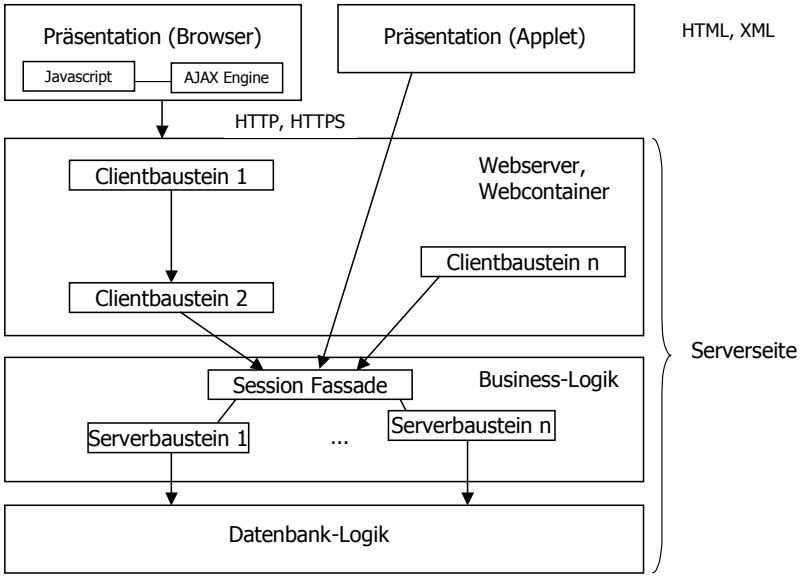


Abbildung 5-44: Typische Bausteine einer Web-Architektur

Der Client läuft üblicherweise auf der Serverseite und zwar in einem Webserver oder in einem Web-Container. Bevor eine Ausgabe an den Browser angestoßen wird, ist eine Generierung von HTML- oder XML-Code erforderlich, da diese Markup-Sprachen von den Browsern interpretiert und dargestellt werden. Scripts

(z.B. Javascripts), die im Browser ablaufen sollen, werden ebenfalls in den HTML-Code eingebracht. Der Browser stellt also nur die Präsentationslogik zur Verfügung. Browser und Webserver kommunizieren über die Protokolle HTTP bzw. HTTPS (Mandl 2008b). Die Kommunikation zwischen Browser und Serverseite erfolgt üblicherweise synchron, bei Einsatz von AJAX-Technologien ist auch eine aus Sicht des Benutzers asynchrone Ausführung durch eine AJAX-Engine möglich. Die Kommunikation bleibt aber weiterhin synchron. Die Asynchronität wird durch die parallele Ausführung von HTTP-Requests (sog. XMLHttpRequests) im Browser in eigenen Threads erreicht. Die Möglichkeiten der Einziehung einer Business-Logik entsprechen ansonsten den üblichen Schichtenarchitekturansätzen.

Die Clientbausteine laufen im Webserver bzw. in einer speziellen Ablaufumgebung namens Web-Container ab. Ein Web-Container ist eine Softwareplattform, in der Servlets ablaufen. Ein typischer Web-Container ist Apache Tomcat (WWW-007). In Abbildung 5-44 ist eine typische Web-Architektur skizziert. Wie man sieht, ist neben einer Browser-basierten Präsentation auch eine andere Möglichkeit gegeben. Sun Microsystems entwickelte hierzu die Java-Technologie *Applets.* Mit dieser Technologie lassen sich klassische Java-Anwendungen in eine gesicherte Umgebung im Browser laden. Die Kommunikation mit dem Server wird meist über http/HTTPS „getunnelt". Für größere Anwendungen, die das Internet nutzen, hat sich diese Technologie aber noch nicht verbreitet.

Wir wollen uns nun als Fallstudie einen eher technisch orientierten Vorschlag für eine moderne webbasierte Schichtenarchitektur im verteilten Umfeld anschauen: Den JEE-Architekturvorschlag von Sun Microsystems.

5.8.2 Fallbeispiel: JEE-Architektur

Die JEE-Spezifikation teilt ihren Architekturvorschlag für verteilte Anwendungen in vier Schichten (Tiers) ein, und berücksichtigt dabie die Web-Anbindung. Es wird also grundsätzlich von Anwendungen, die (auch) über einen Web-basierten Zugang verfügen, ausgegangen (siehe Abbildung 5-45):

- Client-Tier: Diese Schicht entspricht der Präsentationsschicht, die als sog. „native" Client oder als Browser-basierter Client implementiert sein kann.
- Web-Tier: Web-Frontend zur Steuerung der Präsentation. Diese Schicht nimmt Requests der Clients entgegen und bearbeitet sie. Als Ergebnis baut sie in der Regel dynamisch eine HTML-Seite auf und sendet diese an den Client.
- Business-Tier: Diese Schicht dient der Bereitstellung der Business-Logik, also des Anwendungskerns.
- EIS-Tier: Diese Schicht dient dem Zugriff auf Datenhaltungssysteme oder auf sonstige Fremdsysteme (EIS steht für Enterprise Information System).

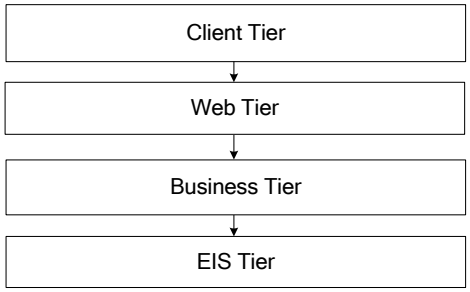


Abbildung 5-45: JEE-Schichtenarchitekturvorschlag

Eine Verfeinerung bildet die verfügbaren JEE-Technologien auf die JEE-Schichten ab. Im JEE-Vorschlag werden mehrere Java-basierte Technologien vorgeschlagen.

Physisch können die Web- und die Business-Tier in einem JEE-Server unterstützt werden. Die EIS Tier kann sehr vielfältig sein, wobei es sich meistens um ein Datenbankmanagementsystem handelt. Es sei aber nochmals erwähnt, dass es sich im Client-Server-Modell um ein Softwarekonzept handelt.

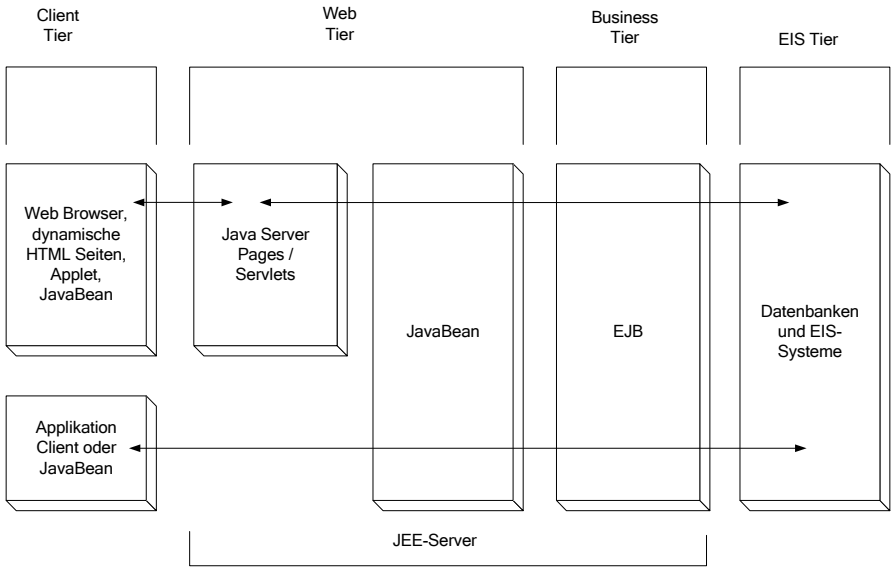


Abbildung 5-46: JEE-Architektur und JEE-Technologien

Die Architektur einer JEE-Anwendung sieht unter Einbeziehung dieser Technologien wie folgt aus (Abbildung 5-46):

– *Client Tier*: Clientseitig sind entweder reine Java-Bausteine meist mit grafischer Oberfläche oder aber Web-basierte Clients vorzufinden, die direkt im

Web-Broswser ablaufen. Die Clientsoftware wird bei Bedarf im Browser selbst zum Ablauf gebracht (Applets) oder es werden HTML-Seiten ausgegeben, die entweder statisch sind oder dynamisch in der Web Tier aufgebaut werden.

- *Web Tier*: Die Erzeugung der HTML-Seiten findet im Web Tier (in einem Web-Container) statt, wobei unter Nutzung des MVC-Modells Java-Bausteine auf Basis der Servlets-Technologie zum Einsatz kommen. Weiterhin werden die Model-Informationen in sog. Java Beans (spezielle Java-Klassen) verwaltet. Views werden durch Java Server Pages (JSP) repräsentiert.
- *Business Tier*: Für die Business-Logik wird die EJB-Technologie, eine serverseitige Komponententechnologie, die in Kapitel 2 erläutert wurde, bereitgestellt.
- *EIS Tier*: Hier sind die klassischen Datenbanken und sonstige Zugangsmechanismen auf Unternehmensdaten zugeordnet. Datenbankzugriffe erfolgen entweder direkt über EJB (container-managed) oder werden mit JDBC (Java Database Connection) auf SQL-Basis ausprogrammiert. JEE stellt auch ein spezielles API für die Implementierung von Konnektoren zum Zugriff auf Drittsysteme bereit.

Die Kommunikation zwischen den Schichten ist über standardisierte Kommunikationsprotokolle (RMI, HTTP, SOAP) möglich.

Als Beispiele für häufig verwendete Webtechnologien sollen noch Servlets und Java Server Pages, die beide Technologien der JEE-Architektur sind, betrachtet werden.

5.8.3 Servlets und Java Server Pages

Java Server Pages (JSP) und *Servlets* sind Spezifikationen aus der JEE von Sun Microsystems und speziell für Webanwendungen entwickelt. Heute enthält jeder Java-basierte Application-Server einen Web-Container, der auch als *Servlet-Container* bezeichnet wird. Ein Servlet-Container stellt auch eine Umgebung für die parallele Abwicklung von Requests zur Verfügung. In der Regel wird dies über einen Thread-Pool ermöglicht, der vom Servlet-Container verwaltet wird. Jedem Request wird ein Thread zur Bearbeitung zugewiesen, der am Ende der Bearbeitung wieder freigegeben wird.

Wie in Abbildung 5-47 dargestellt, ist der Servlet-Container und auch eine JVM im Web-Server integriert, wobei die Interfaces definiert sind. HTTP-Requests, die *Servlets* ansprechen, werden vom Web-Server an den Servlet-Container weitergereicht. Servlet-Container können auch „stand-alone“ laufen. Die Verbindung mit einem Web-Server ist vor allem dann sinnvoll, wenn neben *Servlets* auch noch statische HTML-Seiten benötigt werden. Letztere werden dann im Web-Server verwaltet. Der am meisten verwendete Servlet-Container ist *Apache Tomcat* (WWW-007), der heute in mehreren Application-Server-Produkten integriert ist.

Die Servlet-Basisklassen sind im *Java Servlet Development Kit* (JSDK) unter den Java-Packages *javax.servlet*, *javax.servlet.http* verfügbar. Servlets sind Java-Programme, welche die Schnittstelle *javax.servlet.Servlet* implementieren.

Man kann ein *Servlet* entweder von der Klasse *GenericServlet* ableiten oder von der etwas komfortableren Klasse *HttpServlet*. Bei Nutzung einer Ableitung der Klasse *GenericServlet* muss die Methode *service* überschrieben werden. Die Methode *service* wird vom Web-Server bei jedem Request aufgerufen. Die Klasse *HTTP-Servlet* enthält zwei Ereignisroutinen mit den Namen *doGet*, *doPost*, an die vom Servlet-Container die HTTP-Get- und -Post-Requests mit allen Eingabeparametern zur Bearbeitung weitergeleitet werden. Diese Methoden werden vom Programmierer überschrieben. Hier wird der Code zur Abarbeitung eines Requests eingefügt und der Programmierer entscheidet über den Aufbau der Response-PDU. Auch die Methoden *init* und *destroy* können für Initialisierungs- und Aufräumzwecke überschrieben werden.

