LEHRBUCH

Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Anwendungen entwickeln mit Standard-Technologien

4. Auflage

Springer Vieweg

Masterkurs Client/Server-Programmierung mit Java

Dietmar Abts

Masterkurs Client/Server-Programmierung mit Java

Anwendungen entwickeln mit Standard-Technologien

4. Auflage

Springer Vieweg

Dietmar Abts
FB Wirtschaftswissenschaften
Hochschule Niederrhein
Mönchengladbach, Deutschland

ISBN 978-3-658-09920-6 ISBN 978-3-658-09921-3 (eBook)
DOI 10.1007/978-3-658-09921-3

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

Springer Vieweg
1. Auflage 2003 erschien unter dem Titel “Aufbaukurs JAVA”
2. Auflage 2007 erschien unter dem Titel “Masterkurs JAVA”

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Fachmedien Wiesbaden ist Teil der Fachverlagsgruppe Springer Science+Business Media
(www.springer.com)

Vorwort zur vierten Auflage

Das vorliegende Buch bietet eine kompakte Einführung in aktuelle Technologien zur Entwicklung von modernen Client/Server-Anwendungen auf der Basis von Java SE mit zahlreichen Programmbeispielen und Übungsaufgaben. Die verschiedenen Themen können mit Grundkenntnissen der Programmiersprache Java erarbeitet werden.

Gegenüber der dritten Auflage wurden zahlreiche Verbesserungen und Aktualisierungen vorgenommen. Das Kapitel zu JDBC der dritten Auflage wurde zugunsten neuer Themen entfernt:

- WebSocket-Protokoll für die bidirektionale Verbindung zwischen einer Webanwendung und einem WebSocket-Server,
- REST-basierte Web Services mit JAX-RS.

Die Programmbeispiele wurden mit Java SE 8 entwickelt und sind für die aktuellen Versionen der hier eingesetzten Produkte und Frameworks lauffähig:

- Apache ActiveMQ
- Apache Tomcat
- WebSocket-Implementierung Tyrus
- Apache XML-RPC
- Metro Web Service Stack
- JAX-RS-Implementierung Jersey

Der Quellcode der Programmbeispiele und die Lösungen der Übungsaufgaben sind im Internet auf der Website des Verlags direkt neben den bibliographischen Angaben zu diesem Buch verfügbar:

`www.springer-vieweg.de`

Danken möchte ich Frau Sybille Thelen vom Lektorat IT für die gute Zusammenarbeit sowie meinen Leserinnen und Lesern für die guten Vorschläge, die in dieser Auflage berücksichtigt wurden.

Ratingen, April 2015

Dietmar Abts
`abts@hs-niederrhein.de`
`www.dietmar-abts.de`

Inhaltsverzeichnis

Nachrichtenbasierte Kommunikation

Objekt-basierte Kommunikation

Web-basierte Kommunikation

1 Einleitung

1.1 Vorbemerkungen

Zielsetzung

Das vorliegende Buch beschäftigt sich mit der Implementierung von *Client/Server-Anwendungen* auf Basis der Internet-Protokolle und darauf aufsetzenden Standard-Technologien. Die dazu gehörenden Spezifikationen sind zum großen Teil plattform- und programmiersprachenunabhängig (wie z. B. HTTP, WebSocket, SOAP- und REST-basierte Web Services). Wir nutzen für die Implementierung der Beispiele durchweg Java und bekannte Referenzimplementierungen der oben angesprochenen Spezifikationen.

Angesichts des Umfangs der Themen musste eine Auswahl getroffen werden, die auch in einem vier Semesterwochenstunden umfassenden Kurs behandelt werden kann.

Ziel des Buches ist es, über die Themenvielfalt angemessen zu informieren und in die Einzelthemen systematisch mit der für die Anwendungsentwicklung nötigen Tiefe einzuführen. Besonderer Wert wurde dabei auf praxisnahe Programmbeispiele und Übungsaufgaben gelegt.

Aufbau des Buches

Dieses Buch ist in zehn Kapitel gegliedert, die jeweils einen Schwerpunkt behandeln. Obwohl die Kapitel weitestgehend in sich geschlossen sind und unabhängig voneinander bearbeitet werden können, wird empfohlen, diese in der hier vorgegebenen Reihenfolge durchzuarbeiten. Die vorgestellten Themen werden anhand vollständiger, lauffähiger Programmbeispiele verdeutlicht.

Übungsaufgaben am Ende eines jeden Kapitels regen zur selbständigen Beschäftigung mit dem dargebotenen Stoff an.

Wir haben die verschiedenen Themen anhand der verwendeten Kommunikationsverfahren klassifiziert und in drei Klassen eingeteilt:

- Nachrichten-basierte Kommunikation
- Objekt-basierte Kommunikation
- Web-basierte Kommunikation

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_1, © Springer Fachmedien Wiesbaden 2015

Zu den *Nachrichten-basierten Verfahren* gehören die synchrone Nachrichtenübertragung über UDP- und TCP-Sockets sowie Nachrichtensysteme zur Realisierung eines asynchronen Nachrichtenaustausches mit Zwischenspeicherung (Message Oriented Middleware).

Zum *Objekt-basierten Ansatz* gehört RMI (Remote Method Invocation). RMI ermöglicht die Kommunikation zwischen Java-Objekten in verschiedenen JVM-Instanzen (JVM = Java Virtual Machine).

In die Klasse der *Web-basierten Verfahren* haben wir die Verfahren aufgenommen, die auf der Basis des Transportprotokolls HTTP und damit verwandter Techniken arbeiten. Diese sind alle plattform- und programmiersprachenunabhängig.

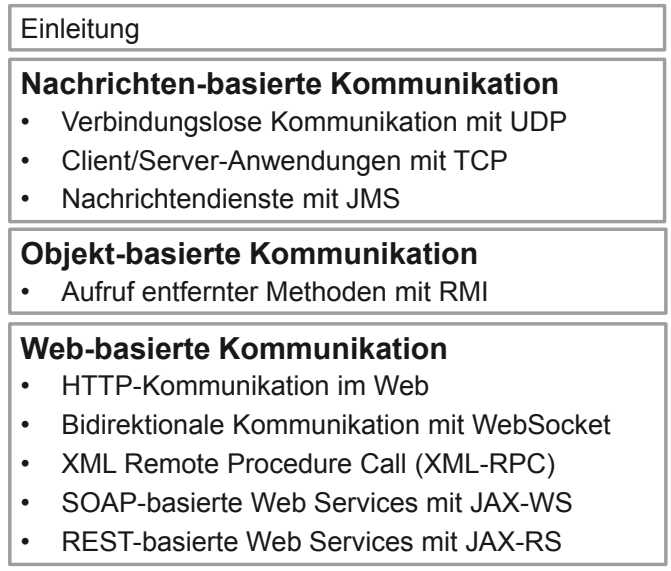

Abbildung 1-1: Einordnung der Themen

Erwartungen an den Leser

Von den Leserinnen und Lesern werden erwartet:

- Grundlegende Java-Kenntnisse, insbesondere: Konzepte der objektorientierten Programmierung, Dateiverarbeitung und Thread-Programmierung,
- das Wissen um Internet und World Wide Web und der Umgang mit einem Webbrowser,
- Grundlagenkenntnisse in HTML und XML.

Buchtipps hierzu finden Sie am Ende des Buches im Abschnitt "Literaturhinweise".

Die Beispiele

Alle Beispielprogramme und Lösungen zu den Übungsaufgaben wurden mit Hilfe von Java SE 8 unter Windows 7 entwickelt und im Netz getestet. Selbstverständlich sind die hier vorgestellten Programme ohne Änderung auch auf UNIX/Linux-Systemen lauffähig.

Als Entwicklungsumgebung wurde *Eclipse* eingesetzt. Selbstverständlich kann auch jede andere Java-Entwicklungsumgebung genutzt werden.

Zur Vereinfachung der Programmentwicklung wurde an einigen Stellen das Build-Werkzeug *Ant* der Apache Software Foundation genutzt (siehe Quellenverzeichnis am Ende des Buches). Ant ist in Eclipse integriert, kann aber auch stand-alone genutzt werden. Das Begleitmaterial zu diesem Buch enthält die benötigten Ant-Skripte.

Einzelheiten zu den verwendeten Tools und Bibliotheken können den entsprechenden Kapiteln entnommen werden. Das Quellenverzeichnis am Ende des Buches enthält die Bezugsquellen.

Download

Sämtliche Programme und Lösungen stehen im Internet zur Verfügung und können heruntergeladen werden.

1.2 Verteilte Systeme

Umfassende gesellschaftliche und wirtschaftliche Entwicklungen zwingen Unternehmen, sich flexibel an Veränderungen anzupassen und schnell auf neue Marktsituationen zu reagieren. Betrieblichen Anwendungssystemen kommt hier eine besondere Bedeutung zu. Wie sind die Anwendungssysteme zu implementieren, um höchstmögliche Flexibilität, Verlässlichkeit und Anpassbarkeit zu erreichen?

Monolithische Systeme (alle Anwendungen laufen auf *einem* Rechner) sind nur noch bedingt geeignet. Heute ist die Strukturierung der Software in *Client* und *Server* weit verbreitet.

Die folgenden Faktoren haben die fundamentalen Änderungen wesentlich beeinflusst:

- Leistungsexplosion in der Mikroprozessortechnik,
- schnelle Datennetze,
- Fortschritte in der Softwaretechnik,
- Abkehr von hierarchischen Organisationsstrukturen,
- Bedürfnis nach Integration bisher nicht integrierter Systeme,
- Internet-Technologie.

Waren die 1960er Jahre noch durch *Batch-Processing-Systeme*, Lochstreifen und Lochkarten geprägt, konnten Anfang der 1970er Jahre in so genannten *Timesharing-Systemen* Transaktionen im Dialog gestartet werden. Minicomputer (UNIX-Systeme) erschienen als zweiter Gerätetyp neben dem Host. Anfang der 1980er Jahre kamen die *Personal Computer* als Stand-alone-Systeme oder vernetzt im LAN hinzu. Der wichtigste Trend ab 1990 ist durch das Aufkommen von *Client/Server-Computing* für die verteilte Verarbeitung bestimmt. Hierzu zählen insbesondere die heutigen Internet-Anwendungen (Web-Informationssysteme, E-Business-Anwendungen, Cloud Computing).

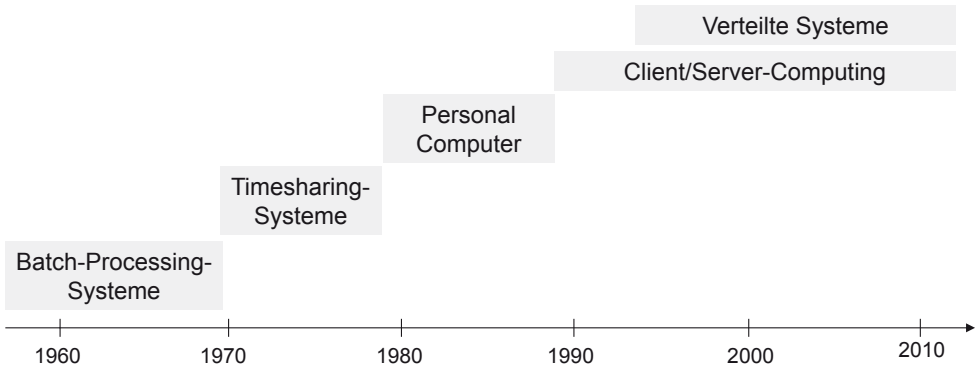

Abbildung 1-2: Entwicklungsstufen

Jede Anwendung kann grob in drei Schichten eingeteilt werden:

1. *Präsentation*
   Diese Schicht (auch Benutzungsschnittstelle genannt) hat die Aufgabe, den Dialog mit dem Benutzer zu steuern sowie Daten und Befehle an das System zu übermitteln.
2. *Verarbeitung*
   Diese Schicht enthält die Verarbeitungslogik. Hier wird der eigentliche Nutzen der Anwendung erzielt.
3. *Datenhaltung*
   Diese Schicht hat die Aufgabe, die Daten abzuspeichern und bei Bedarf wieder zur Verfügung zu stellen.

Abbildung 1-3 zeigt eine Webanwendung, die dem Benutzer das Produktangebot eines Unternehmens präsentiert. Die drei Schichten Präsentation, Verarbeitung und Datenhaltung sind jeweils auf eigenen Rechnern implementiert. Der *Webbrowser* ist die Benutzungsoberfläche, die die abgefragten Informationen grafisch präsentiert. Der *Webserver* erzeugt Datenbankabfragen, bereitet die Ab-

frageergebnisse als Webseite auf und übermittelt diese an den Client. Der *Datenbankserver* verwaltet den Produktkatalog und führt die Datenbankabfragen aus. Webbrowser, Webserver und Datenbankserver sind autonome Teilsysteme, die koordiniert miteinander kooperieren.

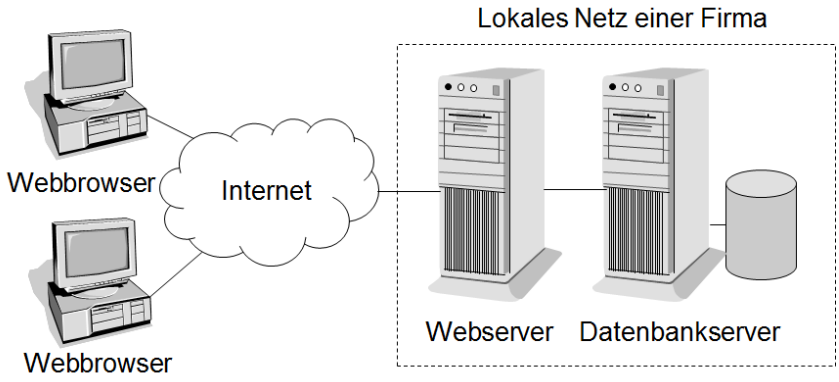

Abbildung 1-3: Beispiel "Webanwendung"

Verteiltes System

Verteilung bedeutet die Zuordnung von Funktionen und Daten auf mehrere Rechner. Ein Anwendungssystem wird dabei in funktionale Komponenten zerlegt, die dann bei der Installation im Netz so verteilt werden, dass die Komponenten optimal unterstützt werden. Abstrakt formuliert ist ein *verteiltes System* eine Menge von Komponenten, die kooperieren, um eine *gemeinsame* Aufgabe zu erfüllen.

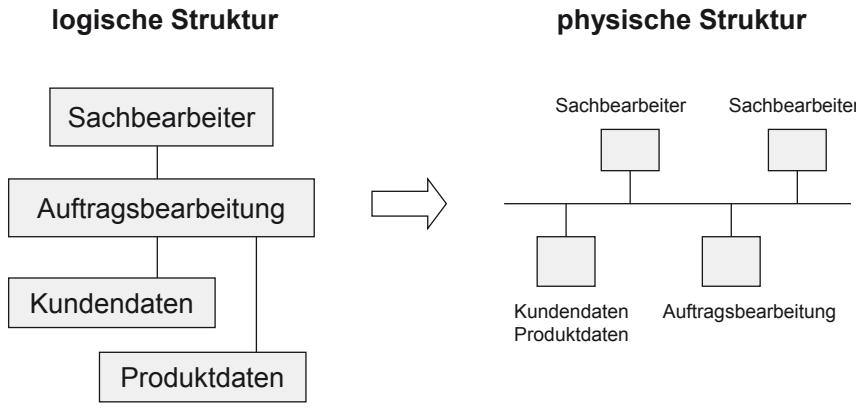

Abbildung 1-4: Abbildung der logischen auf die physische Struktur

Konfigurationsproblem

Da es im Allgemeinen mehrere Varianten für die Installation der Komponenten im Netz gibt, besteht das *Konfigurationsproblem*, die *logische* Struktur der Anwendung auf die *physische* Struktur (Rechner im Netz) so abzubilden, dass bei vorgegebenen Randbedingungen ein maximaler Nutzen erzielt wird (siehe Abbildung 1-4).

Ob eine Verteilung überhaupt möglich ist, entscheidet sich bereits beim Entwurf der Anwendungssoftware. Der Entwurf muss unabhängig von einer späteren physischen Konfiguration sein. Dem Benutzer muss die konkrete physische Aufteilung verborgen bleiben.

Der Vergleich der verteilten Verarbeitung mit der großrechnerdominierten, zentralen Verarbeitung ergibt folgende Vor- und Nachteile:

Vorteile

- *Besseres Abbild der Realität*
  Die Verteilung unterstützt die "schlanke" Organisation. Leistungen werden dort erbracht, wo sie benötigt werden. Daten werden dort erfasst, wo sie im Geschäftsprozess entstehen.
- *Wirtschaftlichkeit*
  Teure Ressourcen (Geräte, Softwarekomponenten) stehen mehreren Rechnern zur Verfügung und können gemeinsam genutzt werden. Automatisierte Aufgaben werden möglichst dort ausgeführt, wo sie am wirtschaftlichsten sind.
- *Bessere Lastverteilung*
  Da mehrere Rechner mit jeweils eigenem Betriebssystem arbeiten, kann durch teilweise parallele Verarbeitung die Gesamtbearbeitungszeit verkürzt werden.
- *Bessere Skalierbarkeit*
  Einzelne Komponenten können leichter an einen steigenden Bedarf angepasst werden.
- *Fehlertoleranz*
  Beim Ausfall eines Rechners bleiben die anderen betriebsbereit. Die Aufgaben des ausgefallenen Rechners können oft von einem anderen Rechner übernommen werden.

Nachteile

- *Höhere Komplexität durch Verteilung und Heterogenität*
  Die Verteilung der Komponenten großer Systeme und die Heterogenität bei Betriebssystemen, Programmiersprachen, Datenformaten und Kom-

munikationsprotokollen stellen hohe Anforderungen an die Entwickler, wenn das Gesamtsystem überschaubar bleiben soll.

- *Komplexe Netzinfrastrukturen*
  Netzarchitektur und Netzmanagement sind das Rückgrat der Systeme eines Unternehmens. Die Integration heterogener Systeme mit Komponenten unterschiedlicher Hersteller- und Standardarchitekturen stellt hohe Anforderungen an die Administration.
- *Höhere Sicherheitsrisiken*
  Verteilte Systeme bieten mehr Möglichkeiten für unberechtigte Zugriffe. Im Netz übertragene Nachrichten können abgehört oder sogar verändert werden. Der Einsatz von Verschlüsselungsverfahren und Firewall-Systemen sind wirksame Schutzmaßnahmen.

Aufgabe von so genannten *Verteilungsplattformen* ist es, diese Komplexität beherrschbar zu machen.

Das *Client/Server-Modell* ist das am weitesten verbreitete Modell für die verteilte Verarbeitung.

Client/Server

Client/Server bezeichnet die Beziehung, in der zwei Programme zueinander stehen. Der *Client* stellt eine Anfrage an den Server, eine gegebene Aufgabe zu erledigen. Der *Server* erledigt die Aufgabe und liefert das Ergebnis beim Client ab. Ein Kommunikationsdienst verbindet Client und Server miteinander.

Ein *Client/Server-System* besteht also aus

- einem oder mehreren Clients (Auftraggebern),
- einem Server (Auftragnehmer) und
- einem Kommunikationsdienst (Netzwerk, Vermittler).

Software-Sicht

Client und Server sind Programme, die auch auf demselben Rechner laufen können. Sie müssen also nicht notwendig über ein Netz verbunden sein. Das macht deutlich, dass es sich hierbei um eine Software- und keine Hardwarearchitektur handelt. In der Praxis werden natürlich Client und Server auf unterschiedlichen Rechnern verteilt, um die bekannten Vorteile eines verteilten Systems zu erzielen.

Hardware-Sicht

Gelegentlich werden auch die *Trägersysteme* (Rechner), auf denen die Programme laufen, als Client bzw. Server bezeichnet. Wir nutzen die Begriffe Client bzw.

Server als Homonyme. Aus dem Zusammenhang wird dann klar, welche Bedeutung gemeint ist.

Ein Server kann die Erledigung einer Aufgabe weiter delegieren und dazu auf andere Server zugreifen. Er spielt dann selbst die Rolle eines Client. Der Webserver in Abbildung 1-3 hat diese Doppelfunktion.

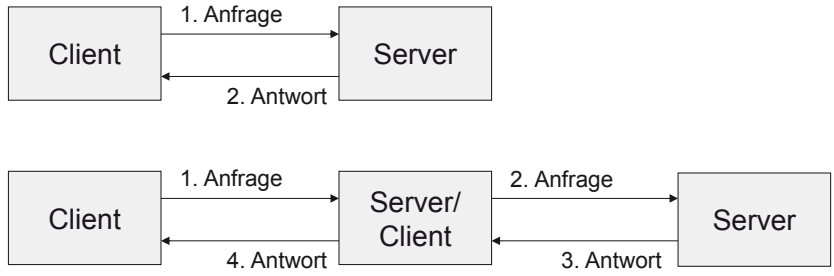

Abbildung 1-5: Client/Server-Modell

Im Folgenden sind charakteristische Merkmale von Client und Server zusammengefasst:

Client-Merkmale

- Ein Programm wird vorübergehend zum Client, wenn es einen Dienst von einem anderen Programm (Server) anfordert. Darüber hinaus kann das Programm andere Aufgaben lokal ausführen.
- Der Client leitet den Kontakt mit einem Server *aktiv* ein.
- Der Client kann im Laufe einer Sitzung auf mehrere Server zugreifen, kontaktiert aber aktiv nur jeweils einen Server.

Server-Merkmale

- Der Server ist ein Programm, das einen bestimmten Dienst bereitstellt. Er kann in der Regel gleichzeitig mehrere Clients bedienen.
- Häufig werden Server automatisch beim Hochfahren des Rechners gestartet und beim Herunterfahren beendet.
- Der Server wartet *passiv* auf die Verbindungsaufnahme durch einen Client.
- Da Server eigenständige Programme sind, können mehrere Server unabhängig voneinander auf einem einzigen Rechner als Trägersystem laufen.
- *Parallelbetrieb* ist ein grundlegendes Merkmal von Servern. Mehrere Clients können einen bestimmten Dienst in Anspruch nehmen, ohne warten zu

müssen, bis der Server mit der Erledigung der laufenden Anfrage fertig ist. In den späteren Java-Programmbeispielen werden die Client-Anfragen jeweils in einem neuen Thread bedient.

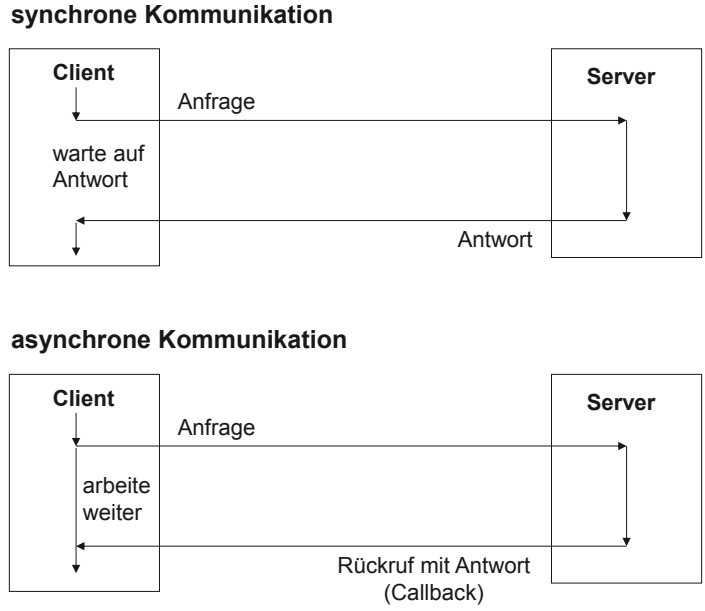

Abbildung 1-6: Interaktionsarten

synchron/asynchron

Die Interaktion zwischen Client und Server kann *synchron* oder *asynchron* erfolgen (siehe Abbildung 1-6).

Bei der *synchronen* Kommunikation wartet der Client nach Absenden der Anfrage an den Server so lange, bis er eine Rückantwort erhält, und kann dann erst andere Aktivitäten ausführen.

Im *asynchronen* Fall sendet der Client die Anfrage an den Server und arbeitet sofort weiter. Beim Eintreffen der Rückantwort werden dann bestimmte Ereignisbehandlungsroutinen beim Client aufgerufen. Die durch den Server initiierten Rückrufe (*Callbacks*) erfordern allerdings zusätzliche Kontrolle beim Client, wenn ungewünschte Unterbrechungen der Arbeit vermieden werden sollen.

Bei *einstufigen Architekturen* wird die gesamte Anwendung (Präsentation, Verarbeitung und Datenhaltung) auf *einem* Rechner implementiert. Die Benutzer-

interaktion erfolgt über Terminals, die direkt oder über einen Terminal- bzw. Kommunikationsserver an den Rechner angeschlossen sind.

Multi-Tier-Architekturen

Verteilte Anwendungen haben eine *mehrstufige Architektur* (*Multi-Tier-Architektur*). Beispielsweise werden Applikations- und Datenbankserver verschiedenen Ebenen zugeordnet.

Bei der *zweistufigen Architektur* ist die Gesamtanwendung zwei Ebenen zugeordnet: Client und Server.

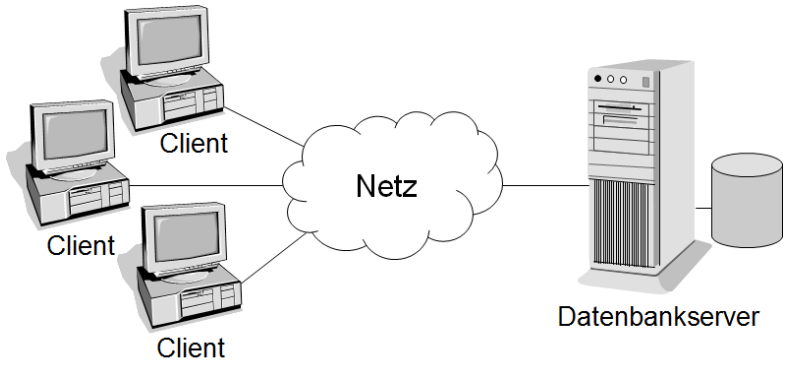

Abbildung 1-7: Eine zweistufige Architektur

Für die Arbeitsteilung zwischen Client und Server existieren verschiedene Alternativen, je nachdem, wo die Schichten *Präsentation*, *Verarbeitung* und *Datenhaltung* angesiedelt sind.

Abbildung 1-8 zeigt fünf Alternativen der Aufgabenverteilung bei der zweistufigen Architektur eines Client/Server-Systems. Von Alternative 1 bis 5 wird immer mehr Funktionalität auf den Client übertragen.

Thin und Rich Clients

Alternative 1 und 2 stellen die extreme Form eines *Thin Clients* dar. Der Client übernimmt nur die Aufgaben einer Benutzungsschnittstelle. Im Gegensatz dazu steht der *Rich Client*, der neben der Präsentation auch (teilweise) die Anwendungslogik und ggf. die lokale Datenhaltung übernimmt.

Einige Merkmale von Thin und Rich Clients:

- Ein Thin Client belastet den Server stärker als ein Rich Client.
- Die Netzbelastung ist bei Thin Clients stärker als bei Rich Clients.
- Rich Clients können meist auch offline arbeiten. Unter Umständen müssen dann aber Daten repliziert und synchronisiert werden.

- Die Versorgung mit neuen Software-Releases ist bei der Verwendung von Thin Clients einfacher, da hier im Allgemeinen nur zentrale Komponenten auf dem Server auf den neuesten Stand gebracht werden müssen.

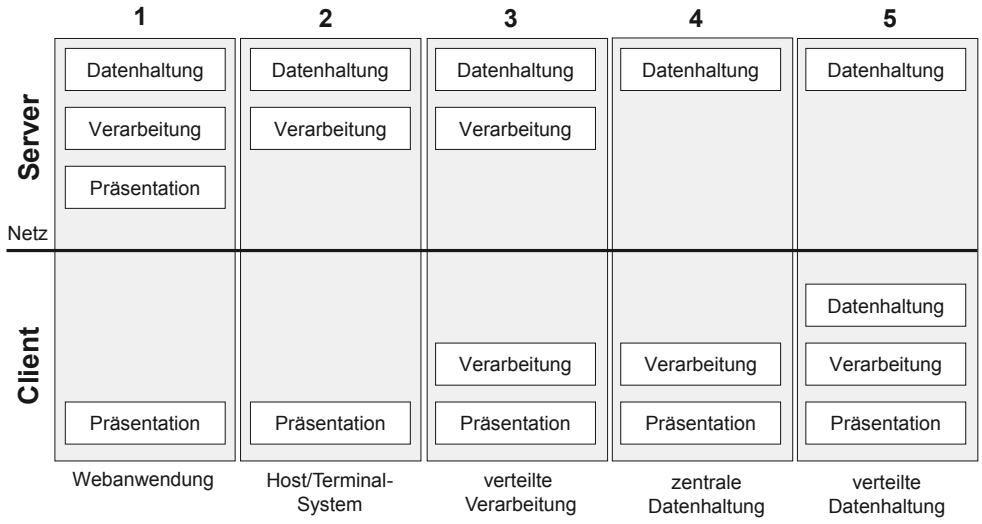

Abbildung 1-8: Alternativen der Aufgabenverteilung

Bei der *dreistufigen Architektur* wird die Gesamtanwendung auf drei Ebenen aufgeteilt: z. B. Client, Applikationsserver und Datenbankserver. Dieses Modell ist auch Basis vieler ERP-Systeme (Enterprise Resource Planning).

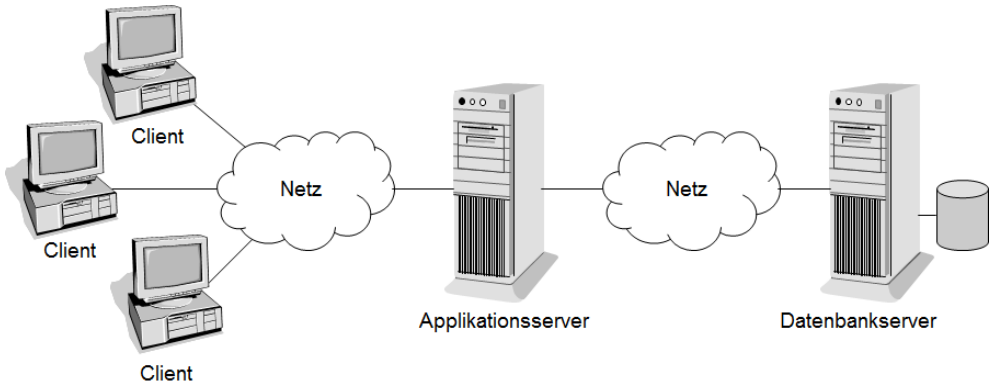

Abbildung 1-9: Eine dreistufige Architektur

Insbesondere bei Internet- und Intranet-Anwendungen findet man eine *vierstufige Architektur*, bei der ein Webserver dem Applikationsserver vorgeschaltet ist.

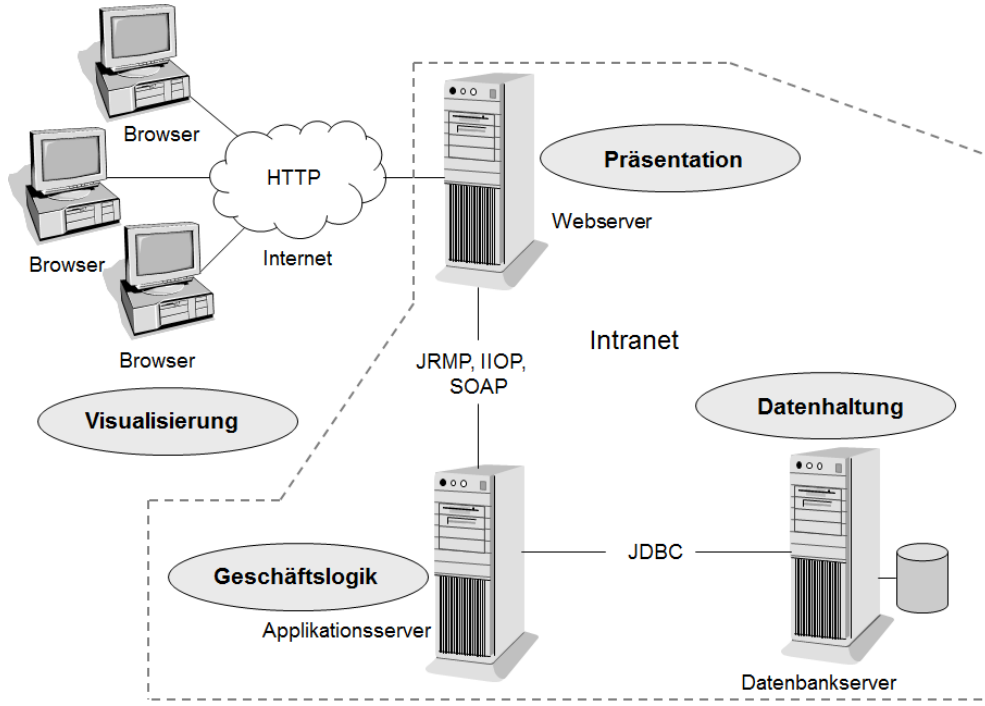

Abbildung 1-10: Eine vierstufige Architektur

Abbildung 1-10 zeigt eine klassische webbasierte Anwendung. Dabei wird die Präsentation in Form von HTML-Seiten auf dem Webserver erzeugt, die dann per HTTP an den Client ausgeliefert werden.

Dank leistungsfähiger Technologien wie HTML5, CSS3 und JavaScript-Frameworks setzen sich immer mehr so genannte *Single-Page-Webanwendungen* durch.

Eine solche Anwendung besteht in der Regel aus einer einzigen HTML-Seite, deren Dateninhalte asynchron nachgeladen werden.

Die HTML-Elemente werden dynamisch mit JavaScript im Browser verändert. Die Validierung von Formulareingaben und die Navigationssteuerung werden ebenfalls mit JavaScript im Browser ausgeführt. Es ist kein klassischer Seitenwechsel mehr nötig.

Die Daten werden meist im XML- oder JSON-Format vom Server übertragen. Dazu werden vorzugsweise REST-basierte Web Services eingesetzt.

In mehrstufigen Architekturen sind die Komponenten eines Anwendungssystems auf potentiell heterogene Trägersysteme in einem Netzwerk verteilt. Um die Komplexität des Gesamtsystems bei der Bedienung oder Entwicklung der Anwendung vor dem Benutzer bzw. Entwickler verborgen zu halten, bedarf es einer geeigneten Software-Infrastruktur, die die Interaktion zwischen den Komponenten in besonderer Weise unterstützt.

Middleware

Die verteilte Verarbeitung hat zu einem neuen Typ systemnaher Software geführt, der so genannten *Middleware*.

Middleware ist Software, die den Austausch von Informationen zwischen den verschiedenen Komponenten eines verteilten, heterogenen Systems unterstützt. Die Anwendungen werden dabei von den komplexen Details der internen Vorgänge abgeschirmt.

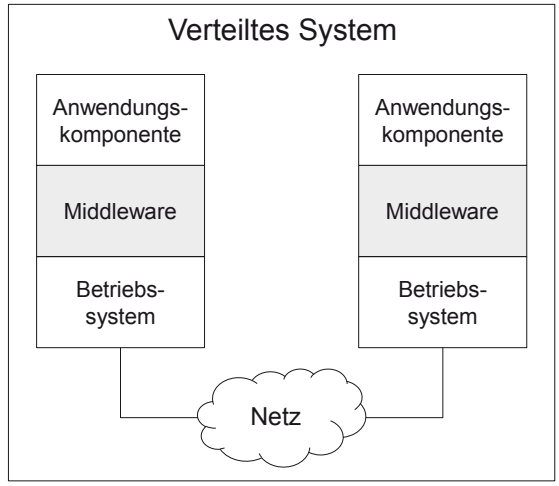

Abbildung 1-11: Middleware

In der Informatik nennt man eine Sache *transparent*, wenn sie "durchsichtig", also nicht sichtbar ist.

Verteilungstransparenz

In verteilten Systemen sollen interne Abläufe und Implementierungsdetails für den Benutzer oder Anwendungsentwickler nicht sichtbar sein. *Verteilungstransparenz* verbirgt die Komplexität verteilter Systeme.

Es existieren mehrere untergeordnete Transparenzbegriffe, die jeweils einen bestimmten Aspekt bezeichnen. Beispiele hierfür sind:

- *Ortstransparenz*
  Der Ort einer Ressource ist dem Benutzer nicht bekannt. Er identifiziert sie über einen Namen, der keine Information über ihren Aufenthaltsort enthält.
- *Zugriffstransparenz*
  Die Form des Zugriffs auf eine Ressource ist einheitlich und unabhängig davon, ob die Ressource lokal oder auf einem entfernten Rechner zur Verfügung steht. Unterschiede verschiedener Betriebssysteme und Dateisysteme werden verborgen.
- *Nebenläufigkeitstransparenz*
  Der gleichzeitige Zugriff mehrerer Benutzer auf dieselbe Ressource (z. B. Datenbanktabelle) erfolgt ohne gegenseitige Beeinflussung und lässt keine falschen Ergebnisse entstehen.
- *Replikationstransparenz*
  Sind aus Verfügbarkeits- oder Performance-Gründen mehrere Kopien einer Ressource (z. B. eines Datenbestandes) vorhanden, so merkt der Benutzer nicht, ob er auf das Original oder eine Kopie zugreift. Das System sorgt dafür, dass alle Kopien bei Änderungen konsistent bleiben.
- *Migrationstransparenz*
  Ressourcen können von einem Rechner auf einen anderen verlagert werden, ohne dass der Benutzer dies bemerkt.
- *Fehlertransparenz*
  Der Benutzer soll nicht mit allen auftretenden Fehlern im System konfrontiert werden. Das Auftreten von Fehlern und die Fehlerbehebung sollen vor dem Benutzer weitestgehend verborgen sein.

Middleware-Produkte stellen dem Entwickler die für die Anwendung benötigten Funktionen meist in Form eines *API* (Application Programming Interface) zur Verfügung.

Datenbank-Middleware ermöglicht den Zugriff auf unterschiedliche Datenbanksysteme, ohne das Anwendungsprogramm beim Wechsel des Datenbanksystems ändern zu müssen. Das *JDBC*-API erlaubt es einem Java-Programm, SQL-Anweisungen an beliebige relationale Datenbanken zu schicken.

Andere Middleware-Produkte verwenden eine synchrone Verbindung zwischen Client und Server, um Prozeduren nach dem *RPC-Modell* aufzurufen.

Der *Remote Procedure Call* (*RPC*) versteckt den Aufruf einer auf einem anderen Rechner im Netz implementierten Prozedur hinter einem lokalen Prozeduraufruf und bietet damit ein sehr vertrautes Programmiermodell an. Das Programm, das

die Prozedur aufruft, agiert als Client, das Programm, das die aufgerufene Prozedur ausführt, als Server.

RMI (*Remote Method Invocation*) ist die objektorientierte Umsetzung des RPC-Modells für Java-Programme (siehe Kapitel 5).

MOM (Message Oriented Middleware) unterstützt den zeitversetzten (asynchronen) Austausch von Nachrichten mit Hilfe eines Vermittlers (Message Broker), der eine Warteschlange verwaltet. Die Nachricht des Senders wird vom Vermittler in die Warteschlange des Empfängers gelegt. Der Empfänger kann diese Nachricht zu einem späteren Zeitpunkt aus der Warteschlange holen. Sender und Empfänger agieren unabhängig voneinander und sind also über den Vermittler nur lose gekoppelt (siehe Kapitel 4).

1.3 Grundlagen zu TCP/IP

Das *Internet* ist ein weltumspannendes Rechnernetz, das aus einer Vielzahl von internationalen und nationalen öffentlichen Netzen sowie privaten lokalen Netzen besteht. Spezielle Kopplungselemente, so genannte Router, verbinden diese Teilnetze miteinander und ermöglichen so den Datenaustausch zwischen Rechnern, die sich in unterschiedlichen Teilnetzen befinden. Das Internet ist ein offenes Netz, zu dem Unternehmen, nicht-kommerzielle Organisationen sowie Privatpersonen gleichermaßen Zugang haben.

Kommunikationsprotokolle beinhalten Sprach- und Handlungsregeln für den Datenaustausch. Sie sind das Verständigungsmittel in einem Rechnernetz.

TCP/IP

TCP/IP (*Transmission Control Protocol/Internet Protocol*) bezeichnet eine Familie von einzelnen, aufeinander abgestimmten Protokollen für die Kommunikation im Internet. *TCP* und *IP* sind die beiden wichtigsten Protokolle dieser Familie.

Im Gegensatz zum offenen Internet ist ein *Intranet* ein privates, innerbetriebliches Rechnernetz auf Basis der Internet-Protokolle ("privates Internet").

Zur Beschreibung der Kommunikation werden allgemein *Schichtenmodelle* eingesetzt. Im Internet werden vier aufeinander aufbauende Schichten unterschieden. Jede Schicht hat ihre eigene Funktionalität, die sie der jeweils höheren Schicht zur Verfügung stellt.