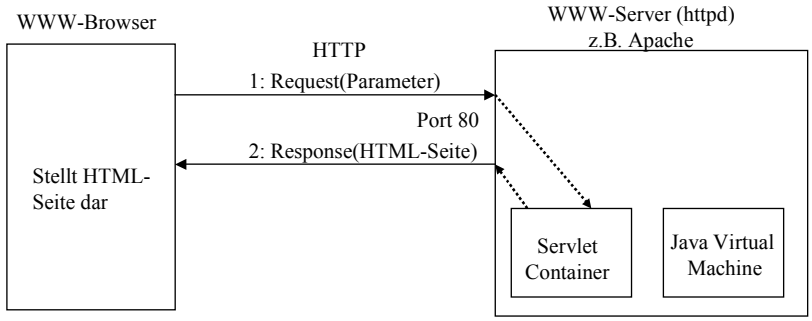


Abbildung 5-47: Zusammenspiel der Web-Komponenten bei Servlets

Der Servlet-Container stellt auch Schnittstellen für den Zugriff auf Request-, Response-Informationen, auf den Sessionkontext und weitere Daten zur Verfügung.

Beispiel: Abbildung 5-48 zeigt ein einfaches Klassendiagramm für eine eigene Servlet-Klasse, die hier von *GenericServlet* abgeleitet ist. *GenericServlet* implementiert wiederum das Basis-Interface mit dem Namen *Servlet*. Der Java-Code für das Beispiel ist nachfolgend etwas vereinfacht notiert. Im Beispiel wird die Methode *service* überschrieben. Die Parameter für die *service*-Methode sind ein Request- und ein Response-Objekt für den aktuell abzuarbeitenden Request. In der Methode wird ein Stream eröffnet, der bereits vom Servlet-Container über ein Objekt vom Typ *ServletResponse* zur Ausgabe der Ergebnisse bereitgestellt wird. In diesen Stream wird nur der Text „Hello World“ geschrieben, der dann in eine HTTP-Response-PDU aufgenommen wird.

```
package de.webapp.Examples.Servlet;
import java.servlet.*;
import java.io.IOException;
```

```
public class HelloWorld extends javax.servlet.GenericServlet
{
   public void init(...) {...}
   public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException
   {
        res.setContentType("text/html");
        ServletOutputStream out = res.getOutputStream();
        out.println("Hello World");
   }
   public void destroy(...) {...}
}
```

Ein Servlet durchläuft drei Lebensabschnitte. Zunächst wird eine Initialisierung durch Aufruf der Methode *init* vorgenommen (Konstruktion). Anschließend bearbeitet es beliebig viele Requests durch Aufruf der Methode *service* und schließlich wird es durch Aufruf der Methode *destroy* wieder freigegeben (Destruktion).

Jede Servlet-Instanz besitzt genau einen Kontext. Der Kontext enthält Eigenschaften aus der Umgebung und bietet umgebungsspezifische Dienste an. Hierfür wird dem Servlet der Zugriff auf ein Kontext-Objekt über das Interface *ServletContext* ermöglicht.

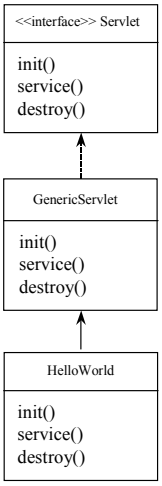


Abbildung 5-48: Hello-World-Servlet im Klassenmodell

JSPs sind prinzipiell nur eine Erweiterung von Servlets, die die HTML-Ausgabe-Kodierung erleichtert. Bei Servlets wird nämlich die Trennung von Präsentations- und Anwendungslogik noch nicht sauber unterstützt. Dies wird durch JSP besser gelöst. Der Aufbau einer JSP-Seite ist vergleichbar mit einer HTML-Seite. Sie ent-

hält neben HTML-Code beliebig viele JSP-Anweisungen. Eine JSP kann auch Java-Codeteile enthalten, die mit HTML-Coding (auch Template-Text genannt) kombiniert werden. Die Teile werden zur Laufzeit aneinandergereiht. Java Server Pages sind Servlets, die erst zur Laufzeit erzeugt werden. Im Prinzip kann man bei der JSP-Technik beliebig viel Java-Code einbinden und mit HTML-Code und JSP-Script-Elementen mischen.

Beispiel: Ohne auf Details einzugehen, wird hier ein kleines JSP-Programm zum Erzeugen eines Artikels aus unserem Artikelmanager-Beispiel dargestellt. Die Java Server Page wird durch einige spezielle Tags als solche gekennzeichnet. Aufgabe dieser JSP ist es lediglich, vier Parameter im Dialog (in einer HTML-Form) zu erfassen und an ein Servlet mit dem Namen *ControllerServlet.do* zur Bearbeitung weiterzuleiten. Bei Betätigen des Buttons „Submit“ wird ein HTTP-Post-Request generiert und an den Servlet-Container gesendet.

```
<%@ page session="true" %>
<%@ page errorPage="/error.jsp" buffer="32kb"%>
<html>
<head><title>Artikelverwaltung mit JSP</title></head>
<body>
<h1 align="center"><font face="Arial"
   size="5">Artikelverwaltung mit JSP</font></h1>
<hr>
<h2>
   Neuen Artikel anlegen
</h2>
<font size="4" face="Arial">
<pre>
   Bitte Artikeldaten eingeben:
<br>
<form method="post" action="ControllerServlet.do">
   Katalognummer <input type="text" name="katalogId" size="8">
   Beschreibung <input type="text" name="beschreibung" size="50">
   Warengruppe <input type="text" name="warengruppe" size="8">
   Preis(Euro) <input type="text" name="preis" size="12">
   <input type="submit" value="Absenden" name="Absenden">
   <input type="reset" value="Reset" name="Reset">
</form>
<hr>
</pre>
</body>
</html>
```

5.9 Resümee zur Architekturbetrachtung

Dieses letzte Kapitel sollte einen Überblick über grundlegende Aspekte von Architekturen für die Entwicklung verteilter betrieblicher Informationssysteme geben. Im Fokus der Betrachtung stand die IT-Architektur. Bei derartigen Anwendungen gibt es oft eine umfangreiche Datenhaltung, die im Sinne serviceorientierter Architekturen über Diensteschnittstellen zugreifbar ist. Daher wurden auch Möglichkeiten für Datenzugriffsarchitekturen betrachtet, die bei einer Client-Server-Sicht in der Regel serverseitig implementiert wird.

Auch der Zugang zu den verteilten Diensten kann aus Sicht des Clients unterschiedlich gestaltet werden. Möchte man eine möglichst starke Technologieunabhängigkeit, so wird der Einsatz bestimmter Patterns wie dem *Business-Delegate-Pattern* empfohlen. Eine grundlegende Einführung in diese Patterns wurde ebenfalls in diesem Kapitel gegeben. Auch Web-Architekturen wurden kurz skizziert. Hier benötigt man zwar einige spezielle Programmiertechniken (Webtechniken) für die Entwicklung der Clientseite, allerdings sind Web-Architekturen im Prinzip klassische Client-Server-Architekturen, bei denen der Client in einem Webserver zum Ablauf kommt. Die Möglichkeiten der Einziehung einer Business-Logik entsprechen den üblichen Schichtenarchitekturansätzen. In unserer Übersicht sollte eine durchgängige Sicht auf alle verteilten Bausteine typischer betrieblicher Informationssysteme gegeben werden.

Neben einer Einordnung in Architekturstile wurden auch einige Qualitätskriterien für Architekturen erläutert. Auch die heute in betrieblichen Informationssystemen noch etwas exotischen Architekturansätze wie *Peer-to-Peer* und *Grid* wurden eingeordnet und beispielhaft beschrieben. Sie sind heute für betriebliche Anwendungen noch nicht so verbreitet, aber man sollte die Weiterentwicklung beobachten.

In Zukunft wird man wohl zunächst großes Augenmerk auf die Weiterentwicklung des SOA-Grundgedankens legen. Aber auch eine automatisiertere Entwicklung von Architekturen ist seit einiger Zeit in Diskussion. Hier handelt es sich um die *Model Driven Architecture (MDA*. Diese noch etwas theoretische Herangehensweise an die Entwicklung von Softwaresystemen wird von der OMG (Object Management Group) vorangetrieben. Weitere Ausführungen hierzu sind über (WWW-009) zu finden.

5.10 Übungsaufgaben

1. Nennen Sie Qualitätskriterien zu Bewertung einer Software-Architektur!
2. Was ist ein datenzentrierter Architekturstil?
3. Was versteht man unter einem „unabhängigen Komponentenstil“?
4. Welchen Vorteil kann eine Peer-to-Peer-Architektur im Vergleich zu einer klassischen Client-Server-Architektur bringen?

5. Nennen Sie Vor- und Nachteile einer dreischichtigen im Vergleich zu einer zweischichtigen Client-Server-Architektur!
6. Ist es in einer dreischichtigen Client-Server-Anwendung möglich, die Transaktionsklammerung im Clientprogramm zu hinterlegen, wenn die Server zustandslos arbeiten? Erläutern Sie Ihre Entscheidung!
7. Wie würden Sie eine Transaktion in einer mehrschichtigen Client-Server-Anwendung programmieren, wenn alle Datenbankzugriffe über Serverdienste abgewickelt werden und die Transaktion mehrere Server nutzen muss?
8. Was sind applikationsspezifische Bausteine und was sind generische Softwarebausteine? Nennen Sie einige Beispiele!
9. Erläutern Sie das DAO-Pattern in Verbindung mit dem DTO-Pattern.
10. Wozu benötigt man das Business-Delegate-Pattern?
11. Diskutieren Sie die Auswirkungen einer ORM-Persistenzschicht auf die Leistungsfähigkeit des Datenzugriffs.
12. Welche Aufgabe hat der Entity-Manager in der Java Persistence API?
13. Was ist ein Persistenzkontext im Sinne der Java Persistence API?

6 Schlussbemerkung

Für eine gute Architektur und ein gutes Design eines komplexen verteilten Systems benötigt man ein breites IT-Wissen und viel Erfahrung. Ein guter Softwarearchitekt wird man erst durch das Üben in konkreten Projekten. Dieses Buch sollte dem Studierenden grundlegende Kenntnisse für den Entwurf und die Implementierung von verteilten Systemen und speziell von verteilten betrieblichen Informationssystemen vermitteln. Der Studierende sollte beim Lesen des Buches einen Einblick in aktuelle und auch historische Themenbereiche verteilter Systeme erhalten und wichtige praktische Gesichtspunkte vermittelt bekommen. Es wurde versucht, neben der Technik immer auch - soweit möglich - den Blick für das Machbare zu entwickeln.

In diesem Buch wurden wichtige Prinzipien, Konzepte, Modelle, Techniken, Technologien und Plattformen verteilter Systeme, deren Entwicklung und deren Nutzung vorgestellt. Die Schwerpunkte der Betrachtung lagen bei verteilten Objekten, verteilten Transaktionen, verteilten Komponenten und bei serviceorientierten Architekturen (im speziellen Webservices als erster konkreter Implementierungsvariante) und zwar immer mit dem Hintergrund einer guten Architektur.

Verteilte Anwendungssysteme spielen im betrieblichen Umfeld eine immer größere Rolle und es ist nicht abzusehen, dass sich dies in nächster Zeit ändert. Im Gegenteil, der Trend scheint sich fortzusetzen, wenngleich man sich auch hier über die Begrenzung der Komplexität Gedanken machen muss und auch macht.

Die Technologien für verteilte Systeme und deren Anwendung entwickeln sich weiterhin rasant. Aktuell entwickelt man in der Praxis mittlerweile viele Anwendungssysteme auf der Basis der Komponententechnologie (JEE/EJB, aber auch .NET 3). Die Standards und die Produkte sind aber noch lange nicht ausgereift. Sie werden ständig weiterentwickelt, was unschwer an der Versionsplanung bei EJB und bei Microsoft .NET erkennbar ist. Neue Ideen kommen hinzu und aus Praxiserfahrungen folgen Verbesserungen. Wie man anhand des Spring-Frameworks sehen kann, versucht man auch ständig, zu komplexe Basissysteme wieder einfacher zu gestalten. Zunehmend kommen auch neue Systembestandteile, die in diesem Buch nicht besprochen wurden, hinzu. So wird in den nächsten Jahren der Trend zum *Mobile Computing* weiter voranschreiten. Zunehmend werden also mobile Geräte auf betriebliche Informationssysteme zugreifen. Für die Entwicklung mobiler Systeme gibt es bereits einige Entwicklungsumgebungen im Java- (J2ME-API) und im .NET-Umfeld (Windows Mobile), die sich weiterentwickeln werden. Zunehmend werden sich auch Anwendungen aus dem Umfeld des *Ubiquitous Computing* entwickeln und Einfluss auf betriebliche Informationssysteme nehmen.

Diese Entwicklungen sind also ebenfalls zu beobachten. Noch sind diese Möglichkeiten nicht ganz für die betriebliche Praxis ausgereift und daher nur in einigen Spezialprojekten zu finden.

Als Softwarearchitekt und Entwickler ist es also weiterhin angesagt, sich kontinuierlich mit neuesten Technologien, Architekurkonzepten und Produkten auseinanderzusetzen und auch ausgereifte Techniken zu identifizieren. Denn eines sollte immer klar sein. Verteilte Softwaresysteme müssen in erster Linie die gewünschten Funktionen der Fachdomäne beherrschen. Wenn Technologien nicht ausgereift sind, führen sie zu zusätzlichen Problemen und Risiken in Entwicklungsprojekten.

7 Lösungen zu den Übungsaufgaben

7.1 Einführung in verteilte Systeme

1. *Versuchen Sie eine Definition für ein verteiltes Informationssystem mit eigenen Worten!*

   Verteilte Informationssysteme stellen meist kompliziertere verteilte Anwendungen dar und sind durch folgende Eigenschaften charakterisiert:

   – Verteilte Informationssysteme sind meist sehr groß, was den Codeumfang anbelangt (mehrere 100.000, teilweise mehrere Millionen Lines of Code).
   – Verteilte Informationssysteme sind sehr datenorientiert, d.h. die Datenhaltung steht im Zentrum der Anwendung. Üblicherweise werden die Daten in einer Datenbank verwaltet. Die zugrundeliegenden Datenmodelle sind meist sehr umfangreich.
   – Verteilte Informationssysteme sind extrem interaktiv und verfügen (neben Hintergrund- und Batch-Funktionalität) überwiegend über graphische Benutzerschnittstellen.
   – Verteilte Informationssysteme sind meistens auch sehr nebenläufig, was sich durch eine große Anzahl an parallel arbeitenden Benutzern äußert.

   Hinzu kommt, dass derartige Anwendungen oft auch sehr unternehmenskritisch sind.

2. *Nennen Sie drei Gründe für die Verteilung eines betrieblichen Informationssystems und erläutern Sie diese!*

   Folgende Gründe können u.a. angeführt werden:

   – Lastverteilung: Man kann gewisse Systembausteine auf mehrere Rechner verteilen, was einen Leistungsgewinn ermöglicht.
   – Ausfallsicherheit: Durch Verteilung bestimmter Services auf mehreren Serversystemen ist es möglich das System redundant auszulegen und damit gegen Ausfälle besser zu schützen.
   – Skalierbarkeit: Man kann ein System so konzipieren, dass es auf höhere Belastungen durch Hinzunahme weiterer Rechnersysteme reagieren kann.

3. *Welche Arten der Heterogentät wurden identifiziert?*

   Die Heterogenität von Betriebssystemen, Rechnerarchitekturen sowie Programmiersprachen und Laufzeitsystemen.

4. *Welche Fehlersituationen muss eine At-Most-Once-Implementierung für einen entfernten Aufruf zusätzlich zu At-Least-Once implementieren und wie kann die Implementierung erfolgen?*

   Der Server muss die Fehlersituation, dass ein Request beim Server mehrmals ankommt, erkennen. Duplikate müssen ausgefiltert werden, so dass die Ausführung des Requests nicht mehrmals erfolgt. Dies kann der Server über eine Requestliste erreichen. Bei jedem ankommenden Request ist zu prüfen, ob für ihn schon ein Eintrag in der Liste steht. Je nachdem sind dann die entsprechenden Aktionen auszuführen. Wurde ein Request schon ausgeführt, ist nur noch die Antwort zu übertragen. Weiterhin benötigt man eine zusätzliche ACK-Nachricht des Clients, die bestätigt, dass eine Antwort angekommen ist.

5. *Wird in lokalen Methodenaufrufen innerhalb eines Prozesses eine Exactly-Once-Fehlersemantik garantiert?*

   Nein, da im lokalen Umfeld auch nicht ohne weiteres garantiert werden kann, dass die Ausführung einer Methode nicht durch einen Systemausfall unterbrochen werden kann.

6. *Ist es sinnvoll, dass eine Java-Anwendung und eine C#-Anwendung ohne weitere Maßnahmen über eine Kommunikationsschnittstelle Objekte austauschen? Begründen Sie Ihre Entscheidung!*

   Nein, da eine Java-Anwendung einen C#-Objektstrom nicht interpretieren kann und umgekehrt. Man braucht also ein gemeinsames Verständnis der auszutauschenden Objekte/Daten oder eine Zwischenschicht, welche die Transformation der Nachrichten vornimmt.

7. *Wie löst Java das Problem der Heterogenität in der reinen Java-Welt?*

   Alle Partner nutzen die gleichen Serialisierungsmechanismen, da diese in der JVM implementiert sind. Damit sind die gleichen Datentypen und auch die gleiche Darstellung für alle Partner sichergestellt. Eine Umwandlung in eine andere Syntax ist damit nicht erforderlich.

8. *Was ist ein Big-Endian- im Unterschied zu einem Little-Endian-Format?*

   Es geht um die Darstellung von Integerwerten. Bei Little Endian wird das höchstwertige Byte eines Integerwerts an der höheren Speicheradresse abgelegt. Bei Big Endian wird das höchstwertige Byte eines Integerwerts dagegen an der niedrigsten Speicheradresse abgelegt.

9. *Wozu braucht man in verteilten Systemen eine symbolische Adressierung der verteilten Bausteine und welche Lösung gibt es hierfür?*

Man benötigt eine symbolische Adressierung, um das Auffinden von entfernten Anwendungsbausteinen ohne zu komplizierte Adressierungsinformationen in den Programmen zu ermöglichen. Hierfür gibt es in verteilten Systemen meistens Basisdienste wie Naming- oder Directory-Services, die eine Abbildung einer symbolischen Adresse auf eine „echte“ Adresse einer Bausteininstanz vornehmen. Dies erhöht die Transparenz für den Programmierer.

10. *Wozu braucht man Object-Relational-Mapper?*

    Einen ORM benötigt man, um Objekte aus objektorientierten Programmen auf Tabellen in relationalen Datenbanken abzubilden und umgekehrt.

11. *Erläutern Sie anhand eines Beispiels, wie eine Verteilung von Anwendungsbausteinen dazu beitragen kann, dass ein Anwendungssystem ausfallsicherer wird!*

    Durch eine redundante Verteilung von Services auf mehrere Rechner kann die Ausfallsicherheit erhöht werden. Fällt ein Baustein oder ein Rechner aus, so kann ein gleichwertiger anderer Baustein oder Rechner die Dienste weiterhin anbieten.

7.2 Konzepte und Modelle verteilter Kommunikation

1. *Wozu wird Threadpooling eingesetzt und welche Aufgabe hat ein Request-Dispatcher im Client-Server-Modell?*

   Ein Threadpool erzeugt und verwaltet mehrere Worker-Threads und stellt diese für die Bearbeitung von Requests bereit. Ein Request-Dispatcher verteilt in einem Server die anfallende Last, indem er die Anfragen analysiert und auf die Worker-Threads verteilt. Dieser Mechanismus dient der Parallelisierung der Requestbearbeitung.

2. *Erläutern Sie die Aufgaben eines Objektservers bei der Implementierung eines verteilten Objektsystems!*

   Ein Objektserver stellt eine Ablaufumgebung für verteilte serverseitige Objekte bereit. Er stellt einen Dispatching-Mechanismus zur Verfügung, um ankommende verteilte Methodenaufrufe über Objektadapter und Skeletons an die adressierten Objektinstanzen weiterzuleiten.

3. *Welche beiden grundlegenden Konzepte werden für die Realisierung von verteilten Objektsystemen herangezogen?*

   Das bereits aus lokalen Anwendungen bekannte Objektmodell und das RPC-Konzept.

4. *Was ist ein Proxy-Objekt und welche Aufgaben hat es?*

   Ein Proxy-Objekt ist ein lokal beim Client vorhandenes Stellvertreter-Objekt, das den Zugriff auf ein entferntes Objekt ermöglicht. Alle Methodenaufrufe eines Clientprogramms gehen lokal an das Proxy-Objekt, welches den Aufruf dann über das Netz an das entfernte Objekt weiterleitet. Weiterhin übernimmt das Proxy-Objekt das Marshalling und das Unmarshalling.

5. *Erläutern Sie das verteilte Garbage-Collection am Beispiel des Reference-Counting-Algorithmus!*

   Jeder Serverprozess verwaltet beim Reference-Counting eine Liste aller Clientprozesse bzw. Proxies, die entfernte Objektreferenzen auf seine Objekte nutzen. Generell wird beim verteilten Reference-Counting zum Zeitpunkt der Erstellung einer neuen Referenz auf ein Objekt und zum Zeitpunkt des Löschens eines Verweises auf das Objekt vom Nutzer (Client) eine Nachricht an den entfernten Objektserver gesendet. Der Server kann dann den zugehörigen Referenzzähler entsprechend inkrementieren bzw. dekrementieren.

6. *Welche Vorteile bieten verteilte Komponentensysteme im Vergleich zu verteilten Objektsystemen für den Anwendungsprogrammierer?*

   Komponentensysteme bieten gegenüber verteilten Objektsystemen folgende Vorteile:

   - Die Verwaltung des Lebenszyklus von Komponenten wird vom Komponentensystem übernommen. Ein Programmierer muss sich nicht mehr um die Erzeugung von Komponenteninstanzen kümmern.
   - Komponentensysteme ermöglichen den nebenläufigen Ablauf mehrerer Instanzen einer Komponente ohne zusätzliche Programmierung. Der Programmierer einer Komponente nutzt ein einfaches, sequentielles Programmiermodell. Die Parallelisierung übernimmt der Container. Komponenten werden vom Container bei Bedarf erzeugt und freigegeben.
   - Komponentensysteme stellen eine Reihe von Systemdiensten bereit, die vom Komponentenentwickler verwendet werden können. Hierzu gehören Transaktionsdienste, Persistenzdienste, Pooling von Datenbankverbindungen und Threadpooling.
   - Mechanismen für die Skalierbarkeit und auch für Fehlertoleranz und Lastverteilung werden durch den Container unterstützt. Komponenten können mehrfach gestartet werden.
   - Die Programmierung ist einfacher, da sich das Komponentensystem um den gesamten Lebenszyklus einer Komponente kümmert.

7. *Wie läuft ein Remote-Procedure-Call aus Sicht eines Clients prinzipiell ab? Erläutern Sie den Ablauf anhand des Fallbeispiels ONC RPC kurz und gehen Sie dabei auch auf die Nutzung des zugehörigen Naming-Service ein!*

   Bei RPC fordet ein Client den Dienst eines Serverbausteins so an, als ob es sich um einen lokalen Prozeduraufruf handeln würde. Für den Client bleibt die Kommunikation über das Netz weitgehend transparent. Die Abwicklung unterstützen generierte Client-Stubs und Server-Skeletons.