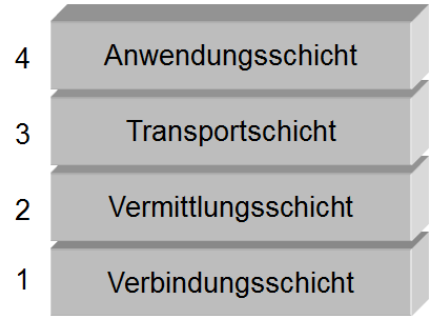

Abbildung 1-12: TCP/IP-Schichtenmodell

1. Die *Verbindungsschicht* umfasst die Netzwerkhardware und Gerätetreiber. Sie ist die Basis für eine zuverlässige physische Verbindung zwischen zwei benachbarten Systemen.
2. Die *Vermittlungsschicht* enthält das Protokoll *IP* (*Internet Protocol*), das die Internet-Adressen (IP-Adressen) definiert, die als Basis für die Wahl der Route im Internet dienen. Einzelne Datenpakete werden vom Sender zum Empfänger über verschiedene Teilnetze des Internets geleitet. Das hier angesiedelte Protokoll *ARP* (*Address Resolution Protocol*) übersetzt logische IP-Adressen in physische Adressen (Hardwareadressen) einer Datenstation im lokalen Netz.
3. Die *Transportschicht* enthält die beiden Protokolle *TCP* (*Transmission Control Protocol*) und *UDP* (*User Datagram Protocol*).

   TCP stellt der Anwendung ein verlässliches, verbindungsorientiertes Protokoll zur Verfügung. Vor der eigentlichen Datenübertragung in Form von Datenpaketen wird eine virtuelle Ende-zu-Ende-Verbindung zwischen Sender und Empfänger aufgebaut.

   UDP ist ein so genanntes verbindungsloses Protokoll. Datenpakete werden ins Internet geschickt, ohne dass vorher eine Verbindung mit dem Empfänger hergestellt wurde.

   UDP und TCP werden in den Kapiteln 2 und 3 ausführlich behandelt.
4. Beispiele für die Protokolle der *Anwendungsschicht* sind: *SMTP* (*Simple Mail Transfer Protocol*), *Telnet*, *FTP* (*File Transfer Protocol*) und *HTTP* (*Hypertext Transfer Protocol*), das die Basis für das *World Wide Web* (*WWW*) ist.

Abbildung 1-13 zeigt den Ablauf einer Kommunikation zwischen Client und Server, die ihre Daten direkt über TCP austauschen.

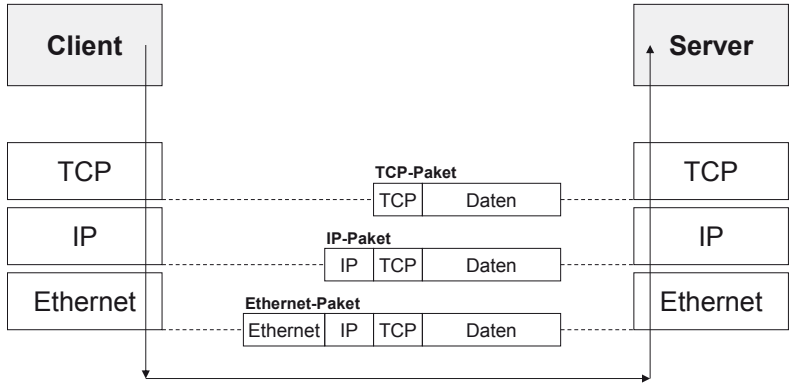

Abbildung 1-13: Datenübertragung mit TCP

Der Client übergibt seine Daten an die Transportschicht. TCP sorgt für die Aufteilung der Daten in Pakete, versieht jeweils die Nutzdaten eines Pakets mit einem Datenkopf (Header) und gibt die Pakete weiter an die Vermittlungsschicht. Die Vermittlungsschicht fügt ihrerseits einen IP-Header hinzu. Schließlich packt die Netzwerkkarte einen Rahmen um die Daten. Beim Empfänger werden in umgekehrter Reihenfolge die Header Schicht für Schicht wieder entfernt und ausgewertet, bevor die Daten an die übergeordnete Schicht weitergegeben werden.

IP-Adressen

Eine Aufgabe des Protokolls IP ist die Bereitstellung eines Adressierungsschemas, das von den physischen Adressen der Netzwerkhardware unabhängig ist.

Jeder Rechner im Internet (genauer: der Rechner als Komponente eines Teilnetzes) hat eine eindeutige *IP-Adresse*, die in der Version *IPv4* aus 32 Bit besteht und durch eine Folge von vier durch Punkte getrennte Zahlen zwischen 0 und 255 dargestellt wird.

Eine IP-Adresse besteht aus zwei Teilen: der *Netzadresse*, die das Netz eindeutig identifiziert, und der *Rechneradresse*, die den Rechner in diesem Netz eindeutig identifiziert. Offizielle Netzadressen werden zentral vergeben, Rechneradressen können frei vom Netzwerkadministrator der jeweiligen Institution vergeben werden.

Die Adressen können in fünf verschiedene Klassen eingeteilt werden. Adressen der *Klasse A* nutzen 8 Bit für die Netzadresse (beginnend mit 0), Adressen der *Klasse B* 16 Bit (beginnend mit 10) und Adressen der *Klasse C* 24 Bit (beginnend mit 110). Adressen der *Klasse D* sind für Multicast-Anwendungen (siehe Kapitel 2.6), Adressen der *Klasse E* sind für zukünftige Zwecke reserviert.

Adressbereiche:

Klasse A: `1.0.0.0 - 127.255.255.255`
Klasse B: `128.0.0.0 - 191.255.255.255`
Klasse C: `192.0.0.0 - 223.255.255.255`
Klasse D: `224.0.0.0 - 239.255.255.255`
Klasse E: `240.0.0.0 - 247.255.255.255`

Beispiel:

Die IP-Adresse `194.94.124.236` gehört zur Klasse C und identifiziert einen Rechner in dem durch die Netzadresse `194.94.124.0` identifizierten Netz.

Die starre Klasseneinteilung der IP-Adressen wird durch das heute übliche Verfahren *CIDR* (*Classless Inter-Domain Routing*) aufgehoben. Hierbei erfolgt die Aufteilung in Netz- und Rechneradresse bitweise mit Hilfe von so genannten Subnetzmasken.

Jeder Rechner im Internet kann sich selbst mit der Adresse `127.0.0.1` adressieren (*Loopback-Adresse*).

Private IP-Adressen wie z. B. `10.0.0.0` bis `10.255.255.255` oder `192.168.0.0` bis `192.168.255.255` werden nie im Internet geroutet. Jede Organisation kann sie frei im internen Netz verwenden. Um eine Verbindung mit dem Internet zu ermöglichen, werden private Adressen in öffentliche Adressen übersetzt und umgekehrt.

Die neue Version *IPv6* bietet vor allem einen größeren Adressraum. Es werden 128 Bit zur Darstellung der Adresse verwendet. IPv6-Adressen werden hexadezimal notiert, wobei die Adresse in 8 Blöcken zu jeweils 16 Bit unterteilt wird. Führende Nullen innerhalb eines Blockes dürfen weggelassen werden. Blöcke werden durch einen Doppelpunkt getrennt. Blöcke mit lauter Nullen können durch zwei Doppelpunkte ersetzt werden. Diese Notation ist aber nur an einer Stelle der Adresse erlaubt, damit die Eindeutigkeit nicht verloren geht.

Beispiel:

`2001:0db8:0:0:0:0:1428:57ab` ist gleichbedeutend mit `2001:db8::1428:57ab`

Domain Name System

Zur besseren Handhabbarkeit wird das Nummernsystem durch ein Namensystem, dem *Domain Name System* (*DNS*), überlagert. IP-Adressen werden einem sprechenden Namen, dem *Domain-Namen*, zugeordnet. Domain-Namen sind hierarchisch aufgebaut.

Beispiel: `www.hs-niederrhein.de`

Hier handelt es sich um den Webserver der Hochschule Niederrhein mit `de` (Deutschland) als *Top Level Domain*.

Der Hostname `localhost` identifiziert den eigenen Rechner und entspricht der IP-Adresse `127.0.0.1`.

DNS-Server sind spezielle Verzeichnisdienste, die die Zuordnung von IP-Adressen zu DNS-Namen und umgekehrt unterstützen. Hierüber kann z. B. die IP-Adresse zu einem Namen erfragt werden.

Zu beachten ist, dass mehrere Rechner mit jeweils eigener IP-Adresse denselben DNS-Namen haben können und zu einer IP-Adresse mehrere DNS-Namen gehören können.

Portnummern

Um einen bestimmten Dienst (Server) im Internet zu identifizieren, reicht die IP-Adresse des Rechners, auf dem der Server läuft, nicht aus, da mehrere Server auf demselben Rechner gleichzeitig laufen können.

Zur Identifizierung eines Servers dient neben der IP-Adresse des Rechners die so genannte *Portnummer*. Portnummern werden auch vom Server benutzt, um den anfragenden Client für die Rückantwort zu adressieren. Portnummern sind ganze Zahlen von 0 bis 65535. Sie werden von den Protokollen der Transportschicht verwendet. TCP und UDP können Portnummern unabhängig voneinander verwenden, d. h. ein TCP-Server und ein UDP-Server können auf einem Rechner unter derselben Portnummer zur gleichen Zeit laufen.

Die Portnummern von 0 bis 1023 (*System Ports*) sind für so genannte *well-known services* reserviert.

Im Bereich von 1024 bis 49151 sind weitere Portnummern (*User Ports*) registriert, z. B. 1099 für die RMI-Registry und 3306 für den MySQL-Datenbankserver.

Der Bereich 49152 – 65535 steht zur freien Verfügung.

Beispiele für System Ports:

Port	Protokoll	Service	Beschreibung
7	TCP/UDP	Echo	liefert die Eingabe als Antwort zurück
13	TCP/UDP	Daytime	liefert die aktuelle Zeit des Servers
20	TCP	FTP (Data)	überträgt Daten zum Server
21	TCP	FTP (Control)	sendet FTP-Kommandos zum Server

23	TCP	Telnet	Terminalemulation
25	TCP	SMTP	überträgt E-Mails zwischen Rechnern
53	TCP/UDP	DNS	Domain Name System
80	TCP	HTTP	Webserver

1.4 Java-Klassen für IP-Adressen und Sockets

InetAddress

Die Klasse `java.net.InetAddress` repräsentiert eine IP-Adresse. Subklassen hiervon sind `Inet4Address` und `Inet6Address`, die IP-Adressen der Version 4 bzw. 6 repräsentieren.

Die folgenden drei statischen Methoden erzeugen `InetAddress`-Objekte:

`static InetAddress getLocalHost() throws UnknownHostException`
ermittelt die IP-Adresse des Rechners, auf dem diese Methode aufgerufen wird. `java.net.UnknownHostException` ist Subklasse von `java.io.IOException`. Eine Ausnahme dieses Typs wird ausgelöst, wenn die IP-Adresse eines Rechners nicht ermittelt werden kann.

`static InetAddress getByName(String host) throws UnknownHostException`
liefert die IP-Adresse zu einem Hostnamen. Enthält das Argument host die numerische Adresse anstelle eines Namens, so wird nur die Gültigkeit des Adressformats geprüft.

`static InetAddress[] getAllByName(String host) throws UnknownHostException`
liefert ein Array von IP-Adressen, die alle zum vorgegebenen Hostnamen gehören.

Weitere `InetAddress`-Methoden:

`String getHostAddress()`
liefert die IP-Adresse eines `InetAddress`-Objekts als Zeichenkette.

`String getHostName()`
liefert den Hostnamen zu dieser IP-Adresse. Falls das `InetAddress`-Objekt mittels einer numerischen Adresse erzeugt wurde und der zugehörige Hostname nicht ermittelt werden konnte, wird diese Adresse wieder zurückgegeben.

`boolean isReachable(int timeout) throws IOException`
prüft, ob die IP-Adresse erreicht werden kann. Dieser Versuch dauert maximal `timeout` Millisekunden. Einstellungen des zu kontaktierenden Servers und Firewalls können allerdings durch den Aufruf dieser Methode initiierte Anfragen blockieren.

Das Programm `Localhost` zeigt den Hostnamen und die IP-Adresse des eigenen Rechners an.

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Localhost {
   public static void main(String[] args) throws UnknownHostException {
      InetAddress address = InetAddress.getLocalHost();
      System.out.printf("%s/%s%n", address.getHostName(),
            address.getHostAddress());
   }
}
```

Das Programm `Lookup` liefert die IP-Adresse zu einem Hostnamen.

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Lookup {
   public static void main(String[] args) throws UnknownHostException {
      InetAddress address = InetAddress.getByName(args[0]);
      System.out.printf("Looking up %s -> %s/%s%n", args[0],
            address.getHostName(), address.getHostAddress());
   }
}
```

Wird im lokalen Netz kein Domain Name Server (DSN) genutzt, so können unter Windows in der Datei `hosts` im Verzeichnis `Windows\System32\drivers\etc` die Zuordnungen von IP-Adressen zu frei wählbaren Hostnamen festgelegt werden.

Aufruf:

```
java -cp bin Lookup www.hs-niederrhein.de
```

Ausgabe des Programms:

```
Looking up www.hs-niederrhein.de -> www.hs-niederrhein.de/194.94.120.91
```

Wird beim Aufruf des Programms eine numerische Adresse anstelle eines Namens mitgegeben und konnte der Hostname nicht ermittelt werden, so wird als Hostname die vorgegebene Adresse ausgegeben.

InetSocketAddress

Java-Programme nutzen so genannte *Sockets* als Programmierschnittstelle für die Kommunikation über UDP/IP und TCP/IP. Sockets stellen die Endpunkte einer

Kommunikationsbeziehung zwischen Prozessen dar. Sie basieren auf einer IP-Socket-Adresse, bestehend aus einer IP-Adresse und einer Portnummer.

Die Klasse `java.net.InetSocketAddress` repräsentiert eine IP-Socket-Adresse (IP-Adresse und Portnummer). Diese Klasse ist von der abstrakten Klasse `java.net.SocketAddress` abgeleitet.

Um ein `InetSocketAddress`-Objekt zu erzeugen, stehen die folgenden Konstruktoren zur Verfügung:

`InetSocketAddress(int port)`
: erzeugt eine Socket-Adresse mit der so genannten *Wildcard-Adresse* `0.0.0.0`. Hiermit ist irgendeine IP-Adresse des Rechners, auf dem der Konstruktor aufgerufen wird, gemeint. Zu beachten ist, dass ein Rechner mehrere Netzwerkadapter mit jeweils eigener IP-Adresse haben kann.

`InetSocketAddress(InetAddress addr, int port)`

`InetSocketAddress(String hostname, int port)`

Einige `InetSocketAddress`-Methoden:

`InetAddress getAddress()`
: liefert die IP-Adresse.

`String getHostName()`
: liefert den Hostnamen.

`int getPort()`
: liefert die Portnummer.

Das Programm `CreateSocketAddress` wendet die verschiedenen Konstruktoren zur Veranschaulichung an.

```
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.UnknownHostException;

public class CreateSocketAddress {
   public static void main(String[] args) {
      InetSocketAddress socketAddress;

      socketAddress = new InetSocketAddress(80);
      output(socketAddress);

      try {
         InetAddress address = InetAddress.getByName("www.google.de");
         socketAddress = new InetSocketAddress(address, 80);
         output(socketAddress);
      } catch (UnknownHostException e) {
         System.out.println(e);
      }
```

```
        socketAddress = new InetSocketAddress("www.google.de", 80);
        output(socketAddress);

        socketAddress = new InetSocketAddress("localhost", 80);
        output(socketAddress);
    }

    private static void output(InetSocketAddress socketAddress) {
        System.out.println(socketAddress.getAddress());
        System.out.println(socketAddress.getHostName());
        System.out.println(socketAddress.getPort());
        System.out.println();
    }
}
```

Ausgabe:

```
0.0.0.0/0.0.0.0
0.0.0.0
80

www.google.de/173.194.112.216
www.google.de
80

www.google.de/173.194.112.216
www.google.de
80

localhost/127.0.0.1
localhost
80
```

1.5 Aufgaben

1. Entwickeln Sie eine verbesserte Version des Programms `Lookup` mit den folgenden Eigenschaften:
   - Sind einem Hostnamen mehrere IP-Adressen zugeordnet, so sollen alle angezeigt werden.
   - Wird statt eines Namens eine numerische Adresse beim Aufruf mitgegeben, so soll eine Fehlermeldung ausgegeben werden, wenn der Hostname nicht ermittelt werden konnte, ansonsten soll der zugeordnete Hostname angezeigt werden.

2. Schreiben Sie das Programm `Localhost`, das den Hostnamen und die IP-Adresse des lokalen Rechners in einem Fenster anzeigt.

Um `Localhost` mit einem Doppelklick auf einem grafischen Symbol starten zu können, muss eine jar-Datei erzeugt werden:

```
jar cfm localhost.jar manifest.txt -C bin Localhost.class
```

Die Datei `manifest.txt` hat den Inhalt:

```
Main-Class: Localhost
```

Diese Textzeile muss mit einem Zeilenvorschub abgeschlossen sein. Eine Verknüpfung mit `localhost.jar` kann z. B. auf den Desktop gelegt werden.

3. Schreiben Sie das Programm `Ping`, das die Erreichbarkeit einer IP-Adresse testet.

2 Verbindungslose Kommunikation mit UDP

Das *User Datagram Protocol* (*UDP*) stellt grundlegende Funktionen zur Verfügung, um mit geringem Aufwand Daten zwischen kommunizierenden Prozessen austauschen zu können. UDP ist als Transportprotokoll der dritten Schicht im TCP/IP-Schichtenmodell zugeordnet und nutzt den Vermittlungsdienst IP.

2.1 Das Protokoll UDP

UDP ist ein verbindungsloses Protokoll, d. h. es bietet keinerlei Kontrolle über die sichere Ankunft versendeter Datenpakete, so genannter Datagramme. UDP ist im *RFC* (Request for Comments) 768 der *IETF* (*Internet Engineering Task Force*) spezifiziert.[1]

Datagramme müssen nicht den Empfänger in der Reihenfolge erreichen, in der sie abgeschickt werden. Datagramme können verloren gehen, weder Sender noch Empfänger werden über den Verlust informiert. Fehlerhafte Datagramme werden nicht nochmals übertragen. Abbildung 2.1 zeigt den Aufbau des *UDP-Datagramms*, das in ein IPv4-Paket eingebettet ist.

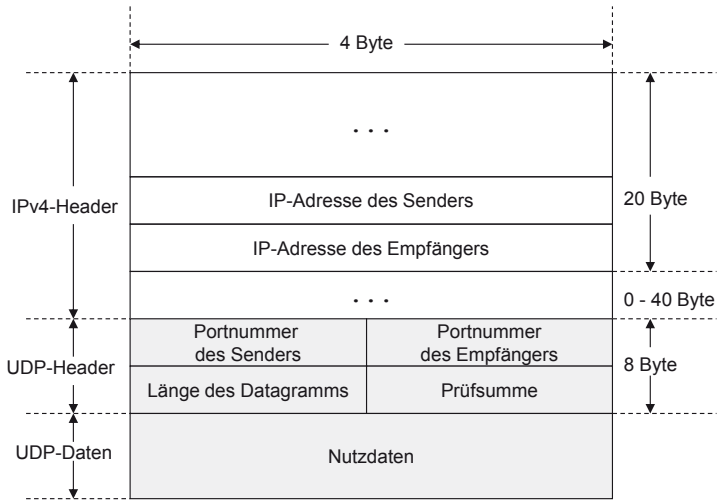

Abbildung 2-1: Aufbau des UDP-Datagramms

[1] `http://www.ietf.org/rfc/rfc768.txt`

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_2, © Springer Fachmedien Wiesbaden 2015

Der UDP-Header enthält vier 16 Bit lange Felder:

- die Portnummer des Senders,
- die Portnummer des Empfängers,
- die Gesamtlänge des UDP-Datagramms und
- die Prüfsumme.

Darauf folgen dann die eigentlichen Nutzdaten des Datagramms.

Die Maximallänge eines UDP-Datagramms ist begrenzt. Anwendungen sollten nicht mehr als 508 Byte Nutzdaten in einem Datagramm versenden, um Probleme infolge einer evtl. nötigen Fragmentierung des IP-Pakets durch Router zu vermeiden.

UDP ist ein einfaches Mittel, um Nachrichten, die keine zuverlässige Übertragung erfordern, zu versenden. UDP hat gegenüber TCP den Vorteil eines viel geringeren Kontroll-Overheads.

UDP-Anwendungen

Einige bekannte Anwendungen, die UDP nutzen, sind:

- das *Domain Name System* (*DNS*) zur automatischen Konvertierung von Domain-Namen (Rechnernamen) in IP-Adressen und umgekehrt,
- das *Simple Network Management Protocol* (*SNMP*) zur Verwaltung von Ressourcen in einem TCP/IP-basierten Netz,
- das *Trivial File Transfer Protocol* (*TFTP*), das einen einfachen Dateitransferdienst zum Sichern und Laden von System- und Konfigurationsdateien bietet,
- das *Dynamic Host Configuration Protocol* (*DHCP*) zur dynamischen Zuweisung von IP-Adressen.

Die unidirektionale Übertragung von Video- und Audiosequenzen zur Wiedergabe in Echtzeit (*Multimedia Streaming*) ist besonders zeitkritisch. Der gelegentliche Verlust eines Datenpakets kann aber toleriert werden. Hier ist UDP sehr gut geeignet.

2.2 DatagramSocket und DatagramPacket

Ein UDP-Socket wird an eine Portnummer auf dem lokalen Rechner gebunden. Er kann dann Datagramme an jeden beliebigen anderen UDP-Socket senden und von jedem beliebigen UDP-Socket empfangen (siehe Abbildung 2.2).

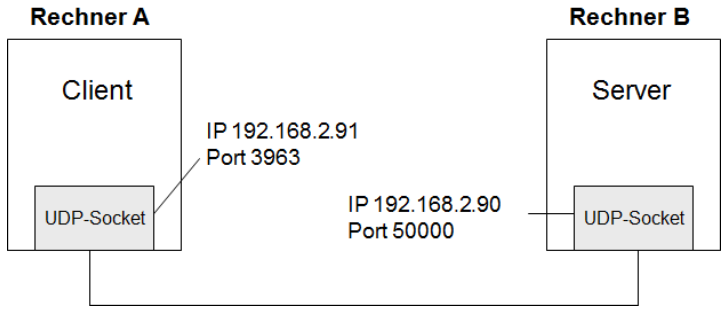

Abbildung 2-2: UDP-Sockets

DatagramSocket

Die Klasse `java.net.DatagramSocket` repräsentiert einen UDP-Socket zum Senden und Empfangen von UDP-Datagrammen. Ein UDP-Socket muss an einen Port des lokalen Rechners gebunden werden. Hierzu stehen Konstruktoren zur Verfügung:

`DatagramSocket() throws SocketException`
: erzeugt einen UDP-Socket und bindet ihn an einen verfügbaren Port des lokalen Rechners.

`DatagramSocket(int port) throws SocketException`
: erzeugt einen UDP-Socket und bindet ihn an die Portnummer `port` des lokalen Rechners. Hierdurch kann ein Server den Port, den er nutzen möchte, festlegen.

`DatagramSocket(int port, InetAddress laddr) throws SocketException`
`DatagramSocket(SocketAddress bindaddr) throws SocketException`
: Die beiden letzten Konstruktoren binden den Socket an eine vorgegebene Adresse.

Alle Konstruktoren können bei Fehlern im zugrunde liegenden Protokoll die Ausnahme `java.net.SocketException` auslösen. `SocketException` ist Subklasse von `java.io.IOException`.

Einige `DatagramSocket`-Methoden:

`void close()`
: schließt den Socket.

`InetAddress getLocalAddress()`
: liefert die IP-Adresse, an die der Socket gebunden ist.

`int getLocalPort()`
: liefert die Portnummer, an die der Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
liefert die Socket-Adresse, an die der Socket gebunden ist.

DatagramPacket

Die Klasse `java.net.DatagramPacket` repräsentiert ein UDP-Datagramm.

Der Konstruktor

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

erzeugt ein UDP-Datagramm zum Senden von Daten. `buf` enthält die Daten, `address` ist die IP-Adresse und `port` die Portnummer des Empfängers. Die Anzahl `length` der zu übertragenden Bytes muss kleiner oder gleich der Pufferlänge sein.

Ein ähnlicher Konstruktor, der eine Socket-Adresse verwendet:

```
DatagramPacket(byte[] buf, int length, SocketAddress address)
       throws SocketException
```

Der Konstruktor

```
DatagramPacket(byte[] buf, int length)
```

erzeugt ein UDP-Datagramm zum Empfangen von Daten. `buf` ist der Puffer, der die empfangenen Daten aufnimmt. `length` legt fest, wie viele Bytes maximal gelesen werden sollen. `length` muss kleiner oder gleich der Pufferlänge sein.

Wendet man auf dieses Paket die Methoden `setAddress` und `setPort` (siehe weiter unten) an, so kann es auch gesendet werden.

Senden und empfangen

Die `DatagramSocket`-Methode

```
void send(DatagramPacket p) throws IOException
```

sendet ein UDP-Datagramm von diesem Socket aus.

Die `DatagramSocket`-Methode

```
void receive(DatagramPacket p) throws IOException
```

empfängt ein UDP-Datagramm von diesem Socket. `p` enthält die Daten sowie die IP-Adresse und die Portnummer des Senders. Die Methode blockiert so lange, bis ein Datagramm empfangen wurde.

Weitere `DatagramPacket`-Methoden sind:

`byte[] getData()`
liefert den Datenpufferinhalt.

`int getLength()`
liefert die Länge der Daten, die empfangen wurden bzw. gesendet werden sollen.

`InetAddress getAddress()`
liefert die (entfernte) IP-Adresse des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`int getPort()`
liefert die (entfernte) Portnummer des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`SocketAddress getSocketAddress()`
liefert die (entfernte) Socket-Adresse des Senders (beim empfangenen Datagramm) bzw. des Empfängers (beim zu sendenden Datagramm).

`void setData(byte[] buf)`
setzt den Datenpuffer des Datagramms.

`void setLength(int length)`
bestimmt die Anzahl Bytes, die gesendet werden sollen bzw. die maximale Anzahl Bytes, die empfangen werden sollen. `length` muss kleiner oder gleich der Pufferlänge des Datagramms sein.

`void setAddress(InetAddress address)`
setzt die IP-Adresse des Rechners, an den das Datagramm gesendet werden soll.

`void setPort(int port)`
setzt die Portnummer, an die das Datagramm gesendet werden soll.

`void setSocketAddress(SocketAddress address)`
setzt die Socket-Adresse des Rechners, an den das Datagramm gesendet werden soll.

In allen Beispielen dieses Kapitels ist die Bearbeitungszeit des Servers nach Eintreffen eines Pakets relativ kurz. Deshalb ist eine iterative Implementierung (Paket empfangen – Anfrage bearbeiten – Paket senden) ausreichend.

Bei längeren Bearbeitungszeiten sollten für die Bearbeitung und das Senden der Antwort zu einer Anfrage Threads benutzt werden (parallele Implementierung), um die durchschnittliche Wartezeit eines Clients zu reduzieren.

2.3 Ein Echo-Server und -Client

Das erste Programmbeispiel ist eine einfache Client-Server-Anwendung, die den Umgang mit den Methoden des vorigen Abschnitts zeigen soll.

Der Client schickt einen Text, den der Server als Echo an den Absender zurückschickt.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class EchoServer {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      int port = Integer.parseInt(args[0]);

      try (DatagramSocket socket = new DatagramSocket(port)) {
         // Pakete zum Empfangen bzw. Senden
         DatagramPacket packetIn = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE);
         DatagramPacket packetOut = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE);

         System.out.println("Server gestartet ...");

         while (true) {
            // Paket empfangen
            socket.receive(packetIn);
            System.out.println("Received: " + packetIn.getLength()
                  + " bytes");

            // Daten und Länge im Antwortpaket speichern
            packetOut.setData(packetIn.getData());
            packetOut.setLength(packetIn.getLength());

            // Zieladresse und Zielport im Antwortpaket setzen
            packetOut.setSocketAddress(packetIn.getSocketAddress());

            // Antwortpaket senden
            socket.send(packetOut);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

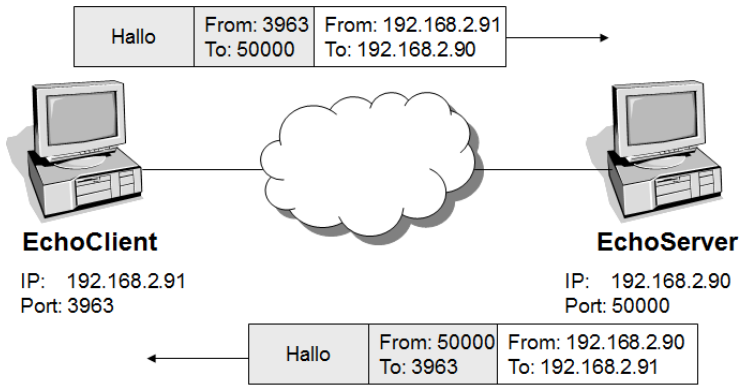

Abbildung 2-3: Echo-Service

`EchoServer` wird mit dem Parameter Portnummer aufgerufen.

Zu Beginn wird ein UDP-Socket erzeugt und an die vorgegebene Portnummer gebunden. Es werden zwei Datagramme zum Empfangen bzw. Senden von Daten mit der Pufferlänge 508 Byte erzeugt. In einer Endlos-Schleife werden Daten empfangen und gesendet. Nutzdaten und Datenlänge sowie IP-Adresse und Portnummer werden aus dem empfangenen Paket gelesen und in das zu sendende Paket übertragen.

Der Server kann über Tastatur mit `Strg + C` abgebrochen werden.

Aufruf des Servers:

```
java -cp bin EchoServer 50000
```

Das Programm `EchoClient` sendet ein Datagramm an den Server. Neben dem Hostnamen und der Portnummer des Servers werden die Nutzdaten des Datagramms als weiterer Parameter beim Aufruf mitgegeben.

Erhält der Client nicht innerhalb von zwei Sekunden eine Antwort vom Server, wird eine Ausnahme ausgelöst und das Programm beendet.

Diese Zeitsteuerung wird durch Aufruf der folgenden `DatagramSocket`-Methode erreicht:

`void setSoTimeout(int timeout) throws SocketException`
: setzt ein Timeout in Millisekunden. `receive` für diesen Socket blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert. `SocketTimeoutException` ist Subklasse von `java.io.InterruptedIOException`.

`int getSoTimeout() throws SocketException`
: liefert die Timeout-Angabe.

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketTimeoutException;

public class EchoClient {
   private static final int BUFSIZE = 508;
   private static final int TIMEOUT = 2000;

   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      byte[] data = args[2].getBytes();

      try (DatagramSocket socket = new DatagramSocket()) {
         // Maximal TIMEOUT Millisekunden auf Antwort warten
```

```
            socket.setSoTimeout(TIMEOUT);

            // Paket an Server senden
            InetAddress addr = InetAddress.getByName(host);
            DatagramPacket packetOut = new DatagramPacket(data, data.length,
                  addr, port);
            socket.send(packetOut);

            // Antwortpaket empfangen
            DatagramPacket packetIn = new DatagramPacket(new byte[BUFSIZE],
                  BUFSIZE);
            socket.receive(packetIn);

            String received = new String(packetIn.getData(), 0,
                  packetIn.getLength());
            System.out.println("Received: " + received);

         } catch (SocketTimeoutException e) {
            System.err.println("Timeout: " + e.getMessage());
         } catch (Exception e) {
            System.err.println(e);
         }
      }
   }
```

Aufruf des Client auf demselben Rechner, auf dem auch der Server läuft:

```
java -cp bin EchoClient localhost 50000 Hallo
```

2.4 Vorgegebene Socket-Verbindungen

Die Klasse `DatagramSocket` enthält eine Methode, die es ermöglicht, die Adresse, an die Datagramme gesendet werden bzw. von der Datagramme empfangen werden, gezielt auszuwählen und alle anderen Datagramme zu verwerfen:

```
void connect(InetAddress address, int port)
void connect(SocketAddress addr) throws SocketException
```

Datagramme können nur an diese IP-Adresse/Portnummer gesendet bzw. von dieser IP-Adresse/Portnummer empfangen werden.

```
void disconnect()
```

hebt die durch `connect` hergestellte Restriktion auf.

Mit den `DatagramSocket`-Methoden

```
InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
```

können IP-Adresse/Portnummer, die in `connect` verwendet wurden, abgefragt werden. Wurde `connect` nicht benutzt, wird `null` bzw. `-1` geliefert.

Im folgenden Beispiel schickt `Sender` eine Nachricht als Datagramm an den Server. `Receiver` zeigt die empfangene Nachricht auf dem Bildschirm an. Er soll nur Datagramme, die von einer bestimmten IP-Adresse und Portnummer aus gesendet wurden, empfangen können. Zu diesem Zweck wird der Receiver mit den folgenden Parametern aufgerufen:

- lokale Portnummer,
- IP-Adresse/Hostname des Client und
- Portnummer des Client.

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Receiver {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      int port = Integer.parseInt(args[0]);
      String remoteHost = args[1];
      int remotePort = Integer.parseInt(args[2]);

      try (DatagramSocket socket = new DatagramSocket(port)) {
         // Es werden nur Datagramme dieser Adresse entgegengenommen
         socket.connect(InetAddress.getByName(remoteHost), remotePort);

         DatagramPacket packet = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE);

         while (true) {
            socket.receive(packet);
            String text = new String(packet.getData(), 0,
                  packet.getLength());
            System.out.println(packet.getAddress().getHostAddress() + ":"
                  + packet.getPort() + " > " + text);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

Beim Client `Sender` wird der UDP-Socket an eine explizit vorgegebene Portnummer gebunden, die beim Aufruf des Programms als Parameter mitgegeben wird.

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Sender {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) throws Exception {
```

```
      int localPort = Integer.parseInt(args[0]);
      String host = args[1];
      int port = Integer.parseInt(args[2]);
      String message = args[3];

      try (DatagramSocket socket = new DatagramSocket(localPort)) {
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket packet = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE, addr, port);
         byte[] data = message.getBytes();
         packet.setData(data);
         packet.setLength(data.length);
         socket.send(packet);
      }
   }
}
```

Aufruf des Serverprogramms:

```
java -cp bin Receiver 50000 localhost 40000
```

Es können nur Datagramme von `localhost` mit Portnummer `40000` empfangen werden.

Aufruf des Clientprogramms auf demselben Rechner:

```
java -cp bin Sender 40000 localhost 50000 "Das ist ein Test!"
```

Der Socket des Clients ist an die Portnummer `40000` gebunden. Wird hier beim Aufruf statt `40000` z. B. `30000` angegeben, so ignoriert der Server die vom Client gesendeten Datagramme.

2.5 Online-Unterhaltung

Mit dem folgenden Programm `Talk` können zwei Benutzer an verschiedenen Rechnern eine Unterhaltung online führen.

Benutzer "Hugo" sitzt am Rechner A (IP-Adresse 192.168.2.90), Benutzer "Emil" am Rechner B (IP-Adresse 192.168.2.91).

Benutzer "Hugo" ruft das Programm wie folgt auf:

```
java -cp bin Talk -user Hugo -localPort 50000 -remoteHost 192.168.2.91
-remotePort 50000
```

Benutzer "Emil" gibt das folgende Kommando ein:

```
java -cp bin Talk -user Emil -localPort 50000 -remoteHost 192.168.2.90
-remotePort 50000
```

Die Eingaben müssen jeweils in einer einzigen Zeile erfolgen. Der UDP-Socket wird hier jeweils an die explizit vorgegebene Portnummer `50000` (zweiter Aufrufparameter) gebunden.

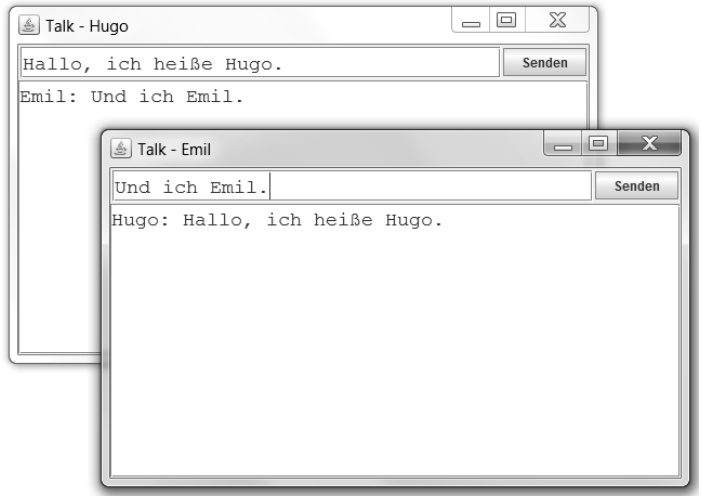

Abbildung 2-4: Online-Unterhaltung mit zwei Teilnehmern

Da das Programm für die beiden Portnummern den Standardwert `50000` eingestellt hat, kann das Programm auch vereinfacht wie folgt aufgerufen werden:

```
java -cp bin Talk -user Hugo -remoteHost 192.168.2.91
java -cp bin Talk -user Emil -remoteHost 192.168.2.90
```

Zum Test kann das Programm auch zweimal auf demselben Rechner (`localhost`) gestartet werden. Allerdings müssen dann die Portnummern unterschiedlich sein:

```
java -cp bin Talk -user Hugo -localPort 40000 -remoteHost localhost
-remotePort 50000
java -cp bin Talk -user Emil -localPort 50000 -remoteHost localhost
-remotePort 40000
```

Die Eingaben müssen wieder jeweils in einer einzigen Zeile erfolgen.

```
import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class Talk implements ActionListener, Runnable {
   private static final int BUFSIZE = 508;

   private String user;
   private int localPort = 50000;
   private String remoteHost;
   private int remotePort = 50000;

   private DatagramSocket socket;
   private DatagramPacket packetOut;
   private JTextField input;
   private JTextArea output;

   public static void main(String[] args) {
      JFrame frame = new JFrame();

      try {
         new Talk(args, frame);
      } catch (Exception e) {
         System.err.println(e.getMessage());
         System.exit(1);
      }

      frame.pack();
      frame.setVisible(true);
   }

   public Talk(String[] args, JFrame frame) throws Exception {
      setParameter(args);
      if (user == null || remoteHost == null || localPort == 0 ||
            remotePort == 0)
         throw new RuntimeException("Parameter fehlen");

      InetAddress remoteAddr = InetAddress.getByName(remoteHost);

      frame.setTitle("Talk - " + user);

      socket = new DatagramSocket(localPort);
      packetOut = new DatagramPacket(new byte[BUFSIZE], BUFSIZE, remoteAddr,
            remotePort);

      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            if (socket != null)
               socket.close();
            System.exit(0);
         }
      });

      JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 2, 2));
      input = new JTextField(40);
      input.setFont(new Font("Monospaced", Font.PLAIN, 18));
      panel.add(input);
      JButton send = new JButton("Senden");
      send.addActionListener(this);
```

```
      panel.add(send);
      frame.add(panel, BorderLayout.NORTH);

      output = new JTextArea(10, 45);
      output.setFont(new Font("Monospaced", Font.PLAIN, 18));
      output.setLineWrap(true);
      output.setWrapStyleWord(true);
      output.setEditable(false);
      frame.add(new JScrollPane(output), BorderLayout.CENTER);

      // Thread zum Empfangen von Nachrichten starten
      Thread t = new Thread(this);
      t.start();
   }

   public void run() {
      DatagramPacket packetIn = new DatagramPacket(new byte[BUFSIZE], BUFSIZE);

      while (true) {
         try {
            receive(packetIn);
         } catch (IOException e) {
            break;
         }
      }
   }

   private void receive(DatagramPacket packetIn) throws IOException {
      socket.receive(packetIn);
      final String text = new String(packetIn.getData(), 0,
            packetIn.getLength());

      try {
         EventQueue.invokeLater(new Runnable() {
            public void run() {
               output.append(text);
               output.append("\n");
            }
         });
      } catch (Exception e) {
      }
   }

   public void actionPerformed(ActionEvent e) {
      try {
         String message = user + ": " + input.getText();
         byte[] data = message.getBytes();
         packetOut.setData(data);
         packetOut.setLength(data.length);
         socket.send(packetOut);
         input.requestFocus();
      } catch (IOException ex) {
         System.err.println(ex);
      }
   }

   private void setParameter(String[] args) {
      for (int i = 0; i < args.length; i++) {
         if (args[i].equals("-user"))
            user = args[i + 1];
         else if (args[i].equals("-localPort"))
```

```
            localPort = Integer.parseInt(args[i + 1]);
         else if (args[i].equals("-remoteHost"))
            remoteHost = args[i + 1];
         else if (args[i].equals("-remotePort"))
            remotePort = Integer.parseInt(args[i + 1]);
      }
   }
}
```

Im Konstruktor werden zunächst IP-Adressen bestimmt, der UDP-Socket an die vorgegebene lokale Portnummer gebunden und ein Paket zum Senden der eingegebenen Daten erstellt. Beim Schließen des Fensters wird der Socket geschlossen. Textfeld, Button und Textfläche werden erzeugt und platziert sowie das aktuelle Objekt als `ActionListener` beim Button registriert. Schließlich wird ein Thread, der Datagramme empfängt (siehe `run`-Methode), erzeugt und gestartet.