   Bei Sun RPC wendet sich der Client zunächst über einen Aufruf der Prozedur *clnt_create* an den Portmapper, der auch auf dem Serverrechner abläuft, auf dem der gesuchte Dienst implementiert ist. Der Portmapper liefert dem Client eine Referenz (Client-Handle) auf die Serverbaustein-Instanz zurück. Bei allen folgenden Aufrufen gibt der Client das Client-Handle an. Anschließend erfolgt der Prozeduraufruf über den Client-Stub.

8. *Wie wird eine entfernte Schnittstelle eines Servers bei ONC RPC beschrieben? Was wird aus der Schnittstellenbeschreibung über einen speziellen Compiler generiert?*

   Zur Beschreibung der Schnitttelle einer entfernten Prozedur wird die C-ähnliche XDR- oder RPC-Sprache verwendet. XDR nutzt das sog. implizite Typing. Ein Parameter wird ohne weitere Zusatzinformation mit dem RPC-Request gesendet.

   Das Tool rpcgen generiert aus der Interface-Beschreibung sowohl den Client-Stub, als auch den Server-Stub in der Sprache C sowie ein Headerfile und einen XDR-Filter (Marshalling-/Unmarshalling-Routinen).

9. *Was steckt hinter dem sog. Bootstrapping-Problem bei verteilten Anwendungen, die einen Naming- oder Directory-Service nutzen?*

   Der Client muss zuerst die Adresse eines Naming-Service herausfinden, bevor er von diesem die Adressen der verteilten Objekte ermitteln kann. Dies wird als Bootstrapping bezeichnet. In der Regel wird vom Basissystem ein initialier Kontext bereitgestellt, der das Weitersuchen ermöglicht.

10. *Wie funktioniert bei Java RMI und bei CORBA ein „Lookup“?*

    Bei Java-RMI funktioniert der Lookup über eine Methode namens *lookup*. Diese Methode dient dem Client dazu, mit Hilfe einer URL eine Objektreferenz aus einer Registry (RMI-Registry) zu lesen. Die RMI-Registry liegt auf dem Serverrechner, auf dem auch die entfernten Objekte platziert sind.

    Bei CORBA wird der CORBA-Naming-Service zum Auffinden einer Objektadresse verwendet. Über die Funktion *resolve_initial_references* kann zu einem Dienstnamen eine IOR (ORB-unabhängige Referenz) auf das entsprechende

CORBA-Objekt ermittelt werden. Der Naming-Service ermöglicht höhrere Orttransparenz als das RMI-Registry.

11. *Wie funktioniert das verteilte Garbage-Collection unter Java-RMI und unter CORBA?*

    Das Garbage-Collection erfolgt bei Java-RMI auf Basis eines Reference-Counting-Algorithmus in Verbindung mit Leases.

    In CORBA wird kein konkreter Garbage-Collection-Algorithmus vorgeschlagen. Aufgrund der Sprachunabhängigkeit ist es in CORBA nicht ohne weiteres möglich, einen verteilten Garbage-Collection-Mechanismus zu unterstützen, der für alle Bindings funktioniert.

12. *Wie wird bei Java-RMI ein Interface eines verteilten Objekts definiert? Grenzen Sie die Vorgehensweise zu der in CORBA ab.*

    In Java-RMI wird ein Interface direkt in Java als Java-Interface, welches das Remote Interface beerbt, definiert. CORBA nutzt für die Interface-Definition eine eigene, sprachunabhängige IDL.

13. *Wie wird in RMI ein Remote-Objekt programmiert? Gehen Sie dabei auf die Nutzung der RMI-Basis-Interfaces- und Klassen ein.*

    Ein RMI-Objekt wird als Java-Klasse programmiert, die von der Basisklasse UnicastRemoteObject erbt und ein Remote-Interface implementiert. Die Implementierungsklasse eines RMI-Objekts ist durch die Vererbung und auch durch die spezielle Behandlung von Remote-Exceptions erkennbar.

14. *Welche Channel-Typen sind bei .NET Remoting für die Kommunikation zwischen Client und Server möglich?*

    Folgende Kanaltypen werden unterstützt:
    - HTTP-Kanal; Transport über HTTP
    - TCP-Kanal: Transport über TCP
    - IPC-Kanal: Lokaler Transport über IPC-Mechanismen

15. *Was versteht man bei .NET Remoting unter serveraktivierten und clientaktivierten Objekten (SAO und DAO)?*

    Serveraktivierte Objekte (SOA) sind Objekte, deren Lebensdauer direkt vom Server gesteuert wird. Fordert ein Client eine Instanz eines serveraktivierten Objekts an, wird in der Anwendungsdomäne des Clients ein Proxy erstellt. Es gibt zwei sog. Aktivierungsmodi für serveraktivierte Objekte. Dies sind Singleton- und SingleCall-Objekte.

Clientaktivierte Objekte (CAO) sind Objekte, deren Lebensdauer von der aufrufenden Anwendungsdomäne (also vom Client) gesteuert werden. Die Objekte liegen aber nicht wirklich im Client. CAOs sind statusbehaftete Serverbausteine. Jeder Client erhält eine eigene Instanz des verteilten Serverbausteins. Die Lebensdauer eines CAO richtet sich nach der Lebensdauer des Clients.

16. *Was ist bei .NET Remoting ein Singleton und was ist der Unterschied zu einem SingleCall-Objekt? Handelt es sich hier um SAOs oder CAOs?*

    Beides sind SAOs. Bei einem Singleton wird im Server nur eine Instanz des verteilten Objekts angelegt. Ein Singleton wird beim ersten Aufruf einer Methode instanziert und kann von beliebig vielen Clients verwendet (aktiviert) werden. SingleCall-Objekte sind verteilte Objekte, bei denen für jeden Request eine eigene Instanz verwendet wird. SingleCall-Instanzen vom gleichen Typ sind untereinander isoliert. SingleCall-Objekte sind zustandslose Serverbausteine.

17. *Was versteht man unter RMI/IIOP (RMI over IIOP) und wozu ist es nützlich?*

    RMI/IIOP bzw. RMI over IIOP bildet die RMI-Mechanismen auf das standardisierte Protokoll IIOP ab. Dies ist sinnvoll, wenn ein Java/EJB-Client mit einem CORBA-Objekt kommunizieren möchte oder umgekehrt ein CORBA-Objekt mit einer EJBean. Für die Übertragung eines Requests wird IIOP genutzt.

18. *Skizzieren Sie typische Protokollstacks für das RMI-Transportprotokoll!*

    Ein typischer Protokollstack für RMI enthält ein Transportprotokoll, wobei derzeit TCP genutzt wird. Darüber ist der sog. *Remote Reference Layer* angesiedelt, der derzeit eine *Point-to-Point*-Kommunikation zwischen Client und Server implementiert. Client-Stub und Server-Skeleton nutzen die Dienste dieser Schicht.

19. *Wie werden bei Java-RMI die Argumente und Returnwerte eines Methodenaufrufs an ein entferntes Objekt übertragen (by-reference, by-value)?*

    Einfache Datentypen (Elementartypen wie *int* und *long*) werden bei Java-RMI an den RMI-Proxy als Werte übergeben (Call-by-value). Die Übergabe von Java-Objekten als Parameter erfolgt zwar lokal im Client zwischen Client-Objekt und RMI-Stub über Referenzen, für die Übertragung werden die Objekte aber in den RMI-Strom kopiert und damit ebenfalls als Wertparameter übergeben (Call-by-value=Call-by-copy). Die Übergabe von Referenzen auf Remote-Objekte erfolgt als Referenzparameter (Call-by-reference).

20. *Wann verwendet man stateless und wann stateful Session-Beans?*

    Bei stateful Session-Beans wird für jeden Client eine eigene Instanz zugeordnet. Es wird ein Konversationszustand verwaltet. Alle Anwendungen, bei denen über einen Client-Request hinweg im Server ein Zustand verwaltet werden muss, sind Kandidaten für die Nutzung von stateful Session-Beans (Beispiel: Warenkorb für einen Online-Shop).

    Stateless Session-Beans sind für Anwendungen oder Anwendungsteile gedacht, die serverseitig zustandslos arbeiten können. Stateless Session-Beans ermöglichen eine effizientere Abarbeitung der Methoden und sind auch robuster als stateful Session-Beans.

21. *Wozu dient das JNDI-API?*

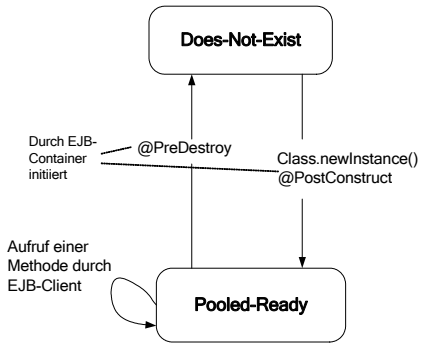
    Das JNDI-API stellt eine Schnittstelle für einen Naming- und Directory-Service zur Verfügung. Sowohl das Registrieren von Ressourcen als auch das Auffinden dieser (Lookup) ist über die API möglich.

22. *Warum kann man sich als Bean-Provider nicht darauf verlassen, dass alle Beans in einer JVM ablaufen? Nennen Sie Restriktionen, die sich für den Programmierer daraus ergeben!*

    Alle Beans werden vom EJB-Container verwaltet. Es ist in der Spezifikation nicht vorgegeben, wie der Container-Hersteller dies implementiert. Er kann dazu eine JVM oder auch mehrere JVMs nutzen und diese auch beliebig verteilen. Daher müssen statische Variablen als *final* deklariert werden (keine globalen Variablen in einer Klasse, die im Container läuft) und es dürfen auch keine Java-Synchronisationsprimitiven (z.B. wait, notify, …) verwendet werden.

23. *Skizzieren und beschreiben Sie den Zustandsautomaten einer stateless Session-Bean!*

    Es gibt nur zwei Zustände: Entweder die Bean-Instanz ist nicht vorhanden („Does-not-Exist“) oder sie liegt in einem Bean-Pool für die Bearbeitung neuer Methodenaufrufe („Pooled-Ready“) bereit. Der Zustandsübergang vom Zustand „Pooled-Ready“ nach „Does not Exist“ wird durch den Container initiiert. Der Pool wird auch durch den Container verwaltet. Bei den Zustandsübergängen werden annotierte Methoden (Annotationen @PreDestroy und @PostConstruct) aufgerufen, sofern sie definiert sind.



24. *Was ist im Messsage-Passing-Modell eine synchrone persistente Kommunikation im Vergleich zu einer asynchronen persistenten Kommunikation?*

    Bei synchroner persistenter Kommunikation speichert das Kommunikationssystem eine Nachricht beim Empfänger (in einem Puffer) ab, bevor der Sender weiterarbeiten kann.

    Bei asynchroner persistenter Kommunikation wird eine gesendete Nachricht dauerhaft vom Kommunikationssystem zwischengespeichert, bis der adressierte Empfänger sie liest, während der Sender gleich weiterarbeiten kann.

25. *Nennen Sie Einsatzmöglichkeiten bzw. Anwendungen für Nachrichtenwarteschlangensysteme!*

    Warteschlangen dienen der Entkopplung von Anwendungssystemen und eignen sich sehr gut für die Integration von „neu entwickelten" Anwendungen in bestehende IT-Landschaften. Sie ermöglichen auch die Entwicklung ereignisorientierter Kommunikationslösungen, bei der z.B. ein Partner mehrere andere Partner (one-to-many) über bestimmte Ereignisse informiert.

26. *Erläutern Sie das P2P-Modell im Vergleich zum Publish-Subscribe-Modell!*

    Bei einer P2P-Kommunikation kommunizieren zwei Kommunikationsparter gleichberechtigt (symmetrisch) und keiner ist, wie im asymmetrischen Client-Server-Modell, in einer speziellen Rolle. Im Publish-Subscribe-Modell registrieren (subscribe) sich Konsumenten von bestimmten Nachrichtentypen für einen bestimmten Kommunikationsaspekt (Topic genannt). Produzenten erzeugen diese Nachrichtentypen und senden (publish) sie an die Konsumentengruppe.

27. *Was ist ein JMS-Topic?*

    Ein JMS-Topic stellt einen abstrakten, allen Beteiligten bekannten „Knotenpunkt" dar. Dieser Knotenpunkt kann Nachrichten empfangen und weiterlei-

ten. Die Nutzer eines Topics heißen auch Produzenten und Konsumenten. Die Anzahl der Produzenten und Konsumenten kann sich dynamisch verändern.

7.3 Verteilte Dienstaufrufe und Webservices

1. *Was ist der große Vorteil von Webservices im Vergleich zu RMI oder CORBA?*

   Webservices sind im Gegensatz zu RMI oder CORBA auch in heterogenen und Umgebungen unabhängig von Technologien einsetzbar. Die Implementierung der Services kann beliebig sein. Somit lassen sich Bausteine, die mit beliebiger Technologie entwickelt wurden, mit einer Schnittstelle versehen, die eine Nutzung über einen neutralen Mechanismus ermöglicht.

2. *Diskutieren Sie den Einfluss von SOAP auf die Leistungsfähigkeit einer Client-Server-Anwendung!*

   Webservices basieren auf XML. Alle SOAP-Nachrichten werden in einem XML-Dialekt übertragen. Das Validieren und Parsen der XML-Nachrichten sowie das Umwandeln der Datentypen eines Service-Endpoints in eine WSDL-konforme Notation und umgekehrt erfordert zusätzliche Ressourcen. Zudem sind je nach Kodierungsstil die Nachrichten länger als bei klassischen Protokollen. Die Leistung wird also insgesamt beeinträchtigt.

3. *Welche Aufgabe hat UDDI bei Webservices?*

   UDDI ist eine Schnittstelle für eine zentrale Dienstregistratur. Die Implementierung dieser Registratur ist nicht festgelegt. Ein Diensterbringer kann seine Webservices über UDDI registrieren und damit können diese durch Dienstnehmer gefunden und genutzt werden. Obwohl bereits konkrete Implementierungen existieren, werden diese noch nicht im großen Stil eingesetzt.

4. *Mit welchen Transportprotokollen arbeitet SOAP?*

   SOAP ist an kein spezielles Transportprotokoll gebunden. Aktuell wird es meistens auf Basis von HTTP implementiert, aber auch eine Implementierung auf Basis von SMTP oder auf Basis eines nachrichtenbasierten Protokolls ist denkbar.

5. *Was ist ein SOAP-Prozessor?*

   Ein SOAP-Prozessor (auch SOAP-Engine) stellt eine Laufzeitumgebung für Webservices bereit und verarbeitet ankommende und abgehende SOAP-Nachrichten. SOAP-Prozessoren sind heute in fast allen Application-Server-Produkten integriert.

6. *Erläutern Sie die Einbettung von SOAP-PDUs in HTTP-PDUs! Warum ist HTTP als „Transportprotokoll" für SOAP so gut geeignet?*

   Der SOAP-Request und die SOAP-Response werden jeweils in einer HTTP-GET- oder in einer HTTP-POST-PDU übertragen. HTTP ist gut geeignet, da SOAP in der Regel einen synchronen entfernten Dienstaufruf realisiert, was sich leicht auf HTTP abbilden lässt. Zudem ist HTTP fast überall verfügbar und genauso wie SOAP textbasiert.

7. *Wie wird einem Webservice ein Port zugeordnet?*

   Ein Port wird in der WSDL-Beschreibung eines Webservice im Service-Element zugeordnet. Dort wird ein Portname für ein Binding festgelegt und eine URL definiert, über die der Webservice erreichbar ist.

8. *Erläutern Sie die Mechanismen der Serialisierung bei Webservices?*

   Die Serialisierung wird über generierte Stubs und Skeletons durchgeführt. Sie übernehmen das Mapping von lokalen Datentypen auf WSDL-Datentypen und umgekehrt. Stubs und Skeletons können über Tools, die der Hersteller einer Entwicklungsumgebung bereitstellt aus den WSDL-Beschreibungen erzeugt werden.

9. *Was ist der Unterschied zwischen JAX-RPC und JAX-WS?*

   Beides sind Java-basierte Spezifikationen zur Implementierung von Webservices. JAX-WS ist die Weiterentwicklung von JAX-RPC und hat das Ziel die Entwicklung und das Deployment von WebService-Anwendungen zu beschleunigen. Bei JAX-WS werden Java-Annotationen verwendet.

10. *Was versteht man unter einer serviceorientierten Architektur? Diskutieren Sie den Ansatz anhand von Webservices! Kann man eine serviceorientierte Architektur auch auf Basis von JEE/EJB realisieren?*

    SOA ist ein Architekturkonzept, dass die Bereitstellung von Diensten in einer heterogenen Umgebung unterstützt. Webservices stellen eine konkrete technische Basis für die Implementierung einer SOA zur Verfügung. Allerdings ist eine SOA nicht an Webservices gebunden. Es können auch andere Technologien zur Umsetzung des Konzepts genutzt werden. Durch die weitgehende Implementierungsunabhängigkeit von Webservices wird der SOA-Gedanke aber stark unterstützt. JEE/EJBs eignen sich für die Implementierung einer SOA gut. Man kann mit dieser Technologie Serverbausteine gut kapseln. Zur Kommunikatioin sollte aber besser SOAP und nicht RMI/IIOP verwendet werden.

11. *Was bedeutet bei SOAP Document/Literal im Unterschied zu RPC/literal?*

    Hier handelt es sich um zwei Kombinationsmöglichkeiten aus Nachrichtenformat (Style) und Kodierungsstil (Use). Die Begriffe sind recht verwirrend und historisch bedingt. Style=Document/Use=Literal bedeutet, dass der SOAP-Message-Body ausschließlich Dokumente nach XML-Schema enthält, d.h. die ganze Nachricht kann vollständig validiert werden. Dieser Stil wird innerhalb der Webservice-Community als favorisierte Lösung propagiert. Bei Style=RPC/Use=Literal werden SOAP-Nachrichten als RPC-Aufrufe mit Parametern und Returncodes realisiert. Jede Nachricht hat einen eigenen Schematyp. Die Datentypen der Parameter werden in der Nachricht nicht explizit angegeben, sondern sie ergeben sich implizit aus der konkreten Schema-Beschreibung.

7.4 Konzepte, Modelle und Standards verteilter Transaktionsverarbeitung

1. *Welche Rolle spielt im DTP-Modell der Transaktionsmanager und welche Schnittstelle bietet er in Richtung Anwendungsprogramm (AP) an?*

   Der TM verwaltet den Kontext von lokalen und globalen Transaktionen. Er koordiniert alle an einer Transaktion beteiligten Ressourcenmanager über die XA-Schnittstelle. Dem AP wird die TX-Schnittstelle bereitgestellt.

2. *Wozu dient der CRM im DTP-Modell?*

   Er ist ein lokaler Stellvertreter aller entfernten Ressourcenmanager (RMs), die in eine verteilte Transaktion involviert sind. Der CRM beauftragt die entfernten TMs mit der Commit-Bearbeitung für ihre Knoten. Der lokale TM führt die Ergebnisse der lokalen RMs und der entfernten TMs zusammen und entscheidet über den Ausgang der Transaktion. Ein entfernter RM kann nicht direkt, sondern nur über ein auf dem Knoten liegendes AP verwendet werden.

3. *Wie startet ein AP eine Transaktion?*

   Ein AP setzt einen Funktionsaufruf tx_begin an den TM ab, der daraufhin eine globale Transaktions-Id (XID) generiert und je nach Registrierungsvariante alle dem AP hinzugebundenen RMs über den Beginn einer Transaktion informiert, also den Transaktionskontext propagiert.

4. *Was ist im DTP-Modell ein Thread-of-Control bzw. ein Ausführungspfad?*

   Bei der Abwicklung einer Transaktion müssen AP, TM, CRMs und RMs das gleiche Verständnis über den Transaktionskontext haben. Dieses gemeinsame Verständnis wird auch logisch als *Thread-of-Control* bezeichnet. Über die Art

und Weise, wie ein *Thread-of-Control* iumplementiert werden soll, macht das DTP-Modell keine Aussage.

5. *Erläutern Sie die beiden Phasen des 2PC-Protokolls!*

   Beim 2PC-Protokoll sendet der Koordinator in der ersten Phase (Prepare-Phase) einen Request (Prepare-Request) an alle Teilnehmer einer Transaktion, damit diese sich auf das Transaktionsende vorbereiten. Die Teilnehmer antworten (votieren) mit "ready" oder "not-ready", je nachdem, ob sie die Transaktion erfolgreich zu Ende führen können oder nicht. Votiert ein Teilnehmer mit "not-ready", setzt er die innerhalb der Transaktion ausgeführten Operationen selbstständig zurück. Nur wenn alle Teilnehmer mit "ready" antworten, wird vom Koordinator die zweite Phase eingeleitet, um die Transaktion erfolgreich zu beenden. Wenn ein Teilnehmer "ready" meldet, muss er in der Lage sein, die Transaktion aus seiner Sicht zu beenden oder zurückzusetzen, je nachdem, wie sich der Koordinator entscheidet. Eine einmal getroffene Entscheidung darf weder vom Koordinator noch von einem Teilnehmer zurückgenommen werden. In der zweiten Phase sendet der Koordinator einen Commit-Request, um die Transaktion abzuschließen. Die Teilnehmer nehmen den Request entgegen, führen ihn aus und beantworten ihn entsprechend.

6. *Was bedeutet im 2PC-Protokoll ein „unsicherer Zustand", wer kann ihn einnehmen und in welcher Situation tritt er auf?*

   Das 2PC-Protokoll ist ein blockierendes Commit-Protokoll. Der Zustand eines Teilnehmers, in dem er blockiert ist und auf die Commit-/Rollback-Entscheidung des Koordinators wartet, wird auch als unsicherer Zustand bezeichnet.

7. *Wie wird üblicherweise eine XA-Unterstützung durch den Hersteller eines DBMS realisiert?*

   Die Realisierung erfolgt üblicherweise in einer XA-Library, die der Datenbankhersteller bereitstellt. Ein Hersteller eines Application-Servers oder eines Transaktionsmanagers kann die Library einbinden. Die Library stellt die RM-Funktionen der XA-Schnittstelle bereit und versteckt die Implementierung der Kommunikation mit dem Datenbankmanagementsystem.

8. *Was passiert bei einer 2PC-Koordinierung zwischen einem Koordinator und zwei Teilnehmern, wenn ein Teilnehmer während der Phase 1 „ready" an den Koordinator meldet, der zweite allerdings „not ready"?*

   In diesem Fall entscheidet der Koordinator auf Abbruch der Transaktion, protokolliert dies in seinem Transaktionslog und sendet eine Abort-PDU an den Teilnehmer, der mit „ready" antwortete.

9. *Sollte ein PC-basierter Client als Transaktionskoordinator dienen? Begründen Sie Ihre Entscheidung!*

   Ein Client, der auf einem eher unsicheren PC abläuft, sollte nicht als Transaktionskoordinator dienen. Fällt nämlich der Koordinator während einer Commit-Bearbeitung aus, sind die Teilnehmer blockiert und bleiben unsicher, bis er wieder aktiv ist. Da sich der unsichere Client in der Regel aber nicht den letzten Zustand der Transaktion besorgen kann, kann der Blockierzustand möglicherweise nicht mehr ohne manuellen Eingriff aufgelöst werden.