Innerhalb der `run`-Methode werden in einer Endlos-Schleife die Daten empfangen und in der Textfläche angezeigt. Die `DatagramSocket`-Methode `receive` blockiert so lange, bis ein Datagramm empfangen wurde. Die Ausgabe des Textes erfolgt in der `run`-Methode eines `Runnable`-Objekts `r`, das mittels `EventQueue.invokeLater(r)` an den Ereignis-Dispatcher, der für die Aktualisierung der Benutzungsoberfläche verantwortlich ist, zur Ausführung weitergeleitet wird.

Beim Drücken des Buttons "Senden" werden die Daten des Eingabefeldes in einem Datagramm an die Zieladresse gesendet.

2.6 Punkt-zu-Mehrpunkt-Verbindungen

Bisher haben wir in diesem Kapitel ausschließlich *Punkt-zu-Punkt-Verbindungen* (*Unicast*) betrachtet. Unter *Multicast* versteht man eine Übertragung von einem Teilnehmer zu einer Gruppe von Teilnehmern (*Punkt-zu-Mehrpunkt-Verbindung*).

Der Vorteil dieses Verfahrens liegt darin, dass gleichzeitig Pakete über eine einzige Adresse an mehrere Teilnehmer gesendet werden. Die Vervielfältigung der Pakete findet nicht beim Sender statt, sondern erst an jedem Verteiler (Switch, Router) auf dem Übertragungsweg. So wird *Multicasting* insbesondere zur Übertragung von Audio- und Videoströmen eingesetzt, um die Netzbelastung zu reduzieren.

Im Internet werden für das Multicasting bei IPv4 die IP-Adressen der Klasse D verwendet: `224.0.0.0 - 239.255.255.255`

Eine *Multicast-Gruppe* wird durch eine Adresse dieses Bereichs (*Multicast-Adresse*) und eine UDP-Portnummer repräsentiert. Die Adresse `224.0.0.0` ist reserviert und sollte nicht benutzt werden.

Teilnehmer können jederzeit einer Multicast-Gruppe beitreten oder diese wieder verlassen. Wird eine Nachricht an eine Multicast-Gruppe gesendet, erhalten alle

Gruppenteilnehmer diese Nachricht, sofern sie sich im Time-to-Live-Bereich des Pakets befinden. Das Feld *Time-to-Live* (*TTL*) im Header eines IP-Pakets gibt an, wie lange ein Paket im Internet verbleiben darf. Es handelt sich um einen Zähler, der von jedem Router, den das Paket passiert, um 1 dekrementiert wird. Bei einem Wert von 0 wird das Paket zerstört. Die Reichweite der gesendeten Pakete kann also durch die Angabe des Time-to-Live-Wertes eingeschränkt werden.

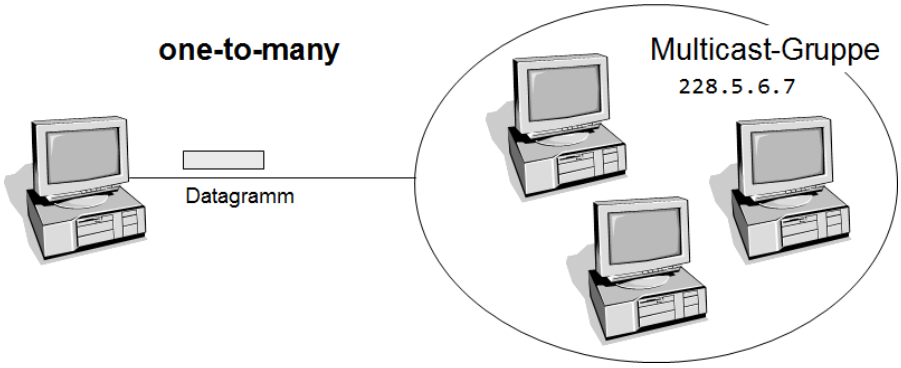

Abbildung 2-5: Multicasting

Die Klasse `java.net.MulticastSocket` (Subklasse von `DatagramSocket`) wird zum Senden und Empfangen von Mulitcast-Datagrammen verwendet.

`void setTimeToLive(int ttl) throws IOException`
setzt den Time-To-Live-Wert. Zulässiger Wert: `0 <= ttl <= 255`. Voreinstellung ist `1`.

`void joinGroup(InetAddress mcastaddr) throws IOException`
tritt der Multicast-Gruppe bei.

`void leaveGroup(InetAddress mcastaddr) throws IOException`
verlässt die Multicast-Gruppe.

Ein Client, der ein Datagramm an eine Multicast-Gruppe sendet, muss selbst nicht Teilnehmer dieser Gruppe sein.

Im folgenden Beispiel bilden zwei Receiver eine Multicast-Gruppe (IP-Adresse: 228.5.6.7, Portnummer: 50000).

Ein Sender schickt in kurzen Abständen Nachrichten an diese Gruppe.

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.util.Random;

public class Sender {
   private static final int BUFSIZE = 508;
   private static final int LOOPS = 10;

   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);

      try (DatagramSocket socket = new DatagramSocket()) {
         InetAddress addr = InetAddress.getByName(host);
         DatagramPacket packet = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE, addr, port);

         for (int i = 0; i < LOOPS; i++) {
            Random random = new Random();
            int value = random.nextInt();
            byte[] data = String.valueOf(value).getBytes();
            packet.setData(data);
            packet.setLength(data.length);
            socket.send(packet);
            Thread.sleep(3000);
         }
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

```
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class Receiver {
   private static final int BUFSIZE = 508;

   public static void main(String[] args) {
      String multicastAddress = args[0];
      int port = Integer.parseInt(args[1]);

      try (MulticastSocket socket = new MulticastSocket(port)) {
         socket.setTimeToLive(1);
         InetAddress group = InetAddress.getByName(multicastAddress);
         socket.joinGroup(group);

         DatagramPacket packetIn = new DatagramPacket(new byte[BUFSIZE],
               BUFSIZE);

         while (true) {
            socket.receive(packetIn);
            String received = new String(packetIn.getData(), 0,
                  packetIn.getLength());
            System.out.println(received);
         }
```

```
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Zum Testen auf einem lokalen Windows-Rechner muss das Netzwerk-Interface mit einer IP-Adresse konfiguriert sein. Ggf. sind Firewall-Einstellungen anzupassen.

```
java -cp bin Receiver 228.5.6.7 50000
java -cp bin Receiver 228.5.6.7 50000
java -cp bin Sender 228.5.6.7 50000
```

2.7 Aufgaben

1. Entwickeln Sie einen Server (`DaytimeServer`), der als Reaktion auf den Empfang eines "leeren" Datagramms die aktuelle Systemzeit in Form einer Zeichenkette an den Sender zurückschickt. Programmieren Sie für den Test auch einen passenden Client (`DaytimeClient`), der maximal zwei Sekunden auf die Antwort des Servers wartet.

2. Ein Client (`MesswertClient`) sendet in Abständen von 5 Sekunden Zufallszahlen mit Zeitpunktangaben an den Server. Der Server (`MesswertServer`) gibt diese Zahlen zusammen mit der IP-Adresse des Senders am Bildschirm aus.

   Beispiel:

   ```
   192.168.2.90:3975 10:45:46 48.72014957156734
   192.168.2.90:3975 10:45:51 48.557565388620226
   192.168.2.90:3975 10:45:56 70.66274469656827
   ```

3. Ein Server (`QuoteServer`) liefert zufallsgesteuert Zitate. Alle Zitate befinden sich in einer Datei `quotes.txt`, die den folgenden Aufbau hat:

   ```
   Zitat (eine Zeile)
   Autor des Zitats (eine Zeile)
   Zitat (eine Zeile)
   Autor des Zitats (eine Zeile)
   ...
   ```

   Erhält der Server ein "leeres" Datagramm, so bestimmt eine Zufallszahl, welches Zitat (mit Autor) an den Client gesendet wird.

   Programmieren Sie eine Anwendung mit grafischer Oberfläche zur Anzeige von Zitaten in regelmäßigen Zeitabständen.

4. Erstellen Sie eine Variante des Programms `Talk` für eine Gruppenkommunikation mit Hilfe des Multicasting-Verfahrens. Aufrufparameter sollen sein: User, Multicast-Adresse, Portnummer.

3 Client/Server-Anwendungen mit TCP

Das wichtigste Protokoll der Transportschicht im TCP/IP-Schichtenmodell ist das *Transmission Control Protocol* (*TCP*). Es ist aufwändiger als UDP, stellt aber dafür eine verlässliche Verbindung zwischen Client und Server her. Viele bekannte Internet-Dienste wie *FTP* (*File Transfer Protocol*), *Telnet*, *SMTP* (*Simple Mail Transfer Protocol*), *POP* (*Post Office Protocol*) und *HTTP* (*Hypertext Transfer Protocol*) nutzen TCP.

3.1 Das Protokoll TCP

TCP ist im Gegensatz zu UDP ein verbindungsorientiertes Protokoll. Zwischen zwei Prozessen (Client und Server) wird eine virtuelle Verbindung hergestellt, über die in Form eines bidirektionalen Datenstroms Bytefolgen beliebiger Länge geschickt werden können.

Vor der Übertragung von Daten muss der Client eine Verbindung zum Server anfordern. Wird die Verbindung nicht mehr gebraucht, fordert einer der beteiligten Prozesse TCP auf, sie abzubauen.

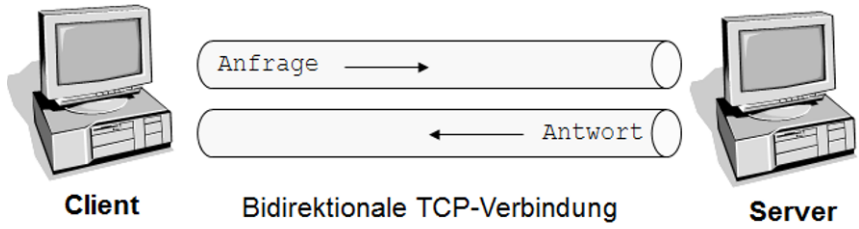

Abbildung 3-1: TCP-Verbindung

TCP sorgt für die Aufteilung der Daten in einzelne Pakete, garantiert, dass die Pakete den Empfänger in der richtigen Reihenfolge erreichen, und initiiert die Neuübertragung von verloren gegangenen oder defekten Paketen. TCP ist im *RFC* 793 der *IETF* spezifiziert.[1]

[1] `http://www.ietf.org/rfc/rfc793.txt`

D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
DOI 10.1007/978-3-658-09921-3_3, © Springer Fachmedien Wiesbaden 2015

Abbildung 3.2 zeigt den Aufbau eines TCP-Pakets, das Bestandteil der Nutzdaten des IPv4-Pakets ist.

Der TCP-Header enthält:

- die Portnummer des Senders (2 Byte),
- die Portnummer des Empfängers (2 Byte),
- weitere Felder mit einer Gesamtlänge von 16 Byte wie z. B. Reihenfolgenummer und Prüfsumme,
- optionale Felder variabler Länge.

Darauf folgen dann die eigentlichen Nutzdaten des TCP-Pakets.

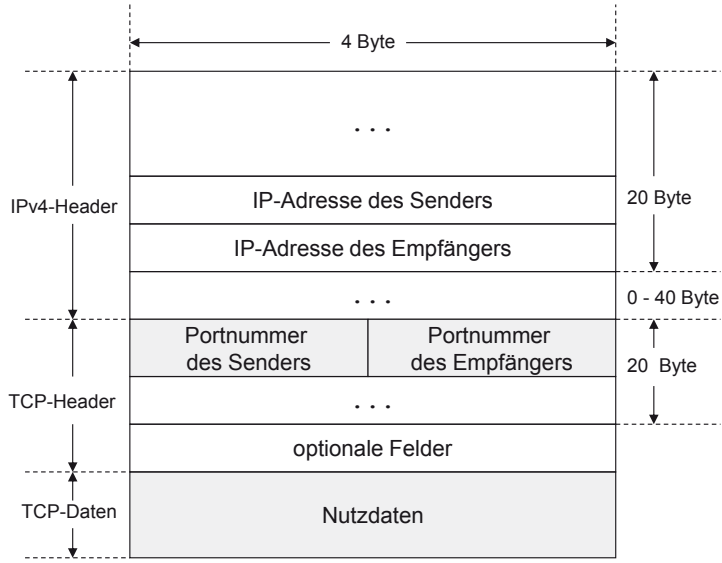

Abbildung 3-2: Aufbau des TCP-Pakets

3.2 TCP-Sockets

Im Gegensatz zu UDP wird bei TCP zwischen Client-Socket und Server-Socket unterschieden.

Der Server-Socket versetzt den Server in die Lage, die Verbindungsanforderungen der Clients abzuhören und dann mehrere Clients zu bedienen.

Bevor Daten zwischen Client und Server ausgetauscht werden können, muss eine Verbindung zwischen Sockets hergestellt werden. Der Aufruf der Methode `accept` des Server-Sockets blockiert den Server so lange, bis ein Client versucht, Kontakt

aufzunehmen. `accept` liefert als Ergebnis einen weiteren Socket, über den dann die eigentliche Kommunikation stattfindet.

Die Kontaktaufnahme auf der Client-Seite geschieht automatisch bei der Erzeugung des Client-Sockets mit einem geeigneten Konstruktor. Eine Verbindung kann natürlich nur hergestellt werden, wenn der Server zeitlich vor dem Start des Client `accept` aufgerufen hat. Über Ein- und Ausgabeströme können nun Daten empfangen bzw. gesendet werden.

Die Verbindung wird abgebaut, wenn einer der beiden Sockets geschlossen wird.

Die Klasse Socket

Die Klasse `java.net.Socket` implementiert einen Client-Socket.

`Socket(InetAddress address, int port) throws IOException`
: erzeugt einen Client-Socket und bindet ihn an die vorgegebene IP-Adresse und Portnummer.

`Socket(String host, int port) throws java.net.UnknownHostException, IOException`
: erzeugt einen Client-Socket und bindet ihn an den vorgegebenen Rechner und an die vorgegebene Portnummer.

`void close() throws IOException`
: schließt den Socket und auch die mit ihm verbundenen Ein- und Ausgabeströme (siehe weiter unten).

`boolean isClosed()`
: liefert true, falls der Socket geschlossen ist, sonst false.

`void setSoTimeout(int timeout) throws SocketException`
: setzt ein Timeout in Millisekunden. `read` für den Eingabestrom dieses Sockets blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert.

`int getSoTimeout() throws SocketException`
: liefert die Timeout-Angabe.

`InetAddress getLocalAddress()`
: liefert die lokale IP-Adresse, an die der Socket gebunden ist.

`int getLocalPort()`
: liefert die lokale Portnummer, an die der Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
: liefert die lokale Socket-Adresse, an die der Socket gebunden ist.

`InetAddress getInetAddress()`
: liefert die entfernte IP-Adresse, mit der der Socket verbunden ist.

`int getPort()`
: liefert die entfernte Portnummer, mit der der Socket verbunden ist.

`SocketAddress getRemoteSocketAddress()`
liefert die entfernte Socket-Adresse, mit der der Socket verbunden ist.

Input/Output

`InputStream getInputStream() throws IOException`
liefert einen Eingabestrom für diesen Socket.

`OutputStream getOutputStream() throws IOException`
liefert einen Ausgabestrom für diesen Socket.

Wird einer der beiden Datenströme mit der entsprechenden `close`-Methode geschlossen, so wird auch der zugehörige Socket geschlossen.

Mit einer der folgenden Methoden kann jeweils nur einer der beiden Datenströme (Eingabe bzw. Ausgabe) geschlossen werden, ohne damit den Socket selbst zu schließen:

`void shutdownInput() throws IOException`
setzt den Eingabestrom für diesen Socket auf EOF (End Of File).

`void shutdownOutput() throws IOException`
deaktiviert den Ausgabestrom für diesen Socket.

Die Klasse ServerSocket

Die Klasse `java.net.ServerSocket` implementiert einen Server-Socket.

`ServerSocket()`
erzeugt einen ungebundenen Server-Socket.

`ServerSocket(int port) throws IOException`
erzeugt einen Server-Socket und bindet ihn an die vorgegebene Portnummer.

`void bind(SocketAddress addr) throws IOException`
bindet den Server-Socket an eine Socket-Adresse (IP-Adresse und Portnummer).

`void close() throws IOException`
schließt den Server-Socket.

`boolean isClosed()`
liefert `true`, falls der Server-Socket geschlossen ist, sonst `false`.

`Socket accept() throws IOException`
nimmt einen Verbindungswunsch an und erzeugt einen neuen Socket für diese Verbindung. accept blockiert so lange, bis eine neue Verbindung aufgenommen wurde.

`void setSoTimeout(int timeout) throws SocketException`
setzt ein Timeout in Millisekunden. accept blockiert höchstens `timeout` Millisekunden und löst dann die Ausnahme `java.net.SocketTimeoutException` aus. Wird `timeout = 0` gesetzt, so wird die Timeout-Steuerung deaktiviert.

`int getSoTimeout() throws IOException`
liefert die Timeout-Angabe.

`InetAddress getInetAddress()`
liefert die IP-Adresse, an die der Server-Socket gebunden ist.

`int getLocalPort()`
liefert die Portnummer, an die der Server-Socket gebunden ist.

`SocketAddress getLocalSocketAddress()`
liefert die Socket-Adresse, an die der Server-Socket gebunden ist.

Verbindungswünsche, für die mittels `accept` noch kein Socket erzeugt werden konnte, werden in einer Warteschlange (*backlog*) verwaltet. Falls die Warteschlange voll ist und ein neuer Verbindungswunsch eingeht, wird beim Client die Ausnahme `java.net.ConnectException` ausgelöst. Die maximale Länge der Warteschlange ist standardmäßig 50.

Mit dem Konstruktor

```
ServerSocket(int port, int backlog) throws IOException
```

kann die maximale Länge der Warteschlange explizit festgelegt werden.

Der folgende Konstruktor bindet den Socket an die explizit vorgegeben IP-Adresse:

```
ServerSocket(int port, int backlog, InetAddress bindAddr)
       throws IOException
```

`void bind(SocketAddress addr, int backlog) throws IOException`
bindet den Server-Socket an eine Socket-Adresse (IP-Adresse und Portnummer).

Ablaufschritte:

1. Der Server erzeugt ein `ServerSocket`-Objekt, das an einen vorgegebenen Port gebunden wird.
2. Der Server ruft die Methode `accept` des Server-Sockets auf und wartet auf den Verbindungswunsch eines Client.
3. Der Client erzeugt ein `Socket`-Objekt mit der Adresse und der Portnummer des Servers und versucht damit, eine Verbindung zum Server aufzunehmen.
4. Der Server erzeugt ein `Socket`-Objekt, das das serverseitige Ende der Kommunikationsverbindung darstellt. Die Methode `accept` gibt dieses Objekt zurück.

5. Um eine bidirektionale Verbindung zwischen Client und Server aufzubauen, stellen Client und Server jeweils einen Aus- und Eingabestrom mit Hilfe ihrer `Socket`-Objekte bereit.
6. Nun können Daten mittels Lese- und Schreiboperationen hin- und hergeschickt werden.

Abbildung 3.3 zeigt den Ablauf beim Aufbau einer TCP-Verbindung.

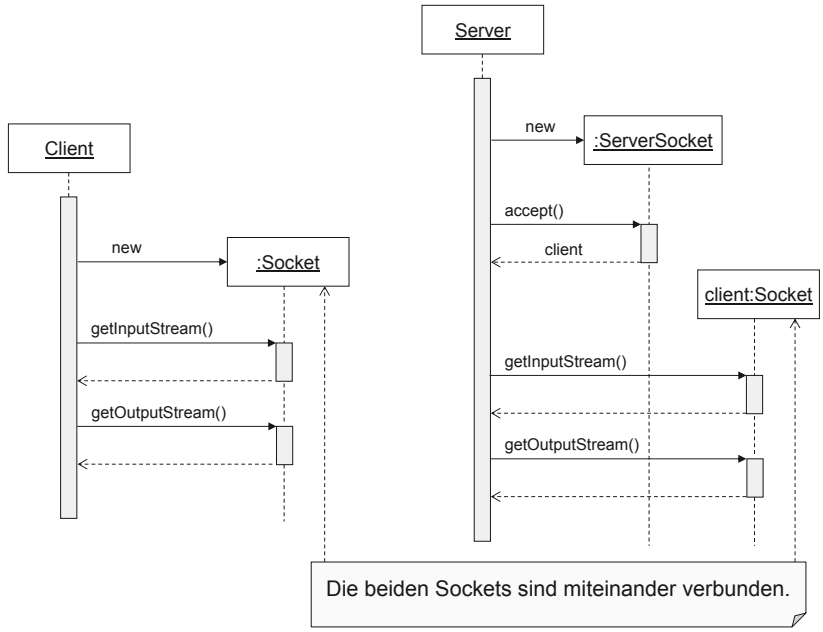

Abbildung 3-3: Aufbau einer TCP-Verbindung

3.3 Ein Echo-Server und -Client

Wir stellen nun eine verbindungsorientierte Version des Echo-Service vor.

Der Client schickt in einer Schleife jeweils eine Textzeile zum Server, der diese dann wieder an den Client zurückschickt. Die Eingabe erfolgt über Tastatur.

Hier ist zunächst der Client:

Die eigentliche Verarbeitung findet in einer `try`-Anweisung statt. Die Verbindung zum Server wird am Ende durch Schließen des Sockets (hier automatisch dank des *try-with-resources*-Konstrukts) abgebaut.

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class EchoClient {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);

      try (Socket socket = new Socket(host, port);
            BufferedReader in = new BufferedReader(new InputStreamReader(
                  socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                  true);
            BufferedReader input = new BufferedReader(
                  new InputStreamReader(System.in))) {

         String msg = in.readLine();
         System.out.println(msg);

         String line;
         while (true) {
            line = input.readLine();
            if (line == null || line.equals("q"))
               break;
            out.println(line);
            System.out.println(in.readLine());
         }
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Zunächst wird ein Client-Socket erzeugt und damit versucht, die Verbindung zum Server herzustellen. Das `BufferedReader`-Objekt `in` ist der Eingabestrom, das `PrintWriter`-Objekt `out` der Ausgabestrom für diesen Socket.

Dann wird eine Nachricht des Servers ausgegeben. In einer Schleife werden nun von der Tastatur Textzeilen eingelesen, in den Ausgabestrom `out` geschrieben und damit zum Server geschickt. `in.readLine()` blockiert so lange, bis die Antwort vom Server vorliegt. Diese wird dann angezeigt.

Wir zeigen zwei Server-Versionen.

Die *erste Version* kann nicht mehrere Clients gleichzeitig bedienen. Werden z. B. zwei Clients kurz hintereinander gestartet, so werden sie der Reihe nach bedient. Erst wenn der erste Client beendet wurde, kommt der zweite zum Zuge. Es handelt sich also um einen so genannten *iterativen Server*.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

public class EchoServer1 {
   private int port;
   private int backlog;

   public EchoServer1(int port, int backlog) {
      this.port = port;
      this.backlog = backlog;
   }

   public void startServer() {
      try (ServerSocket serverSocket = new ServerSocket(port, backlog)) {

         System.out.println("EchoServer1 auf "
               + serverSocket.getLocalSocketAddress() + " gestartet ...");

         process(serverSocket);
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private void process(ServerSocket server) throws IOException {
      while (true) {
         SocketAddress socketAddress = null;

         try (Socket socket = server.accept();
               BufferedReader in = new BufferedReader(
                     new InputStreamReader(socket.getInputStream()));
               PrintWriter out = new PrintWriter(socket.getOutputStream(),
                     true)) {

            socketAddress = socket.getRemoteSocketAddress();
            System.out.println("Verbindung zu " + socketAddress
                  + " aufgebaut");

            out.println("Server ist bereit ...");

            String input;
            while ((input = in.readLine()) != null) {
               out.println(input);
            }
         } catch (IOException e) {
            System.err.println(e);
         } finally {
            System.out.println("Verbindung zu " + socketAddress
                  + " abgebaut");
         }
      }
   }

   public static void main(String[] args) {
      int port = Integer.parseInt(args[0]);
      int backlog = 50;
```

```
      if (args.length == 2)
         backlog = Integer.parseInt(args[1]);

      new EchoServer1(port, backlog).startServer();
   }
}
```

Der Server-Socket wird erzeugt und an die vorgegebene Portnummer des lokalen Rechners gebunden. Hier wird auch die maximale Länge der Warteschlange (`backlog`) vorgegeben. Somit kann das Verhalten der Anwendung mit unterschiedlichen Werten getestet werden.

Je Schleifendurchgang in der Methode `process` erfolgen diese Schritte:

- Aufruf der Methode `accept`, die so lange blockiert, bis ein Client versucht, Verbindung aufzunehmen,
- Bereitstellung der Ein- und Ausgabeströme `in` und `out`,
- Sendung einer Nachricht an den Client.

In einer Schleife werden Textzeilen eingelesen und sofort in den Ausgabestrom geschrieben. Die Schleife läuft so lange, bis der Client die Verbindung und damit seinen Ausgabestrom geschlossen hat.

Aufruf des Servers:

```
java -cp bin EchoServer1 50000
```

Ausgabe des Servers:

```
EchoServer1 auf 0.0.0.0/0.0.0.0:50000 gestartet ...
```

Aufruf des Client (hier auf demselben Rechner):

```
java -cp bin EchoClient localhost 50000
```

Der Server meldet den Auf- und Abbau der Verbindung:

```
Verbindung zu /127.0.0.1:50275 aufgebaut
Verbindung zu /127.0.0.1:50275 abgebaut
```

Der Server kann über Tastatur mit `Strg + C` abgebrochen werden.

Die *zweite Version* des Echo-Servers ermöglicht die *parallele Bedienung* mehrerer Clients. Dazu erfolgt die Kommunikation mit einem Client innerhalb eines Threads.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;

public class EchoServer2 {
   private int port;

   public EchoServer2(int port) {
      this.port = port;
   }

   public void startServer() {
      try (ServerSocket serverSocket = new ServerSocket(port)) {

         System.out.println("EchoServer2 auf "
               + serverSocket.getLocalSocketAddress() + " gestartet ...");

         while (true) {
            Socket socket = serverSocket.accept();
            new EchoThread(socket).start();
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private class EchoThread extends Thread {
      private Socket socket;

      public EchoThread(Socket socket) {
         this.socket = socket;
      }

      public void run() {
         SocketAddress socketAddress = socket.getRemoteSocketAddress();
         System.out.println("Verbindung zu " + socketAddress + " aufgebaut");

         try (BufferedReader in = new BufferedReader(new InputStreamReader(
               socket.getInputStream()));
               PrintWriter out = new PrintWriter(socket.getOutputStream(),
                     true)) {

            out.println("Server ist bereit ...");

            String input;
            while ((input = in.readLine()) != null) {
               out.println(input);
            }
         } catch (IOException e) {
            System.err.println(e);
         } finally {
            System.out.println("Verbindung zu " + socketAddress
                  + " abgebaut");
         }
      }
   }
```

```
    public static void main(String[] args) {
       int port = Integer.parseInt(args[0]);
       new EchoServer2(port).startServer();
    }
}
```

Sobald die Methode `accept` einen Verbindungswunsch angenommen hat, wird mit Hilfe des gelieferten Sockets ein `EchoThread`-Objekt erzeugt und dieser Thread dann gestartet. Sofort ist der Server wieder bereit, eine neue Verbindung aufzunehmen. Pro Verbindung existiert also ein eigener Thread, der den entsprechenden Client bedient.

3.4 Thread-Pooling

Die Fähigkeit des Servers, mehrere Clients quasi gleichzeitig zu bedienen, wurde bisher so gelöst, dass für jede Anfrage eines Clients ein neuer Thread erzeugt wurde. Im Vergleich zu Prozessen ist die Erzeugung eines Threads weniger aufwändig. Trotzdem sollte man hiermit sparsam umgehen; insbesondere dann, wenn kurzzeitig sehr viele Threads mit kurzer Laufzeit benötigt werden.

Ab der Version Java SE 5 gibt es die Möglichkeit, einen *Thread-Pool* einzusetzen. Dieser bietet die Möglichkeit, mehrere separate Aufgaben vom selben Thread nacheinander ausführen zu lassen. Nicht mehr benutzte Threads werden in den Pool zurückgelegt und können wiederverwendet werden.

Wir benutzen hier eine spezielle Thread-Pool-Variante, einen so genannten *Cached Thread Pool*. Ein solcher Pool wächst bzw. schrumpft nach Bedarf. Steht eine neue Aufgabe (hier eine Client-Anfrage) zur Bearbeitung an und gibt es keinen "freien" Thread im Pool, so wird ein neuer erzeugt, der die Ausführung übernimmt. Ist die Aufgabe ausgeführt, steht dieser Thread als wieder "freier" Thread im Pool zur Verfügung. Wird er *innerhalb von 60 Sekunden* nicht benötigt, so wird er terminiert und ist dann nicht mehr verwendbar. Der Pool passt sich also dynamisch den momentanen Anforderungen an und ist optimal für kleinere Aufgaben, die in hoher Zahl kurzfristig anstehen.

Für unsere Zwecke benutzen wir die Klasse `java.util.concurrent.Executors` und das Interface `java.util.concurrent.ExecutorService.`

Die statische `Executors`-Methode

```
static ExecutorService newCachedThreadPool()
```

erzeugt einen *Cached Thread Pool* und liefert diesen als Objekt vom Typ `ExecutorService` zurück.

Das Interface `ExecutorService` enthält u. a. die folgenden Methoden:

`void execute(Runnable task)`
führt die `run`-Methode von `task` in einem Thread des Pools aus.

`void shutdown()`
bewirkt, dass vor dem Aufruf dieser Methode übergebene Aufgaben noch ausgeführt, neue aber nicht mehr akzeptiert werden; alle vom Pool verwalteten Threads werden dann terminiert.

Die folgenden Programme demonstrieren den Einsatz eines Thread-Pools.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Server {
   private int port;
   private ExecutorService pool;

   public Server(int port) {
      this.port = port;
   }

   public void startServer() {
      try (ServerSocket server = new ServerSocket(port)) {

         System.out.println("Server gestartet ...");

         pool = Executors.newCachedThreadPool();

         while (true) {
            Socket socket = server.accept();
            pool.execute(new Task(socket));
         }
      } catch (IOException e) {
         System.err.println(e);
         pool.shutdown();
      }
   }

   private class Task implements Runnable {
      private Socket socket;

      public Task(Socket socket) {
         this.socket = socket;
      }

      public void run() {
         try (BufferedReader in = new BufferedReader(new InputStreamReader(
               socket.getInputStream()));
```

```
                PrintWriter out = new PrintWriter(socket.getOutputStream(),
                      true)) {

             String input = in.readLine();

             if (input != null) {
                int delay = 3000 + (int) (Math.random() * 7000);

                long begin = System.currentTimeMillis();
                System.out.printf("Beginn Auftrag %3s: %tT, Thread; %s%n", input,
                      new Date(begin), Thread.currentThread().getName());
                Thread.sleep(delay);
                long end = System.currentTimeMillis();
                System.out.printf("Ende   Auftrag %3s: %tT, Thread; %s%n", input,
                      new Date(end), Thread.currentThread().getName());

                out.printf("Auftrag %3s, Beginn: %tT, Ende: %tT%n", input,
                      new Date(begin), new Date(end));
             }
          } catch (Exception e) {
             System.err.println(e);
          }
       }
    }

    public static void main(String[] args) {
       int port = Integer.parseInt(args[0]);
       new Server(port).startServer();
    }
}
```

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Client {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      int id = 1;

      for (int i = 0; i < 20; i++) {
         new Handler(host, port, id++).start();
         try {
            Thread.sleep(4000);
         } catch (InterruptedException e) {
         }
      }
   }

   private static class Handler extends Thread {
      private String host;
      private int port;
      private int id;

      public Handler(String host, int port, int id) {
         this.host = host;
```

```
            this.port = port;
            this.id = id;
        }

        public void run() {
            try (Socket socket = new Socket(host, port);
                  BufferedReader in = new BufferedReader(
                        new InputStreamReader(socket.getInputStream()));
                  PrintWriter out = new PrintWriter(socket.getOutputStream(),
                        true)) {

               out.println(id);

               String response = in.readLine();
               System.out.println(response);
            } catch (Exception e) {
               System.err.println(e);
            }
        }
    }
}
```

3.5 Ein Framework für TCP-Server

Die Beispiele der vorhergehenden Abschnitte zeigen, dass die Implementierungen der TCP-Server im Großen und Ganzen fast immer dem gleichen Muster folgen. Es liegt also nahe, die immer wiederkehrenden Codeteile zu standardisieren und als Framework für eigene Server-Implementierungen anzubieten.

Das Framework

Das Framework besteht aus den beiden Klassen `TCPServer` und `AbstractHandler`, die zum Paket `tcpframework` gehören.

Der Konstruktor von `TCPServer` erwartet eine Portnummer und die Klasse des Handlers, der die eigentliche Kommunikation mit dem Client durchführt. Hier wird das `Class`-Objekt des Handlers angegeben. Anstelle der Handler-Klasse kann auch eine Handler-Instanz angegeben werden.

Der Handler ist eine Subklasse von `AbstractHandler`. Innerhalb des Konstruktors werden u. a. ein `ServerSocket`-Objekt und ein Thread-Pool (siehe Kapitel 3.4) erzeugt.

`TCPServer` ist von `Thread` abgeleitet. Innerhalb der `run`-Methode wird in einer Schleife die Methode `accept` aufgerufen. Für die im Konstruktor erzeugte bzw. übergebene Handler-Instanz wird die Methode `handle` mit den Referenzen für das `Socket`-Objekt und den Thread-Pool aufgerufen.

Die Methode `stopServer` schließt das `ServerSocket`-Objekt und terminiert den Thread-Pool.

```
package tcpframework;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TCPServer extends Thread {
   private AbstractHandler handler;
   private ServerSocket serverSocket;
   private ExecutorService pool;

   public TCPServer(int port, Class<?> handlerClass) throws Exception {
      handler = (AbstractHandler) handlerClass.newInstance();
      serverSocket = new ServerSocket(port);
      pool = Executors.newCachedThreadPool();
   }

   public TCPServer(int port, Object handlerObject) throws IOException {
      handler = (AbstractHandler) handlerObject;
      serverSocket = new ServerSocket(port);
      pool = Executors.newCachedThreadPool();
   }

   public void run() {
      try {
         while (true) {
            Socket socket = serverSocket.accept();
            handler.handle(socket, pool);
         }
      } catch (SocketException e) {
         // Beim Aufruf von stopServer() wird eine SocketException ausgeloest
      } catch (Exception e) {
         System.err.println(e);
      }
   }

   public void stopServer() {
      try {
         serverSocket.close();
      } catch (IOException e) {
      }

      pool.shutdown();
   }
}
```

Ein konkreter Handler ist von `AbstractHandler` abgeleitet und implementiert die Methode `runTask`.

```
package tcpframework;

import java.io.IOException;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.concurrent.ExecutorService;
```

```
public abstract class AbstractHandler {
   public void handle(final Socket socket, ExecutorService pool) {
      pool.execute(new Runnable() {
         @Override
         public void run() {
            SocketAddress socketAddress = socket.getRemoteSocketAddress();
            System.out.println("Verbindung zu " + socketAddress + " aufgebaut");

            runTask(socket);

            try {
               if (socket != null)
                  socket.close();
            } catch (IOException e) {
            }

            System.out.println("Verbindung zu " + socketAddress + " abgebaut");
         }
      });
   }

   public abstract void runTask(Socket socket);
}
```

Anwendung des Frameworks

Das Framework wird nun in einem Beispiel verwendet. Hierbei handelt es sich um den Echo-Server aus Kapitel 3.3.

```
import tcpframework.TCPServer;

public class EchoServer {
   public static void main(String[] args) throws Exception {
      int port = Integer.parseInt(args[0]);

      TCPServer server = new TCPServer(port, EchoHandler.class);

      server.start();
      System.out.println("EchoServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("EchoServer wird gestoppt.");
   }
}
```

Der Server wird nach Betätigung der ENTER-Taste gestoppt; allerdings erst dann, wenn alle momentan aktiven Client-Bearbeitungen beendet sind.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

import tcpframework.AbstractHandler;

public class EchoHandler extends AbstractHandler {
   public void runTask(Socket socket) {
      try (BufferedReader in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                  true)) {

         out.println("Server ist bereit ...");

         String input;
         while ((input = in.readLine()) != null) {
            out.println(input);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

3.6 Beenden eines Datenstroms ohne Verbindungsabbau

Die `Socket`-Methode `close` schließt nicht nur den Socket, sondern auch gleichzeitig die mit ihm verbundenen Datenströme. Umgekehrt führt das Schließen eines der beiden Datenströme zum Schließen des Sockets.

Will man nur einen der Datenströme beenden, ohne auch den Socket zu schließen, kann die `Socket`-Methode `shutdownInput` bzw. `shutdownOutput` verwendet werden (siehe Kapitel 3.2).

Wir nutzen diese Möglichkeit im folgenden Programm. Der `UploadClient` sendet den Inhalt einer Datei zum Server und beendet dann den Ausgabestrom. Der `UploadServer` liest Bytes aus dem Eingabestrom, bis dieser beendet ist. Danach schickt er eine Meldung zum Client.

Würde der Ausgabestrom vom Client nicht durch `shutdownOutput` beendet, so würde der Server in Erwartung weiterer Daten in der Schleife zum Einlesen verbleiben und so das Programm nicht zum Ende kommen.

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.Socket;
```

```
public class UploadClient {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      String file = args[2];

      try (Socket socket = new Socket(host, port);
            BufferedReader in = new BufferedReader(new InputStreamReader(
                  socket.getInputStream()));
            OutputStream out = socket.getOutputStream()) {

         try (InputStream is = new FileInputStream(file)) {
            byte[] buffer = new byte[8192];
            int c;
            while ((c = is.read(buffer)) != -1) {
               out.write(buffer, 0, c);
            }
         }

         socket.shutdownOutput();

         String msg = in.readLine();
         System.out.println(msg);
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Für den Server nutzen wir das Framework aus Kapitel 3.5. Die hochgeladene Datei wird unter einem eindeutigen Namen (mittels `UUID.randomUUID()` erzeugt) gespeichert.

```
import tcpframework.TCPServer;

public class UploadServer {
   public static String DIR;

   public static void main(String[] args) throws Exception {
      int port = Integer.parseInt(args[0]);
      DIR = args[1];

      TCPServer server = new TCPServer(port, UploadHandler.class);

      server.start();
      System.out.println("UploadServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("UploadServer wird gestoppt.");
   }
}
```

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.UUID;

import tcpframework.AbstractHandler;

public class UploadHandler extends AbstractHandler {
   public void runTask(Socket socket) {
      try (InputStream in = socket.getInputStream();
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                  true)) {

         String message;
         String filename = UUID.randomUUID().toString();
         File file = new File(UploadServer.DIR, filename);
         try (OutputStream os = new FileOutputStream(file)) {
            byte[] buffer = new byte[8192];
            int c;
            while ((c = in.read(buffer)) != -1) {
               os.write(buffer, 0, c);
            }

            message = "Die Datei wurde unter dem Namen '" + filename
                  + "' gespeichert.";
         } catch (IOException e) {
            System.err.println(e);
            message = e.getMessage();
         }

         out.println(message);
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

3.7 Ein Proxy zum Aufruf entfernter Methoden

In diesem Abschnitt entwickeln wir ein allgemeingültiges Verfahren zum Aufruf von Methoden, die auf einem entfernten Rechner implementiert sind (*RPC = Remote Procedure Call*). Hierzu nutzen wir die in den vorhergehenden Abschnitten dargestellten Klassen und Methoden zur TCP/IP-Kommunikation, werden aber die Methodenaufrufe auf der Client-Seite von den konkreten Details zur Adressierung und Netzwerkkommunikation durch einen so genannten *Proxy* (lokalen Stellvertreter für die im Server implementierten Methoden) abschirmen.

Wir nutzen auch hier wieder das Framework für den TCP-Server aus Kapitel 3.5.

Die Methoden, die der Client aufrufen können soll, werden in einem Interface definiert. Einzige Voraussetzung für diese Methoden ist:

- Alle Parameterwerte müssen Objekte sein,
- die Klassen der Parameterwerte und der Rückgabewert müssen das Interface `java.io.Serializable` implementieren.

```
package service;

import java.io.IOException;
import java.util.Date;
import java.util.Vector;

public interface DemoService {
   String getEcho(String text);
   int getSumme(Integer x, Integer y);
   Date getDate();
   void sendMessage(String msg);
   Vector<String> getMessages(String file) throws IOException;
}
```

Die Klasse `DemoServiceImpl` implementiert dieses Interface. `DemoServiceImpl` wird nur auf der Server-Seite benötigt. Die Klasse muss den parameterlosen Konstruktor zur Verfügung stellen (hier implizit).

```
package service;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Date;
import java.util.Vector;

public class DemoServiceImpl implements DemoService {
   public String getEcho(String text) {
      return text;
   }

   public int getSumme(Integer x, Integer y) {
      return x + y;
   }

   public Date getDate() {
      return new Date();
   }

   public void sendMessage(String msg) {
      System.out.println(msg);
   }

   public Vector<String> getMessages(String file) throws IOException {
      Vector<String> lines = new Vector<String>();
      try (BufferedReader in = new BufferedReader(new FileReader(file))) {
         String line;
```

```
            while ((line = in.readLine()) != null) {
               lines.add(line);
            }
            return lines;
         }
      }
   }
```

Wir lernen zunächst einen "generischen" Aufruf für eine entfernte Methode kennen:

```
Object call(String name, Object[] params)
```

Diese Methode ist in der Klasse `RPCClient` implementiert. `name` ist der Methodenname, `params` enthält die Parameterwerte. `params` darf auch `null` sein, d. h. die Methode hat keine Parameter.

Ein Beispiel:

Lautet der Kopf der entfernten Methode

```
public int getSumme(Integer x, Integer y),
```

so sieht der Codeabschnitt des Clients für den Aufruf beispielsweise so aus:

```
Object[] params = {10, 33};
int summe = (Integer) rpcClient.call("getSumme", params);
```

Hierbei referenziert `rpcClient eine Instanz der Klasse RPCClient.`

Dieser Aufruf funktioniert zwar auch dann, wenn `DemoServiceImpl` kein entsprechendes Interface implementiert. Da eine später vorzustellende Ergänzung (dynamischer Proxy) dieses Interface jedoch benötigt, erfolgt bereits jetzt seine Einführung.

Die Klasse `RPCClient` nutzt die Methoden `writeObject` und `readObject` der Klassen `ObjectOutputStream` bzw. `ObjectInputStream`, um den Methodennamen, das Parameter-Array und den Rückgabewert über das Netz zu transferieren. Bei einem Rückgabewert vom Typ `Exception` wird eine Ausnahme dieses Typs ausgelöst.

```
package rpc;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class RPCClient {
   private String host;
   private int port;

   public RPCClient(String host, int port) {
```

```
      this.host = host;
      this.port = port;
   }

   public Object call(String name, Object[] params) throws Exception {
      try (Socket socket = new Socket(host, port);
            ObjectOutputStream out = new ObjectOutputStream(
                  socket.getOutputStream())) {

         out.writeObject(name);
         out.writeObject(params);
         out.flush();

         ObjectInputStream in = new ObjectInputStream(
               socket.getInputStream());
         Object ret = in.readObject();

         if (ret instanceof Exception)
            throw (Exception) ret;

         return ret;
      }
   }
}
```

Die Klasse `RPCServer` lädt diejenige Klasse (im Beispiel: `service.DemoServiceImpl`), die die Implementierung der entfernten Methoden enthält, und erzeugt eine Instanz dieser Klasse.

```
package rpc;

import tcpframework.TCPServer;

public class RPCServer {
   public static void main(String[] args) throws Exception {
      int port = Integer.parseInt(args[0]);
      String service = args[1];

      Class<?> serviceClass = Class.forName(service);
      Object serviceObject = serviceClass.newInstance();
      RPCHandler handlerObject = new RPCHandler(serviceObject);

      TCPServer server = new TCPServer(port, handlerObject);

      server.start();
      System.out.println("RPCServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("RPCServer wird gestoppt.");
   }
}
```

Die eigentliche Arbeit leistet der `RPCHandler`.

```
package rpc;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.net.Socket;

import tcpframework.AbstractHandler;

public class RPCHandler extends AbstractHandler {
   private Object serviceObject;

   public RPCHandler(Object serviceObject) {
      this.serviceObject = serviceObject;
   }

   public void runTask(Socket socket) {
      try (ObjectInputStream in = new ObjectInputStream(
            socket.getInputStream());
            ObjectOutputStream out = new ObjectOutputStream(
                  socket.getOutputStream())) {

         String name = (String) in.readObject();
         Object[] params = (Object[]) in.readObject();

         Object ret = null;
         try {
            Class<?>[] types = null;

            if (params != null) {
               types = new Class[params.length];
               for (int i = 0; i < params.length; i++)
                  types[i] = params[i].getClass();
            }

            Method m = serviceObject.getClass().getMethod(name, types);
            ret = m.invoke(serviceObject, params);
         }

         // Eine Ausnahme wird als Ergebnisobjekt zurückgeliefert
         catch (InvocationTargetException e) {
            ret = e.getTargetException();
         } catch (Exception e) {
            ret = e;
         }

         out.writeObject(ret);
         out.flush();
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Anhand des Methodennamens und der Parameterwerte, die vom Client als Objekte übermittelt wurden, wird mit Hilfe der `Class`-Methode `getMethod` die gewünschte Methode des Service als `Method`-Objekt zur Verfügung gestellt.

Anschließend wird mit `invoke` die Service-Methode aufgerufen. Wenn die Service-Methode eine Ausnahme auslöst, so löst `invoke` die Ausnahme

```
java.lang.reflect.InvocationTargetException
```

aus.

Die `InvocationTargetException`-Methode

```
Throwable getTargetException()
```

gibt die von der aufgerufenen Service-Methode ausgelöste Ausnahme zurück.

Ausnahmen (Typ `Exception`), die beim Suchen bzw. bei der Ausführung der Service-Methode auftreten können, werden als "normales" Ergebnisobjekt des Aufrufs zurücktransportiert.

Der `RPCServer` kann wie folgt gestartet werden:

```
java -cp bin;../framework/bin rpc.RPCServer 50000 service.DemoServiceImpl
```

Der Klassenpfad bezieht auch die Framework-Klassen aus Kapitel 3.5 mit ein.

`TestClient1` enthält beispielhafte Aufrufe der entfernten Methoden.

```
import java.util.Date;
import java.util.Vector;

import rpc.RPCClient;

public class TestClient1 {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);

      RPCClient rpcClient = new RPCClient(host, port);

      try {
         Object[] params = { "Hallo" };
         String s = (String) rpcClient.call("getEcho", params);
         System.out.println("getEcho: " + s);
      } catch (Exception e) {
         System.err.println(e);
      }

      try {
         Object[] params = { 10, 33 };
         int summe = (Integer) rpcClient.call("getSumme", params);
         System.out.println("getSumme: " + summe);
      } catch (Exception e) {
         System.err.println(e);
      }

      try {
         Date date = (Date) rpcClient.call("getDate", null);
         System.out.println("getDate: " + date);
      } catch (Exception e) {
```

```
            System.err.println(e);
         }

         try {
            Object[] params = { "Dies ist ein Test." };
            rpcClient.call("sendMessage", params);
         } catch (Exception e) {
            System.err.println(e);
         }

         try {
            Object[] params = { "msg.txt" };
            @SuppressWarnings("unchecked")
            Vector<String> v = (Vector<String>) rpcClient.call("getMessages",
                  params);
            System.out.println("getMessages:");
            for (String msg : v) {
               System.out.println(msg);
            }
         } catch (Exception e) {
            System.err.println(e);
         }
      }
   }
```

Aufruf des Client:

```
java -cp bin TestClient1 localhost 50000
```

Ausgabe des Client (Beispiel):

```
getEcho: Hallo
getSumme: 43
getDate: Tue Feb 24 14:13:55 CET 2015
getMessages:
Das ist ein Test.
```

Ausgabe des Servers (Beispiel):

```
Verbindung zu /127.0.0.1:50488 aufgebaut
Verbindung zu /127.0.0.1:50488 abgebaut
Verbindung zu /127.0.0.1:50490 aufgebaut
Verbindung zu /127.0.0.1:50490 abgebaut
Verbindung zu /127.0.0.1:50492 aufgebaut
Verbindung zu /127.0.0.1:50492 abgebaut
Verbindung zu /127.0.0.1:50494 aufgebaut
Dies ist ein Test.
Verbindung zu /127.0.0.1:50494 abgebaut
Verbindung zu /127.0.0.1:50496 aufgebaut
Verbindung zu /127.0.0.1:50496 abgebaut
```

Damit die entfernten Methoden wie "normale" lokale Methoden aufgerufen werden können, wird ein *dynamischer Proxy* zur Laufzeit generiert.

Die für den Client nötigen Anweisungen sehen dann wie folgt aus:

```
RPCClient rpcClient = new RPCClient(host, port);
ClientFactory factory = new ClientFactory(rpcClient);
DemoService demo = (DemoService) factory.newInstance(DemoService.class);

// Beispiel
String s = demo.getEcho("Hallo");
```

Die `ClientFactory`-Methode `newInstance` liefert einen Proxy, der das Interface `DemoService` implementiert.

```
package rpc;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class ClientFactory {
   private RPCClient client;

   public ClientFactory(RPCClient client) {
      this.client = client;
   }

   public Object newInstance(Class<?> c) {
      ClassLoader loader = client.getClass().getClassLoader();
      Class<?>[] interfaces = new Class<?>[] { c };
      Handler handler = new Handler(client);
      return Proxy.newProxyInstance(loader, interfaces, handler);
   }

   private class Handler implements InvocationHandler {
      private Object object;

      public Handler(Object object) {
         this.object = object;
      }

      public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
         Object result = ((RPCClient) this.object).call(method.getName(), args);
         return result;
      }
   }
}
```

Die Klasse `java.lang.reflect.Proxy` besitzt u. a. die Methode

```
public static Object newProxyInstance(
       ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
       throws IllegalArgumentException
```

Sie liefert eine Instanz für die vorgegebenen Interfaces `interfaces`. Diese leitet die Methodenaufrufe an den `InvocationHandler h` weiter. `loader` ist für die Definition und das Laden der Proxy-Klasse zuständig.

Das Interface `java.lang.reflect.InvocationHandler` gibt die zu implementierende Methode vor:

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

`invoke` verarbeitet den Methodenaufruf und liefert das Ergebnis dieses Aufrufs zurück. `proxy` ist die Proxy-Instanz, für die die Methode aufgerufen wurde. `method` ist die `Method`-Instanz passend zur Interface-Methode, die für den Proxy aufgerufen wurde. `args` enthält die Argumente des Methodenaufrufs oder hat den Wert `null`, falls die zugehörige Interface-Methode keine Parameter hat.

Wie die Implementierung von `invoke` in der Klasse `ClientFactory` zeigt, wird der Aufruf der Interface-Methode auf den `call`-Aufruf der Klasse `RPCClient` zurückgeführt.

```
import java.io.IOException;
import java.util.Date;
import java.util.Vector;

import rpc.ClientFactory;
import rpc.RPCClient;
import service.DemoService;

public class TestClient2 {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);

      RPCClient rpcClient = new RPCClient(host, port);
      ClientFactory factory = new ClientFactory(rpcClient);
      DemoService demo = (DemoService) factory.newInstance(DemoService.class);

      String s = demo.getEcho("Hallo");
      System.out.println("getEcho: " + s);

      int summe = demo.getSumme(10, 33);
      System.out.println("getSumme: " + summe);

      Date date = demo.getDate();
      System.out.println("getDate: " + date);

      demo.sendMessage("Dies ist ein Test.");

      try {
         Vector<String> v = demo.getMessages("msg.txt");
         System.out.println("getMessages:");
         for (String msg : v) {
            System.out.println(msg);
         }
      } catch (IOException e) {
         System.err.println(e);
      }
   }
}
```

3.8 Ein Chat-Programm

Mit dem Chat-Client ist das so genannte "Chatten" mit mehreren Teilnehmern im Netz möglich. Der Teilnehmer kann sich an- und abmelden und Texte durch Eingabe von ENTER im Eingabefeld an alle anderen aktiven Teilnehmer senden. Der Chat-Server registriert die angemeldeten Teilnehmer und verteilt eingehende Nachrichten an alle registrierten Teilnehmer.

```
import tcpframework.TCPServer;

public class ChatServer {
   public static void main(String[] args) throws Exception {
      int port = Integer.parseInt(args[0]);

      TCPServer server = new TCPServer(port, ChatHandler.class);

      server.start();
      System.out.println("ChatServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("ChatServer wird gestoppt.");
   }
}
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

import tcpframework.AbstractHandler;

public class ChatHandler extends AbstractHandler {
   private final static int TIMEOUT = 600000; // 10 Min.
   private List<PrintWriter> manager = new CopyOnWriteArrayList<PrintWriter>();

   public void runTask(Socket socket) {
      String name = null;
      BufferedReader in = null;
      PrintWriter out = null;

      try {
         socket.setSoTimeout(TIMEOUT);
         in = new BufferedReader(new InputStreamReader(
               socket.getInputStream()));
         out = new PrintWriter(socket.getOutputStream(), true);

         manager.add(out);
         name = in.readLine();
         sendMessage(name + " ist angemeldet.");
```

```
            String message;
            while ((message = in.readLine()) != null) {
               sendMessage(name + ": " + message);
            }
         } catch (IOException e) {
            System.err.println(e);
         } finally {
            manager.remove(out);
            sendMessage(name + " ist abgemeldet.");
         }
      }

      private void sendMessage(String message) {
         for (PrintWriter out : manager) {
            synchronized (out) {
               out.println(message);
            }
         }
      }
   }
```

Das Programmgerüst entspricht den in den letzten Abschnitten behandelten Beispielen. Der Server verwaltet eine Liste vom Typ `CopyOnWriteArrayList`. Mit Hilfe dieses Objekts "merkt" sich der Server, wer sich als Teilnehmer angemeldet hat. Nach Aufnahme der Verbindung wird die Referenz `out` auf den Ausgabestrom für diesen Teilnehmer in der Liste gespeichert und am Ende wieder entfernt. `CopyOnWriteArrayList` verhindert Ausnahmen vom Typ `Concurrent-ModificationException` während der Iteration über die Elemente der Liste.

Die erste Textzeile, die der Server über `in` erhält, ist der Login-Name des Teilnehmers. Alle anderen empfangenen Zeilen sind Nachrichten des Teilnehmers an alle aktiven Teilnehmer. Alle Nachrichten werden mit dem Teilnehmernamen versehen und mit der Methode `sendMessage` an alle angemeldeten Teilnehmer gesendet (siehe Abbildung 3.4). Mit derselben Methode werden die Teilnehmer darüber informiert, wer sich an- oder abgemeldet hat.

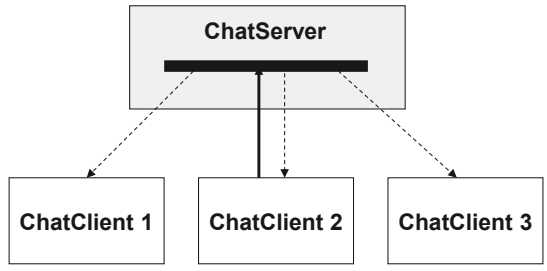

Abbildung 3-4: Der Chat-Server als Reflektor

```
import java.awt.BorderLayout;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.Socket;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

@SuppressWarnings("serial")
public class ChatClient extends JFrame implements Runnable, ActionListener {
   private final static int width = 500;
   private final static int hight = 400;

   private JLabel label;
   private JTextArea area;
   private JTextField text;
   private JButton button;

   private String host;
   private int port;
   private Socket socket;
   private BufferedReader in;
   private BufferedWriter out;

   private volatile Thread t;
   private boolean login = false;
   private String name;

   public ChatClient(String host, int port) {
      super("Chat-Client");

      this.host = host;
      this.port = port;

      label = new JLabel(" ");
      JPanel top = new JPanel();
      top.add(label);

      area = new JTextArea();
      area.setFont(new Font("Monospaced", Font.PLAIN, 14));
      area.setLineWrap(true);
      area.setEditable(false);

      text = new JTextField(48);
      text.setFont(new Font("Monospaced", Font.PLAIN, 14));
      text.addActionListener(this);
```

```
      button = new JButton("Login");
      button.addActionListener(this);

      JPanel input = new JPanel();
      input.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));
      input.add(text);
      input.add(button);

      add(top, BorderLayout.NORTH);
      add(new JScrollPane(area), BorderLayout.CENTER);
      add(input, BorderLayout.SOUTH);

      setSize(width, hight);
      setVisible(true);
      text.requestFocus();
   }

   public void actionPerformed(ActionEvent e) {
      Object obj = e.getSource();
      String cmd = e.getActionCommand();

      try {
         if (obj == button) {
            if (cmd.equals("Login")) {
               name = text.getText();
               if (name.length() != 0) {
                  login();
               }
            } else {
               logout();
            }
         }

         if (obj == text) {
            if (login) {
               out.write(text.getText());
               out.newLine();
               out.flush();
            }
         }
      } catch (IOException ex) {
         area.append(ex.getMessage() + "\n");
         logout();
      } finally {
         text.setText("");
         text.requestFocus();
      }
   }

   private void login() throws IOException {
      socket = new Socket(host, port);
      in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      out = new BufferedWriter(new OutputStreamWriter(
            socket.getOutputStream()));

      out.write(name);
      out.newLine();
      out.flush();

      login = true;
```

```
      label.setText(name);
      button.setText("Logout");

      t = new Thread(this);
      t.start();
   }

   public void logout() {
      if (login) {
         try {
            login = false;
            label.setText(" ");
            button.setText("Login");
            t = null;
            if (socket != null)
               socket.close();
            if (in != null)
               in.close();
            if (out != null)
               out.close();
         } catch (IOException e) {
         }
      }
   }

   public void run() {
      try {
         while (Thread.currentThread() == t) {
            final String msg = in.readLine();
            if (msg == null)
               break;

            try {
               EventQueue.invokeLater(new Runnable() {
                  public void run() {
                     area.append(msg + "\n");
                  }
               });
            } catch (Exception e) {
            }
         }
      } catch (IOException e) {
      }
   }

   static public void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);

      final ChatClient client = new ChatClient(host, port);

      client.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            client.logout();
            System.exit(0);
         }
      });
   }
}
```

Das Einlesen der Nachrichten vom Server erfolgt in einem eigenen Thread (siehe `run`-Methode), der in der Methode `login` gestartet wird.

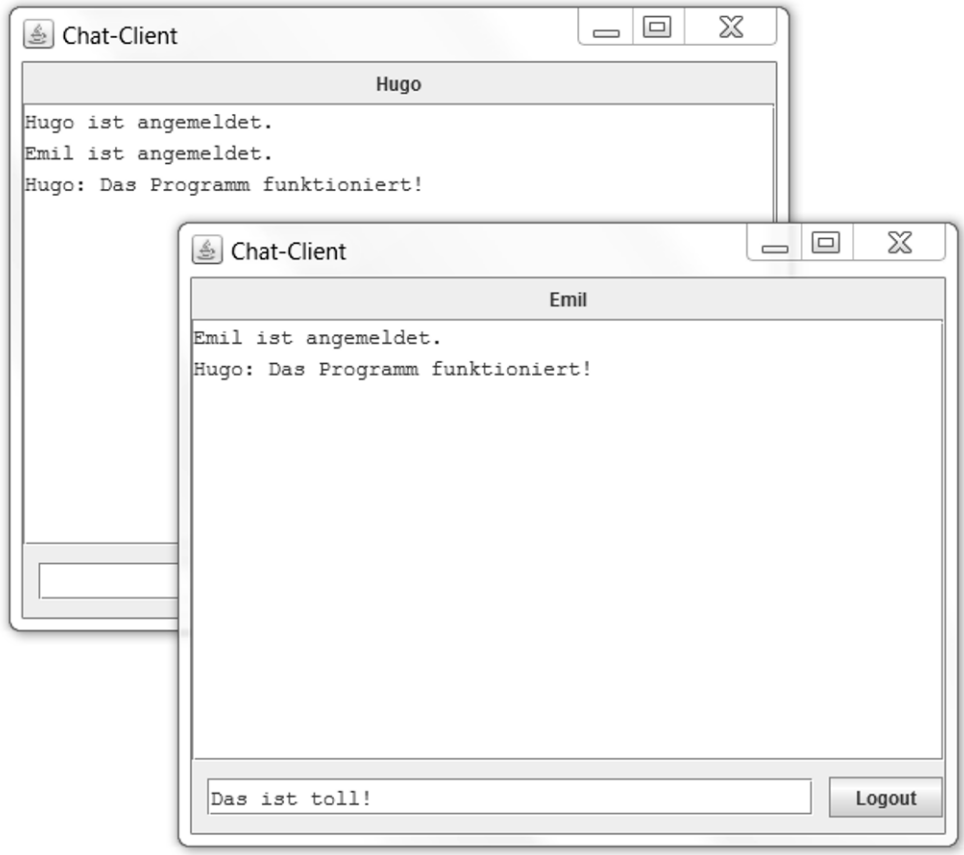

Abbildung 3-5: Der Chat-Client

Start des Servers:

```
java -cp bin;../framework/bin ChatServer 50000
```

Der Klassenpfad bezieht auch die Framework-Klassen aus Kapitel 3.5 mit ein.

Aufruf des Client:

```
java -cp bin ChatClient localhost 50000
```

bzw.

```
java -cp bin ChatClient <IP-Adresse des Servers> 50000
```

3.9 Klassen über das Netz laden

In Java müssen Klassen erst dann physisch vorhanden sein, wenn sie verwendet werden sollen (dynamisches Laden von Klassen). Ein so genannter *Klassenlader* ist verantwortlich für das Auffinden und Laden von Klassen.

Die Klasse `java.lang.ClassLoader` ist eine abstrakte Klasse. Sie enthält die Methode `loadClass`, die die Klasse mit dem angegebenen Namen lädt:

```
public Class<?> loadClass(String name) throws ClassNotFoundException
```

Ein eigener Klassenlader (als Subklasse von `ClassLoader`) kann z. B. eine Klasse von einem Server im Netz laden.

Hierzu muss nur die `ClassLoader`-Methode

```
protected Class<?> findClass(String name) throws ClassNotFoundException
```

in geeigneter Weise überschrieben werden.

Im Folgenden entwickeln wir einen Server (`ClassServer`), der auf Anfrage eine Klasse in seinem Serververzeichnis sucht und den Inhalt als Byte-Array liefert.

Der Client (`NetworkClassLoader`) überschreibt als Subklasse von `ClassLoader` die Methode `findClass`. Dazu nutzt er die `ClassLoader`-Methode `defineClass`, um aus dem vom Server gelieferten Byte-Array ein `Class`-Objekt zu erzeugen:

```
protected final Class<?> defineClass(
    String name, byte[] b, int off, int len)
```

Beim Einsatz eines Programms mit eigenem Klassenlader ist generell zu beachten:

Es wird versucht, weitere in der soeben geladenen Klasse referenzierte Klassen mit demselben Klassenlader zu laden, mit dem auch die erste Klasse geladen wurde.

Befinden sich das aufgerufene Programm (die Start-Klasse) und die zu ladende Klasse im gleichen Klassenpfad, so wird diese Klasse vom Standard-Klassenlader des Systems und nicht vom eigenen Klassenlader geladen.

Der Server wird mit Portnummer und Suchpfad aufgerufen. Da der Name der angeforderten Klasse auch den Paketnamen enthalten kann, werden Punkte durch Schrägstriche ersetzt und somit die einzelnen Teile des Paketnamens auf Verzeichnisnamen abgebildet.

```
import tcpframework.TCPServer;

public class ClassServer {
   public static String searchPath;

   public static void main(String[] args) throws Exception {
```

```
      int port = Integer.parseInt(args[0]);
      searchPath = args[1];

      searchPath = searchPath.replace('\\', '/');
      if (!searchPath.endsWith("/"))
         searchPath += "/";

      TCPServer server = new TCPServer(port, ClassHandler.class);

      server.start();
      System.out.println("ClassServer gestartet.");
      System.out.println("ENTER stoppt den Server.");

      System.in.read();

      server.stopServer();
      System.out.println("ClassServer wird gestoppt.");
   }
}
```

```
import java.io.EOFException;
import java.io.ObjectInputStream;
import java.io.PrintWriter;
import java.net.ConnectException;
import java.net.Socket;

public class NetworkClassLoader extends ClassLoader {
   private String host;
   private int port;

   public NetworkClassLoader(String host, int port) {
      this.host = host;
      this.port = port;
   }

   public Class<?> findClass(String name) throws ClassNotFoundException {
      byte[] b = loadClassData(name);
      return defineClass(name, b, 0, b.length);
   }

   private byte[] loadClassData(String name) {
      try (Socket socket = new Socket(host, port);
            ObjectInputStream in = new ObjectInputStream(
                  socket.getInputStream());
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                  true)) {

         out.println(name);
         Object obj = in.readObject();
         return (byte[]) obj;
      } catch (ConnectException e) {
         throw new RuntimeException(
               "Verbindung zum Server konnte nicht hergestellt werden.");
      } catch (EOFException e) {
         throw new RuntimeException("Klasse wurde nicht gefunden.");
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
}
```

Das Programm `Start` wird mit den folgenden Parametern aufgerufen:

- Name bzw. IP-Adresse des entfernten Rechners,
- Portnummer des entfernten Rechners,
- Name der zu ladenden Klasse, die die `main`-Methode enthält,
- evtl. Aufrufparameter für die `main`-Methode.

Das Reflection-API wird genutzt, um die `main`-Methode der geladenen Klasse mit evtl. Parametern aufzurufen.

```
public class Start {
   public static void main(String[] args) {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      String className = args[2];

      // Kommandozeilenparameter
      String[] params = new String[args.length - 3];
      for (int i = 3; i < args.length; i++)
         params[i - 3] = args[i];

      try {
         NetworkClassLoader loader = new NetworkClassLoader(host, port);
         Class<?> c = loader.loadClass(className);

         java.lang.reflect.Method m = c.getMethod("main",
               new Class[] { String[].class });
         m.invoke(null, new Object[] { params });
      } catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

Zur Demonstration des Verfahrens kann z. B. der `ChatClient` aus Kapitel 3.8 genutzt werden.

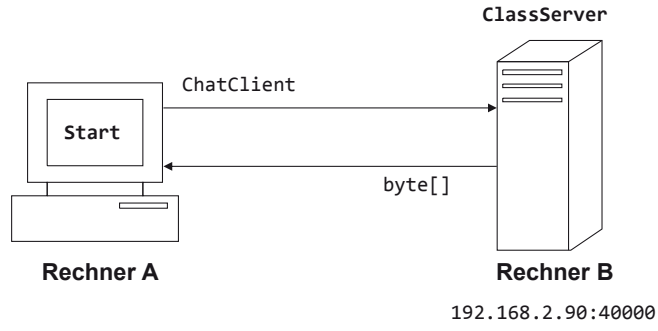

Abbildung 3-6: Dynamisches Laden über das Netz

Das Laden unbekannter Klassen aus dem Netz stellt ein Sicherheitsrisiko dar. Eine Möglichkeit, sich zu schützen, besteht darin, den *Security Manager* zu nutzen und nur so viele Zugriffsrechte wie nötig für die Ausführung der Anwendung zuzulassen.

Die Zugriffsrechte, passend zum obigen Beispiel, werden in der Policy-Datei `policy.txt` beschrieben:

```
grant {
   permission java.lang.RuntimePermission "createClassLoader";
   permission java.lang.RuntimePermission "exitVM";
   permission java.awt.AWTPermission "showWindowWithoutWarningBanner";

   // ClassServer
   permission java.net.SocketPermission "192.168.2.90:40000", "connect";

   // ChatServer
   permission java.net.SocketPermission "192.168.2.90:50000",  "connect";
};
```

Die hier beispielhaft angegebene IP-Nummer ist die Adresse des Servers.

Start des `ClassServer`:

```
java -cp bin;../framework/bin ClassServer 40000 ..\chat\bin
```

Der Klassenpfad bezieht auch die Framework-Klassen aus Kapitel 3.5 mit ein.

Start des Chat-Servers: siehe Kapitel 3.8

Aufruf des Start-Programms (Kommando in einer Zeile):

```
java -Djava.security.manager -Djava.security.policy= policy.txt -cp bin
Start 192.168.2.90 40000 ChatClient 192.168.2.90 50000
```

3.10 Aufgaben

1. Entwickeln Sie einen TCP-Server, der als Reaktion auf die Verbindungsaufnahme durch den Client an diesen einen in einer Datei gespeicherten Text sendet und dann von sich aus die Verbindung beendet. Testen Sie den Server mit Hilfe von Telnet:

   ```
   telnet localhost 50000
   ```

2. Entwickeln Sie einen TCP-Server, der die aktuelle Systemzeit des Servers als Zeichenkette liefert. Programmieren Sie auch einen passenden Client hierzu.

3. Entwickeln Sie ein Programm, mit dem Dateien von einem Rechner auf einen anderen kopiert werden können.

   Aufruf beim Empfänger (Server):

   ```
   java -cp bin NetCopy > ausgabe
   ```

   Aufrufbeispiel beim Sender (Client):

   ```
   java -cp bin NetCopy 192.168.2.90 < eingabe
   ```

4. Entwickeln Sie einen Server `SecretMessageReceiver`, der verschlüsselte Nachrichten empfangen kann und diese entschlüsselt auf der Konsole ausgibt, sowie einen Client `SecretMessageSender`, der Nachrichten verschlüsselt an den Server sendet.

   Aufruf des Servers:

   ```
   java -cp bin SecretMessageReceiver 50000 password
   ```

   Aufruf des Client (Beispiel):

   ```
   java -cp bin SecretMessageSender 192.168.2.90 50000 password < eingabe
   ```

   Benutzen Sie den symmetrischen Verschlüsselungsalgorithmus *Blowfish*.

   Das Programm `CryptDemo` zeigt, wie Datenströme mit Blowfish verschlüsselt bzw. entschlüsselt werden können:

```
import java.io.InputStream;
import java.io.OutputStream;
import java.security.Key;

import javax.crypto.Cipher;
import javax.crypto.CipherInputStream;
import javax.crypto.CipherOutputStream;
import javax.crypto.spec.SecretKeySpec;

public class CryptDemo {
   private static void encode(byte[] password) throws Exception {
      Cipher cipher = Cipher.getInstance("Blowfish");
      Key key = new SecretKeySpec(password, "Blowfish");
      cipher.init(Cipher.ENCRYPT_MODE, key);

      try (OutputStream out = new CipherOutputStream(System.out, cipher)) {
         byte[] buffer = new byte[8 * 1024];
         int c;
         while ((c = System.in.read(buffer)) != -1) {
            out.write(buffer, 0, c);
         }
      }
   }

   private static void decode(byte[] password) throws Exception {
      Cipher cipher = Cipher.getInstance("Blowfish");
      Key key = new SecretKeySpec(password, "Blowfish");
      cipher.init(Cipher.DECRYPT_MODE, key);

      try (InputStream in = new CipherInputStream(System.in, cipher)) {
```

```
            byte[] buffer = new byte[8 * 1024];
            int c;
            while ((c = in.read(buffer)) != -1) {
               System.out.write(buffer, 0, c);
            }
         }
      }

      public static void main(String[] args) throws Exception {
         char op = args[0].charAt(0);
         byte[] password = args[1].getBytes();

         if (op == 'e')
            encode(password);
         else
            decode(password);
      }
}
```

4 Nachrichtendienste mit JMS

Die Kommunikation zwischen Client und Server in den vorangegangenen Kapiteln ist dadurch gekennzeichnet, dass die Teilnehmer direkt und zeitgleich miteinander in Verbindung treten. In der Regel ist der Client solange blockiert, bis der Server die Verarbeitung abgeschlossen und die Antwort zurückgesendet hat (*synchrone Kommunikation*).

Die Kommunikation auf der Basis von nachrichtenorientierter Middleware (*Message Oriented Middleware*, *MOM*) macht sich von dieser engen Kopplung frei. Der Austausch von Nachrichten erfolgt *asynchron* mit Hilfe eines Vermittlers (*Message Broker*, *MOM-Server*), der Warteschlangen verwaltet. Die Nachricht des Senders wird vom Vermittler in die Warteschlange des Empfängers gelegt. Der Empfänger kann diese Nachricht zu einem späteren Zeitpunkt aus der Warteschlange holen. Sender und Empfänger agieren unabhängig voneinander und sind also über den Vermittler nur lose gekoppelt.

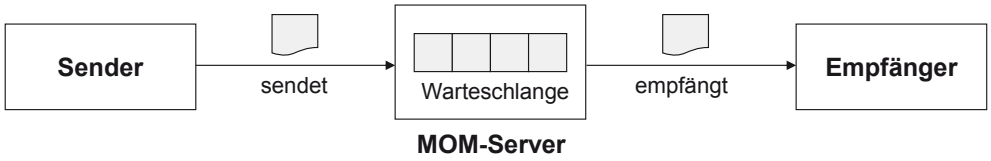

Abbildung 4-1: Nachrichtenaustausch

Vorteile dieser *asynchronen Kommunikation* sind:

- Sender und Empfänger arbeiten unabhängig voneinander. Auch bei Ausfall eines Teils (Sender, Empfänger oder Vermittler) kann die Nachricht noch nachträglich zugestellt werden. Die beteiligten Komponenten können getrennt voneinander wieder gestartet werden. Daraus ergibt sich eine hohe Fehlertoleranz.
- Eine Nachricht kann dank des Vermittlers an mehrere Empfänger gesendet werden, ohne dass jeweils eine explizite Verbindung vom Client zu jedem Empfänger aufgebaut werden muss.

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_4, © Springer Fachmedien Wiesbaden 2015

- Mehrere gesendete Nachrichten können gebündelt werden und zur Steigerung der Effizienz erst zu einem späteren Zeitpunkt in einem Rutsch abgeholt und verarbeitet werden.
- Durch Verwendung nachrichtenorientierter Middleware ist die Kommunikation unabhängig von Programmiersprache und Plattform.
- Die weitgehende Unabhängigkeit von Sender und Empfänger fördert den Einsatz dieser Technologie im Bereich der Anwendungsintegration.

4.1 Java Message Service

Java Message Service (*JMS*) wurde 1998 von Sun Microsystems veröffentlicht und ist inzwischen integraler Bestandteil der Java Enterprise Edition (Java EE). JMS ist ein API, das Syntax und Semantik für den Zugriff von in Java geschriebenen Clients auf nachrichtenorientierte Middleware definiert. Verschiedene Hersteller bieten Implementierungen an, die dieser Spezifikation genügen. Jeder zu Java EE konforme Application Server muss über eine MOM-Implementierung verfügen.

Wir nutzen in diesem Kapitel

- JMS 1.1

  JAR-Datei und Dokumentation können über die Webseite `http://www.oracle.com/technetwork/java/javaee/docs-136352.html` heruntergeladen werden.

- Open-Source-JMS-Implementierung *Apache ActiveMQ* Version 5.11.1 [1]

  Sie kann über die Webseite `http://activemq.apache.org` heruntergeladen werden.

Komponenten einer JMS-Anwendung

Eine JMS-Anwendung besteht aus einem JMS-Provider und mehreren JMS-Clients.

- *JMS-Provider*

  Der JMS-Provider ist der MOM-Server, in unserem Fall also ActiveMQ.

- *JMS-Clients*

  JMS-Clients sind die Java-Programme, die Nachrichten senden (*Message Producer*) bzw. empfangen (*Message Consumer*).

---

[1] Selbstverständlich kann auch eine neuere Version verwendet werden. Kleinere Änderungen, was z. B. die Administration des Tools angeht, müssen ggf. berücksichtigt werden.

- *Messages*

  Messages (Nachrichten) haben ein festgelegtes Format und werden von JMS-Clients erzeugt, versandt und empfangen.

- *Administrierte Objekte*

  Administrierte Objekte werden vom *JMS-Provider* bereitgestellt und in einem so genannten Namensdienst veröffentlicht. Diese Objekte werden von JMS-Clients über *JNDI* (*Java Naming and Directory Interface*) angefordert, um Nachrichten senden oder empfangen zu können. *ConnectionFactory* und *Destination* sind administrierte Objekte.

Abbildung 4.2 zeigt die JMS-Architektur und Abbildung 4.3 wichtige Schnittstellen des JMS-API.

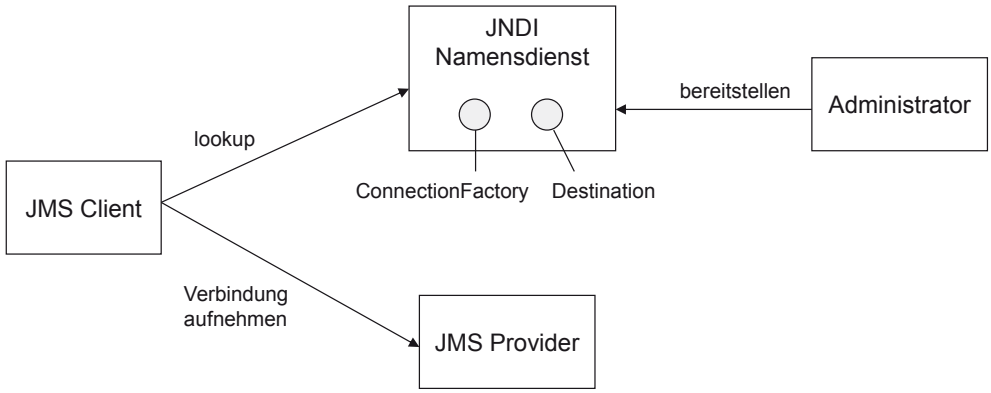

Abbildung 4-2: JMS-Architektur

Die Interfaces und Klassen des JMS-API gehören alle zum Paket `javax.jms`. Im Folgenden werden die in Abbildung 4.3 aufgeführten Interfaces kurz beschrieben.

`ConnectionFactory`

Eine Instanz dieses Typs wird vom JMS-Client genutzt, um Verbindungen zu einem JMS-Provider aufzubauen. Sie wird vom Administrator konfiguriert (administriertes Objekt) und über einen Namensdienst zur Verfügung gestellt.

`Connection`

Eine Verbindung wird von einer Verbindungsfabrik erzeugt und stellt einen Kommunikationskanal zum JMS-Provider dar.

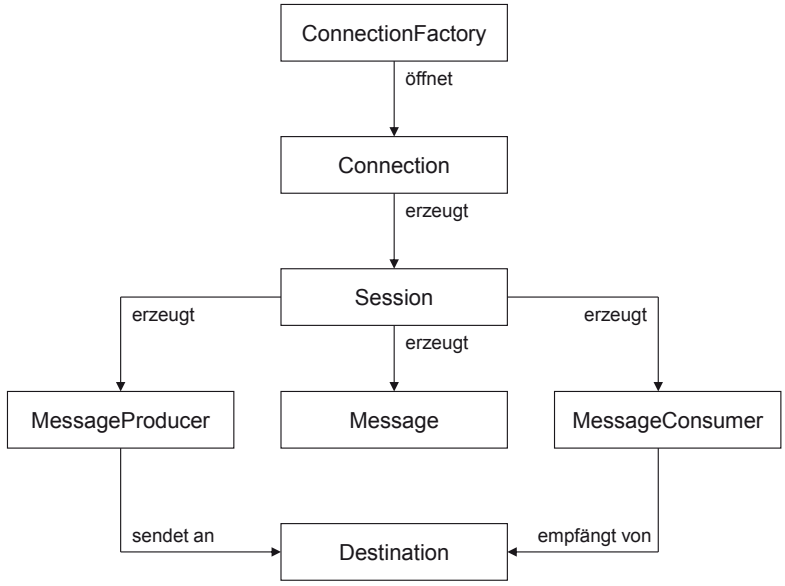

Abbildung 4-3: Wichtige JMS-Schnittstellen

`Session`

Eine Sitzung wird von einer geöffneten Verbindung erstellt und repräsentiert einen Kontext, in dem Nachrichten, Sender oder Empfänger erzeugt werden.

`Message`

Eine Nachricht besteht aus einem Kopf (Header), Eigenschaften (Properties) und einem Rumpf (Body).

Der Kopf enthält verschiedene Felder, die zur Identifizierung und Verwaltung genutzt werden und zum Teil vom JMS-Provider automatisch gesetzt werden. Einige Header werden in den Beispielen dieses Kapitels verwendet.

Abbildung 4-4: Aufbau einer Nachricht

Anwendungsbezogen können Properties hinterlegt werden. Hierzu stehen die folgenden `Message`-Methoden zur Verfügung:

```
void setXxxProperty(String name, xxx value) throws JMSException
xxx getXxxProperty(String name) throws JMSException
```

Hierbei steht *xxx* für `boolean`, `byte`, `short`, `int`, `long`, `float`, `double` oder `String`.

Der Rumpf speichert die Nutzdaten. Es existieren unterschiedliche Nachrichtentypen in Form von Subinterfaces:

`BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage` und `TextMessage`.

`MessageProducer`

Der `MessageProducer` hat die Aufgabe, Nachrichten an ein Ziel zu versenden.

`MessageConsumer`

Der `MessageConsumer` empfängt Nachrichten vom JMS-Provider.

`Destination`

Ein Nachrichtenziel (`Destination`-Objekt) repräsentiert je nach Nachrichtenmodell (Point-to-Point, Publish/Subscribe) eine Warteschlange des JMS-Providers. Das Nachrichtenziel kann vom Administrator eingerichtet (administriertes Objekt) und über einen Namensdienst bereitgestellt werden.

4.2 Das Point-to-Point-Modell

Beim *Point-to-Point-Modell* (*P2P*) wird eine vom Sender erzeugte Nachricht über eine *Queue* an genau einen Empfänger übermittelt (siehe Abbildung 4.1).

Sobald der Empfänger den Erhalt der Nachricht bestätigt hat, gilt sie als verbraucht und kann nicht erneut geholt werden. MOM-Server bieten das Konzept der *persistenten Warteschlange*. Nachrichten werden bis zu ihrer Auslieferung dauerhaft in einer Datenbank gespeichert, sodass sie ihren Empfänger auf jeden Fall erreichen, wenn er wieder aktiv ist.

In den Programmbeispielen dieses Abschnitts werden einfache Textnachrichten gesendet und empfangen.

ActiveMQ

Im Installationsverzeichnis von *ActiveMQ* befindet sich das Unterverzeichnis `bin`.

Hier muss nun mittels Konsole (Eingabeaufforderung)