10. *Warum ist ein persistenter Logging-Mechanismus wie z.B. nach dem WAL-Prinzip für die Realisierung von (verteilten) ACID-Transaktionen notwendig?*

    WAL ist notwendig, da während einer Transaktion zu jeder Zeit ein Ausfall einer Systemkomponente möglich ist und daher ein Recovery notwendig wird. Wenn vor der Ausführung einer Operation ein Logsatz dauerhaft geschrieben wird, der den Zustand vor und nach der auszuführenden Operation enthält, kann beim anschließenden Recovery mit dieser Log-Information sowohl ein Redo, als auch ein Undo durchgeführt werden. Damit können alle Recovery-Protokolle unterstützt werden. Was genau protokolliert wird, hängt davon ab, welche Recovery-Variante unterstützt wird. Heutige Implementierungen nutzen meist Undo/Redo. Diese Variante benötigt ein Before- und ein After-Image.

11. *Wozu muss ein Teilnehmer an einer Transaktion, der in der Phase 1 einer 2PC-Koordination mit „ready" antwortet, in der Lage sein?*

    Der Teilnehmer muss die Transaktion zu Ende führen oder diese zurücksetzen können. Die Entscheidung trifft der Koordinator.

12. *Welche Aufgabe hat JTA in einer JEE/EJB-Umgebung und welche Schnittstellen stellt JTA zur Verfügung?*

    Für die Unterstützung von Transaktionen ist innerhalb der EJB-Architektur die Java Transaction API (JTA) als grundlegende Schnittstelle bzw. als Satz von Schnittstellen vorgesehen. JTA liefert höherwertige Transaktionsdienste, die aus drei Teilen bestehen:

    – Eine Schnittstelle für Anwendungsprogrammierer, mit denen diese Transaktionsgrenzen setzen können. Sie enthält die klassischen Methoden *begin*, *commit* und *rollback*.
    – Ein Java-Mapping für das standardisierte XA-Protokoll der Open Group für die Teilnahme an globalen Transaktionen, die von einem Transaktionsmanager koordiniert werden.
    – Eine Schnittstelle für einen Application-Server, um mit einem Transaktionsmanager zu kommunizieren. Diese Schnittstelle ist für die Transakti-

onsabwicklung durch den Application-Server (bzw. EJB-Container) notwendig und wird für *container-managed* Transactions verwendet.

13. *Welches Transaktionsmodell unterstützt EJB?*

    Es werden nur flache Transaktionen unterstützt.

14. *Erläutern Sie den Unterschied zwischen bean-managed, container-managed und client-managed Transaktionen.*

    Bei bean-managed Transaktionen setzt der Bean-Entwickler serverseitig die Transaktionsgrenzen.

    Bei container-managed Transaktionen sieht der Bean- und Client-Enwickler nichts von Transaktionen. Die Transaktionssteuerung erfolgt implizit über den Application-Server, der üblicherweise einen Transaktionsmanager implementiert. Die Transaktionseinstellungen werden über Annotationen oder über einen Deployment-Descriptor vorgenommen.

    Bei client-managed Transaktionen definiert der Client die Transaktionsgrenzen und führt bei Bedarf ein Commit oder einen Rollback aus.

15. *Wie kann ein EJB-Client nach einem Ausfall während einer Commit-Bearbeitung erfahren, ob die gerade ausgeführte Transaktion erfolgreich zu Ende geführt wurde oder nicht?*

    Er kann es gar nicht erfahren, da er über keine Recovery-Information verfügt. Die Anwendung muss selbst Operationen an der Benutzerschnittstelle implementieren, die eine Überprüfung der letzten Aktionen ermöglicht. Dies ist aber nicht immer möglich.

16. *Können ein EJB-Container und ein .NET-Enterprise-Services-Container eine gemeinsame Transaktion ausführen und diese koordinieren? Begründen Sie Ihre Entscheidung!*

    Dies ist grundsätzlich nicht möglich, da .NET den Microsoft Transaction Server (MTS) und EJB den auf dem OTS-Standard basierenden Java Transaction Service (JTS) verwendet. MTS und JTS sind nicht miteinander kompatibel und können daher auch keine Transaktionskontexte austauschen und auch kein verteiltes Commit durchführen.

17. *Warum funktioniert das starre ACID-Konzept für Transaktionen mit Webservices nicht? Welches Transaktionsmodell würden Sie für eine Webservice-basierte Anwendung, die drei Webservice-Operationen unterschiedlicher Diensteprovider in einer zusammenhängenden (atomaren) Aktionsfolge nutzen muss, vorschlagen?*

Das ACID-Transaktionskonzept ist für Webservice-Transaktionen zu starr. Es müssten alle drei Diensteprovider die gleichen Transaktionsmechanismen unterstützen.

Sinnvoll erscheint trotz einiger anderer Vorschläge aus Standardisierungsgremien wie OASIS derzeit immer noch, jeden Webservice für sich als eigenständige Transaktion zu implementieren. Ein Recovery kann im Fehlerfall dann nur durch eine Kompensationstransaktion erfolgen, die aber in der Anwendung selbst realisiert werden muss. Dies ist ein pragmatischer Ansatz, der natürlich nicht immer ganz sicher funktioniert, aber dafür robust ist.

18. *Was sind im BTP-Transaktionsstandard Cohesive Business Transactions?*

Cohesive Business Transactions sind offen geschachtelte Transaktionen. Der Koordinator erzeugt die Transaktion und ruft innerhalb dieser mehrere Services in Subtransaktionen auf. Wenn eine Subtransaktion nicht erfolgreich verläuft, kann der Koordinator über die weitere Ausführung bzw. den Abbruch entscheiden. Dies muss vom Entwickler der Haupttransaktion bei der Programmierung festgelegt werden. Die ACID-Korrektheitskriterien können hier nicht erfüllt werden.

7.5 Architekturen verteilter betrieblicher Anwendungen

1. *Nennen Sie Qualitätskriterien zu Bewertung einer Software-Architektur!*

Es gibt viele Kriterien, die zur Bewertung herangezogen werden können. „Sichtbare Kriterien (bei der Ausführung des Systems erkennbar) sind Funktionsfähigkeit, Leistung und Skalierbarkeit, Transaktionssicherheit, Sicherheit, Verfügbarkeit und Usability.

Zu den nicht beobachtbaren Qualitätskriterien gehören die Einhaltung von Standards, die Strukturiertheit der Software, die Portabilität, die Wiederverwendbarkeit, die Integrierbarkeit und die Testbarkeit.

Wichtig ist auch die Einhaltung der Prinizipien der Softwareentwicklung. Dazu gehören: Separation of Concerns, Comprehension (intellektuelle Beherrschbarkeit), Korrektheit und Vollständigkeit, Ersetzbarkeit, Buildability, lose Kopplung und hohe Kohäsion sowie die Verringerung der Komplexität.

2. *Was ist ein datenzentrierter Architekturstil?*

Hauptziel dieses Architekturstils ist die Verwaltung von Daten und der Zugriff auf die Daten.

3. *Was versteht man unter einem „unabhängigen Komponentenstil“?*

Dieser Architekturstil umfasst eigenständige Peers (Objekte oder Prozesse), die unabhängig voneinander arbeiten und miteinander über Nachrichten kommunizieren.

4. *Welchen Vorteil kann eine Peer-to-Peer-Architektur im Vergleich zu einer klassischen Client-Server-Architektur bringen?*

   Peer-to-Peer-Architekturen sind im Vergleich zu klassischen Client-Server-Architekturen robuster. Es gibt keinen Server, der als Single Point of Failure einen Engpass bilden kann. Am robustesten sind reine Peer-to-Peer-Architekturen, da hier alle Peers gleichberechtigt sind. Nicht alle Anwendungen können aber auf Basis einer Peer-to-Peer-Architektur realisiert werden.

5. *Ist es in einer dreischichtigen Client-Server-Anwendung möglich, die Transaktionsklammerung im Clientprogramm zu hinterlegen, wenn die Server zustandslos arbeiten? Erläutern Sie Ihre Entscheidung!*

   Diese Vorgehensweise ist grundsätzlich schwierig, da ein zustandsloser Server jeden Methodenaufruf abschließt und damit auch evtl. Transaktionen, die während der Methodenbearbeitung ausgeführt werden, beendet. Ein Rollback, das der Client initiiert, könnte nur über Kompensationstransaktionen ausgeführt werden. Clientsysteme können aber leichter ausfallen, da sie meist nicht hochverfügbar sind. Bei Ausfall eines Clients ist das Transaktionsende dann undefiniert. Generell sollte in Client-Server-Anwendungen die Transaktionssteuerung nicht in einem doch recht unsicheren Client liegen.

6. *Nennen Sie Vor- und Nachteile einer dreischichtigen im Vergleich zu einer zweischichtigen Client-Server-Architektur!*

   Vorteile einer dreischichtigen Architektur sind, dass man die Business-Logik sauber in Serverbausteinen kapseln und auch den Ressourcenzugriff von der Präsentationsschicht entkoppeln kann. Nachteilig ist der zusätzliche Aufwand für die Programmierung und Pflege der Business-Logik-Schicht.

   Vorteile einer zweischichtigen Architektur sind der geringere Programmieraufwand und bis zu einer bestimmten Größe des Systems auch die vereinfachte Pflege. Nachteilig ist die fehlende Entkopplung der Präsentationsschicht vom Ressourcenzugriff. Datenbankänderungen wirken sich unmittelbar in der Präsentationsschicht aus, was beim dreischichtigen Ansatz nicht immer so ist.

   Grundsätzlich ist ab einer bestimmten Größe des Systems eine dreischichtige Architektur vorteilhafter, allerdings sind auch immer alle Randbedingungen zu betrachten. Bei einer Software, die zwar groß ist, aber nur einmal genutzt wird, kann dies schon wieder anders aussehen. Wie man sieht, ist die Entscheidung nicht einfach und es gibt kein Standardvorgehen.

7. *Wie würden Sie eine Transaktion in einer mehrschichtigen Client-Server-Anwendung programmieren, wenn alle Datenbankzugriffe über Serverdienste abgewickelt werden und die Transaktion mehrere Server nutzen muss?*

   In diesem Fall ist die Anwendung von verteilten Transaktionen möglich. Über einen Transaktionskoordinator, der in der Regel in einem Application-Server vorhanden ist, erfolgt die verteilte Koordination. Die Transaktionen sollten im Server ausgeführt werden, der Client nutzt nur die Serverdienste und sieht die Transaktionssteuerung nicht.

   Da verteilte Transaktionen in der Praxis noch recht problematisch sind (2PC-Protokoll ist blockierend), kann man auch auf verteilte Transaktionen verzichten. Jeder Dienst führt dann für sich abgeschlossene Transaktionen aus. Rollback-Situationen sind dann aber in der Anwendung durch passende Kompensationsstrategien anzuwickeln.

   Es kommt also auf die konkrete Anwendung an. Wenn man verteilte Transaktionen vermeiden kann, ist dies heute zu empfehlen.

8. *Was sind applikationsspezifische Bausteine und was sind generische Softwarebausteine? Nennen Sie einige Beispiele!*

   Applikationsspezifische Softwarebausteine sind nur für spezielle Anwendungen konzipiert und einsetzbar. Sie implementieren Anwendungslogik und die Wiederverwendbarkeit ist eingeschränkt. Beispiel hierfür ist ein spezieller Logigbaustein für eine Bankanwendung.

   Generische Softwarebausteine sind unabhängig von der Anwendungslogik und bieten eher technische Funktionen. Sie sind für verschiedene Applikationen verwendbar. Beispiele hierfür sind Bausteine für das Tracing, das Logging, die Namensauflösung oder allgemein nutzbare GUI-Basisfunktionen.

9. *Erläutern Sie das DAO-Pattern in Verbindung mit dem DTO-Pattern.*

   DAOs (Data Access Objects) sind Patterns, mit denen man den Zugriff auf Datenbanken realisieren kann. Der Zugriff auf Datenbanken erfolgt hier ausschließlich über DAO-Instanzen. Insbesondere bei mehrschichtigen Architekturen ist es sinnvoll, die DAO-Objekte nicht direkt an die Präsentationsschicht zu übertragen, sondern dafür spezielle DTO-Objekte (Data Transfer Objects) zu nutzen. DTOs sind eigenständige Objektklassen, die von der Datenbankstruktur entkoppelt sind, mit einfachen Methoden zur Bearbeitung. Durch Einsatz des DAO und des DTO-Patterns kann man eine Entkopplung der Datenbank von der Business-Logik und eine Entkopplung der Präsentationslogik von der Business-Logik erreichen. Allerdings ist dies mit Overhead verbunden.