```
activemq start
```

eingegeben werden.

Der Message Broker wird gestartet. Durch Eingabe von `Strg + C` kann er später beendet werden.

Die Aktivitäten des Brokers sowie Nachrichtenziele (Queues, Topics) können mit Hilfe einer *Web Console* überwacht werden:

```
http://localhost:8161/admin
```

Diese Anwendung ist geschützt.

Benutzername: admin, Passwort: admin.

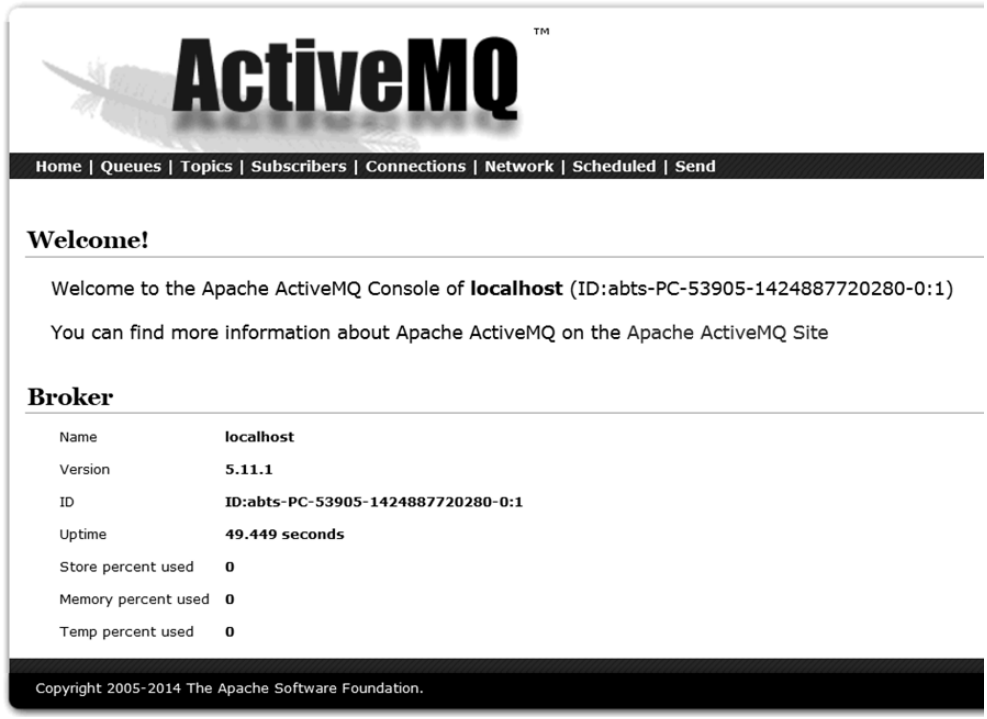

Abbildung 4-5: ActiveMQ Console

Ein JMS-Client muss zunächst `ConnectionFactory` und `Destination` über *JNDI* abfragen (*lookup*). Die Verbindungsdaten für diesen Zugriff befinden sich in der Datei `jndi.properties` (die Schlüssel-Wert-Paare stehen jeweils in einer Zeile):

```
java.naming.factory.initial =
       org.apache.activemq.jndi.ActiveMQInitialContextFactory
java.naming.provider.url = tcp://localhost:61616
# queue.[jndiName] = [physicalName]
queue.myQueue = demo.queue1
```

Arbeitet man mit verschiedenen Rechnern, so kann natürlich statt `localhost` die IP-Adresse des Rechners, auf dem ActiveMQ läuft, angegeben werden.

Der JNDI-Name der Queue ist hier `myQueue`. Im Fall, dass die zugehörige physische Ressource `demo.queue1` nicht existiert, wird sie von ActiveMQ erzeugt, sobald eine Nachricht an dieses Ziel gesendet wird. Gleiches gilt auch für Topics im Publish/Subscriber-Modell.

Zunächst werden die in den Programmbeispielen benutzten Typen und Methoden vorgestellt.

Die meisten Methoden des JMS-API lösen bei Fehlern eine Ausnahme vom Typ `JMSException` (Subklasse von `Exception`) aus.

`ConnectionFactory`-Methode:

`Connection createConnection() throws JMSException`
baut eine Verbindung auf.

`Connection`-Methoden:

`void start() throws JMSException`
startet die Auslieferung eingegangener Nachrichten beim Message Consumer.

`Session createSession(boolean transacted, int acknowledgeMode) throws JMSException`
erzeugt eine Session. `transacted` gibt an, ob das Senden bzw. Empfangen von Nachrichten innerhalb einer Transaktion stattfinden soll. `acknowledgeMode` legt fest, wer für die Empfangsbestätigung zuständig ist. Im Beispiel wird `Session.AUTO_ACKNOWLEDGE` (automatische Bestätigung durch die Session) genutzt.

`void close() throws JMSException`
schließt die Verbindung.

`Session`-Methoden:

`TextMessage createTextMessage() throws JMSException`
erzeugt eine Textnachricht.

`Message createMessage() throws JMSException`
erzeugt eine Nachricht ohne Rumpf.

`MessageProducer createProducer(Destination destination) throws JMSException`
erzeugt einen Message Producer für das Nachrichtenziel.

`MessageConsumer createConsumer(Destination destination) throws JMSException`
erzeugt einen Message Consumer für das Nachrichtenziel.

`void close() throws JMSException`
schließt die Sitzung.

`MessageProducer`-Methoden:

`void setTimeToLive(long timeToLive) throws JMSException`
setzt die Gültigkeitsdauer für Nachrichten in Millisekunden (0 steht für unbegrenzt).

`void setPriority(int priority)`
setzt die Priorität der Nachricht. 0 ist die niedrigste, 9 die höchste Priorität.

`void send(Message message) throws JMSException`
sendet die Nachricht message.

`void close() throws JMSException`
beendet den Message Producer.

`MessageConsumer`-Methoden:

`Message receive() throws JMSException`
wartet auf das Eintreffen einer Nachricht und kehrt dann zurück.

`Message receive(long timeout) throws JMSException`
wartet maximal `timeout` Millisekunden auf das Eintreffen einer Nachricht und kehrt dann zurück.

`void setMessageListener(MessageListener listener) throws JMSException`
registriert ein Objekt vom Typ `MessageListener`, das asynchron Nachrichten empfängt.

`void close() throws JMSException`
beendet den Message Consumer.

Das Interface `MessageListener` definiert die Methode

`void onMessage(Message message)`

`TextMessage` ist Subinterface von `Message`.

`TextMessage`-Methoden:

`void setText(String string) throws JMSException`
setzt den Inhalt der Textnachricht.

`String getText() throws JMSException`
liefert den Inhalt der Textnachricht.

Im folgenden ersten Beispiel demonstrieren wir, auf welche Arten ein Consumer Nachrichten empfangen kann.

Der Producer sendet eine Textnachricht an den JMS-Provider ActiveMQ. Optional kann eine Gültigkeitsdauer in Millisekunden angegeben werden.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Producer {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageProducer messageProducer;
   private String text;
   private long expiration;

   public Producer(String text, long expiration) throws NamingException,
         JMSException {
      this.text = text;
      this.expiration = expiration;

      // JNDI-Kontext erzeugen
      Context ctx = new InitialContext();

      // ConnectionFactory über Namensdienst auslesen
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");

      // Nachrichtenziel über Namensdienst auslesen
      Destination queue = (Destination) ctx.lookup(DESTINATION);

      // Verbindung aufbauen
      connection = factory.createConnection();

      // Session erzeugen
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // Sender erzeugen
      messageProducer = session.createProducer(queue);
   }
```

```
   // Nachricht erzeugen und senden
   public void sendMessage() throws JMSException {
      TextMessage message = session.createTextMessage();
      message.setText(text);
      message.setStringProperty("Status", "Testnachricht");
      messageProducer.setTimeToLive(expiration);
      messageProducer.send(message);
   }

   // Ressourcen freigeben
   public void close() {
      try {
         if (messageProducer != null)
            messageProducer.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   public static void main(String[] args) {
      Producer producer = null;
      try {
         String text = args[0];
         long expiration = (args.length == 2) ? Long.parseLong(args[1]) : 0;
         producer = new Producer(text, expiration);
         producer.sendMessage();
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (producer != null)
            producer.close();
      }
   }
}
```

Pull-Prinzip

Im folgenden Programm wartet der Empfänger aktiv auf das Eintreffen von Nachrichten (*Pull-Prinzip*). Nach `timeout` Millisekunden Wartezeit wird das Programm beendet.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.naming.NamingException;

public class Consumer1 {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;
   private long timeout;

   public Consumer1(long timeout) throws NamingException, JMSException {
      this.timeout = timeout;

      // JNDI-Kontext erzeugen
      Context ctx = new InitialContext();

      // ConnectionFactory ueber Namensdienst auslesen
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");

      // Zieladresse ueber Namensdienst auslesen
      Destination queue = (Destination) ctx.lookup(DESTINATION);

      // Verbindung aufbauen
      connection = factory.createConnection();

      // Session erzeugen
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // Empfaenger erzeugen
      messageConsumer = session.createConsumer(queue);

      // Empfang von Nachrichten starten
      connection.start();
   }

   // Aktives Warten auf Nachrichten (Pull-Prinzip)
   public void receiveMessage() throws JMSException {
      Message message;

      while ((message = messageConsumer.receive(timeout)) != null) {
         if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println(textMessage.getText());
         }
      }
   }

   // Ressourcen freigeben
   public void close() {
      try {
         if (messageConsumer != null)
            messageConsumer.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }
```

```
   public static void main(String[] args) {
      Consumer1 consumer = null;
      try {
         long timeout = Long.parseLong(args[0]);
         consumer = new Consumer1(timeout);
         consumer.receiveMessage();
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (consumer != null)
            consumer.close();
      }
   }
}
```

Push-Prinzip

Im Programm `Consumer2` wird der Empfänger vom MOM-Server durch die Callback-Methode `onMessage` über das Eintreffen einer Nachricht informiert (*Push-Prinzip*). Das Programm wird nach `timeout` Millisekunden beendet.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer2 implements MessageListener {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;

   public Consumer2() throws NamingException, JMSException {
      // JNDI-Kontext erzeugen
      Context ctx = new InitialContext();

      // ConnectionFactory ueber Namensdienst auslesen
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");

      // Zieladresse ueber Namensdienst auslesen
      Destination queue = (Destination) ctx.lookup(DESTINATION);

      // Verbindung aufbauen
      connection = factory.createConnection();

      // Session erzeugen
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // Empfaenger erzeugen
```

```
      messageConsumer = session.createConsumer(queue);

      // MessageListener setzen
      messageConsumer.setMessageListener(this);

      // Empfang von Nachrichten starten
      connection.start();
   }

   // Nachrichten werden im Push-Verfahren empfangen
   public void onMessage(Message message) {
      try {
         if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println(textMessage.getText());
         }
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   // Ressourcen freigeben
   public void close() {
      try {
         if (messageConsumer != null)
            messageConsumer.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   public static void main(String[] args) throws Exception {
      Consumer2 consumer = null;
      try {
         long timeout = Long.parseLong(args[0]);
         consumer = new Consumer2();
         Thread.sleep(timeout);
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (consumer != null)
            consumer.close();
      }
   }
}
```

Zur Compilierung und Ausführung der Programme wird die JAR-Datei

`activemq-all-5.11.1.jar`

aus der ActiveMQ-Installation verwendet.

`jndi.properties` muss in das Verzeichnis `bin` kopiert werden.

Wir nutzen die Umgebungsvariable `ACTIVEMQ_HOME`. Sie verweist auf den Installationsort von ActiveMQ und kann über die Systemeinstellungen zentral gesetzt werden.

Ausführung:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Producer Hallo
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Consumer1 30000
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Consumer2 30000
```

Das folgende Programm ermöglicht die Abfrage des Inhalts einer Warteschlange, ohne die Nachrichten zu löschen.

Hierzu gibt es das Interface `QueueBrowser` mit den folgenden Methoden:

`java.util.Enumeration getEnumeration() throws JMSException`
: liefert ein `Enumeration`-Objekt zum sequentiellen Durchlaufen der in der Warteschlange enthaltenen Nachrichten.

`void close() throws JMSException`
: schließt den Browser.

Mit der `Session`-Methode

```
QueueBrowser createBrowser(Queue queue) throws JMSException
```

wird ein Browser für die Warteschlange `queue` erzeugt. `Queue` ist Subinterface von `Destination`.

Header-Felder einer Nachricht können mit get-Methoden des Interface `Message` abgefragt werden. Das Programm verwendet die folgenden vom JMS-Provider automatisch gesetzten Felder:

`JMSDestination`

Das Nachrichtenziel, an das die Nachricht gesendet wurde (Typ `Destination`).

`JMSMessageID`

Eindeutiger Bezeichner zur Identifikation einer Nachricht (Typ `String`).

`JMSPriority`

Wert zwischen 0 (niedrig) und 9 (hoch), der die Priorität der Auslieferung einer Nachricht festlegt (Typ `int`). Die Priorität kann mit der `MessageProducer`-Methode `setPriority` (siehe oben) gesetzt werden.

`JMSTimestamp`

Zeitpunkt der Übergabe der Nachricht an den JMS-Provider in Millisekunden (Typ `long`).

`JMSExpiration`

Zeitpunkt des Verfalls einer Nachricht in Millisekunden (Typ `long`). Dieser Zeitpunkt kann mit der `MessageProducer`-Methode `setTimeToLive` (siehe oben) bestimmt werden.

Die `Message`-Methode

```
java.util.Enumeration getPropertyNames() throws JMSException
```

liefert ein `Enumeration`-Objekt mit den Namen der in der Nachricht hinterlegten Properties.

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Enumeration;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class QueueInfo {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private QueueBrowser browser;

   public QueueInfo() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
      Destination queue = (Destination) ctx.lookup(DESTINATION);
      connection = factory.createConnection();
```

```
      connection.start();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // QueueBrowser erzeugen
      browser = session.createBrowser((Queue) queue);
   }

   public void list() throws JMSException {
      @SuppressWarnings("rawtypes")
      Enumeration e = browser.getEnumeration();
      SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      int cnt = 0;

      while (e.hasMoreElements()) {
         cnt++;

         Message message = (Message) e.nextElement();

         System.out.print(cnt + ".");
         System.out.println("\tDestination: " + message.getJMSDestination());
         System.out.println("\tMessageID: " + message.getJMSMessageID());
         System.out.println("\tTimestamp: "
               + f.format(new Date(message.getJMSTimestamp())));
         System.out.println("\tPriority: " + message.getJMSPriority());

         long expiration = message.getJMSExpiration();

         if (expiration == 0) {
            System.out.println("\tExpiration: 0");
         } else {
            System.out.println("\tExpiration: "
                  + f.format(new Date(expiration)));
         }

         System.out.println("\tProperties:");

         @SuppressWarnings("rawtypes")
         Enumeration names = message.getPropertyNames();

         while (names.hasMoreElements()) {
            String name = (String) names.nextElement();
            System.out.println("\t\t" + name + " = "
                  + message.getStringProperty(name));
         }

         System.out.println();
      }
   }

   public void close() {
      try {
         if (browser != null)
            browser.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }
```

```
    public static void main(String[] args) throws Exception {
       QueueInfo info = null;
       try {
          info = new QueueInfo();
          info.list();
       } catch (Exception e) {
          System.err.println(e);
       } finally {
          if (info != null)
             info.close();
       }
    }
}
```

Ausgabebeispiel:

```
1.      Destination: queue://demo.queue1
        MessageID: ID:abts-PC-54108-1424888600452-1:1:1:1:1
        Timestamp: 25.02.2015 19:23:20
        Priority: 4
        Expiration: 0
        Properties:
                Status = Testnachricht
...
```

4.3 Das Request/Reply-Modell

Ein Spezialfall des Point-to-Point-Modells ist das *Request/Reply-Modell*, das eine synchrone Kommunikation zwischen Sender und Empfänger simuliert. Der Sender wartet solange, bis die Antwortnachricht eintrifft. Hierzu wird eine temporäre Warteschlange für die Antwortnachricht genutzt. Diese wird vom Sender für die Dauer der Kommunikation erzeugt und dem Empfänger über den Nachrichten-Header `JMSReplyTo` (Typ `Destination`) mitgeteilt.

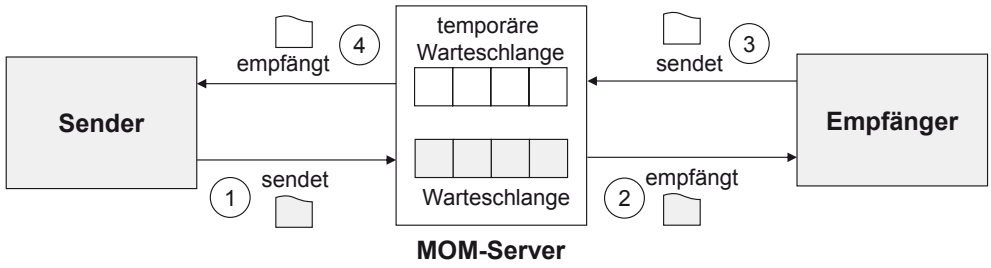

Abbildung 4-6: Request/Reply-Modell

Das Interface `TemoraryQueue` ist Subinterface von `Queue` und enthält die Lösch-Methode

```
void delete() throws JMSException
```

Die folgende `Session`-Methode erzeugt eine temporäre Warteschlange:

```
TemporaryQueue createTemporaryQueue() throws JMSException
```

Wir realisieren einen synchronen Echo-Dienst, der bereits aus vorhergehenden Kapiteln bekannt ist.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class EchoServer {
   private static final String DESTINATION = "myQueue";

   private ConnectionFactory factory;
   private Destination queue;
   private volatile boolean stop = false;

   public EchoServer() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      queue = (Destination) ctx.lookup(DESTINATION);
   }

   public void process() throws JMSException {
      Connection connection = factory.createConnection();
      Session session = connection.createSession(false,
           Session.AUTO_ACKNOWLEDGE);
      MessageConsumer messageConsumer = session.createConsumer(queue);
      connection.start();
      System.out.println("EchoServer gestartet ...");

      while (!stop) {
         TextMessage request = (TextMessage) messageConsumer.receive(5000);
         if (request == null)
            continue;
         String text = request.getText();
         Destination tempQueue = (Destination) request.getJMSReplyTo();
         TextMessage response = session.createTextMessage();
         response.setText(text);
         MessageProducer producer = session.createProducer(tempQueue);
         producer.send(response);
         producer.close();
      }
```

```
      messageConsumer.close();
      session.close();
      connection.close();
   }

   public void stop() {
      stop = true;
   }

   public static void main(String[] args) throws Exception  {
      final EchoServer server = new EchoServer();

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            if (server != null) {
               server.stop();
            }
            System.out.println("Server gestoppt");
         }
      });

      server.process();
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TemporaryQueue;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class EchoClient {
   private static final String DESTINATION = "myQueue";

   private String text;
   private ConnectionFactory factory;
   private Destination queue;

   public EchoClient(String text) throws NamingException, JMSException {
      this.text = text;
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      queue = (Destination) ctx.lookup(DESTINATION);
   }

   public void process() throws JMSException {
      Connection connection = null;
      Session session = null;
      TemporaryQueue tempQueue = null;
      MessageProducer messageProducer = null;
      MessageConsumer messageConsumer = null;
```

```
      try {
         connection = factory.createConnection();
         connection.start();
         session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

         // Temporaere Queue fuer die Antwort erzeugen
         tempQueue = session.createTemporaryQueue();

         messageProducer = session.createProducer(queue);
         messageConsumer = session.createConsumer(tempQueue);

         TextMessage request = session.createTextMessage();
         request.setText(text);

         // Hiermit wird das Nachrichtenziel fuer die Antwort festgelegt:
         request.setJMSReplyTo(tempQueue);

         messageProducer.send(request);

         // Auf Antwort warten
         TextMessage response = (TextMessage) messageConsumer.receive();

         System.out.println(response.getText());
      } finally {
         if (messageProducer != null)
            messageProducer.close();
         if (messageConsumer != null)
            messageConsumer.close();
         if (tempQueue != null)
            tempQueue.delete();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      }
   }

   public static void main(String[] args) throws Exception {
      String text = args[0];
      EchoClient client = new EchoClient(text);
      client.process();
   }
}
```

4.4 Das Publish/Subscribe-Modell

Beim *Publish/Subscribe-Modell* (*Pub/Sub*) sendet der *Publisher* Nachrichten zu einem bestimmten Thema (*Topic*), die vom Vermittler (MOM-Server) verteilt werden. *Subscriber* können beim Vermittler ein bestimmtes Thema abonnieren. Alle Subscriber erhalten dann die Nachrichten, die zu diesem Thema veröffentlicht wurden. Im Unterschied zum P2P-Modell erhalten sie aber nur die Nachrichten, die *während ihrer aktiven Verbindung* mit dem Vermittler versandt wurden. Publisher und Subscriber sind vollständig unabhängig voneinander.

Es können jedoch auch dauerhafte Abonnements eingerichtet werden. Ein *dauerhafter Subscriber* erhält dann später auch die Nachrichten zu einem Thema, die veröffentlicht wurden, während er keine Verbindung zum Vermittler hatte (siehe Kapitel 4.5).

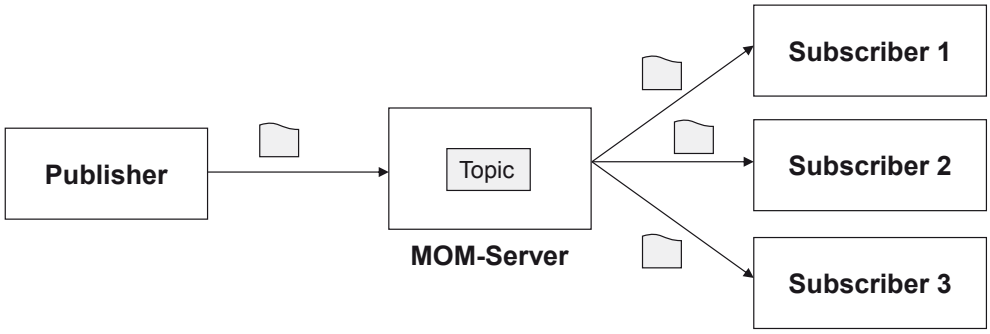

Abbildung 4-7: Publish/Subscribe-Modell

Das folgende Programmbeispiel nutzt den Nachrichtentyp `MapMessage`. Eine solche Nachricht besteht im Nachrichtenrumpf aus Schlüssel-Wert-Paaren. Der Schlüssel ist vom Typ `String`, der Wert vom Typ `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `String` oder `byte[]`.

Zu jedem Datentyp existieren die get- und set-Methoden:

```
void setXxx(xxx value) throws JMSException
xxx getXxx(String name) throws JMSException
```

Für den Typ `byte[]` heißen die Methoden `setBytes` und `getBytes`.

Eine `MapMessage` wird mit der folgenden `Session`-Methode erzeugt:

```
MapMessage createMapMessage() throws JMSException
```

Der zur Queue analoge Eintrag für ein Topic in der Datei `jndi.properties` ist hier:

```
topic.myTopic = demo.topic1
```

Der Publisher veröffentlicht in regelmäßigen Abständen Messdaten mit Zeit- und Wertangabe. Subscriber werten diese Messdaten aus.

```
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Publisher {
   private static final String DESTINATION = "myTopic";

   private ConnectionFactory factory;
   private Connection connection;
   private Session session;
   private MessageProducer messageProducer;
   private Destination topic;
   private long timeout;

   public Publisher(long timeout) throws NamingException, JMSException {
      this.timeout = timeout;
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      topic = (Destination) ctx.lookup(DESTINATION);
   }

   public void process() throws JMSException {
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageProducer = session.createProducer(topic);

      SimpleDateFormat formatter = new SimpleDateFormat("HH:mm:ss");
      long ende = System.currentTimeMillis() + timeout;

      while (System.currentTimeMillis() < ende) {
         String time = formatter.format(new Date());
         double value = Math.random() * 100;
         MapMessage message = session.createMapMessage();
         message.setString("Time", time);
         message.setDouble("Value", value);
         messageProducer.send(message);
         try {
            Thread.sleep(2000);
         } catch (InterruptedException e) {
         }
      }
   }

   public void close() {
      try {
         if (messageProducer != null)
            messageProducer.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
```

```
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   public static void main(String[] args) {
      Publisher pub = null;
      try {
         long timeout = Long.parseLong(args[0]);
         pub = new Publisher(timeout);
         System.out.println("Publisher gestartet ...");
         pub.process();
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (pub != null)
            pub.close();
         System.out.println("Publisher beendet");
      }
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber implements MessageListener {
   private static final String DESTINATION = "myTopic";

   private ConnectionFactory factory;
   private Destination topic;
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;

   public Subscriber() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      topic = (Destination) ctx.lookup(DESTINATION);
   }

   public void subscribe() throws JMSException {
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createConsumer(topic);
      messageConsumer.setMessageListener(this);
      connection.start();
   }
```

```
    public void close() {
       try {
          if (messageConsumer != null)
             messageConsumer.close();
          if (session != null)
             session.close();
          if (connection != null)
             connection.close();
       } catch (JMSException e) {
          System.err.println(e);
       }
    }

    public void onMessage(Message message) {
       try {
          if (message instanceof MapMessage) {
             MapMessage mapMessage = (MapMessage) message;
             System.out.println(mapMessage.getString("Time"));
             System.out.println(mapMessage.getDouble("Value"));
             System.out.println();
          }
       } catch (JMSException e) {
          System.err.println(e);
       }
    }

    public static void main(String[] args) {
       Subscriber sub = null;
       try {
          long time = Long.parseLong(args[0]);
          sub = new Subscriber();
          sub.subscribe();
          System.out.println("Subscriber gestartet ...");
          Thread.sleep(time);
       } catch (Exception e) {
          System.err.println(e);
       } finally {
          if (sub != null)
             sub.close();
          System.out.println("Subscriber beendet");
       }
    }
}
```

Test

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Publisher 120000
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Subscriber 60000
```

Mehrere Subscriber können in verschiedenen Konsolen mit unterschiedlichen Aktivitätszeiten gestartet werden.

4.5 Dauerhafte Subscriber

Wie bereits in Kapitel 4.4 erwähnt, erhält ein Subscriber standardmäßig nur solche Nachrichten, die an ein Topic geschickt werden, während er eine Verbindung zum MOM-Server hat. Ein *dauerhafter Subscriber* kann auch nachträglich Nachrichten abrufen, die geschickt wurden, während er inaktiv war. Hierzu muss eine Subskription eindeutig identifiziert werden können.

Zur Identifizierung einer Subskription dienen

- die Client-Identifikation für die Verbindung und
- der Name, der die Subskription innerhalb dieser Client-Identifikation eindeutig identifiziert.

Die Client-Identifikation wird mit der folgenden `Connection`-Methode gesetzt, unmittelbar nachdem das `Connection`-Objekt erzeugt wurde:

```
void setClientID(String clientID) throws JMSException
```

Die `Connection`-Methode

```
String getClientID() throws JMSException
```

liefert die Client-Identifikation der Verbindung.

Die `Session`-Methode

```
TopicSubscriber createDurableSubscriber(Topic topic,  String name)
    throws JMSException
```

erzeugt einen dauerhaften Subscriber. `name` ist der oben erwähnte Subskriptionsname. `Topic` ist Subinterface von Destination. `TopicSubscriber` ist Subinterface von `MessageConsumer`.

Mit der `Session`-Methode

```
void unsubscribe(String name) throws JMSException
```

kann sich der Subscriber vom Server abmelden. `name` ist der Subskriptionsname.

Das folgende Beispiel demonstriert Anmeldung, Abruf von Nachrichten und Abmeldung eines dauerhaften Subscribers.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
```

```
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Publisher {
   private static final String DESTINATION = "myTopic";

   private ConnectionFactory factory;
   private Connection connection;
   private Session session;
   private MessageProducer messageProducer;
   private Destination topic;

   public Publisher() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      topic = (Destination) ctx.lookup(DESTINATION);
   }

   public void publish(String text) throws JMSException {
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageProducer = session.createProducer(topic);

      TextMessage message = session.createTextMessage();
      message.setText(text);
      messageProducer.send(message);
   }

   public void close() {
      try {
         if (messageProducer != null)
            messageProducer.close();
         if (session != null)
            session.close();
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   public static void main(String[] args) throws Exception {
      Publisher pub = null;
      try {
         String text = args[0];
         pub = new Publisher();
         pub.publish(text);
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (pub != null)
            pub.close();
      }
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber {
   private static final String DESTINATION = "myTopic";

   private ConnectionFactory factory;
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;
   private Destination topic;

   public Subscriber() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      factory = (ConnectionFactory) ctx.lookup("ConnectionFactory");
      topic = (Destination) ctx.lookup(DESTINATION);
   }

   public void subscribe(String id, String name) throws JMSException {
      connection = factory.createConnection();
      connection.setClientID(id);
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createDurableSubscriber((Topic) topic, name);
   }

   public void unsubscribe(String id, String name) throws JMSException {
      connection = factory.createConnection();
      connection.setClientID(id);
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      session.unsubscribe(name);
   }

   public void receive(String id, String name, long timeout)
         throws JMSException {
      connection = factory.createConnection();
      connection.setClientID(id);
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createDurableSubscriber((Topic) topic, name);
      connection.start();

      Message message;
      while ((message = messageConsumer.receive(timeout)) != null) {
         if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            System.out.println(textMessage.getText());
         }
      }
   }
```

```
    public void close() {
       try {
          if (messageConsumer != null)
             messageConsumer.close();
          if (session != null)
             session.close();
          if (connection != null)
             connection.close();
       } catch (JMSException e) {
          System.err.println(e);
       }
    }

    public static void main(String[] args) {
       Subscriber sub = null;
       try {
          String option = args[0];
          String id = args[1];
          String name = args[2];

          sub = new Subscriber();
          if (option.equals("subscribe")) {
             sub.subscribe(id, name);
          } else if (option.equals("unsubscribe")) {
             sub.unsubscribe(id, name);
          } else if (option.equals("receive")) {
             long timeout = Long.parseLong(args[3]);
             sub.receive(id, name, timeout);
          }
       } catch (Exception e) {
          System.err.println(e);
       } finally {
          if (sub != null)
             sub.close();
       }
    }
}
```

Anmelden:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Subscriber
subscribe 0001 test
```

Nachricht publizieren:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Publisher
Hallo
```

Nachrichten abrufen:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Subscriber
receive 0001 test 5000
```

Ausgabe:

```
Hallo
```

Abmelden:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Subscriber
unsubscribe 0001 test
```

Die Kommandos sind jeweils in einer einzigen Zeile einzugeben.

Mit der Web Console (`http://localhost:8161/admin`) kann das An- und Abmelden verfolgt werden.

4.6 Filtern von Nachrichten

Um nur die Nachrichten zu empfangen, die bestimmte Bedingungen erfüllen, kann ein Filtermechanismus verwendet werden.

Die Nachrichten werden beim JMS-Provider auf der Basis von Bedingungen, die sich auf *Header-Felder* und *Properties* beziehen können, selektiert.

Zur Formulierung dieser Bedingungen wird eine *SQL-ähnliche Notation* verwendet. Es können Vergleichsoperatoren (`=`, `<`, `<=`, `>`, `>=`), logische Operatoren (`AND`, `OR`, `NOT`), sowie `IN`, `LIKE` und `IS NULL` benutzt werden.

Ein *Message Selector* ist ein String, der einen so gebildeten Bedingungsausdruck enthält.

Im Programmbeispiel wird z. B. der folgende Message Selector verwendet:

```
"Priority = 'normal' OR Priority = 'high'"
```

Beim Erzeugen des Empfängers kann der Message Selector festgelegt werden. Hierzu stehen die folgenden `Session`-Methoden zur Verfügung:

```
MessageConsumer createConsumer(Destination destination, String
       messageSelector, boolean noLocal) throws JMSException

TopicSubscriber createDurableSubscriber(Topic topic, String name, String
       messageSelector, boolean noLocal) throws JMSException
```

Für den Fall, dass über dieselbe Verbindung für ein Topic Nachrichten sowohl gesendet als auch empfangen werden, kann der Empfänger festlegen, ob von ihm selbst gesendete Nachrichten auch von ihm wieder empfangen werden sollen. Hat `noLocal` den Wert `true`, so wird dies ausgeschlossen.

Im folgenden Beispiel erzeugt der Publisher Nachrichten unterschiedlicher Priorität (`low`, `normal`, `high`). Der Subscriber kann so aufgerufen werden, dass er alle Nachrichten, Nachrichten mit normaler und hoher Priorität oder nur Nachrichten mit hoher Priorität empfängt.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Publisher {
   private static final String DESTINATION = "myTopic";
   private Connection connection;
   private Session session;
   private MessageProducer messageProducer;

   public Publisher() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
      Destination queue = (Destination) ctx.lookup(DESTINATION);
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageProducer = session.createProducer(queue);
   }

   public void sendMessage() throws JMSException {
      TextMessage message = session.createTextMessage();

      for (int i = 0; i < 60; i++) {
         message.setText("Nachricht " + (i + 1));

         String prio = "";

         if (Math.random() < 0.25) {
            prio = "low";
         } else if (Math.random() < 0.75) {
            prio = "normal";
         } else {
            prio = "high";
         }

         message.setStringProperty("Priority", prio);

         messageProducer.send(message);

         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            System.err.println(e);
         }
      }
   }

   public void close() {
      try {
         if (messageProducer != null)
            messageProducer.close();
         if (session != null)
            session.close();
```

```
         if (connection != null)
            connection.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   public static void main(String[] args) {
      Publisher pub = null;
      try {
         pub = new Publisher();
         pub.sendMessage();
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (pub != null)
            pub.close();
      }
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Subscriber implements MessageListener {
   private static final String DESTINATION = "myTopic";
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;

   public Subscriber(String messageSelector) throws NamingException,
         JMSException {
      Context ctx = new InitialContext();
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
      Destination queue = (Destination) ctx.lookup(DESTINATION);
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

      // Selektion von Messages
      messageConsumer = session.createConsumer(queue, messageSelector, true);

      messageConsumer.setMessageListener(this);
      connection.start();
   }

   public void onMessage(Message message) {
      try {
         TextMessage textMessage = (TextMessage) message;
         System.out.print(textMessage.getText());
```

```
            System.out
                  .println(": " + textMessage.getStringProperty("Priority"));
         } catch (JMSException e) {
            System.err.println(e);
         }
      }

      public void close() {
         try {
            if (messageConsumer != null)
               messageConsumer.close();
            if (session != null)
               session.close();
            if (connection != null)
               connection.close();
         } catch (JMSException e) {
            System.err.println(e);
         }
      }

      public static void main(String[] args) throws Exception {
         int typ = Integer.parseInt(args[0]);
         String messageSelector = "";

         if (typ == 2) {
            messageSelector = "Priority = 'normal' OR " + "Priority = 'high'";
         } else if (typ == 3) {
            messageSelector = "Priority = 'high'";
         }

         Subscriber sub = null;
         try {
            sub = new Subscriber(messageSelector);
            Thread.sleep(30000);
         } catch (Exception e) {
            System.err.println(e);
         } finally {
            if (sub != null)
               sub.close();
         }
      }
   }
```

4.7 Transaktionen

Producer und Consumer bauen jeweils eigene und unabhängige Sessions mit dem JMS-Provider auf. Die Kommunikation zwischen Sender und JMS-Provider einerseits und Empfänger und JMS-Provider andererseits kann *transaktionsorientiert* erfolgen. Es wird also zwischen Transaktionssteuerung beim Senden und Empfangen unterschieden.

Der *Transaktionsmodus* wird durch den ersten Aufrufparameter der `Connection`-Methode `createSession` eingestellt.

Mehrere Nachrichten können auf der Senderseite in einer Transaktion gebündelt werden. Der Versand der Nachrichten wird solange zurückgehalten, bis die Transaktion mit dem Aufruf der `Session`-Methode `commit` abgeschlossen wird. Die gesamte Transaktion kann durch den Aufruf der `Session`-Methode `rollback` rückgängig gemacht werden. `commit` bzw. `rollback` öffnet sofort wieder eine neue Transaktion.

Auf der Empfängerseite können innerhalb einer Transaktion nacheinander mehrere Nachrichten empfangen werden. Der Aufruf von `commit` bestätigt den Empfang und schließt die Transaktion ab. Wird die Transaktion mit `rollback` beendet, wird versucht, alle Nachrichten, die in der letzten Transaktion empfangen wurden, erneut zuzustellen. Ob eine Nachricht schon einmal zugestellt wurde, kann durch Auswertung des Header-Feldes `JMSRedelivered` (`true` oder `false`) ermittelt werden.

`Session-Methoden:`

`void commit() throws JMSException`

`void rollback() throws JMSException`

`boolean getTransacted() throws JMSException`
prüft, ob die aktuelle Session mit Transaktionssteuerung arbeitet.

Das folgende Beispiel demonstriert das Arbeiten mit Transaktionen. Der Sender schickt innerhalb einer Transaktion zwei Textnachrichten und eine weitere Nachricht als Abschlussnachricht, die das Ende einer Nachrichtenfolge signalisiert. Wird der *Producer* mit dem Kommandozeilenparameter "`true`" aufgerufen, so wird eine Fehlersituation simuliert, die zum Abbruch der Transaktion mit `rollback` führt. Der *Consumer* kann ebenfalls mit einem solchen Kommandozeilenparameter gestartet werden. Rollback führt dazu, dass die bisher empfangenen Nachrichten nochmals zugestellt werden.

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Producer {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageProducer messageProducer;
```

```
public Producer() throws NamingException, JMSException {
   Context ctx = new InitialContext();
   ConnectionFactory factory = (ConnectionFactory) ctx
         .lookup("ConnectionFactory");
   Destination queue = (Destination) ctx.lookup(DESTINATION);
   connection = factory.createConnection();

   // transaktionsfähige Session
   session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

   messageProducer = session.createProducer(queue);
}

public void sendMessages(boolean exception) {
   try {
      System.out.println("getTransacted: " + session.getTransacted());

      // Es werden 2 Nachrichten gesendet.
      TextMessage message = session.createTextMessage();

      message.setText("Nachricht 1");
      message.setIntProperty("number", 1);
      message.setIntProperty("total", 2);
      messageProducer.send(message);

      message.setText("Nachricht 2");
      message.setIntProperty("number", 2);
      message.setIntProperty("total", 2);
      messageProducer.send(message);

      if (exception) {
         throw new JMSException("Simulierter Fehler");
      }

      session.commit();
      System.out.println("Committed");
   } catch (JMSException e) {
      System.err.println(e);
      try {
         session.rollback();
         System.out.println("Rolled back");
      } catch (JMSException e1) {
         System.err.println(e1);
      }
   }
}

public void close() {
   try {
      if (messageProducer != null)
         messageProducer.close();
      if (session != null)
         session.close();
      if (connection != null)
         connection.close();
   } catch (JMSException e) {
      System.err.println(e);
   }
}
```

```
   public static void main(String[] args) {
      Producer producer = null;
      try {
         boolean exception = Boolean.parseBoolean(args[0]);
         producer = new Producer();
         producer.sendMessages(exception);
      } catch (Exception e) {
         System.err.println(e);
      } finally {
         if (producer != null)
            producer.close();
      }
   }
}
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Consumer {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;

   public Consumer() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
      Destination queue = (Destination) ctx.lookup(DESTINATION);
      connection = factory.createConnection();

      // transaktionsfähige Session
      session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);

      messageConsumer = session.createConsumer(queue);
      connection.start();
   }

   public void receiveMessages(boolean exception) {
      String[] messages = null;
      boolean first = true;
      int total = 0;
      int count = 0;

      while (true) {
         try {
            Message message = messageConsumer.receive();
            TextMessage textMessage = (TextMessage) message;

            if (first) {
               total = textMessage.getIntProperty("total");
```

```
            messages = new String[total];
            first = false;
         }

         int number = textMessage.getIntProperty("number");
         messages[number - 1] = textMessage.getText();
         count++;

         System.out.println(textMessage.getText());
         System.out.println("\tRedelivered: "
               + textMessage.getJMSRedelivered());

         if (exception) {
            exception = false;
            throw new JMSException("Simulierter Fehler");
         }

         if (count == total) {
            session.commit();
            System.out.println("Committed");
            break;
         }
      } catch (JMSException e) {
         System.err.println(e);
         try {
            session.rollback();
            System.out.println("Rolled back");
            count = 0;
         } catch (JMSException e1) {
            System.err.println(e1);
         }
      }
   }

   System.out.println();
   for (int i = 0; i < messages.length; i++) {
      System.out.println(messages[i]);
   }
}

public void close() {
   try {
      if (messageConsumer != null)
         messageConsumer.close();
      if (session != null)
         session.close();
      if (connection != null)
         connection.close();
   } catch (JMSException e) {
      System.err.println(e);
   }
}

public static void main(String[] args) {
   Consumer consumer = null;
   try {
      boolean exception = Boolean.parseBoolean(args[0]);
      consumer = new Consumer();
      consumer.receiveMessages(exception);
   } catch (Exception e) {
      System.err.println(e);
```

```
      } finally {
         if (consumer != null)
            consumer.close();
      }
   }
}
```

Test

1. Aufruf des Programms Producer mit Fehlersimulation:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Producer true
```

Ausgabe:

```
getTransacted: true
javax.jms.JMSException: Simulierter Fehler
Rolled back
```

2. Aufruf des Programms Producer ohne Fehler:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.11.1.jar Producer false
```

Ausgabe:

```
getTransacted: true
Committed
```

3. Aufruf des Programms Consumer mit Fehlersimulation:

```
java -cp bin;%ACTIVEMQ_HOME%/activemq-all-5.10.0.jar Consumer true
```

Ausgabe:

```
Nachricht 1
   Redelivered: false
javax.jms.JMSException: Simulierter Fehler
Rolled back
Nachricht 1
   Redelivered: true
Nachricht 2
   Redelivered: false
Committed

Nachricht 1
Nachricht 2
```

4.8 Fallbeispiel Händler/Lieferant

Ein Händler (`Vendor`) bestellt mehrere Artikel in bestimmten Stückzahlen bei einem Lieferanten. Die Bestellung (`ObjectMessage`) wird an die Warteschlage `orderQueue` versandt. Dabei wartet er auf die Antwort, ob die bestellten Artikel lieferbar sind

oder nicht. Für die Antwort wird eine temporäre Warteschlage (`replyQueue`) erzeugt.

Der Lieferant (`Supplier`) prüft für jeden bestellten Artikel, ob er in der gewünschten Stückzahl lieferbar ist. Er sendet das Ergebnis an die Warteschlange `replyQueue` und wartet dann auf weitere Aufträge.

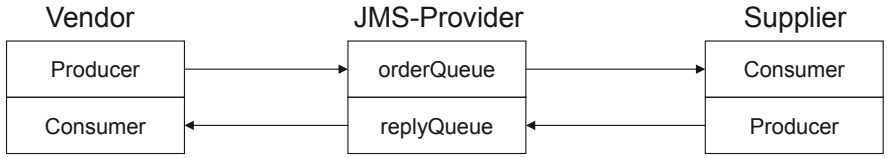

Abbildung 4-8: Kommunikation Vendor – Supplier

Die `Session`-Methode

```
ObjectMessage createObjectMessage()
```

erzeugt eine Nachricht zur Aufnahme eines serialisierbaren Objekt.

Die `ObjectMessage`-Methode

```
void setObject(Object obj)
```

setzt `obj` als Inhalt der Nachricht. `obj` muss serialisierbar sein.

```
Object getObject()
```

liefert den Inhalt der Nachricht.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Supplier {
   private static final String DESTINATION = "myQueue";
```

```
private Connection connection;
private Session session;
private MessageConsumer messageConsumer;
private MessageProducer messageProducer;
private volatile boolean stop = false;

public Supplier() throws NamingException, JMSException {
   Context ctx = new InitialContext();
   ConnectionFactory factory = (ConnectionFactory) ctx
         .lookup("ConnectionFactory");
   Destination queue = (Destination) ctx.lookup(DESTINATION);
   connection = factory.createConnection();
   session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   messageConsumer = session.createConsumer(queue);
   connection.start();
   System.out.println("Supplier gestartet ...");
}

public void processOrders() throws JMSException {
   while (!stop) {
      try {
         ObjectMessage inMessage =
            (ObjectMessage) messageConsumer.receive(5000);
         if (inMessage == null)
            continue;

         System.out.println(inMessage.getJMSMessageID());

         @SuppressWarnings("unchecked")
         List<Order> orders = (List<Order>) inMessage.getObject();

         messageProducer = session.createProducer(inMessage.getJMSReplyTo());
         ObjectMessage outMessage = session.createObjectMessage();
         ArrayList<String> result = new ArrayList<String>();

         for (Order order : orders) {
            String item = order.getItem();
            int quantity = order.getQuantity();

            // Simulation: Lieferbarkeit pruefen
            Random r = new Random();
            boolean available = r.nextBoolean();
            String status = (available == true) ? "lieferbar"
                  : "nicht lieferbar";

            System.out.printf("%s %3d: %s%n", item, quantity, status);
            result.add(item + " " + status);
         }

         outMessage.setObject(result);
         messageProducer.send(outMessage);
         messageProducer.close();
      } catch (JMSException e) {
         System.err.println(e);
      }
   }

   messageConsumer.close();
   session.close();
   connection.close();
}
```

```
   public void stop() {
      stop = true;
   }

   public static void main(String[] args) throws Exception {
      final Supplier supplier = new Supplier();

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            if (supplier != null) {
               supplier.stop();
            }
            System.out.println("Supplier gestoppt");
         }
      });

      supplier.processOrders();
   }
}
```

```
import java.io.Serializable;

@SuppressWarnings("serial")
public class Order implements Serializable {
   private String item;
   private int quantity;

   public Order(String item, int quantity) {
      this.item = item;
      this.quantity = quantity;
   }

   public String getItem() {
      return item;
   }

   public void setItem(String item) {
      this.item = item;
   }

   public int getQuantity() {
      return quantity;
   }

   public void setQuantity(int quantity) {
      this.quantity = quantity;
   }

   public String toString() {
      return "Order [item=" + item + ", quantity=" + quantity + "]";
   }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import javax.jms.Connection;
```

```
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.ObjectMessage;
import javax.jms.Session;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class Supplier {
   private static final String DESTINATION = "myQueue";
   private Connection connection;
   private Session session;
   private MessageConsumer messageConsumer;
   private MessageProducer messageProducer;
   private volatile boolean stop = false;

   public Supplier() throws NamingException, JMSException {
      Context ctx = new InitialContext();
      ConnectionFactory factory = (ConnectionFactory) ctx
            .lookup("ConnectionFactory");
      Destination queue = (Destination) ctx.lookup(DESTINATION);
      connection = factory.createConnection();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      messageConsumer = session.createConsumer(queue);
      connection.start();
      System.out.println("Supplier gestartet ...");
   }

   public void processOrders() throws JMSException {
      while (!stop) {
         try {
            ObjectMessage inMessage =
               (ObjectMessage) messageConsumer.receive(5000);
            if (inMessage == null)
               continue;

            System.out.println(inMessage.getJMSMessageID());

            @SuppressWarnings("unchecked")
            List<Order> orders = (List<Order>) inMessage.getObject();

            messageProducer = session.createProducer(inMessage.getJMSReplyTo());
            ObjectMessage outMessage = session.createObjectMessage();
            ArrayList<String> result = new ArrayList<String>();

            for (Order order : orders) {
               String item = order.getItem();
               int quantity = order.getQuantity();

               // Simulation: Lieferbarkeit pruefen
               Random r = new Random();
               boolean available = r.nextBoolean();
               String status = (available == true) ? "lieferbar"
                     : "nicht lieferbar";

               System.out.printf("%s %3d: %s%n", item, quantity, status);
               result.add(item + " " + status);
            }
```

```
            outMessage.setObject(result);
            messageProducer.send(outMessage);
            messageProducer.close();
         } catch (JMSException e) {
            System.err.println(e);
         }
      }

      messageConsumer.close();
      session.close();
      connection.close();
   }

   public void stop() {
      stop = true;
   }

   public static void main(String[] args) throws Exception {
      final Supplier supplier = new Supplier();

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            if (supplier != null) {
               supplier.stop();
            }
            System.out.println("Supplier gestoppt");
         }
      });

      supplier.processOrders();
   }
}
```

Beispielhafte Ausgabe des Vendor:

```
Bestellt:
Order [item=4711, quantity=5]
Order [item=4712, quantity=15]
Order [item=4713, quantity=8]

Antwort:
4711 lieferbar
4712 nicht lieferbar
4713 lieferbar
```

Beispielhafte Ausgabe des Supplier:

```
Supplier gestartet ...
ID:abts-PC-50959-1424966613029-1:1:1:1:1
4711   5: lieferbar
4712  15: nicht lieferbar
4713   8: lieferbar
Supplier gestoppt
```

4.9 Aufgaben

1. Entwickeln Sie auf Basis des P2P-Modells einen Zeitangabe-Service, der jede Minute die aktuelle Uhrzeit übermittelt. Ein Client soll diese Zeitangabe ausgeben.

2. Durch Vergabe von Prioritäten (siehe Kapitel 4.2) kann die Reihenfolge der ausgelieferten Nachrichten beeinflusst werden. Testen Sie diesen Sachverhalt, indem Sie einen geeigneten Producer bzw. Consumer entwickeln.

   Die Prioritätensteuerung für Queues muss eigens konfiguriert werden. Hierzu muss die XML-Datei

   `<ActiveMQ-Installationsverzeichnis>/conf/activemq.xml`

   angepasst werden:

```
<destinationPolicy>
    <policyMap>
      <policyEntries>
        ...
        <policyEntry queue=">" prioritizedMessages="true" />
      </policyEntries>
    </policyMap>
</destinationPolicy>
```

   `queue=">"` bedeutet, dass für alle Queues Prioritäten berücksichtigt werden.

3. Implementieren Sie auf Basis des Pub/Sub-Modells ein einfaches Chat-Programm. Die Bedienung soll über Kommandozeilen erfolgen. Damit Nachrichten sowohl gesendet als auch empfangen werden können, muss das Programm einen Publisher und einen Subscriber enthalten.

4. Mit dem Nachrichtentyp `BytesMessage` können Daten byteweise übertragen werden. Diese Bytes werden als Rohdaten behandelt und von der Java Virtual Machine nicht interpretiert.

   Erstellen Sie JMS-Clients, die Binärdaten (z. B. ein GIF-Bild) senden bzw. empfangen.

   Die `Session`-Methode

   `BytesMessage createBytesMessage() throws JMSException`

   erzeugt eine `BytesMessage`.

   Zur Lösung dieser Aufgabe sollen die folgenden `BytesMessage`-Methoden verwendet werden:

   `void writeByte(byte value) throws JMSException`
   schreibt ein Byte.

`long getBodyLength() throws JMSException`
liefert die Anzahl Bytes, die der Nachrichtenrumpf als Nutzdaten enthält.

`byte readByte() throws JMSException`
liest ein Byte aus dem Nachrichtenrumpf.

5 Aufruf entfernter Methoden mit RMI

In diesem Kapitel stellen wir ein System vor, das die Kommunikation zwischen Objekten, die auf verschiedenen Rechnern erzeugt sind, ermöglicht. Neben (fast) beliebigen Objekten kann auch das Verhalten (Bytecode) übertragen werden, so dass flexible Anwendungen realisiert werden können. Allerdings muss sowohl der Client als auch der Server in Java programmiert sein.

Das klassische *RPC-Modell* (RPC = Remote Procedure Call) versteckt den Aufruf einer auf einem anderen Rechner im Netz implementierten Prozedur hinter einem lokalen Prozeduraufruf und bietet damit ein sehr vertrautes Programmiermodell an. Das Programm, das die Prozedur aufruft, agiert als Client, das Programm, das die aufgerufene Prozedur ausführt, als Server.

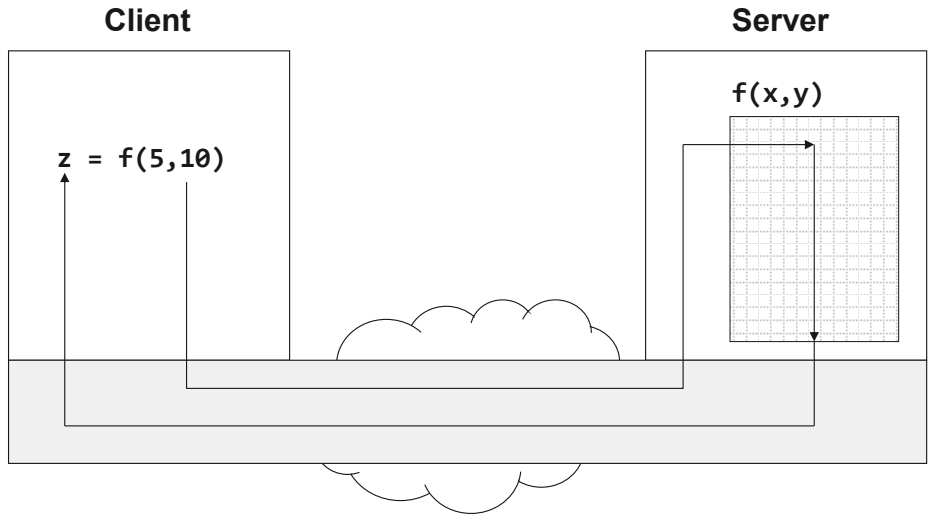

Abbildung 5-1: Entfernter Prozeduraufruf im RPC-Modell

RMI (*Remote Method Invocation*) ist die objektorientierte Umsetzung des RPC-Modells in Java.

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_5, © Springer Fachmedien Wiesbaden 2015

5.1 Ein einführendes Beispiel

Das folgende Programmbeispiel zeigt, wie einfach eine Anwendung mit lokalem Methodenaufruf in eine Client/Server-Anwendung mit entferntem Methodenaufruf (RMI) umgewandelt werden kann. Die im Beispiel benutze Methode berechnet die Summe zweier Zahlen. Zunächst die lokale Anwendung.

```
public class AddServer {
   public double add(double x, double y) {
      return x + y;
   }
}
```

```
public class AddClient {
   public static void main(String[] args) {
      AddServer service = new AddServer();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

Die beiden Klassen werden nun mittels eines Interface entkoppelt. Um ein Server-Objekt zu erzeugen, wird eine statische Methode benutzt, die den konkreten Namen der Implementierungsklasse (hier: `AddServer`) gegenüber dem Client verbirgt (Factory-Methode).

```
public interface Add {
   double add(double x, double y);
}
```

```
public class AddServer implements Add {
   public double add(double x, double y) {
      return x + y;
   }
}
```

```
public class AddFactory {
   public static Add getInstance() {
      return new AddServer();
   }
}
```

```
public class AddClient {
   public static void main(String[] args) {
      Add service = AddFactory.getInstance();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

Das Klassendiagramm in Abbildung 5.2 verdeutlicht die Zusammenhänge.

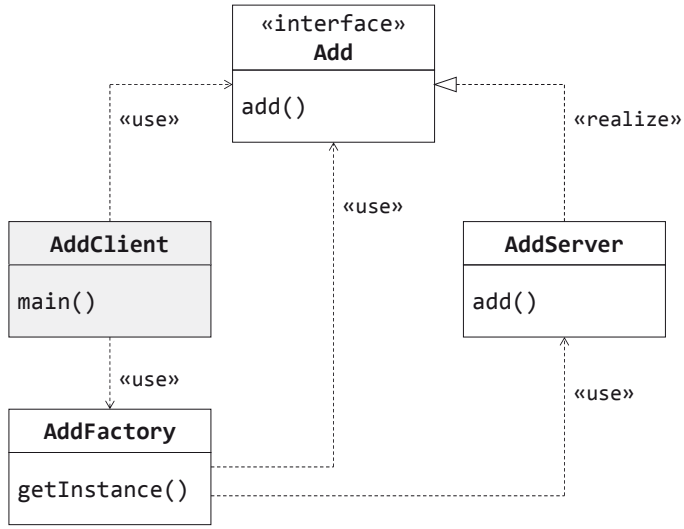

Abbildung 5-2: Klassendiagramm

Die RMI-Version der Anwendung entsteht nun durch leichte Änderung.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Add extends Remote {
   double add(double x, double y) throws RemoteException;
}
```

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class AddServer extends UnicastRemoteObject implements Add {
   public AddServer() throws RemoteException {
   }

   public double add(double x, double y) {
      return x + y;
   }

   public static void main(String[] args) throws Exception {
      AddServer server = new AddServer();
      Naming.rebind("add", server);
   }
}
```

```
import java.rmi.Naming;

public class AddFactory {
   public static Add getInstance() throws Exception {
      return (Add) Naming.lookup("//localhost/add");
   }
}
```

```
public class AddClient {
   public static void main(String[] args) throws Exception {
      Add service = AddFactory.getInstance();
      System.out.println(service.add(2.3, 5.7));
   }
}
```

`Add`, `AddServer` und `AddFactory` importieren RMI-Pakete. Das Interface Add ist von `java.rmi.Remote` abgeleitet, die Methode `add` enthält eine `throws`-Klausel mit der Ausnahme `java.rmi.RemoteException`. Die Klasse `AddServer` ist von `java.rmi.server.UnicastRemoteObject` abgeleitet und besitzt eine `main`-Methode, die das erzeugte Server-Objekt bei einem Verzeichnisdienst registriert. Die Methode `getInstance` der Klasse `AddFactory` erhält ein `Add`-Objekt mit Hilfe dieses Verzeichnisdienstes.

Zur Vereinfachung sollen Client und Server auf demselben Rechner (`localhost`) laufen.

Test (Aufrufe jeweils im Projektverzeichnis)

Start des Verzeichnisdienstes:

```
set CLASSPATH=bin
rmiregistry
```

Start des Servers:

```
java -cp bin AddServer
```

Aufruf des Client:

```
java -cp bin AddClient
```

Verzeichnisdienst und Server müssen am Ende mit `Strg + C` abgebrochen werden.

5.2 Remote Method Invocation

Das Protokoll *Remote Method Invocation* (RMI) setzt auf TCP/IP auf und verbirgt die Details einer Netzverbindung. RMI hat folgende Eigenschaften:

- Mit RMI können Methoden für Objekte aufgerufen werden, die von einer anderen virtuellen Maschine (JVM) erzeugt und verwaltet werden – in der Regel auf einem anderen Rechner.
- Für den Entwickler sieht der entfernte Methodenaufruf wie ein ganz normaler lokaler Aufruf aus.
- Entfernt aufrufbare Methoden werden in einem *Interface* deklariert. Nur hierüber kann der Client mit einem entfernten Objekt kommunizieren. Das Interface stellt einen so genannten *Vertrag* zwischen Client und Server dar.
- *Netzspezifischer Code*, der die Codierung und Übertragung von Aufrufparametern und Rückgabewerten ermöglicht, wird ab Java SE 5 dynamisch zur Laufzeit generiert.
- Um für den ersten Aufruf einer entfernten Methode eine "Referenz" auf das entfernte Objekt, das diese Methode anbietet, zu erhalten, kann der Client einen so genannten *Namensdienst* (Registry) nutzen.
- RMI bietet Mechanismen sowohl für die Übertragung von Objekten als auch für die Übertragung und das Laden des Bytecodes der zugehörigen Klassen, falls diese lokal nicht vorhanden sind.
- RMI ist eine rein Java-basierte Lösung, d. h. Client und Server müssen in Java programmiert sein.

Mit Hilfe eines *Namensdienstes* können Dienste zentral veröffentlicht werden, sodass Clients diese über ein Netz finden und nutzen können. Namensdienste ordnen den Adressen von Ressourcen (beispielsweise entfernten Objekten) eindeutige Namen zu. Die Adresse einer Ressource enthält alle Informationen, die ein Client braucht, um mit der Ressource zu kommunizieren.

Um eine "Referenz" auf die gewünschte Ressource zu erhalten, übergibt der Client dem Namensdienst den Namen, unter dem die Ressource angemeldet ist. Als Ergebnis erhält er die Referenz, mit der nun eine Verbindung zur Ressource aufgebaut werden kann.

Abbildung 5.3 zeigt den allgemeinen Fall einer mit RMI realisierten Client/Server-Anwendung.

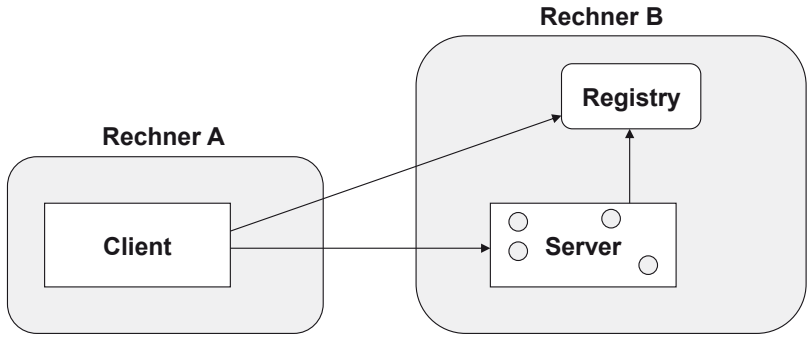

Abbildung 5-3: RMI-Anwendung

Der Client erhält über die Registry eine "Referenz" auf ein entferntes Objekt, das der Server vorher dort registriert hat. Für dieses Objekt ruft der Client eine Methode auf. Gegebenenfalls kann das RMI-System auch einen Webserver nutzen, um Bytecode für Objekte vom Server zum Client bzw. umgekehrt zu übertragen.

Die Zusammenhänge, Begriffe und die erforderlichen Klassen und Methoden zur Entwicklung einer RMI-Anwendung werden nun im Folgenden schrittweise erläutert.

Das Beispiel zeigt die Implementierung eines *Echo-Dienstes*.

Remote Interface

Das *Remote Interface* definiert die Sicht des Client auf das entfernte Objekt. Dieses Interface enthält die Methoden, die für dieses Objekt entfernt aufgerufen werden können.

Ein *Remote Interface* ist von dem Interface

```
java.rmi.Remote
```

abgeleitet: `Remote` dient dazu, Interfaces zu kennzeichnen, deren Methoden entfernt aufgerufen werden sollen. `Remote` muss von allen Interfaces erweitert werden, die entfernte Methoden deklarieren.

`java.rmi.RemoteException` ist von `java.io.IOException` abgeleitet und ist Superklasse einer Reihe von Ausnahmen, die beim Aufruf einer entfernten Methode bei Netz- bzw. Protokollfehlern ausgelöst werden können. Jede Methode eines von `Remote` abgeleiteten Interfaces (*Remote Interface*) muss diese Klasse in der `throws`-Klausel aufführen.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Echo extends Remote {
   String getEcho(String s) throws RemoteException;
}
```

Remote Object

Jedes Objekt, dessen Klasse ein Remote Interface implementiert, ist ein so genanntes entferntes Objekt (*Remote Object*), d. h. es implementiert die vorgeschriebenen entfernt aufrufbaren Methoden. Konstruktoren müssen `RemoteException` in der `throws`-Klausel enthalten. Methoden können in der Regel hierauf verzichten, falls sie selbst keine entfernten Methoden aufrufen.

Damit eine Verbindung zwischen Client und Server aufgenommen werden kann und Methoden des Objekts entfernt aufgerufen werden können, muss das entfernte Objekt *exportiert* und damit *remote-fähig* gemacht werden.

Dies geschieht durch Ableiten von der Klasse

`java.rmi.server.UnicastRemoteObject`.

Bei Erzeugung des entfernten Objekts wird der Konstruktor dieser Superklasse aufgerufen, der das Objekt *exportiert*. Das exportierte Objekt kann nun über TCP/IP eingehende Nachrichten erhalten.

Implementierung des Remote Interface

Wir nutzen die folgende Namenskonvention: Der Name der Klasse, die das Interface `Xxx` implementiert, ist `XxxImpl`.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

@SuppressWarnings("serial")
public class EchoImpl extends UnicastRemoteObject implements Echo {
   public EchoImpl() throws RemoteException {
   }

   public String getEcho(String s) {
      return s;
   }
}
```

Stub und Skeleton

Die Codierung bzw. Decodierung der Aufrufparameter und Rückgabewerte von Methodenaufrufen und die Übermittlung der Daten zwischen Client und Server wird vom RMI-System und von generierten Klassen, beim Client *Stub* und beim Server *Skeleton* genannt, geregelt (siehe Abbildung 5.4).

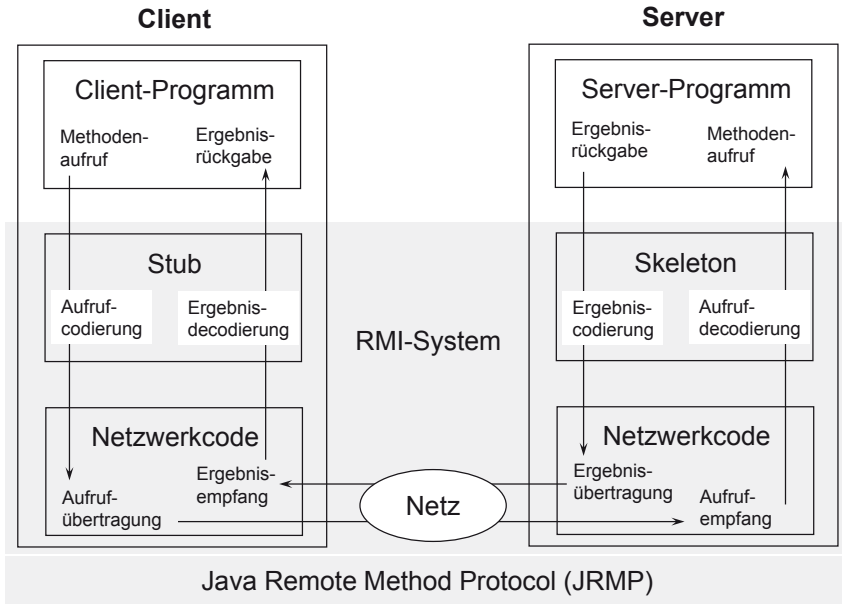

Abbildung 5-4: Aufruf einer entfernten Methode

Ein Stub-Objekt fungiert als *lokaler Stellvertreter* (Proxy) des entfernten Objekts. Ein Stub implementiert dasselbe Remote Interface, das auch die Klasse des entfernten Objekts implementiert.

Remote Reference

Ein Client erhält Zugriff auf ein entferntes Objekt durch eine so genannte *entfernte Referenz* (*Remote Reference*). Diese wird beim Erzeugen des lokalen Stub-Objekts bereitgestellt. Das Stub-Objekt kapselt Informationen, die für den Zugriff auf das entfernte Objekt benötigt werden. Der Client ruft eine Methode des Stub-Objekts auf, die dann für den Methodenaufruf des entfernten Objekts auf der Serverseite sorgt (siehe Abbildung 5.5). Kurz gesagt wird eine entfernte Referenz durch eine Referenz auf ein entsprechendes Stub-Objekt realisiert.

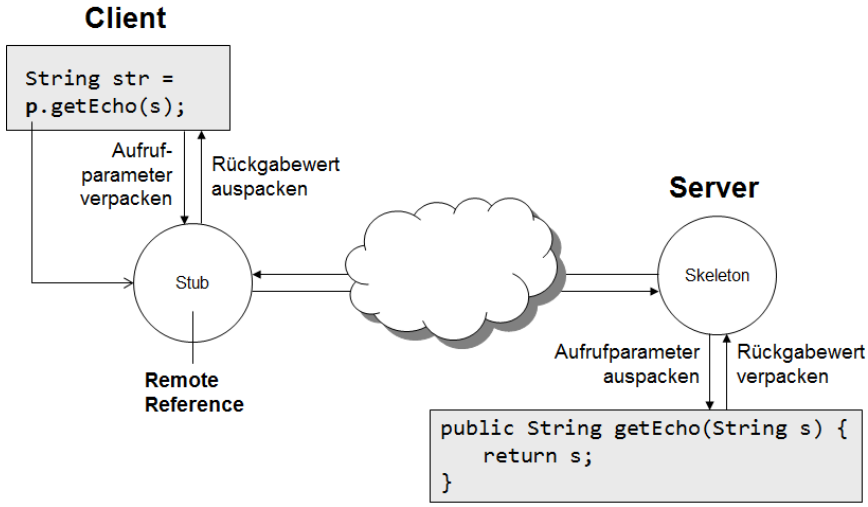

Abbildung 5-5: Remote Reference

Generierung

Das *Java Remote Method Protocol* (JRMP) wird vom Stub genutzt, um mit dem Server zu kommunizieren. Es liegt in zwei Versionen vor: 1.1 und 1.2.

- Version 1.1 kommuniziert mit einem Skeleton (wird genutzt für Clients, die unter JDK 1.1 laufen),
- Version 1.2 benötigt keine Skeleton-Klasse.

Das Tool `rmic` erzeugt Stubs und Skeletons aus den kompilierten Klassen, die die Implementierung des Remote Interface enthalten.

- `rmic -v1.1 ...` erzeugt Stubs und Skeletons für JRMP 1.1,
- `rmic -v1.2 ...` erzeugt nur Stubs für JRMP 1.2.

Ab Java SE 5 werden Stubs zur Laufzeit dynamisch generiert, sodass `rmic` nicht mehr gebraucht wird. `rmic` steht aber weiterhin zur Verfügung, um Clients unter früheren Java-Versionen zu unterstützen. Der Aufruf von `rmic` ohne weitere Option verhält sich wie `rmic -v1.2 ...` Um Stubs und Skeletons, die mit JRMP 1.1 und 1.2 kompatibel sein sollen, zu generieren, muss `rmic -vcompat ...` genutzt werden.

Die Klasse `EchoServer` (siehe unten) erzeugt ein entferntes Objekt vom Typ `EchoImpl` und registriert dieses bei einem Namensdienst (Registry).

Registry

Das vom JDK bereitgestellte Programm `rmiregistry` stellt einen einfachen Dienst zur Verfügung, der es dem Client erlaubt, eine *erste Referenz* auf ein entferntes Objekt als *Einstiegspunkt* zu erhalten. Weitere Referenzen auf andere entfernte Objekte können von hier aus dann anwendungsspezifisch, z. B. als Rückgabewerte von entfernten Methodenaufrufen, geliefert werden.

Jeder Eintrag in der Registry besteht aus einem *Namen* und einer *Objektreferenz*.

Der Name hat die Form eines URL:

```
//host:port/Dienstname
```

Bis auf *`Dienstname`* können alle Bestandteile entfallen. Der Rechnername ist in diesem Fall `localhost` und die Portnummer `1099`.

Soll für die Registry eine andere als die standardmäßig vorgesehene Portnummer `1099` benutzt werden, so muss sie als Aufrufparameter beim Start von `rmiregistry` angegeben werden.

Die Klasse `java.rmi.Naming` wird von Clients und Servern benutzt, um mit der Registry zu kommunizieren.

```
static void bind(String name, Remote obj)
       throws java.rmi.AlreadyBoundException,
       java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. `AlreadyBoundException` wird ausgelöst, wenn `name` bereits eingetragen ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

```
static void rebind(String name, Remote obj)
       throws java.net.MalformedURLException, java.rmi.RemoteException
```

registriert einen Eintrag für ein entferntes Objekt. `name` wird an `obj` gebunden. Besteht bereits ein Eintrag zu diesem Namen, so wird der bestehende Eintrag überschrieben. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

```
static void unbind(String name)
       throws java.rmi.NotBoundException, java.net.MalformedURLException,
       java.rmi.RemoteException
```

entfernt den Eintrag zu `name`. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist.

Diese drei `Naming`-Methoden können nur auf dem Rechner ausgeführt werden, auf dem auch der Namensdienst läuft.

```
import java.rmi.Naming;
import java.rmi.Remote;

public class EchoServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new EchoImpl();
      Naming.rebind("echo", remote);
      System.out.println("EchoServer gestartet ...");
   }
}
```

Das RMI-System sorgt dafür, dass der Server läuft, auch wenn alle Anweisungen der `main`-Methode ausgeführt sind.

```
import java.rmi.Naming;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      String text = args[1];

      Echo remote = (Echo) Naming.lookup("//" + host + "/echo");
      String received = remote.getEcho(text);
      System.out.println(received);
   }
}
```

Mittels der `Naming`-Methode `lookup` erhält der Client das Stub-Objekt zum entfernten Objekt, das unter dem Namen "`echo`" in der Registry eingetragen ist.

```
static Remote lookup(String name) throws java.rmi.NotBoundException,
       java.net.MalformedURLException, java.rmi.RemoteException
```

liefert die Referenz auf das Stub-Objekt für das unter `name` eingetragene entfernte Objekt. `MalformedURLException` wird ausgelöst, wenn der Aufbau von `name` nicht korrekt ist. `NotBoundException` wird ausgelöst, wenn zu `name` kein Eintrag vorhanden ist.

Test (Aufrufe jeweils im Projektverzeichnis)

Aufruf der Registry:

```
set CLASSPATH=bin
rmiregistry
```

oder

```
rmiregistry -J-Djava.rmi.server.logCalls=true
```
[1]

[1] Dies bewirkt, dass der Kommunikationsfluss zwischen Client und Server protokolliert wird.

Aufruf des Servers:

```
java -cp bin EchoServer
```

Mit der Systemeigenschaft `java.rmi.server.hostname` kann der Hostname bzw. die IP-Adresse des Servers explizit festgelegt werden.

Beispiel:

```
java -Djava.rmi.server.hostname=192.168.2.90 -cp bin EchoServer
```

Aufruf des Client:

```
java -cp bin EchoClient localhost Hallo
```

Um sich gegen Einschleusen schädlichen Codes in den Client zu schützen, kann die Ausführung von einem *Security Manager* kontrolliert werden. Hierzu muss eine *Policy-Datei* mit beispielsweise folgendem Inhalt angelegt werden:

`policy.txt`:

```
grant {
  permission java.net.SocketPermission "*:1024-", "connect";
};
```

Der Client ist dann wie folgt aufzurufen (in einer Zeile einzugeben):

```
java -Djava.security.manager -Djava.security.policy=policy.txt -cp bin
EchoClient localhost Hallo
```

Client und Server können natürlich auch auf unterschiedlichen Rechnern installiert und getestet werden.

LocateRegistry

`rmiregistry` startet die Registry auf einem Rechner, die dann für mehrere RMI-Server genutzt werden kann. Eine individuelle Registry kann aber auch innerhalb des Servers gestartet werden. Dazu steht die Klasse `LocateRegistry` im Paket `java.rmi.registry` zur Verfügung.

Die Methode

```
static Registry createRegistry(int port)
    throws java.rmi.RemoteException
```

erzeugt eine Registry die auf dem Port `port` Anfragen akzeptiert.

Wir demonstrieren den Einsatz von `LocateRegistry` und zeigen, wie die Nummer des Ports, an dem das entfernte Objekt Methodenaufrufe empfängt, vorgegeben werden kann.

Hierzu wird der folgende Konstruktor der Klasse `UnicastRemoteObject` genutzt:

```
protected UnicastRemoteObject(int port) throws RemoteException
```

Gegenüber dem vorhergehenden Beispiel werden `EchoImpl`, `EchoServer` und `EchoClient` wie folgt angepasst:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

@SuppressWarnings("serial")
public class EchoImpl extends UnicastRemoteObject implements Echo {
   public EchoImpl(int port) throws RemoteException {
      super(port);
   }

   public String getEcho(String s) {
      return s;
   }
}
```

```
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.registry.LocateRegistry;

public class EchoServer {
   public static void main(String args[]) throws Exception {
      int registryPort = Integer.parseInt(args[0]);
      int port = Integer.parseInt(args[1]);

      LocateRegistry.createRegistry(registryPort);
      Remote remote = new EchoImpl(port);
      Naming.rebind("//:" + registryPort + "/echo", remote);
      System.out.println("EchoServer gestartet ...");
   }
}
```

```
import java.rmi.Naming;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      int port = Integer.parseInt(args[1]);
      String text = args[2];

      Echo remote = (Echo) Naming.lookup("//" + host + ":" + port + "/echo");
      String received = remote.getEcho(text);
      System.out.println(received);
   }
}
```

Aufrufparameter und Rückgabewerte

Die Aufrufparameter und der Rückgabewert einer entfernten Methode können von einem einfachen Datentyp, Referenzen auf "normale" lokale Objekte oder Referenzen auf entfernte Objekte sein.

Für *entfernte Methoden* gelten die folgenden Übertragungsregeln:

Werte von einfachem Datentyp (z. B. `int`, `double`) werden wie bei lokalen Methodenaufrufen *by value* übertragen.

Lokale Objekte werden serialisiert, übertragen und vom Server deserialisiert. Dafür sorgen Stub und Skeleton. Lokale Objekte werden, anders als beim lokalen Methodenaufruf, als Kopie *by value* übertragen. Diese Objekte müssen also das Interface `java.io.Serializable` implementieren.

Exportierte *entfernte Objekte* werden *by reference* übertragen, d. h. es werden die Stub-Objekte, nicht Kopien der Originale übertragen.

Die folgende Tabelle fasst die Regeln zusammen:

Typ	lokale Methode	entfernte Methode
einfacher Typ	by value	by value
Objekt	by reference	by value (Serialisierung)
entferntes Objekt	by reference	by remote reference (Stub-Objekt)

5.3 Dienstauskunft

Das folgende Programm gibt eine Liste aller in der Registry gebundenen Namen aus.

Die `Naming`-Methode

```
static String[] list(String name)
       throws java.rmi.RemoteException, java.net.MalformedURLException
```

liefert ein Array von Namen, die an entfernte Objekte gebunden sind. `name` spezifiziert die Registry in der Form `//host:port`.

```
import java.rmi.Naming;

public class ListRegistry {
   public static void main(String args[]) throws Exception {
      String registryName = args[0];
```

```
      String list[] = Naming.list(registryName);
      for (String name : list) {
         System.out.println(name);
      }
   }
}
```

```
java -cp bin ListRegistry //localhost:1099
```

Die Ausgabe hat die Form:

//host:port/Dienstname

5.4 Transport by reference

Beim Aufruf entfernter Methoden werden lokale Objekte als Kopie in serialisierter Form übertragen. Das folgende Beispiel zeigt eine entfernte Methode, die als Rückgabewert eine *entfernte Referenz* liefert, mit deren Hilfe eine entfernte Methode eines weiteren entfernten Objekts vom Client aufgerufen werden kann.

Der RMI-Server soll Konten mit den Attributen Kontonummer (id), PIN und Saldo verwalten. Der Client kann ein neues Konto mit Angabe von Kontonummer und PIN anlegen oder ein bereits unter seiner Kontonummer eingerichtetes Konto öffnen. Für dieses Konto kann er dann einen Betrag einzahlen oder abheben sowie sich seinen Kontostand anzeigen lassen.

Für dieses Beispiel nutzen wir eine eigene `Exception`-Klasse:

KontoException

```
@SuppressWarnings("serial")
public class KontoException extends Exception {
   public KontoException() {
   }

   public KontoException(String msg) {
      super(msg);
   }
}
```

Remote Interface Konto

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Konto extends Remote {
```

```
   int getSaldo() throws RemoteException;

   void add(int betrag) throws RemoteException, KontoException;
}
```

Remote Interface KontoManager

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface KontoManager extends Remote {
   Konto getKonto(int id, int pin) throws RemoteException, KontoException;
}
```

Die entfernte Methode `getKonto` gibt eine *entfernte Referenz* auf ein `Konto`-Objekt zurück.

KontoImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

@SuppressWarnings("serial")
public class KontoImpl extends UnicastRemoteObject implements Konto {
   private int pin;
   private int saldo;

   public KontoImpl(int pin) throws RemoteException {
      this.pin = pin;
   }

   public int getSaldo() {
      return saldo;
   }

   public void add(int betrag) throws KontoException {
      if (saldo + betrag < 0) {
         throw new KontoException("Das Konto kann nicht ueberzogen werden.");
      }
      saldo += betrag;
   }

   public int getPin() {
      return pin;
   }
}
```

Wird versucht, das Konto zu überziehen, so wird eine Ausnahme vom Typ `KontoException` ausgelöst.

KontoManagerImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;

@SuppressWarnings("serial")
public class KontoManagerImpl extends UnicastRemoteObject implements
      KontoManager {

   private Hashtable<Integer, KontoImpl> hashtable;

   public KontoManagerImpl() throws RemoteException {
      hashtable = new Hashtable<Integer, KontoImpl>();
   }

   public Konto getKonto(int id, int pin) throws RemoteException,
         KontoException {

      KontoImpl konto = hashtable.get(id);
      if (konto == null) {
         konto = new KontoImpl(pin);
         hashtable.put(id, konto);
         System.out.println("Konto " + id + " wurde eingerichtet.");
         return konto;
      } else {
         if (konto.getPin() == pin)
            return konto;
         else
            throw new KontoException("PIN ist ungueltig.");
      }
   }
}
```

Die Konten werden in einer *Hashtable* unter der jeweiligen Kontonummer gespeichert. Ist bereits unter der angegebenen Kontonummer ein Konto vorhanden, so wird geprüft, ob die angegebene PIN gültig ist. Ist die PIN nicht korrekt, wird eine Ausnahme ausgelöst. Ist die Kontonummer neu, so wird ein neues Konto eingerichtet. Bei fehlerfreier Verarbeitung wird in beiden Fällen die entfernte Referenz auf ein Konto-Objekt zurückgeliefert.

BankServer

```
import java.rmi.Naming;
import java.rmi.Remote;

public class BankServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new KontoManagerImpl();
      Naming.rebind("bank", remote);
      System.out.println("BankServer gestartet ...");
   }
}
```

Das "Einstiegsobjekt" vom Typ `KontoManagerImpl` wird registriert.