10. *Wozu benötigt man das Business-Delegate-Pattern?*
    Das Business-Delegate-Pattern dient der Entkopplung von Geschäftslogik-Bausteinen von der technischen Implementierung. Damit kann der Lookup-Code und auch das Exception-Handling und weitere technische Details vor dem nutzenden Clientbaustein verborgen werden. Die technische Serverzugriffsschnittstelle wird also gekapselt.

11. *Diskutieren Sie die Auswirkungen einer ORM-Persistenzschicht auf die Leistungsfähigkeit des Datenzugriffs.*

    Die zusätzliche Schicht bedeutet zunächst Overhead und damit eine Leistungseinbuße. Eingriffe aus Optimierungsgründen sind schwieriger. Allerdings wird in der ORM-Schicht eine gezielte Optimierung der Datenbankzugriffe ermöglicht werden. Die Optimierung muss allerdings der Hersteller des ORM-Produkts entwickeln. Es kommt also auf das ORM-Produkt an, wie die konkreten Auswirkungen sind.

12. *Welche Aufgabe hat der Entity-Manager in der Java Persistence API?*

    Der Entity-Manager stellt eine zentrale Instanz für Persistenzoperationen dar. Hier wird ein Object-Relational-Mapping durchgeführt. Die *Entity-Manager*-Schnittstelle stellt die Dienste bereit, um Entities zu verwalten (*create, find,…*) und Queries oder sonstige Operationen auszuführen. Zudem werden Optimierungen für den Zugriff auf die Datenbank, wie etwas Caching, vorgenommen.

13. *Was ist ein Persistenzkontext im Sinne der Java Persistence API?*

    Der Persistenzkontext verwaltet alle Entities, die automatisch in die Datenbank geschrieben werden müssen. Diese Entities werden als managed Entities bezeichnet. Das Schreiben auf die Datenbank hängt von der verwendeten Strategie ab. Wenn beispielsweise in einer Transaktion mehrere Entities verändert und wieder abgespeichert werden, besteht der Persistenzkontext während dieser Transaktion aus den bearbeiteten Entities.

Literaturhinweise

[Abts 2003] *Abts, D.:* Client/Server-Programmierung mit Java, Vieweg Verlag, 2003

[Andresen 2003] *Andresen, A.:* Komponentenbasierte Softwareenwicklung mit MDA, UML und XML, Hanser Verlag, 2003

[Bauke 2006] *Bauke, H.; Mertens, S.:* Cluster Computing, Springer Verlag, 2006

[Backschat 2007] *Backschat, M.; Rücker, B*.: Enterprise JavaBans 3.0, 2. Auflage, Spektrum Akademischer Verlag, 2007

[Bauer 2000] *Bauer, N.; Hauptmann, J.; Peter* Mandl, *Weise, T.:* CORBA; Schaltzentrale; Verteilte Transaktionen auf Basis von CORBA OTS; iX 1/2000, S. 116

[Beeger 2006] *Beeger, R.; Haase, A.; Roock, S.; Sanitz, S.:* Hibernate - Persistenz in Java-Systemen mit Hibernate, 3, dpunkt.verlag, 2006

[Bengel 2004] *Bengel, G.:* Verteilte Systeme, 4. Auflage, Vieweg-Verlag, 2004

[Bell 1992] Bell, David; Grimson, Jane: Distributed Database Systems, Addison Wesley, 1992

[Bernstein 1987] *Bernstein, P. A.; Hadzilacos, V.; Goodman, N.:* Concurrency Control and Recovery, Addison-Wesley, 1987

[Birrel 1993] *Birrell, A.; Evers, D.; Nelson, G.; Owicki, S., Wobber, E.:* Distributed Garbage Collection for Network Objects, Digital Systems Research Center, Palo Alto, 1993

[Bloomer 1993] *Bloomer, J.:* Power Programming with RPC, O'Reilly&Associates Inc., 1993

[Burke 2006] *Burke, B.; Monson-Haefel, R.:* Enterprise JavaBeans 3.0, 5. Ausgabe, O'Reilly, 2006

[Comer 2002] *Comer, D.:* Computernetzwerke und Internets, 3. Auflage, Pearson Studium, 2002

[Coulouris 2002] *Coulouris, G.;* Dollimore, *J.; Kindberg, T.:* Verteilte Systeme Konzepte und Design, Pearson Studium, 2002

[Dadam 1996] *Dadam, P.:* Verteilte Datenbanken und Client-/Server-Systeme, Springer Verlag, 1996

[DTP 1991] *X/Open:* Distributed Transaction Processing: The XA Specification, CAE Specification, X/Open Company Ltd., U.K., 1991

[DTP 1992] *X/Open:* Distributed Transaction Processing: The TX (Transaction Demarcaton) Specification, Preliminary Specification, X/Open Company Ltd., U.K., 1992

[DTP 1993a] *X/Open:* Distributed Transaction Processing: Reference Model Version 2, X/Open Company Ltd., U.K., 1993

[DTP 1993b] *X/Open:* Distributed Transaction Processing: The TxRPC Specification, Preliminary Specification, X/Open Company Ltd., U.K., 1993

[DTP 1993c] *X/Open:* Distributed Transaction Processing: The Peer-to-Peer Specification, Snapshot, X/Open Company Ltd., U.K., 1993

[DTP 1993d] *X/Open:* Distributed Transaction Processing: The XATMI Specification, Preliminary Specification, X/Open Company Ltd., U.K., 1993

[DTP 1993e] *X/Open:* ACSE/Presentation: Transaction Processing API (XAP-TP), Preliminary Specification, X/Open Company Ltd., U.K., 1993

[DTP 1993f] *X/Open:* Distributed Transaction Processing: The XA+-Specification, X/Open Company Ltd., U.K., 1993

[Dustdar 2003] *Dustdar, S.; Gall, H.; Hauswirth M.:* Software-Architekturen für verteilte Systeme, Springer-Verlag, 2003

[Elmagarmid 1992] *Elmagarmid, A.*: Database Transaction Models for Advanced Applications, Morgan Kaufmann, 1992

[Gamma 1997] *Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.:* Design Patterns - Elements of Reusable Object Oriented Software, Addison-Wesley, 1997

[Gerdsen 1994a] *Gerdsen, P.; Kröger, P.:* Kommunikationssysteme 1 Theorie Entwurf Meßtechnik, Springer Verlag, 1994

[Gerdsen 1994b] *Gerdsen, P.; Kröger, P.:* Kommunikationssysteme 2 Anleitung zum praktischen Entwurf (SDL), Springer Verlag, 1994

[Gray 1993] *Gray, J.; Reuter, A.:* Transaction Processing, Concepts and Techniques, Morgan Kaufmann Publisher, 1993

[Gray 1995] *Gray, J.: Edwards, J.:* Scale Up with TP Monitors, BYTE, April, S. 123-128, 1995

[Greulich 1997]: *Greulich, M.:* Diskussion und Implementierung des von der OMG definierten Transaktionsservices auf Basis des Transaktionsmonitors UTM, TU München, 1997

[Haase 2001] *Haase, O.:* Kommunikation in verteilten Anwendungen, Einführung in Sockets, Java RMI, CORBA und Jini, Oldenbourg Verlag, 2001

[ISO 1984] *ISO/IEC 7498:* Open Systems Interconnection - Basic Reference Model, 1984

[ISO 1991] *ISO/IEC 9804:* Information Technology - Open Systems Interconnection - Service Definition for the Commitment, Concurrency and Recovery Service Element, 1990

[ISO 1992a] *ISO/IEC 10026-1:* Information Technology - Open Systems Interconnection - Distributed Transaction Processing - Part 2: OSI TP Model, 1992

[ISO 1992b] *ISO/IEC 10026-2*: Information Technology - Open Systems Interconnection - Distributed Transaction Processing - Part 2: OSI TP Service, 1992

[ISO 1992c] *ISO/IEC: 10165-4:* Information Technology - Open Systems Interconnection - Structure of Management Information: Guidelines for the Definition of Managed Objects (GDMO); ISO/IEC, 1992

[Jablonski 1997] *Jablonski, S.; Böhm, M.: Schulze, W.:* Workflow Management, Entwicklung von Anwendungen und Systemen, dpunkt.verlag, 1997

[Jeckle 2004] Jeckle, M.; Rupp, C.; Hahn, J.; Zengler, B., Queins, S.: UML2 glasklar, Hanser Verlag, 2004

bourg Verlag, München Wien, 1996 [Keller 2007] *Keller, W.*: IT-Unternehmensarchitektur, dpunkt.verlag, 2007

[Kemper 1996] *Kemper, A.; Eickler, A.:* Datenbanksysteme, Eine Einführung, Olden [Kerner 1993] *Kerner, H.:* Rechnernetze nach OSI, 2. Auflage, Addison-Wesley, 1993

[Koning 2002] *Koning, H.; Dormann, C.; can Vliet, H.*: Practical Guidelines for the Readability of IT-Architecture Diagrams, SIGDOC'02, October 20-23, 2002

[Krafzig 2005] *Krafzig, D.; Banke, K.; Slama, D.:* Enterprise SOA Service-Oriented Architecture Best-Practices, Prentice Hall, 2005

[Kreuz 1996] *Kreuz, D.; Lange, T.:* Offene Objektorientierte Transaktionsverarbeitung, TU Hamburg-Harburg, 1996

[Kreuz 1997] *Kreuz, D., Vogt, F.*: Transaktionales OLE = Viper?, Technische Universität Hamburg-Harburg, Arbeitsbereich Telematik, 1997

[Kruchten 1995] Kruchten, P.: Architectural Blueprints – The „4+1" View Model of Software Architecture, IEEE Software 12(6), S. 42-50

[Kuhrman 2004] *Kuhrmann, M. u.a.:* Verteilte Systeme mit .NET Remoting, Spektrum Akademischer Verlag, 2004

[Kurose 2002] *Kurose, J. F.; Ross, K. W.:* Computernetze, Pearson Studium, 2002

[Laureys 1995] *Laureys, M., Tricaud, E.:* ACTranS Presentation Document, Open Group, 1995

[Lockermann 1993] *Lockemann, P., C., Krüger, G.; Krumm, H.:* Telekommunikation und Datenhaltung, Carl Hanser Verlag, 1993

[McGoveran 1993] *McGoveran, D.; Date, C., J.*: A Guide To SYBASE and SQL Server, Addison-Wesley, 1993

[Mehnert 2006] *Mehnert, H.*: Architektur-basierte Strategien zur Selbst-Optimierung, TU Berlin, 2006

[Meyer 1988] *Meyer-Wegener, K.:* Transaktionssysteme, Teubner Verlag, Stuttgart, 1988

[Monson 1999] *Monson-Haefel, R.:* Enterprise JavaBeans, O'Reilly, 1999

[Mahlmann 2007] *Mahlmann, P.; Schindelhauer, Ch.*: Peer-to-Peer-Netzwerke, Springer-Verlag, 2007

[Mandl 2008a] *Mandl, P.:* Grundkurs Betriebssysteme, Vieweg-Teubner Verlag, 2008

[Mandl 2008b] *Mandl, P., Bakomenko, A.; Weiß, J.*: Grundkurs Datenkommunikation, Vieweg-Teubner Verlag, 2008

[Mandl 1993a] *Mandl, P.:* Entwicklung von Kommunikationsanwendungen mit ONC XDR, Teil 1, iX 4/1993, S. 208-213, Heise Verlag, 1993

[Mandl 1993b] Mandl, P.: Entwicklung von Kommunikationsanwendungen mit ONC XDR, Teil 2, iX 5, S. 193-201, Heise Verlag, 1993