BankClient

Beim Aufruf des Client werden als Parameter u. a. Kontonummer und PIN mitgegeben. Das Programm ermöglicht die Ausführung verschiedener Aktionen:

`get`	Anzeige des Saldos
`+nnn`	Betrag `nnn` einzahlen
`-nnn`	Betrag `nnn` auszahlen
`q`	Beenden

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.rmi.Naming;

public class BankClient {
   public static void main(String args[]) throws Exception {
      if (args.length != 3) {
         System.err.println("java BankClient <host> <id> <pin>");
         System.exit(1);
      }

      String host = args[0];
      int id = Integer.parseInt(args[1]);
      int pin = Integer.parseInt(args[2]);

      KontoManager manager = (KontoManager) Naming.lookup("//" + host
            + "/bank");
      Konto konto = manager.getKonto(id, pin);

      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
      String input;
      while (true) {
         try {
            System.out.println("Kommando eingeben (get | <zahl>| q):");
            input = in.readLine();
            if (input == null || input.length() == 0 || input.equals("q"))
               break;
            if (input.equals("get")) {
               System.out.println("Aktueller Kontostand: "
                     + konto.getSaldo());
            } else {
               int betrag = Integer.parseInt(input);
               konto.add(betrag);
            }
         } catch (NumberFormatException e) {
         } catch (KontoException e) {
            System.out.println(e.getMessage());
         }
      }
   }
}
```

Mit `lookup` erhält der Client Zugriff auf das entfernte `KontoManager`-Objekt. Hierfür ruft er die Methode `getKonto` auf und erhält als Rückgabewert eine entfernte Referenz auf ein `Konto`-Objekt, für das er dann die entfernten `Konto`-Methoden aufruft.

Registry und Server werden wie in Kapitel 5.2 aufgerufen.

Aufruf des Client:

```
java -cp bin BankClient localhost 1 1234
Kommando eingeben (get | <zahl>| q):
1000
Kommando eingeben (get | <zahl>| q):
get
Aktueller Kontostand: 1000
Kommando eingeben (get | <zahl>| q):
-2000
Das Konto kann nicht ueberzogen werden.
Kommando eingeben (get | <zahl>| q):
q
```

5.5 Mobile Agenten

Ein (serialisierbares) Objekt kann auch dann vom Client zum Server transportiert werden, wenn der Server den Bytecode der zugehörigen Klasse noch nicht zur Verfügung hat. Der Code muss dann ebenfalls zur Laufzeit ad hoc übertragen werden.

Wir nutzen hier diese Möglichkeit, um Methoden dieser Klasse auf dem Server *lokal* auszuführen. Derartige Nachrichten entsprechen von ihrem Wesen her einem *mobilen Agenten*, der im Auftrag eines Client bestimmte Aufgaben auf einem anderen Rechner erledigt und danach zurückkehrt.

Wir entwickeln einen Server, der beliebige Aufgaben vom Client entgegennimmt, diese ausführt und die Ergebnisse zurück liefert. Das ist z. B. dann hilfreich, wenn der Server auf einer sehr schnellen Maschine läuft und komplexe mathematische Berechnungen ausgeführt werden müssen.

Eine Aufgabe kann durch ein beliebiges Objekt repräsentiert werden, dessen Klasse das Interface `Agent` implementiert:

Agent

```
package agent;

public interface Agent extends java.io.Serializable {
   void execute();
}
```

Remote Interface ServerAgent

```
package agent;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerAgent extends Remote {
   Agent execute(Agent agent) throws RemoteException;
}
```

ServerAgentImpl

Die entfernte Methode `execute` des Remote Interface `ServerAgent` initiiert den Transport des `Agent`-Objekts und ruft die `Agent`-Methode `execute` auf:

```
package server;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import agent.Agent;
import agent.ServerAgent;

@SuppressWarnings("serial")
public class ServerAgentImpl extends UnicastRemoteObject implements ServerAgent {
   public ServerAgentImpl() throws RemoteException {
   }

   public Agent execute(Agent agent) {
      agent.execute();
      return agent;
   }
}
```

Server

```
package server;

import java.rmi.Naming;
import java.rmi.Remote;

public class Server {
   public static void main(String args[]) throws Exception {
      Remote remote = new ServerAgentImpl();
```

```
      Naming.rebind("agent", remote);
      System.out.println("Server gestartet ...");
   }
}
```

DemoAgent

Die Klasse `DemoAgent` implementiert das Interface `Agent`. Die Methode `execute` berechnet die Summe der Zahlen von `1` bis zu einer vorgegebenen Zahl `n`. Diese Klasse soll später über das Netz zum Server transportiert werden.

```
package client;

import agent.Agent;

@SuppressWarnings("serial")
public class DemoAgent implements Agent {
   private int n;
   private int sum;

   public DemoAgent(int n) {
      this.n = n;
   }

   public void execute() {
      for (int i = 1; i <= n; i++) {
         sum += i;
      }
   }

   public int getResult() {
      return sum;
   }
}
```

Client

```
package client;

import java.rmi.Naming;

import agent.Agent;
import agent.ServerAgent;

public class Client {
   public static void main(String args[]) throws Exception {
      String host = args[0];

      ServerAgent remote = (ServerAgent) Naming
            .lookup("//" + host + "/agent");
      Agent demo = new DemoAgent(100);
      DemoAgent result = (DemoAgent) remote.execute(demo);
      System.out.println(result.getResult());
   }
}
```

Zum Test werden für Client und Server verschiedene Projektverzeichnisse eingerichtet. Die Klasse `DemoAgent` ist so für den Server über seinen `CLASSPATH` nicht erreichbar.

Um den Bytecode von `DemoAgent` herunterladen zu können, nutzt der Server einen HTTP-Server. Hierzu kann ein beliebiger Webserver eingesetzt werden (siehe Kapitel 6).

Für den Start des Client ist der URL für den zu übertragenden Bytecode als Wert der Property

```
java.rmi.server.codebase
```

anzugeben.[2]

Diese Information wird zum Server übertragen, sodass dieser dann die geeignete HTTP-Anfrage stellen kann.

Der Server muss einen *Security Manager* nutzen, da das Laden von Bytecode im Allgemeinen eine unsichere Aktivität ist.

Die *Policy-Datei* `policy.txt` hat für unsere Zwecke den folgenden Inhalt:

```
grant {
  permission java.net.SocketPermission "*:1024-", "connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
};
```

Der HTTP-Server wird an die Portnummer `80` gebunden. Abbildung 5.6 zeigt die Konfiguration. Insbesondere geht aus der Abbildung hervor, welche Klassen dem Client, welche dem Server lokal zur Verfügung stehen.

Test (Aufrufe jeweils im geeigneten Projektverzeichnis)

Aufruf der Registry (im Projektverzeichnis des RMI-Servers):

```
set CLASSPATH=bin
rmiregistry
```

Aufruf des RMI-Servers (in einer Zeile):

```
java -Djava.rmi.server.useCodebaseOnly=false -Djava.security.manager
-Djava.security.policy=policy.txt -cp bin server.Server
```

Aufruf des HTTP-Servers (im Projektverzeichnis des RMI-Client):

```
java -cp bin MyWebserver
```

[2] `http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/javarmiproperties.html`

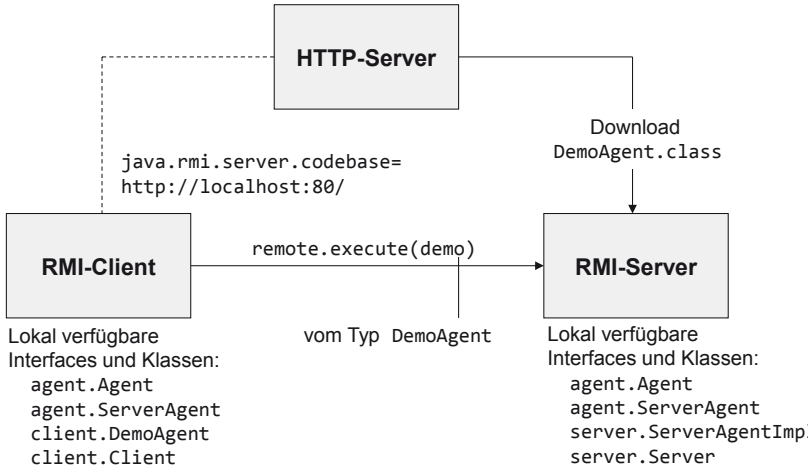

Abbildung 5-6: Dynamisches Laden einer Klasse

Aufruf des Client (in einer Zeile):

```
java -Djava.rmi.server.codebase=http://localhost:80/ -cp bin client.Client
localhost
```

Beim Test mit zwei Rechnern im Netz könnte der Aufruf wie folgt sein (Server: `192.168.2.90`, Client: `192.168.2.91`):

```
java -Djava.rmi.server.codebase=http://192.168.2.91:80/ -cp bin
client.Client 192.168.2.90
```

5.6 Callbacks

In diesem Kapitel sehen wir, dass ein Client auch zeitweise selbst Dienste anbieten kann. Der Server ruft eine entfernte Methode des Client auf.

Polling oder Callback

Eine typische Anwendungssituation ist: Der Client will Ereignisse beobachten, die auf dem Server eintreten. Statt nun regelmäßig in bestimmten Abständen eine Anfrage an den Server zu stellen, ob das interessierende Ereignis eingetreten ist oder nicht (*Polling*, *Pull-Prinzip*), lässt sich der Client vom Server über das Eintreten des Ereignisses informieren (*Callback*, *Push-Prinzip*).

Damit dieser *Callback-Mechanismus* funktioniert, muss der Client sich beim Server registrieren (der Server speichert eine *entfernte Referenz auf ein Remote-Objekt des Client*). Dann kann der Server bei Eintreten des Ereignisses eine entfernte Methode

des Client aufrufen. Bei dieser Lösung fällt im Vergleich zum Polling unnötige Rechenzeit und Netzlast weg.

Zur Veranschaulichung dieses Mechanismus entwickeln wir eine Anwendung (Client und Server), mit der Textnachrichten, die ein so genannter *Publisher* veröffentlicht, an interessierte Abonnenten (*Subscriber*) gesendet werden können. Der Server hat die Aufgabe, eine an ihn gerichtete Nachricht sofort an alle Abonnenten weiterzuleiten. Zu diesem Zweck muss er diese "kennen".

Textnachrichten werden in ein `Message`-Objekt verpackt, das zusätzlich den Zeitpunkt der Veröffentlichung enthält.

Message

```
@SuppressWarnings("serial")
public class Message implements java.io.Serializable {
   private long timestamp;
   private String text;

   public void setTimestamp(long timestamp) {
      this.timestamp = timestamp;
   }

   public long getTimestamp() {
      return timestamp;
   }

   public void setText(String text) {
      this.text = text;
   }

   public String getText() {
      return text;
   }
}
```

Das *entfernte Objekt des Servers* (vom Typ `MessageManager`) hat drei Methoden:

`void setMessageListener(MessageListener listener)`
: meldet einen Subscriber an.

`void removeMessageListener(MessageListener listener)`
: meldet einen Subscriber ab.

`void send(Message msg)`
: sendet eine Nachricht.

MessageListener

Das *entfernte Objekt des Client* (vom Typ `MessageListener`) hat die Methode:

`void onMessage(Message msg)`
gibt die Nachricht aus.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MessageListener extends Remote {
   void onMessage(Message msg) throws RemoteException;
}
```

MessageManager

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MessageManager extends Remote {
   void setMessageListener(MessageListener listener) throws RemoteException;
   void removeMessageListener(MessageListener listener) throws RemoteException;
   void send(Message msg) throws RemoteException;
}
```

Abbildung 5.7 zeigt den Zusammenhang.

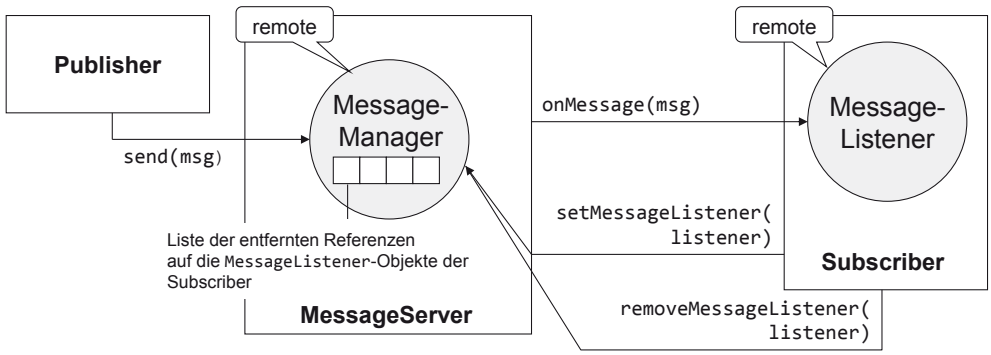

Abbildung 5-7: Callback-Mechanismus

Wir implementieren zunächst den RMI-Server. Die Liste `listeners` vom Typ `CopyOnWriteArrayList` speichert die entfernten Referenzen auf die `MessageListener`-Objekte der Abonnenten. `CopyOnWriteArrayList` verhindert Ausnahmen vom Typ `ConcurrentModificationException` während der Iteration über die Elemente der Liste. Die Methode `send` ruft für jede in der Liste gespeicherte Referenz die entfernte `MessageListener`-Methode `onMessage` auf. Wird hierbei (z. B. aufgrund des Abbruchs eines Subscribers) eine `RemoteException` ausgelöst, so wird die entsprechende Referenz aus der Liste entfernt.

Ein Thread gibt alle 5 Sekunden die Anzahl der Abonnenten am Bildschirm aus.

MessageManagerImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.List;
import java.util.Vector;
import java.util.concurrent.CopyOnWriteArrayList;

@SuppressWarnings("serial")
public class MessageManagerImpl extends UnicastRemoteObject implements
      MessageManager {

   private List<MessageListener> listeners =
                  new CopyOnWriteArrayList<MessageListener>();

   public MessageManagerImpl() throws RemoteException {
      listeners = new Vector<MessageListener>();
      new ControlThread().start();
   }

   public void setMessageListener(MessageListener listener) {
      listeners.add(listener);
   }

   public void removeMessageListener(MessageListener listener) {
      listeners.remove(listener);
   }

   public synchronized void send(Message msg) {
      for (MessageListener listener : listeners) {
         try {
            synchronized (listener) {
               listener.onMessage(msg);
            }
         } catch (RemoteException e) {
            listeners.remove(listener);
         }
      }
   }

   private class ControlThread extends Thread {
      public void run() {
         while (true) {
            try {
               Thread.sleep(5000);
            } catch (InterruptedException e) {
            }

            System.out.println("Anzahl Subscribers: " + listeners.size());
         }
      }
   }
}
```

MessageServer

```
import java.rmi.Naming;
import java.rmi.Remote;
```

```
public class MessageServer {
   public static void main(String args[]) throws Exception {
      Remote remote = new MessageManagerImpl();
      Naming.rebind("message", remote);
      System.out.println("MessageServer gestartet ...");
   }
}
```

Nun implementieren wir *Publisher* und *Subscriber*.

Publisher

```
import java.rmi.Naming;

public class Publisher {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      String text = args[1];

      MessageManager manager = (MessageManager) Naming.lookup("//" + host
            + "/message");
      Message msg = new Message();
      msg.setTimestamp(System.currentTimeMillis());
      msg.setText(text);
      manager.send(msg);
   }
}
```

MessageListenerImpl

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.text.SimpleDateFormat;
import java.util.Date;

@SuppressWarnings("serial")
public class MessageListenerImpl extends UnicastRemoteObject implements
      MessageListener {

   private SimpleDateFormat formatter;

   public MessageListenerImpl() throws RemoteException {
      formatter = new SimpleDateFormat("E, MMM d, yyyy HH:mm:ss z");
   }

   public void onMessage(Message msg) {
      String timestamp = formatter.format(new Date(msg.getTimestamp()));
      System.out.println(timestamp);
      System.out.println(msg.getText());
   }
}
```

Subscriber

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Subscriber {
   public static void main(String args[]) throws Exception {
      String host = args[0];
      int millis = Integer.parseInt(args[1]);

      final MessageManager manager = (MessageManager) Naming.lookup("//"
            + host + "/message");

      final MessageListener listener = new MessageListenerImpl();
      manager.setMessageListener(listener);

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            if (manager != null) {
               try {
                  manager.removeMessageListener(listener);
               } catch (RemoteException e) {
                  System.err.println(e);
               }
            }
         }
      });

      try {
         Thread.sleep(millis);
      } catch (InterruptedException e) {
      }

      manager.removeMessageListener(listener);
      System.exit(0);
   }
}
```

Der Subscriber wird nach einer vorgegebenen Anzahl Millisekunden beendet.

Test (Aufrufe jeweils im Projektverzeichnis)

Aufruf von Registry und Server:

```
set CLASSPATH=bin
rmiregistry
java -cp bin MessageServer
```

Aufruf zweier Abonnenten:

```
java -cp bin Subscriber localhost 60000
java -cp bin Subscriber localhost 60000
```

Aufruf eines Publishers:

```
java -cp bin Publisher localhost "Das ist ein Test"
```

5.7 RMI mit IIOP

CORBA (*Common Object Request Broker Architecture*), eine Spezifikation der *Object Management Group* (*OMG*), stellt eine Kommunikations- und Dienstinfrastruktur für verteilte objektorientierte Anwendungen bereit. Die Methodenaufrufe sind *sprachunabhängig*, d. h. Client und Server können mit unterschiedlichen Programmiersprachen implementiert werden. Schnittstellen werden mit der speziellen Beschreibungssprache *IDL* (*Interface Definition Language*) unabhängig von der Implementierungssprache definiert. Im Vergleich zu RMI ist CORBA jedoch komplizierter und aufwändiger in der Umsetzung.

CORBA nutzt das Kommunikationsprotokoll *IIOP* (*Internet Inter-ORB Protocol*) auf der Basis von TCP/IP.

Neben dem Protokoll JRMP für eine reine Java-Umgebung (siehe Kapitel 5.2) unterstützt RMI auch IIOP (*Java RMI over IIOP*) und hat damit Zugang zu anderen CORBA-Anwendungen. *Enterprise JavaBeans* (*EJB*) der Plattform Java EE unterstützen RMI/IIOP.

Im Folgenden wird gezeigt, wie eine einfache verteilte Anwendung auf der Basis von RMI mit IIOP implementiert werden kann. Analog zur RMI-Registry wird hier ein IIOP-fähiger Namensdienst, der vom *Object Request Broker Daemon* (`orbd`) angeboten wird, eingesetzt.

Das Interface LagerService

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface LagerService extends Remote {
   Lager getLager(String id) throws RemoteException;
}
```

Zu einer Artikelnummer *id* liefert die Methode `getLager` das zugehörige `Lager`-Objekt. Die Klasse `Lager` muss das Interface `java.io.Serializable` implementieren.

Die Klasse Lager

```
import java.io.Serializable;
import java.util.Date;

@SuppressWarnings("serial")
public class Lager implements Serializable {
   private String id;
   private int bestand;
   private Date datum;

   public Lager(String id, int bestand, Date datum) {
      this.id = id;
      this.bestand = bestand;
      this.datum = datum;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public int getBestand() {
      return bestand;
   }

   public void setBestand(int bestand) {
      this.bestand = bestand;
   }

   public Date getDatum() {
      return datum;
   }

   public void setDatum(Date datum) {
      this.datum = datum;
   }
}
```

Die Implementierung LagerServiceImpl

```
import java.rmi.RemoteException;
import java.util.Date;
import java.util.Hashtable;

import javax.rmi.PortableRemoteObject;

public class LagerServiceImpl extends PortableRemoteObject implements
      LagerService {

   private Hashtable<String, Lager> table;

   public LagerServiceImpl() throws RemoteException {
      table = new Hashtable<String, Lager>();
      Date datum = new Date();
```

```
      Lager[] list = new Lager[] { new Lager("4711", 100, datum),
            new Lager("4712", 80, datum), new Lager("4713", 55, datum) };
      for (int i = 0; i < list.length; i++) {
         table.put(list[i].getId(), list[i]);
      }
   }

   public Lager getLager(String id) {
      return table.get(id);
   }
}
```

Da Client und Server über IIOP kommunizieren sollen, muss die Klasse von

`javax.rmi.PortableRemoteObject`

abgeleitet werden.

Der Server LagerServer

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class LagerServer {
   public static void main(String args[]) throws Exception {
      LagerService service = new LagerServiceImpl();
      Context ctx = new InitialContext();
      ctx.rebind("LagerService", service);
      System.out.println("LagerServer gestartet ...");
   }
}
```

Das Programm erzeugt ein Service-Objekt und meldet dieses beim Namensdienst an. Auf den Namensdienst wird mit Hilfe von *JNDI* (*Java Naming and Directory Interface*) zugegriffen. Hierzu werden das Interface `Context` und die Klasse `InitialContext`, beide aus dem Paket `javax.naming`, genutzt. Der Service wird unter dem Namen "`LagerService`" eingetragen. Konkrete Angaben zum gewählten Namensdienst werden beim Aufruf des Programms über *Properties* eingestellt (siehe unten).

Der Client LagerClient

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class LagerClient {
   public static void main(String args[]) throws Exception {
      String id = args[0];

      Context ctx = new InitialContext();
```

```
      Object objref = ctx.lookup("LagerService");
      LagerService service = (LagerService) PortableRemoteObject.narrow(
            objref, LagerService.class);

      Lager lager = service.getLager(id);
      if (lager == null) {
         System.out.println("Lager nicht vorhanden");
      } else {
         System.out.println(lager.getId());
         System.out.println(lager.getBestand());
         System.out.println(lager.getDatum());
      }
   }
}
```

Der Client kontaktiert den Namensdienst und erfragt mit `lookup` das als "`LagerService`" registrierte Service-Objekt. Die gelieferte Referenz muss mit `narrow` auf den entsprechenden Interface-Typ "gecastet" werden.

Nach Compilierung der Sourcen müssen mit `rmic` ein Stub und eine so genannte *Tie-Klasse* generiert werden:

```
rmic -classpath bin -d bin -iiop LagerServiceImpl
```

Mit Hilfe der zusätzlichen Option `-idl` können nach Bedarf auch die IDL-Beschreibungen zum Interface erzeugt werden.

Der Namensdienst wird wie folgt auf Port 50000 gestartet:

```
orbd -ORBInitialPort 50000
```

Die im `bin`-Verzeichnis abgelegte Datei `jndi.properties` enthält den Namen der JNDI-Treiberklasse für den Namensdienst sowie dessen URL:

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://localhost:50000
```

Server und Client benötigen die Angaben in der Datei `jndi.properties`.

Start des Servers:

```
java -cp bin LagerServer
```

Start des Client:

```
java -cp bin LagerClient 4711
```

Zur Ausführung des Client werden die folgenden Bytecode-Dateien benötigt:

`Lager.class`,
`LagerService.class`,
`LagerClient.class` und
`_LagerService_Stub.class`.

Befinden sich Client und Server auf unterschiedlichen Rechnern im Netz, muss in der Datei `jndi.properties` die Angabe `localhost` durch den Hostnamen bzw. die IP-Adresse des Servers ersetzt werden.

5.8 Aufgaben

1. Programmieren Sie einen RMI-Dienst, dessen entfernte Methode

   ```
   String getDaytime()
   ```

   die aktuelle Systemzeit des Servers liefert.

2. Erstellen Sie für den RMI-Server aus Kapitel 5.5 einen neuen *Agenten*, der zu einer vorgegebenen Zahl `n` die Fakultät `n!` ermittelt. Zur Berechnung soll die Klasse `java.math.BigInteger` genutzt werden.

3. Entwickeln Sie nach der Vorlage in Kapitel 3.8 ein auf RMI basierendes Chat-Programm: RMI-Server, RMI-Client. Der Client soll die gleiche Funktionalität und Benutzungsoberfläche haben wie das in Kapitel 3.8 entwickelte Programm. Nutzen Sie den Callback-Mechanismus.

4. Erstellen Sie einen universellen RMI-Server (`MultiServer`), der *mehrere Dienste gleichzeitig* anbieten kann. Zur Laufzeit sollen bestehende Dienste beendet und neue Dienste hinzugefügt werden können. Ebenso soll der Server kontrolliert beendet werden können.

   Der Server soll das Interface `MultiServerManager` mit den folgenden Methoden implementieren:

   ```
   void shutdown()
   void reconfigure()
   ```

   Die neu hinzuzufügenden Dienste sind in einer Textdatei in folgender Form gespeichert (Beispiel):

   ```
   echo      EchoImpl
   daytime   DaytimeImpl
   ```

Jede Zeile entspricht einem Dienst. Sie enthält den Dienstnamen und den Namen der Klasse, die den Dienst erbringt. Diese Datei ist bei der Rekonfiguration einzulesen.

Für jeden Eintrag soll mittels Reflection (Nutzung der `Class`-Methoden `forName` und `newInstance`) ein neues entferntes Objekt erzeugt und bei der Registry angemeldet werden.

Des Weiteren ist ein Client `MultiServerManagerClient` zu erstellen, der den Aufruf der Methoden `shutdown` und `reconfigure` ermöglicht.

6 HTTP-Kommunikation im Web

Ziel dieses Kapitels ist es, eine Einführung in das *Hypertext Transfer Protocol* (*HTTP*) zu geben und einen einfachen Webserver zu entwickeln.

6.1 Das Protokoll HTTP

HTTP ist ein Protokoll der Anwendungsschicht im TCP/IP-Schichtenmodell und regelt insbesondere, wie ein Webbrowser mit einem Webserver im *World Wide Web* (*WWW*) kommuniziert. HTTP verwendet auf der Transportschicht TCP.

Damit ein Webbrowser eine Webseite im Web abrufen kann, muss er sie zunächst adressieren.

Ein *Uniform Resource Locator* (*URL*) ist eine standardisierte Adresse, mit der eine beliebige Ressource (z. B. eine HTML-Seite, ein GIF-Bild, eine PDF-Datei, ein Programm) lokalisiert werden kann.

So lokalisiert z. B. `http://www.hs-niederrhein.de` die Website der Hochschule Niederrhein.

Der URL hat im Allgemeinen den folgenden Aufbau:

```
protocol://server[:port][/resource]
```

Hierbei sind die eingeklammerten Teile optional.

Die Angaben bedeuten:

`protocol`	Protokollname, hier: `http`
`server`	Host-Name oder IP-Adresse des Servers
`port`	Portnummer, unter der der Server läuft; die standardmäßige Portnummer für einen HTTP-Server ist 80 und muss nicht angegeben werden
`resource`	Bezeichnung der Ressource, z. B. Pfad- und Dateiname

Pfad- und Dateiname müssen keine reale Datei im Dateisystem bezeichnen. Sie können auf der Serverseite als logischer Name zur eindeutigen Bezeichnung einer Ressource interpretiert werden.

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_6, © Springer Fachmedien Wiesbaden 2015

Die Interaktion zwischen HTTP-Client und HTTP-Server für eine Anfrage umfasst die folgenden Schritte:

1. Der Server wartet auf eine eingehende HTTP-Anfrage.
2. Der Client erzeugt einen URL `http://...`
3. Der Client versucht, eine TCP-Verbindung zum Server aufzubauen.
4. Der Server akzeptiert den Verbindungswunsch.
5. Der Client sendet eine Nachricht (*HTTP-Anfrage*) an den Server und fordert die Ressource mit dem spezifizierten URL an.
6. Der Server verarbeitet die Anfrage (z. B. Ausführung einer Datenbankabfrage und Generierung einer HTML-Seite mit dem Abfrageergebnis).
7. Der Server sendet eine Rückantwort (*HTTP-Antwort*) an den Client, die die angeforderte Ressource oder eine Fehlermeldung enthält.
8. Der Client verarbeitet die Antwort.
9. Der Client und/oder der Server schließen die TCP-Verbindung.

Der Server hat keine Kenntnis über vorangegangene Anfragen desselben Client. Jede Anfrage wird unabhängig von vorhergehenden Anfragen bearbeitet. HTTP ist also ein *zustandsloses Protokoll*.

Für jede HTTP-Anfrage wird bei HTTP/1.0 in der Regel eine neue TCP-Verbindung aufgebaut. Enthält z. B. eine angeforderte HTML-Seite mehrere Grafiken, so muss jede dieser Grafiken separat angefordert werden (jeweils mit Verbindungsaufbau und -abbau).

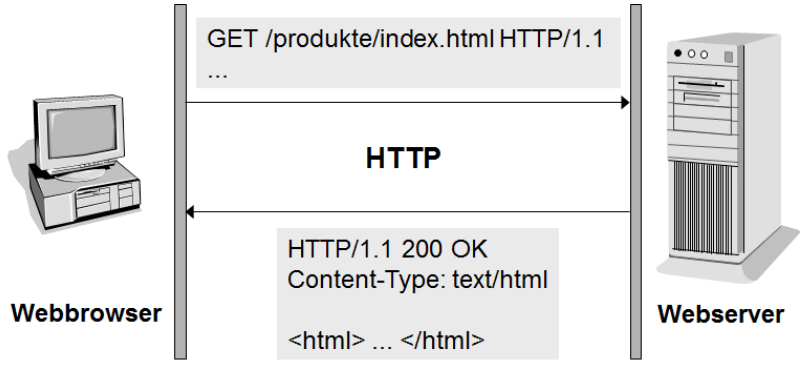

Abbildung 6-1: Eine HTTP-Transaktion

HTTP/1.0 ist im RFC 1945 der IETF spezifiziert.[1]

Die Version *HTTP/1.1* unterstützt so genannte *persistente Verbindungen*. Während die TCP-Verbindung steht, können *mehrere* HTTP-Anfragen über diese Verbindung durchgeführt werden. Die Verbindung kann vom Client oder Server abgebaut werden. HTTP/1.1 ist im RFC 2616 der IETF spezifiziert.[2]

Die neue Version HTTP/2 soll die Übertragung durch Zusammenfassung mehrerer Anfragen, Datenkompression und binäre Übertragung beschleunigen und optimieren.[3]

Das folgende Programm zeigt den Inhalt der Nachricht (HTTP-Anfrage), die ein Webbrowser an den Webserver schickt.

```
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class ShowRequest {
   public static void main(String[] args) {
      int port = Integer.parseInt(args[0]);

      try (ServerSocket server = new ServerSocket(port)) {
         while (true) {
            try (Socket client = server.accept();
                  InputStream in = client.getInputStream()) {

               client.shutdownOutput();
               int c;
               while ((c = in.read()) != -1) {
                  System.out.print((char) c);
               }

               System.out.println();
            }
         }
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Aufruf des Programms:

```
java -cp bin ShowRequest 50000
```

---

1 http: //www.ietf.org/rfc/rfc1945.txt

2 http: //www.ietf.org/rfc/rfc2616.txt

3 HTTP/2 wurde am 18. Februar 2015 von der IETF verabschiedet.

Anforderung einer fiktiven HTML-Seite im Webbrowser (hier: Firefox):

```
http://localhost:50000/abc/xyz.html
```

Ausgabe:

```
GET /abc/xyz.html HTTP/1.1
Host: localhost:50000
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gecko/20100101
Firefox/36.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Eine *HTTP-Anfrage* hat den folgenden Aufbau:

- *Kopfzeile*
  Sie enthält die *HTTP-Methode* (im Beispiel `GET`), den Namen der angeforderten Ressource ohne Protokoll und Domain-Namen (im Beispiel `/abc/xyz.html`) und die verwendete Protokollversion (im Beispiel `HTTP/1.1`).
- *Anfrageparameter (optional)*
  Anfrageparameter liefern dem Server zusätzliche Informationen über den Client und seine Anfrage. Jeder Parameter benutzt eine eigene Zeile und besteht aus dem Namen, einem Doppelpunkt und dem Wert.
- eine *Leerzeile*
- *Nutzdatenteil (optional)*
  Hier stehen z. B. die in einem Formular eingetragenen Daten bei Anwendung der HTTP-Methode POST.

Kopfzeile, Anfrageparameter und Leerzeile enden jeweils mit *Carriage Return* und *Linefeed*: `\r\n`.

Im einfachsten Fall reicht eine Anfrage der Form

```
GET /index.html HTTP/1.1
```

aus, um bereits von einem Webserver verstanden zu werden.

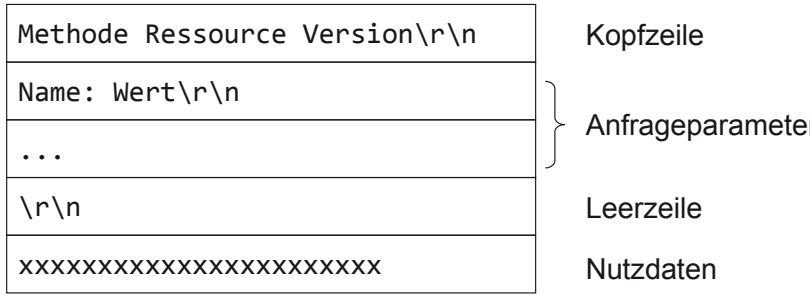

Abbildung 6-2: Aufbau der HTTP-Anfrage

Die *HTTP-Methode* spezifiziert die vom Server durchzuführende Aktion.

Wichtige HTTP-Methoden sind:

- `GET`
  Diese Methode fordert eine Ressource an.
- `POST`
  Diese Methode überträgt Benutzerdaten an den Server.
- `HEAD`
  Diese Methode fordert Informationen über die Ressource an; die Ressource selbst wird nicht benötigt.
- `PUT`
  Diese Methode wird verwendet, um eine Ressource auf dem Server zu erstellen bzw. zu ändern.
- `DELETE`
  Diese Methode wird verwendet, um eine Ressource auf dem Server zu löschen.

`PUT` und `DELETE` werden aus Sicherheitsgründen von vielen Webservern ignoriert.

Anfrageparameter können in beliebiger Reihenfolge angegeben werden. Groß- und Kleinschreibung wird ignoriert.

Die gebräuchlichsten Anfrageparameter sind (siehe obiges Testbeispiel):

- `Host`
  Rechnername und optionale Portnummer des Servers
- `User-Agent`
  Kenndaten über den HTTP-Client

- `Connection`
  Dieser Parameter wird benutzt, um eine persistente TCP-Verbindung anzufordern bzw. zu schließen.
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil
- `Accept-Language`
  Dieser Parameter gibt die vom Client bevorzugte Sprache an. Der Server kann dann z. B. eine HTML-Seite, die in mehreren Sprachvarianten vorliegt, in der vom Client gewünschten Sprache senden.
- `Accept-Encoding`
  Mit diesem Parameter gibt der Client an, welche Komprimierungsalgorithmen er versteht. Der Server kann z. B. große Dateien komprimieren, um die Übertragungszeit zu minimieren.
- `Accept`
  Dieser Parameter gibt die vom Client akzeptierten Medientypen an. Die gültigen Parameterwerte sind durch den MIME-Standard definiert.

MIME steht für den Standard *Multipurpose Internet Mail Extension*, der ursprünglich für E-Mails entworfen wurde.

MIME-Formatangaben werden von HTTP-Clients und HTTP-Servern benutzt. Clients nutzen sie, um dem Server mitzuteilen, welche Medientypen sie handhaben können. Server nutzen sie, um den Client über den Inhaltstyp der gesendeten Ressource zu informieren.

Die MIME-Formatangabe besteht aus einer Typ- und einer Subtypangabe:

```
typ/subtyp
```

Der MIME-Standard ist im RFC 1521 der IETF spezifiziert.[4]

Beispiele für Medientypen:

Typ/Subtyp	Beschreibung und übliche Erweiterung
`text/html`	HTML-Datei (*.htm, *.html)
`text/plain`	ASCII-Text (*.txt)
`text/xml`	XML-Datei (*.xml, *.dtd)
`image/gif`	GIF-Bild (*.gif)
`image/jpeg`	JPEG-Bild (*.jpeg, *.jpg)

[4] `http://www.ietf.org/rfc/rfc1521.txt`

`image/png`	PNG-Bild (*.png)
`application/pdf`	PDF-Datei (*.pdf)
`application/octet-stream`	Binärdaten (*.bin, *.exe)
`application/zip`	ZIP-Datei (*.zip)

Für nicht standardisierte Subtypen wird das Präfix `x-` benutzt, z. B. bezeichnet `audio/x-wav` Audio-Dateien `*.wav`.

Mit der HTTP-Methode `POST` können Benutzerdaten zum Server geschickt werden.

Das folgende Beispiel zeigt ein HTML-Formular, dessen Daten durch Betätigen des Buttons "Senden" zum Server gesendet werden.

HTML-Code des Formulars

```
<html>
<head><title>POST</title></head>
<body>
<form action="http://localhost:50000/xxx" method="POST">
<pre>
Artikelnummer: <input type="text" name="nr" size="5"/>
Bezeichnung:   <input type="text" name="bez" size="30"/>
Preis:         <input type="text" name="preis" size="10"/>
</pre>
Beschreibung:<br/>
<textarea name="beschr" cols="60" rows="5"></textarea>
<p/>
<input type="submit" value="Senden"/>
<input type="reset" value="Zurücksetzen"/>
</form>
</body>
</html>
```

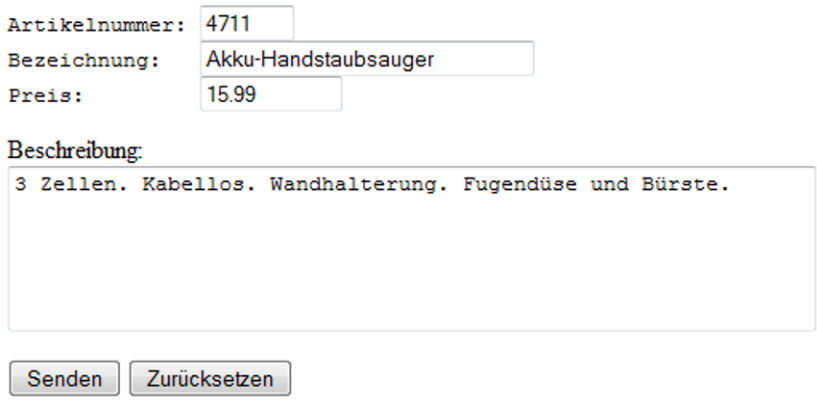

Abbildung 6-3: Ein Formular

Das Programm `ShowRequest` protokolliert:

```
POST /xxx HTTP/1.1
Host: localhost:50000
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 112

nr=4711&bez=Akku-
Handstaubsauger&preis=15.99&beschr=3+Zellen.+Kabellos.+Wandhalterung.+
Fugend%FCse+und+B%FCrste.
```

Die im Formular eingetragenen Daten werden im *URL-codierten Format* übertragen (`application/x-www-form-urlencoded`).

Die im Formular definierten Variablennamen (im Beispiel: `nr`, `bez`, `preis`, `beschr`) sind mit den vom Benutzer eingegebenen Werten verknüpft. Variable und Wert werden jeweils durch ein Gleichheitszeichen voneinander getrennt. Die einzelnen Variable/Wert-Paare sind durch das Zeichen `&` getrennt. Alle Zeichen außer `a-z`, `A-Z`, `0-9`, `.`, `-`, `*` werden zuerst in Bytes nach einem Codierungsschema (z. B. `ISO-8859-1` oder `UTF-8`) konvertiert. Jedes Byte wird dann durch ein Prozentzeichen, gefolgt vom Hexadezimalwert des Bytes dargestellt. Leerzeichen werden durch `+` ersetzt.

Formulare können auch die `GET`-Methode nutzen, um Daten zu übertragen. Die URL-codierten Daten werden nach einem Fragezeichen `?` an den URL angehängt. Im HTML-Code des obigen Formulars muss nur `POST` durch `GET` ersetzt werden. Die Kopfzeile der HTTP-Anfrage hat dann das folgende Aussehen:

```
GET /xxx?nr=4711&bez=Akku-Handstaubsauger&preis=15.99&beschr=3+Zellen.+
Kabellos.+Wandhalterung.+Fugend%FCse+und+B%FCrste. HTTP/1.1
```

Die so codierte Anfragezeichenkette nach dem Fragezeichen wird auch als *Query String* bezeichnet.

Eine *HTTP-Antwort* hat den folgenden Aufbau:

- *Kopfzeile*
  Sie besteht aus der *Protokollversion*, dem *Status-Code* und einer optionalen *Status-Meldung*.
- *Antwortparameter (optional)*
  Antwortparameter liefern dem Client zusätzliche Informationen über den Server und die Antwort.
- eine *Leerzeile*

- *Nutzdatenteil (optional)*
  Dieser enthält die angeforderte Ressource.

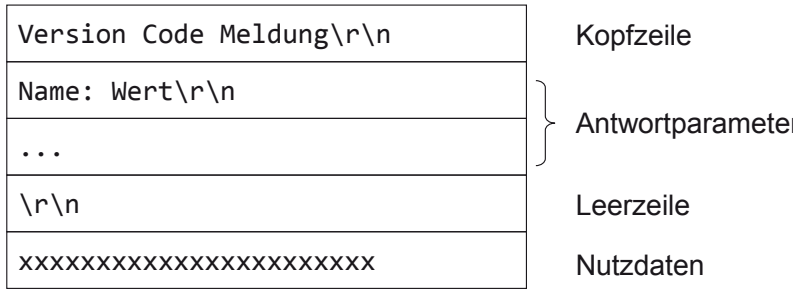

Abbildung 6-4: Aufbau der HTTP-Antwort

Beispiel:

```
HTTP/1.1 200 OK
Date: Fri, 20 Feb 2015 11:51:42 GMT
Server: Apache-Coyote/1.1
...
Content-Length: 3205
Content-Type: text/html

<html>...</html>
```

Der *Status-Code* [5] teilt dem Client mit, wie die gewünschte Aktion vom Server ausgeführt wurde.

Die Status-Codes sind wie folgt gruppiert:

`100 – 199`	Informative Meldungen
`200 – 299`	Die Anfrage war erfolgreich
`300 – 399`	Die Anfrage wurde weitergeleitet
`400 – 499`	Die Anfrage war fehlerhaft
`500 – 599`	Server-Fehler

Beispiele:

`200`	OK
`400`	Bad Request

[5] `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

`404` Not Found

`500` Internal Server Error

`501` Not Implemented

Gebräuchliche *Antwortparameter* sind:

- `Date`
  Aktuelles Datum des Servers zum Zeitpunkt der Beantwortung der Anfrage. Ein Beispiel:
  `Date: Date: Fri, 20 Feb 2015 11:51:42 GMT`
- `Server`
  Kenndaten über den HTTP-Server
- `Content-Type`
  MIME-Format der Nutzdaten
- `Content-Length`
  Länge der Daten (in Byte) im Nutzdatenteil. Werden Webseiten dynamisch erzeugt, ist die Länge oft nicht bekannt, weshalb dieser Parameter dann weggelassen wird.
- `Last-Modified`
  Zeitpunkt der letzten Änderung der Ressource. Ein Beispiel:
  `Last-Modified: Date: Fri, 20 Feb 2015 11:51:42 GMT`
- `Connection`
  Legt für diese Verbindung gewünschte Optionen fest. Ein HTTP/1.1-Server, der persistente Verbindungen nicht unterstützt, muss den Header "`Connection: close`" verwenden.