[Niemann 2000] *Niemann, H:* EKKIS4-Transaktionskonzepte, 2000

[OASIS 2002] *OASIS*: Business Transaction Protocol, 2002

[OASIS 2005] *OASIS*: Web Services Coordination Framework Specification (WS-CF), Editor's Draft 1.0, 2005

[OASIS 2007a] *OASIS*: Web Services Coordination (WS-Coordination) Version 1.1, 2007

[OASIS 2007b] *OASIS*: Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.1, 2007

[OASIS 2007c] *OASIS*: Web Services Business Activity (WS-Business Activity,1.1, 2007

[OMG 1999] *Object Management Group:* CORBA Components - Volume I, 1999

[OMG 1995a] *Object Management Group:* Object Management Architecture Guide, OMG TC Document 92.11.1, John Wiley & Sons, 1995

[OMG 1995b] *Object Management Group*: CORBAservices: Common Object Services Specification, OMG TC Document 95-3-31, 1995

[OMG 2001] *Object Management Group:* CORBAservices: Transaction Service Specification; V1.2, 2001

[OMG 2002a] *Object Management Group*: The Portable Object Adapter, Common Object Request Broker Architecture (CORBA), v.3.0, July 2002,

[OMG 2002b] *Object Management Group*: CORBA Components, June 2002, Version 3.0

[OMG 2004] *Object Management Group*: Common Object Request Broker Architecture: Core Specification, v.3.0, March 2004

[Parnas 1984] *David L. Parnas, Paul C.*: The Modular Structure of Complex Systems. In Proc. Software Engineering, pages 408–419, 1984

[Ramamritham 1997] *Ramamritham, K.; Chrysanthis, P. K.:* Advances in Concurrency Control and Transaction Processing, IEEE Computer Society Press, 1997

[Riggert 2001] *Riggert, W.:* Rechnernetze, München, Wien 2001

[Roth 2002] *Roth, J.:* Mobile Computing, Grundlagen, Technik, Konzepte, dpunkt.verlag, 2002

[Schill 2007] *Schill, A.; Springer, T.*: Verteilte Systeme, Springer-Verlag, 2007

[Schiller 2000] *Schiller, J.:* Mobilkommunikation, Techniken für das allgegenwärtige Internet, Addison-Wesley, 2000

[Schwenkert 2004] *Schwenkert, R.:* Skriptum zur Vorlesung Datenbanksysteme, Fachhochschule München, FB07, 2004

[Schwichtenberg 2007] *Schwichtenberg, H.:* Microsoft .net 3.0 Crashkurs, Microsoft Press 2007

[Seidenfaden 2003] *Seidenfaden, L.; Gehrke, N.; Schuhmann, M.*: Peer-to-Peer Grid Computing – eine prototypische Realisierung, HMD 234, 98-107, dpunkt.verlag, 2003

[Shaw 1996] *Shaw, M.; Garlan, D.*: Software Architecture - Perspectives on an Emerging Discipline, Prentice Hall, 1996

[Siedersleben 2004] *Siedersleben, J.:* Moderne Softwarearchitektur, dpunkt.verlag, 2004

[Stevens 2000] *Stevens, P.; Pooley R.:* UML Softwareentwicklung mit Objekten und Komponenten, Pearson Studium, 2000

[Stevens 2000b] *Stevens, R.W.:* Programmieren von UNIX-Netzen, Hanser Verlag, 2000

[Starke 2005] *Starke, G.:* Effektive Softwarearchitekturen, 2. Auflage, Hanser Verlag, 2005.

[Sun 2004a] *Sun Microsystems:* Java 2 SDK, Standard Edition Documentation, Version 1.4.2

[Sun 2004b] *Sun* Microsystems: Java Remote Method Invocation Specification, Revision 1.10, Java 2 SDK, Standard Edition, v1.5.0

[Sun 1999a] *Sun Microsystems:* Java Transaction API (JTA), Version 1.01, 1999

[Sun 1999b] *Sun Microsystems:* JTS Transction Service Specification Version 1.0, Release 8.12.1999

[Sun 2002] *Sun Microsystems,* Java™ BluePrints, java.sun.com/blueprints/about/index.html, Accessed March 2002

[Sun 2003] *Sun Microsystems:* Enterprise JavaBeans Specification Version 2.1, 12/2003

[Sun 2005a] *Sun Microsystems:* Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements, Juni 2005

[Sun 2005b] *Sun Microsystems:* Enterprise JavaBeans, Version 3.0. Java Persistency API, Juni 2005

[Sun 2005c] *Sun Microsystems:* Enterprise JavaBeans, Version 3.0. EJB 3.0 Simplified API, Juni 2005

[Sun 2006a] *Sun Microsystems:* JDBC 4.0. Specification, JSR 221, November 2006

[Sun 2007a] Sun Microsystems: Enterprise JavaBeans,:Version vom 03.01.2007.

[Sun 2007b] Sun Microsystems: Java Message Service, Version vom 27.01.2007

[SuSe 2001] *SuSE:* SuSE Linux 7.2 Netwerk, suSE GmbH, 2001

[Szyperski 1999] *Szyperski, C.:* Component Software, Beyond Object-Oriented Programming, Addison-Wesley, 1998

[Tanenbaum 2001] *Tanenbaum, A. S.:* Computernetze, 3. revidierte Auflage, Prentice Hall, 2001

[Tanenbaum 2003] *Tanenbaum, A.S., van Stehen, M.*: Verteilte Systeme Grundlagen und Paradigmen, Pearson Studium, 2003

[Tanenbaum 2007] *Tanenbaum, A.S., van Stehen, M.*: Distributed Systems Principles and Pradigms, Pearson Prentice Hall, 2007

[Vasters 2002] *Vasters, C.:* .NET Enterprise Services, Hanser Verlag, 2002

[Vogt 1996] *Vogt, F., H.:* Werkzeuge für Transaktionsverarbeitung, heute - morgen: das ACTranS-Projekt, TU Hamburg-Harburg, 1996

[Vossen 1992] *Vossen, G., Groß-Hardt, M.:* Grundlagen der Transaktionsverarbeitung, Addisson-Wesley, 1993

[Wächter 1993a] *Wächter, A.:* X/Open-Standard für verteilte Transaktionsverarbeitung, in Fokus 07/08, IT Verlag, S. 65-69, 1993

[Wächter 1993b] *Wächter, A.:* Client/Server-Anwendungen auf Basis des X/Open-DTP-Standards X/Open-Standard, Fokus 11/12, IT Verlag, S. 75-80, 1993

[Weber 1998] *Weber, M.:* Verteilte Systeme, Spektum Akademischer Verlag, 1998

[Weikum 1989] *Weikum, G.:* Geschachtelte Transaktionen, in Informatik-Spektrum 12, S. 102-106, 1989

[Wolff 2006] *Wolff, E.:* Spring Framework für die Java-Entwicklung, dpunkt.verlag, 2006

[Wöhr 2004] *Wöhr, H.:* Web-Technologien, Konzepte - Programmiermodelle - Architekturen, dpunkt Verlag, 2004

[W3C 2003a] *W3C:* SOAP Version 1.2 Part 0: Primer, W2C Recommendation 24 June 2003

[W3C 2003b] *W3C:* SOAP Version 1.2 Part 1: Messaging Framework, W2C Recommendation 24 June 2003

[W3C 2004a] *W3C:* XML Schema Part 2: Datatypes Second Edition, W3C Recommendation Oktober 2004

[W3C 2004b] *W3C*: Web Services Architecture Usage Scenarios, W3C Working Group Note 11 February 2004

[Zimmer 2000] *Zimmermann, J.; Beneken, G.:* Verteilte Komponenten und Datenbankanbindung, Addison Wesley, 2000

[Zitterbart 1995] *Zitterbart, M.:* Hochleistungskommunikation Band 1: Technologie und Netze, Oldenbourg Verlag, 1995

[Zitterbart 1996] *Zitterbart, M.; Braun, T.:* Hochleistungskommunikation Band 2: Transportdienste und -protokolle, Oldenbourg Verlag, 1996

Interessante Web-Links

[WWW-001] http://www.opengroup.org/publications/catalog/: Das DTP-Modell (Distributed Transaction Processing) der Open Group

[WWW-002] http://www.omg.org/library/: Object Transaction Services (OTS) der OMG

[WWW-003] http://www.sun.java.com: Website von Sun mit Inhalten zu den Java-Technologien

[WWW-004] http://java.sun.com/j2ee/1.4/docs/index.html: J2EE-Dokumentation

[WWW-005] http://java.sun.com/j2se/index.jsp: J2SE-Dokumentation, Sun Microsystems

[WWW-006] http://msdn.microsoft.com/: Dokumentaton zu den Microsoft-Produkten

[WWW-007] http://www.apache.org/: Webseite der Apache Group. Informationen zu Open-Source-Produkten wie Apache Tomcat und Apache Webserver

[WWW-008] http://www.mpi-forum.org: Homepage des MPI-Forums.

[WWW-009] http://www.omg.org: Webseite der Object Management Group

[WWW-010] http://www.w3.org: Webseite des W3C

[WWW-011] http://www.ws-i.org: Webseite der Web Service Interoperability Organisation

[WWW-012] http://java.cs.uni-magdeburg.de/, letzter Zugriff am 20.09.2005

[WWW-013] http://uddi.microsoft.com

[WWW-014] http://uddi.ibm.com

[WWW-015] www.sourceforge.net:Hier werden einige Open-Source-Lösungen gehostet, u.a. das Spring-Framework

[WWW-016] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

[WWW-017] http://sourceforge.net/projects/aopalliance

[WWW-018] http://www.martinfowler.com/articles/injection.html

[WWW-019] http://web1.jcp.org/: Homepage des Java Community Process (JCP), in dem die Standardisierung der Java-APIs organisiert wird

[WWW-020] http://www.eclipse.org: Homepage von Eclipse

[WWW-021] http://www.opengroup.org/bookstore/catalog: Open Group: Technical Standard Distributed Transaction Processing: The XA Specification, CAE Specification, 1992

[WWW-022] http://www.opengroup.org/bookstore/catalogOpen Group: Distributed Transaction Processing: The TX (Transaction Demarcaton), 1995

[WWW-023] http://www.opengroup.org/bookstore/catalog/g504: Open Group: Technical Standard Distributed Transaction Processing: Reference Model Version 3, 1996

[WWW-024] http:// www.opengroup.org/bookstore/catalog/p305: Open Group: Technical Standard Distributed Transaction Processing: The TxRPC Specification, 1995

[WWW-025] http:// www.opengroup.org/bookstore/catalog/c409: Open Group: Technical Standard Distributed Transaction Processing: The Peer-to-Peer Specification, Snapshot, 1993

[WWW-026] http:// www.opengroup.org/bookstore/catalog/p306: Open Group: Distributed Transaction Processing: The XATMI Specification, 1995

[WWW-027] http:// www.opengroup.org/bookstore/catalog/c409: Open Group: Technical Standard ACSE/Presentation: Transaction Processing API (XAP-TP), 1994

[WWW-028] http:// www.opengroup.org/bookstore/catalog/s201: Open Group: Distributed Transaction Processing: The XA+-Specification, Version 2, 1994

[WWW-029] http://www.p-grid.org: The P-Grid-Project

[WWW-030] http://www.napster.de: Napster

[WWW-031] http://www.gnutella.com: Gnutella

[WWW-032] http://www.freenet.de: Freenet

[WWW-033] http://www.blackberry.com: BlackBerry

[WWW-034] http://www.d-grid.de: D-Grid-Initiative,

[WWW-035] www.springframework.org, Spring Framework

[WWW-036] www.oasis.org: OASIS-Webseite

[WWW-037] https://jira.amqp.org/: AMQP Working Group

[WWW-038] http://de.wikipedia.org/wiki: Wikipedia

[WWW-039] http://openccm.objectweb.org: OpenCCM-Webseite

Sachwortverzeichnis