6.2 Ein einfacher Webserver

Im Folgenden entwickeln wir einen einfachen Webserver, der nur die HTTP-Methode GET versteht, Query Strings aber nicht verarbeitet.

Der Konstruktor der Klasse `SimpleWebserver` wird mit drei Parametern aufgerufen:

- die Wurzel des Webverzeichnisses, das alle abrufbaren Ressourcen enthält,
- die Portnummer und
- ein boolescher Wert, der angibt, ob Zugriffe protokolliert werden sollen.

`SimpleWebserver`-Methoden:

`void start()`
startet den Webserver.

`void shutdown()`
beendet den Webserver.

Die Bearbeitung einer HTTP-Anfrage erfolgt in einem Thread (siehe Klasse `Request`).

```
package webserver;

import java.io.File;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class SimpleWebserver extends Thread {
   private static final String LOG_FILE = "http.log";
   private static final int SOCKET_TIMEOUT = 30000;
   private File root;
   private boolean log;
   private ServerSocket serverSocket;

   public SimpleWebserver(File root, int port, boolean log) throws IOException {
      this.root = root.getCanonicalFile();
      this.log = log;

      if (!root.isDirectory()) {
         throw new IOException("No directory");
      }

      if (log)
         HttpLog.initializeLogger(new File(LOG_FILE));

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            shutdown();
         }
      });

      serverSocket = new ServerSocket(port);
   }

   @Override
   public void run() {
      while (true) {
         try {
            Socket socket;
            try {
               socket = serverSocket.accept();
            } catch (SocketException e) {
               break;
            }

            socket.setSoTimeout(SOCKET_TIMEOUT);

            Request request = new Request(socket, root, log);
            request.start();
         } catch (IOException e) {
```

```
            System.err.println(e);
         }
      }
   }

   public void shutdown() {
      try {
         if (serverSocket != null) {
            serverSocket.close();
         }

         if (HttpLog.logger != null) {
            HttpLog.logger.close();
         }
      } catch (IOException e) {
      }
   }
}
```

Ein *Shutdown hook* sorgt für das ordnungsgemäße Beenden des Servers, auch wenn das Programm mit `Strg + C` abgebrochen wird.

In Eclipse (zumindest bei Version "Luna Release 4.4.0") wird der Shutdown-Hook nicht gestartet, wenn das Programm über den Terminate-Button beendet wird. Dieser Button "killt" den JVM-Prozess direkt, sodass der Shutdown hook nicht mehr ausgeführt werden kann.

```
package webserver;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.SocketTimeoutException;
import java.net.URLConnection;
import java.net.URLDecoder;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class Request extends Thread {
   private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
         "dd.MM.yyyy HH:mm:ss");
   private File root;
   private boolean log;
   private Socket socket;

   public Request(Socket socket, File root, boolean log) {
      this.socket = socket;
      this.root = root;
```

```
      this.log = log;
   }

   @Override
   public void run() {
      try (BufferedReader in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
            BufferedOutputStream out = new BufferedOutputStream(
                  socket.getOutputStream())) {

         String request = in.readLine();
         if (request == null || request.trim().length() == 0)
            return;

         if (log) {
            HttpLog.logger.log("[" + simpleDateFormat.format(new Date())
                  + " " + socket.getInetAddress().getHostAddress() + ":"
                  + socket.getPort() + "] " + request);
         }

         // Nur die HTTP-Methode GET ist implementiert.
         if (!request.startsWith("GET")) {
            sendError(out, Status.NOT_IMPLEMENTED);
            return;
         }

         String path = getPath(request);
         File file = new File(root, URLDecoder.decode(path, "UTF-8"))
               .getCanonicalFile();

         if (file.isDirectory()) {
            File indexFile = new File(file, "index.html");
            if (indexFile.exists() && !indexFile.isDirectory()) {
               file = indexFile;
            } else {
               sendError(out, Status.FORBIDDEN);
               return;
            }
         }

         // Zugriff außerhalb von root ist nicht erlaubt.
         if (!file.getCanonicalPath().startsWith(root.getCanonicalPath())) {
            sendError(out, Status.FORBIDDEN);
            return;
         }

         if (!file.exists()) {
            sendError(out, Status.NOT_FOUND);
            return;
         }

         try (InputStream is = new BufferedInputStream(new FileInputStream(
               file))) {

            String contentType = URLConnection.getFileNameMap()
                  .getContentTypeFor(file.getName());

            if (contentType == null) {
               contentType = "application/octet-stream";
            }
```

```
            sendHeader(out, Status.OK, contentType, file.length(),
                  file.lastModified());

            byte[] buffer = new byte[8192];
            int bytesRead;
            while ((bytesRead = is.read(buffer)) != -1) {
               out.write(buffer, 0, bytesRead);
            }
         }

         out.flush();
      } catch (SocketTimeoutException e) {
      } catch (IOException e) {
         System.err.println(e);
      }
   }

   private void sendError(BufferedOutputStream out, Status status)
         throws IOException {

      String msg = status.getMessage();
      sendHeader(out, status, "text/html", msg.length(),
            System.currentTimeMillis());
      out.write(msg.getBytes());
      out.flush();
   }

   private void sendHeader(BufferedOutputStream out, Status status,
         String contentType, long length, long time) throws IOException {

      String header =
            "HTTP/1.1 " + status.getCode() + " " + status.getMessage()
            + "\r\nDate: " + getTime(System.currentTimeMillis())
            + "\r\nServer: SimpleWebserver"
            + "\r\nContent-Type: " + contentType
            + "\r\nContent-Length: " + length
            + "\r\nLast-Modified: " + getTime(time)
            + "\r\nConnection: close"
            + "\r\n\r\n";

      out.write(header.getBytes());
   }

   private static String getPath(String request) {
      return request.substring(4, request.length() - 9);
   }

   private static String getTime(long time) {
      Calendar calendar = Calendar.getInstance();
      calendar.setTimeInMillis(time);
      SimpleDateFormat dateFormat = new SimpleDateFormat(
            "EEE, dd MMM yyyy HH:mm:ss 'GMT'", Locale.US);
      dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
      return dateFormat.format(calendar.getTime());
   }
}
```

Die einzelne HTTP-Anfrage wird in der `run`-Methode der von Thread abgeleiteten Klasse Request bearbeitet. Die Kopfzeile der Anfrage wird gelesen und analysiert. Nur Anfragen mit der HTTP-Methode GET werden berücksichtigt.

Die Methode `getPath` extrahiert den Pfad-/Dateinamen `xxx` aus der Kopfzeile:

```
GET /xxx HTTP/1.1
```

Ein URL-codierter Pfadname (siehe Kapitel 6.1) wird mit Hilfe der Klassenmethode `decode` der Klasse `java.net.URLDecoder` decodiert und dieser dann mit dem Wurzelverzeichnis verbunden.

Handelt es sich bei dieser File-Ressource um ein Verzeichnis und enthält dieses die Datei `index.html`, so wird der Inhalt von `index.html` gesendet.

Der Versuch, mit Hilfe von Angaben wie `../` innerhalb des Pfadnamens auf Dateien außerhalb des Wurzelverzeichnisses zuzugreifen, führt zu einer Fehlermeldung.

Zur Bestimmung des Content-Type in der HTTP-Antwort wird die Methode

```
URLConnection.getFileNameMap().getContentTypeFor(...)
```

aufgerufen.

Die abstrakte Klasse `java.net.URLConnection` ist die Superklasse derjenigen Klassen, die eine Netzverbindung zu einer durch den URL adressierten Ressource repräsentieren.

`static FileNameMap getFileNameMap()`
: lädt die Tabelle der MIME-Typen aus einer Datei wie folgt: Existiert die System-Property `-Dcontent.types.user.table=<dateiname>`, wird `<dateiname>` geladen. Ansonsten wird die Datei `lib/content-types.properties` im JRE-Verzeichnis geladen.

Das Interface `java.net.FileNameMap` enthält die Methode

```
String getContentTypeFor(String fileName)
```

Sie liefert den MIME-Typ zu einem Dateinamen.

`Status` vom Typ `enum` enthält die hier benötigten Fehlercodes.

```
package webserver;

public enum Status {
   OK(200, "OK"), FORBIDDEN(403, "Forbidden"), NOT_FOUND(404, "Not Found"),
         NOT_IMPLEMENTED(501, "Not Implemented");

   private int code;
   private String message;

   private Status(int code, String message) {
      this.code = code;
      this.message = message;
   }
```

```
   public int getCode() {
      return code;
   }

   public String getMessage() {
      return message;
   }
}
```

```
package webserver;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class HttpLog {
   public static HttpLog logger;
   private BufferedWriter logfileWriter;

   private HttpLog(File logfile) throws IOException {
      logfileWriter = new BufferedWriter(new FileWriter(logfile, true));
   }

   public static void initializeLogger(File logfile) throws IOException {
      if (logger == null)
         logger = new HttpLog(logfile);
   }

   public void log(String info) throws IOException {
      synchronized (logfileWriter) {
         logfileWriter.write(info);
         logfileWriter.newLine();
         logfileWriter.flush();
         System.out.println(info);
      }
   }

   public void close() {
      try {
         logfileWriter.flush();
         logfileWriter.close();
      } catch (IOException e) {
      }
   }
}
```

```
import java.io.File;
import java.io.IOException;

import webserver.SimpleWebserver;

public class MyWebserver {
   private static final String ROOT = "web";
   private static final int PORT = 80;
   private static final boolean LOG = true;

   public static void main(String[] args) {
```

```
      try {
         SimpleWebserver server = new SimpleWebserver(new File(ROOT), PORT,
               LOG);
         server.start();
         System.out.println("MyWebserver gestartet ...");
         System.out
               .println("Eingabe von RETURN oder Strg + C stoppt den Server.");
         System.in.read();
         System.exit(0);
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Aufruf des Servers (in einer Zeile einzugeben):

```
java -cp bin MyWebserver
```

bzw.

```
java -Dcontent.types.user.table=content-types.properties -cp bin
MyWebserver
```

Alternative Implementierung

Java SE enthält seit Version 6 ein einfaches API zur Entwicklung von HTTP-Servern (Paket `com.sun.net.httpserver`).

Die abstrakte Klasse `com.sun.net.httpserver.HttpServer` besitzt die statische Methode `create`, mit der eine konkrete Implementierung bereitgestellt wird:

```
static HttpServer create(InetSocketAddress addr, int backlog)
       throws IOException
```

`addr` ist die Socket-Adresse. `backlog` bestimmt die maximale Länge der Warteschlange (siehe Kapitel 3.2).

Methoden der Klasse `HttpServer`:

```
abstract HttpContext createContext(String path, HttpHandler handler)
```

liefert eine Instanz vom Typ `com.sun.net.httpserver.HttpContext`, die eine Verbindung eines Pfads (z. B. "`/`") mit einer Instanz vom Typ des Interfaces `com.sun.net.httpserver.HttpHandler`, die die HTTP-Anfrage bearbeitet, festlegt.

Für einen HTTP-Server können mehrere Kontexte (mit unterschiedlichen Handlern) eingerichtet werden.

Ein vom Browser übermittelter URL wird auf den so genannten Kontext-Pfad "bestmöglich" abgebildet.

Beispiele:

Kontext 1: `/`

Kontext 2: `/demo/`

`http://localhost/` wird abgebildet auf Kontext 1

`http://localhost/demo/xyz` wird abgebildet auf Kontext 2

`http://localhost/webapp` wird abgebildet auf Kontext 1

`abstract void setExecutor(Executor executor)`
: legt das `java.util.concurrent.Executor`-Objekt fest. `Executor` ist Superinterface von `ExecutorService` (siehe Kapitel 3.4). Alle HTTP-Anfragen werden diesem Objekt zur Ausführung übergeben.

`abstract void start()`
: startet den Server.

`abstract void stop(int delay)`
: stoppt den Server, wobei maximal `delay` Sekunden auf die Beendigung der laufenden Anfragen gewartet wird.

```
package webserver;

import java.io.File;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.util.concurrent.Executors;

import com.sun.net.httpserver.HttpServer;

@SuppressWarnings("restriction")
public class SimpleWebserver {
   private static final String LOG_FILE = "http.log";
   private HttpServer server;

   public SimpleWebserver(File root, int port, boolean log) throws IOException {
      server = HttpServer.create(new InetSocketAddress(port), 0);
      server.createContext("/", new SimpleHttpHandler(root, log));
      server.setExecutor(Executors.newCachedThreadPool());

      if (log)
         HttpLog.initializeLogger(new File(LOG_FILE));

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            shutdown();
         }
      });
   }

   public void start() {
      if (server != null)
         server.start();
   }
```

```
    public void shutdown() {
       if (server != null) {
          server.stop(0);
       }

       if (HttpLog.logger != null) {
          HttpLog.logger.close();
       }
    }
}
```

Die Klasse `SimpleHttpHandler` implementiert `HttpHandler` und nutzt diverse Methoden der Klasse `HttpExchange`.

Ein Handler, der die HTTP-Anfrage bearbeitet, muss das Interface `com.sun.net.httpserver.HttpHandler` implementieren.

`HttpHandler` enthält die Methode:

```
void handle(HttpExchange exchange) throws IOException
```

Die abstrakte Klasse `com.sun.net.httpserver.HttpExchange` kapselt HTTP-Anfrage und -Antwort und stellt insbesondere die folgenden Methoden zur Verfügung:

`abstract String getRequestMethod()`
liefert die HTTP-Methode.

`abstract URI getRequestURI()`
liefert den *Uniform Resource Identifier* (*URI*) der Anfrage, eine Verallgemeinerung des *Uniform Resource Locator* (*URL*).

Die `java.net.URI`-Methode

```
String getPath()
```

liefert den Pfad des URI als Zeichenkette.

`abstract InetSocketAddress getRemoteAddress()`
liefert die Socket-Adresse des Client.

`abstract Headers getResponseHeaders()`
liefert eine Map zur Speicherung der Antwortparameter.

In dieser Map (`com.sun.net.httpserver.Headers`) können mit Hilfe der Methode

```
void add(String key, String value)
```

Parameter aufgenommen werden.

`abstract void sendResponseHeaders(int rCode, long responseLength) throws IOException`
sendet die Kopfzeile mit dem Status-Code `rCode` und die Antwortparameter. `responseLength` gibt die Anzahl Bytes der Nutzdaten an (0 = nicht festgelegt).

`abstract OutputStream getResponseBody()`
liefert den Datenstrom, in den die Nutzdaten geschrieben werden.

`java.net.HttpURLConnection` ist Subklasse von `URLConnection` und enthält insbesondere die HTTP-Status-Codes.

```
package webserver;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URLConnection;
import java.net.URLDecoder;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

import com.sun.net.httpserver.Headers;
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;

@SuppressWarnings("restriction")
public class SimpleHttpHandler implements HttpHandler {
   private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat(
         "dd.MM.yyyy HH:mm:ss");
   private File root;
   private boolean log;

   public SimpleHttpHandler(File root, boolean log) throws IOException {
      this.root = root.getCanonicalFile();
      this.log = log;

      if (!this.root.isDirectory()) {
         throw new IOException("No directory");
      }
   }

   @Override
   public void handle(HttpExchange exchange) throws IOException {
      if (log) {
         HttpLog.logger.log("[" + simpleDateFormat.format(new Date()) + " "
               + exchange.getRemoteAddress() + "] "
               + exchange.getRequestMethod() + " "
```

```
            + exchange.getRequestURI());
      }

      String method = exchange.getRequestMethod();
      if (!method.equals("GET")) {
         sendError(exchange, HttpURLConnection.HTTP_NOT_IMPLEMENTED,
               "Not Implemented");
         return;
      }

      URI uri = exchange.getRequestURI();
      String path = uri.getPath();

      File file = new File(root, URLDecoder.decode(path, "UTF-8"))
            .getCanonicalFile();

      if (file.isDirectory()) {
         File indexFile = new File(file, "index.html");
         if (indexFile.exists() && !indexFile.isDirectory()) {
            file = indexFile;
         } else {
            sendError(exchange, HttpURLConnection.HTTP_FORBIDDEN,
                  "Forbidden");
            return;
         }
      }

      if (!file.getCanonicalPath().startsWith(root.getCanonicalPath())) {
         sendError(exchange, HttpURLConnection.HTTP_FORBIDDEN, "Forbidden");
         return;
      }

      if (!file.exists()) {
         sendError(exchange, HttpURLConnection.HTTP_NOT_FOUND, "Not Found");
         return;
      }

      try (InputStream is = new FileInputStream(file);
            OutputStream os = exchange.getResponseBody()) {

         String contentType = URLConnection.getFileNameMap()
               .getContentTypeFor(file.getName());

         if (contentType == null) {
            contentType = "application/octet-stream";
         }

         sendHeader(exchange, HttpURLConnection.HTTP_OK, contentType,
               file.length(), file.lastModified());

         byte[] buffer = new byte[8192];
         int bytesRead;
         while ((bytesRead = is.read(buffer)) != -1) {
            os.write(buffer, 0, bytesRead);
         }

         os.flush();
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }
```

```
   private void sendError(HttpExchange exchange, int status, String msg)
         throws IOException {

      sendHeader(exchange, status, "text/html", msg.length(),
            System.currentTimeMillis());
      try (OutputStream os = exchange.getResponseBody()) {
         os.write(msg.getBytes());
         os.flush();
      }
   }

   private void sendHeader(HttpExchange exchange, int status,
         String contentType, long length, long time) throws IOException {

      Headers headers = exchange.getResponseHeaders();
      headers.add("Date", getTime(System.currentTimeMillis()));
      headers.add("Server", "com.sun.net.httpserver.HttpServer");
      headers.add("Content-Type", contentType);
      headers.add("Last-Modified", getTime(time));
      exchange.sendResponseHeaders(status, length);
   }

   private static String getTime(long time) {
      Calendar calendar = Calendar.getInstance();
      calendar.setTimeInMillis(time);
      SimpleDateFormat dateFormat = new SimpleDateFormat(
            "EEE, dd MMM yyyy HH:mm:ss 'GMT'", Locale.US);
      dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
      return dateFormat.format(calendar.getTime());
   }
}
```

```
package webserver;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class HttpLog {

   // wie oben

}
```

```
import java.io.File;
import java.io.IOException;

import webserver.SimpleWebserver;

public class MyWebserver {
   private static final String ROOT = "web";
   private static final int PORT = 80;
   private static final boolean LOG = true;

   public static void main(String[] args) {
      try {
         SimpleWebserver server = new SimpleWebserver(new File(ROOT), PORT, LOG);
```

```
            server.start();
            System.out.println("MyWebserver gestartet ...");
            System.out.
                    println("Eingabe von RETURN oder Strg + C stoppt den Server.");
            System.in.read();
            System.exit(0);
         } catch (IOException e) {
            System.err.println(e.getMessage());
         }
      }
   }
```

In Eclipse kann die Fehlermeldung "Access restriction ..." wie folgt ausgeschaltet werden:

```
Window > Preferences > Java > Compiler > Errors/Warnings
```

Unter "Deprecated and restricted API" ist die Option "Forbidden reference (access rules)" auf "Warning" oder "Ignore" zu setzen.

Eclipse IDE for Java EE Developers enthält einen *TCP/IP-Monitor*, der zwischen Client und Server platziert werden kann und die Daten, die über eine TCP/IP-Verbindung gesendet werden, anzeigt. Der Client verbindet sich mit dem Monitor. Dieser leitet die Daten weiter an den Server und protokolliert die Ein- und Ausgabe.

Der Monitor kann über

```
Window > Show View > Other... > Debug > TCP/IP-Monitor
```

eingerichtet und gestartet werden.

6.3 Protokollierung von HTTP-Nachrichten

Das Programm `Monitor` kann genutzt werden, um die zwischen Client und Server ausgetauschten Daten aufzuzeichnen.

Ein Shutdown hook sorgt u. a. für das ordnungsgemäße Schließen der Log-Dateien, wenn das Programm mit `Strg + C` beendet wird.

In Eclipse (zumindest bei Version "Luna Release 4.4.0") wird der Shutdown-Hook nicht gestartet, wenn das Programm über den Terminate-Button beendet wird. Dieser Button "killt" den JVM-Prozess direkt, sodass der Shutdown hook nicht mehr ausgeführt werden kann.

`Monitor` sollte also in der Eingabeaufforderung bzw. Shell ausgeführt werden.

Die folgende Abbildung skizziert die Architektur des Programms.

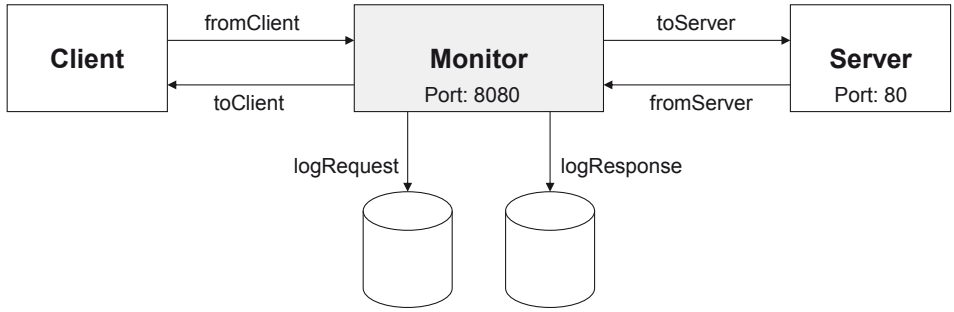

Abbildung 6-5: Monitor zeichnet HTTP-Anfragen und -Antworten auf

Das Programm `Monitor` fungiert hier als *Proxy*. Dieser leitet die Datenströme weiter an den Server (die Ausgabe des Client als Eingabe für den Server) bzw. an den Client (die Ausgabe des Servers als Eingabe für den Client). Anfragen und Antworten werden in getrennten Dateien protokolliert.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class Monitor extends Thread {
   private static OutputStream logRequest;
   private static OutputStream logResponse;
   private static ServerSocket srv;

   public static void main(String[] args) {
      int localPort = Integer.parseInt(args[0]);
      final String host = args[1];
      final int serverPort = Integer.parseInt(args[2]);
      String request = args[3];
      String response = args[4];

      Runtime.getRuntime().addShutdownHook(new Thread() {
         public void run() {
            try {
               if (logRequest != null) {
                  logRequest.close();
               }
               if (logResponse != null) {
                  logResponse.close();
               }
               if (srv != null)
                  srv.close();
            } catch (IOException e) {
            }
         }
```

```
      });

      try {
         logRequest = new FileOutputStream(request);
         logResponse = new FileOutputStream(response);
         srv = new ServerSocket(localPort);

         while (true) {
            Socket client = srv.accept();
            new ConnectionHandler(client, host, serverPort, logRequest,
                  logResponse).start();

         }
      } catch (SocketException e) {
      } catch (IOException e) {
         System.err.println("Monitor::main: " + e);
      }
   }
}
```

Die run-Methode von `ConnectionHandler` verarbeitet die Eingabe eines Client. `StreamThread` kopiert den Eingabestrom in den Ausgabestrom sowie in Log-Datei jeweils für eine Richtung (vom Client zum Server bzw. umgekehrt). Die beiden run-Methoden der Threads `forward` und `reverse` laufen quasi gleichzeitig.

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class ConnectionHandler extends Thread {
   private static final int SOCKET_TIMEOUT = 30000;

   private Socket client;
   private String host;
   private int serverPort;
   private OutputStream logRequest;
   private OutputStream logResponse;

   public ConnectionHandler(Socket client, String host, int serverPort,
         OutputStream logRequest, OutputStream logResponse) {

      this.client = client;
      this.host = host;
      this.serverPort = serverPort;
      this.logRequest = logRequest;
      this.logResponse = logResponse;
   }

   @Override
   public void run() {
      try (InputStream fromClient = client.getInputStream();
            OutputStream toClient = client.getOutputStream();
            Socket server = new Socket(host, serverPort);
            InputStream fromServer = server.getInputStream();
            OutputStream toServer = server.getOutputStream()) {
```

```
            client.setSoTimeout(SOCKET_TIMEOUT);
            server.setSoTimeout(SOCKET_TIMEOUT);

            Thread forward = new StreamThread(fromClient, toServer, logRequest);
            Thread reverse = new StreamThread(fromServer, toClient, logResponse);

            forward.start();
            reverse.run();

            forward.join();
         } catch (InterruptedException e) {
         } catch (IOException e) {
            System.err.println("ConnectionHandler::run: " + e);
         }
      }
   }
```

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.SocketException;

public class StreamThread extends Thread {
   private static final int BUFFER_SIZE = 8192;

   private InputStream from;
   private OutputStream to;
   private OutputStream log;

   public StreamThread(InputStream from, OutputStream to, OutputStream log) {
      this.from = from;
      this.to = to;
      this.log = log;
   }

   @Override
   public void run() {
      try {
         byte[] buffer = new byte[BUFFER_SIZE];
         int bytesRead;
         while ((bytesRead = from.read(buffer)) != -1) {
            to.write(buffer, 0, bytesRead);
            to.flush();
            log.write(buffer, 0, bytesRead);
            log.flush();
         }
      } catch (SocketException e) {
      } catch (IOException e) {
         System.err.println("StreamThread::run " + e);
      } finally {
         try {
            from.close();
            to.close();
         } catch (IOException e) {
         }
      }
   }
}
```

Das Programm kann wie folgt gestartet werden:

```
java -cp bin Monitor localPort remoteHost remotePort logRequest
logResponse
```

Beispiel:

```
java -cp bin Monitor 8080 localhost 80 logRequest.txt logResponse.txt
```

6.4 Aufgaben

1. Entwickeln Sie einen Client, mit dem Dateien vom Webserver mittels HTTP herunter geladen werden können. Nutzen Sie hierzu die Socket-Programmierung mit TCP/IP (siehe Kapitel 3).

2. Realisieren Sie eine Variante zum Downloadprogramm in Aufgabe 1, indem Sie die folgenden Methoden der Klassen `URL` und `URLConnection` benutzen.

   Die Klasse `java.net.URL` repräsentiert einen *Uniform Resource Locator*.

   `URL(String spec) throws MalformedURLException`
   erzeugt ein `URL`-Objekt aus der Zeichenkette `spec`. `java.net.MalformedURLException` ist von `java.io.IOException` abgeleitet.

   Die `URL`-Methode
   `URLConnection openConnection() throws IOException`
   liefert ein `java.net.URLConnection`-Objekt, das eine Verbindung zu einer durch den URL adressierten Ressource repräsentiert.

   `URLConnection`-Methoden:

   `int getContentLength()`
   liefert den Wert des Antwortparameters `Content-Length`.

   `InputStream getInputStream() throws IOException`
   liefert einen Eingabestrom zum Lesen über diese Verbindung.

3. Entwickeln Sie einen HTTP-Server, der als einzigen Dienst nach Verbindungsaufnahme mit einem Browser immer dieselbe PDF-Datei sendet. Zu diesem Zweck muss er also die HTTP-Anfrage des Client gar nicht auswerten. Der Server wird mit der Portnummer und dem Dateinamen als Parameter aufgerufen. Die HTTP-Antwort besteht aus der Kopfzeile mit Status-Code 200, den beiden Antwortparametern `Content-Type` und `Content-Length`, einer Leerzeile und dem Inhalt der Datei.

4. Entwickeln Sie einen HTTP-Server, der einen User auf eine andere Website umleitet (Redirection).

Beispiel:

Start des Servers:
```
java -cp bin Redirector 80 http://de.wikipedia.org/wiki
```

URL im Browser:
```
http://localhost/HTTP-Statuscode
```

Die Weiterleitung erfolgt hier an:
```
http://de.wikipedia.org/wiki/HTTP-Statuscode
```

Nutzen Sie hierzu in der HTTP-Antwort den Status "307 Temporary Redirect" und den `Location`-Header, im Beispiel:

```
Location: http://de.wikipedia.org/wiki/HTTP-Statuscode
```

7 Bidirektionale Kommunikation mit WebSocket

Bei einer reinen HTTP-Verbindung wird jede HTTP-Anfrage mit einer HTTP-Antwort abgeschlossen (siehe Kapitel 6). Anfrage und Antwort bestehen jeweils aus Header-Informationen und Nutzdaten. Jede vom Server an den Client gesendete Nachricht erfordert also eine vorhergehende Anfrage des Client. Die Kommunikation wird also stets vom Client initiiert.

Viele modere Webanwendungen, z. B. Anzeige von Börsenkursen, Verkehrsinformationen, Online-Spiele) müssen aber auf Ereignisse, die beim Server eintreten, reagieren. Hier ist eine vom Server initiierte Übertragung (*Server-Push*) erforderlich. Bisher wurden hierfür proprietäre Verfahren eingesetzt wie z. B. *Long Polling*. Beim Long Polling antwortet der Server verzögert auf eine vom Client initiierte Anfrage. Er antwortet erst, wenn eine Nachricht vorliegt oder eine gewisse Zeitspanne verstrichen ist.

7.1 Das WebSocket-Protokoll

Das *WebSocket*-Protokoll[1] ist ein auf TCP basierendes Netzwerkprotokoll, das eine Ergänzung zu HTTP darstellt. Der Client startet wie bei HTTP eine Anfrage. Die zugrundeliegende TCP/IP-Verbindung bleibt aber nach der Übertragung der Client-Daten bestehen. Client und Server können nun jederzeit Nachrichten übertragen (bidirektionale Kommunikation). Ein ständiger Verbindungsauf- und -abbau entfällt. Zudem entfallen die HTTP-Header-Informationen bei jeder Anfrage und Antwort.

Das WebSocket-Protokoll besteht aus zwei Phasen: Handshake und Datenverbindung.

Der *Handshake* wird vom Client eingeleitet. Er leitet einen Protokollwechsel ein (Upgrade). Die zufällig generierte Zeichenkette `Sec-WebSocket-Key` dient zur Überprüfung (Challenge-Response-Verfahren), ob beide Seiten den Wechsel wollen.

Der Server wendet eine beiden Seiten bekannte Operation auf den Wert von `Sec-WebSocket-Key` an und erzeugt damit den Wert von `Sec-WebSocket-Accept`. Der Client führt dieselbe Operation aus. Stimmt das Ergebnis mit dem vom Server gesendeten Wert überein, ist die Verbindung erfolgreich hergestellt.

---

[1] `http://www.ietf.org/rfc/rfc6455.txt`

D. Abts, *Masterkurs Client/Server-Programmierung mit Java*,
DOI 10.1007/978-3-658-09921-3_7, © Springer Fachmedien Wiesbaden 2015

Daten werden in Frames transportiert. Jeder Frame enthält einen 2 - 14 Byte langen Header und die Nutzdaten. Der Header besteht u. a. aus dem Operationscode und der Angabe zur Länge der Nutzdaten.

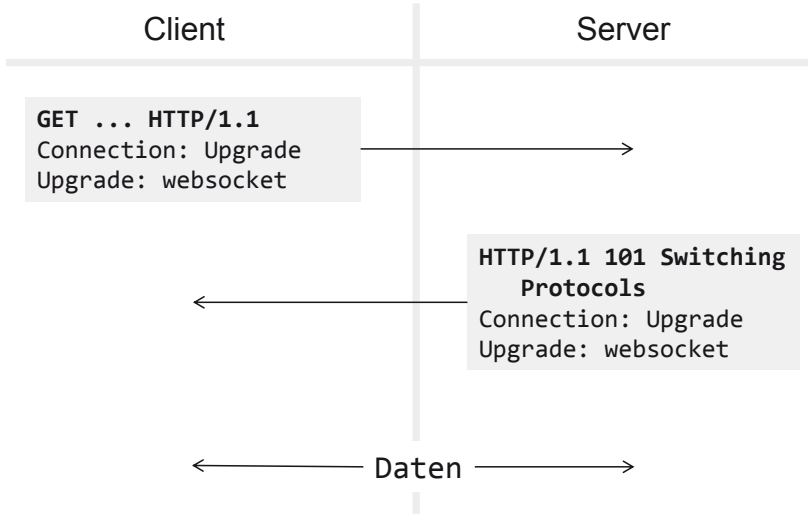

Abbildung 7-1: Ablauf des WebSocket-Protokolls

WebSocket Handshake

```
GET /websocket/echo HTTP/1.1
Host: localhost
Sec-WebSocket-Version: 13
Origin: http://localhost
Sec-WebSocket-Key: CKpmXgNe1Hsp+XBQeDZo/w==
Connection: keep-alive, Upgrade
Upgrade: websocket
```

```
HTTP/1.1 101 Switching Protocols
Server: Apache-Coyote/1.1
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept: QVrIEBDw5cqV1FEBC3VuyLnNmU4=
```

Analog zu HTTP hat der URL den folgenden Aufbau:

```
ws://server[:port][/resource]
```

Die führenden Browser unterstützen das WebSocket-Protokoll im Rahmen von HTML5. [2]

7.2 Implementierung einer einfachen WebSocket-Anwendung

Der Java-Webserver *Apache Tomcat* unterstützt das WebSocket-Protokoll. Wir nutzen hier die Tomcat-Version 8.x mit dem API Java WebSocket 1.1 (JSR-356). Die Dokumentation hierzu ist im Internet verfügbar. [3]

Wir setzen für das Folgende den grundsätzlichen Umgang mit Tomcat voraus. Alles Weitere zur Implementierung und Konfiguration wird hier behandelt.

Beispielhaft wollen wir einen einfachen Echo-Dienst realisieren.

Die Klasse `websocket.echo.Echo` implementiert diesen Dienst.

```
package websocket.echo;

import java.io.IOException;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class Echo {
   private Session session;
   private int counter;

   @OnOpen
   public void open(Session session) {
      System.out.println(session.getId() + ": onOpen");
      this.session = session;
      try {
         session.getBasicRemote().sendText("Verbindung wurde hergestellt.");
      } catch (IOException e) {
      }
   }

   @OnClose
   public void close() {
      System.out.println(session.getId() + ": onClose");
   }

   @OnError
   public void error(Throwable error) {
```

[2] http://html5test.com

[3] http://tomcat.apache.org/tomcat-8.0-doc/websocketapi/index.html

```
        System.out.println(session.getId() + ": " + error);
    }

    @OnMessage
    public void message(String msg) {
        try {
            if (session.isOpen()) {
                System.out.println(msg);
                session.getBasicRemote().sendText(++counter + ". " + msg);
            }
        } catch (IOException e) {
            try {
                session.close();
            } catch (IOException e1) {
            }
        }
    }
}
```

Die Tomcat-Installation enthält die jar-Datei `websocket-api.jar` im Verzeichnis `<Tomcat-Installationsort>/lib`. Diese muss sich im Klassenpfad befinden.

Die Annotation `ServerEndpoint` legt den Pfad fest, mit dem die Anwendung aufgerufen werden kann. Die mit den Annotationen `OnOpen`, `OnClose`, `OnError` und `OnMessage` versehenen Callback-Methoden werden beim Öffnen einer neuen WebSocket-Session, beim Schließen der Session, beim Auftreten eines Fehlers (z. B. bei Verbindungsproblemen) und beim Empfang von Daten aufgerufen.

Das Interface `Session` repräsentiert eine Kommunikationsverbindung zwischen Client und Server.

`Session`-Methoden:

`String getId()`
: liefert die eindeutige Session-Id.

`boolean isOpen()`
: prüft, ob der Socket offen ist.

`void close()`
: schließt die Verbindung.

`RemoteEndpoint.Basic getBasicRemote()`
: liefert ein Objekt vom Typ `RemoteEndpoint.Basic`. Dieses Interface repräsentiert die Gegenseite der Verbindung.

Die `RemoteEndpoint.Basic`-Methode

```
void sendText(String text) throws IOException
```

sendet eine Textnachricht.

Für *jede neue Verbindung* erzeugt Tomcat eine neue Instanz der Klasse `Echo`, sodass clientspezifische Status-Informationen in dieser Instanz gespeichert werden können.

Außer Textdaten können auch Binärdaten (z. B. als `byte`-Array) empfangen und gesendet werden. Diese Möglichkeit betrachten wir hier nicht.

Wir testen die Funktionalität mit Hilfe des Browsers und einer HTML-Seite, die JavaScript-Code enthält.

```
<html>
<head>
<title>EchoClient</title>
<script>
   var uri = 'ws://' + window.location.host + '/websocket/echo';
   var socket;

   function $(a) {
      return document.getElementById(a);
   }

   function sendMessage() {
      socket.send($('userInput').value);
   }

   window.onload = function() {
      if (! 'WebSocket' in window) {
         $('status').innerHTML = 'Der Browser unterstützt keine WebSockets.';
         return;
      }

      socket = new WebSocket(uri);

      socket.onopen = function() {
         $('status').innerHTML = 'onOpen';
      };

      socket.onclose = function() {
         $('status').innerHTML = 'onClose';
      };

      socket.onerror = function(error) {
         $('status').innerHTML = error;
      };

      socket.onmessage = function(message) {
         $('data').innerHTML = message.data;
      };
   }

   window.onunload = function() {
      socket.close();
   }
</script>
</head>

<body>
   <input id="userInput" type="text" />
```

```
    <button onclick="sendMessage();">Senden</button>
    <p><div id="status"></div></p>
    <p><div id="data"></div></p>
</body>
</html>
```

Beim Laden der HTML-Seite werden die verschiedenen CallBack-Funktionen registriert. Die Ereignisse `onopen`, `onclose`, `onerror` und `onmessage` haben die gleiche Bedeutung wie oben.

Deployment

`<Tomcat-Installationsort>/conf/Catalina/localhost` enthält die XML-Datei `websocket.xml` mit dem folgenden Inhalt:

```
<Context docBase="<Pfad-zum-Projekt>/ws/WebContent" reloadable="true" />
```

`docBase` verweist auf das Verzeichnis, das die Webanwendung (Klasse `Echo` und HTML-Seite `echo.html`) enthält.

Dieses Verzeichnis hat die folgende Struktur:

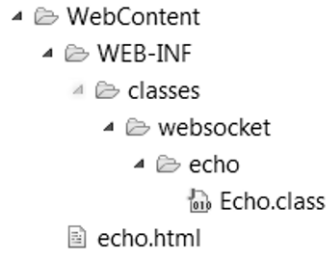

Abbildung 7-2: Inhalt von WebContent

Das Verzeichnis `WebContent/WEB-INF/classes` enthält den Bytecode (mit Paketnamen als Unterverzeichnisse).

Tomcat wird mit dem Skript `startup.bat` bzw. `startup.sh` im Verzeichnis `<Tomcat-Installationsort>/bin` gestartet.

Der Name der oben aufgeführten XML-Datei bestimmt den Kontext der Webanwendung.

Im Browser wird nun der URL aufgerufen:

```
http://localhost:8080/websocket/echo.html
```

Zur Entwicklung des Java-Client benutzen wir den *Tyrus Standalone Client* (hier in der Version 1.10). *Tyrus* ist die Open-Source-Referenzimplementierung für *JSR 356 - Java API for WebSocket*. [4]

Die jar-Datei `tyrus-standalone-client-1.10.jar` muss sich im Klassenpfad der Anwendung befinden.

```
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

import javax.websocket.ClientEndpoint;
import javax.websocket.DeploymentException;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;

import org.glassfish.tyrus.client.ClientManager;

@ClientEndpoint
public class EchoClient {
   private static final String URI = "ws://localhost:8080/websocket/echo";
   private Session session;

   @OnOpen
   public void open(Session session) {
      System.out.println("onOpen");
      this.session = session;
   }

   @OnClose
   public void close() {
      System.out.println("onClose");
   }

   @OnMessage
   public void message(String message) {
      System.out.println(message);
   }

   public void sendMessage(String message) throws IOException {
      if (session.isOpen())
         session.getBasicRemote().sendText(message);
   }

   public static void main(String[] args) {
      ClientManager clientManager = ClientManager.createClient();
      EchoClient client = new EchoClient();

      try {
         Session session = clientManager.connectToServer(client, new URI(URI));
```

[4] https://tyrus.java.net, https://tyrus.java.net/apidocs/1.10/

```
            for (int i = 0; i < 10; i++) {
               client.sendMessage(String.valueOf(Math.random()));
               Thread.sleep(1000);
            }

            session.close();
            clientManager.shutdown();
         } catch (DeploymentException | IOException | URISyntaxException
               | InterruptedException e) {
            System.out.println(e.getMessage());
         }
      }
   }
```

Die Annotation `ClientEndpoint` kennzeichnet die Klasse als WebSocket-Client.

Die `ClientManager`-Klassenmethode

```
ClientManager createClient()
```

erzeugt eine `ClientManager`-Instanz.

`ClientManager`-Methoden:

```
Session connectToServer(Class annotatedEndpointClass, URI path)
       throws DeploymentException, IOException
```

stellt die Verbindung zum Server her und liefert im Erfolgsfall eine `Session`-Instanz.

```
void shutdown()
```

gibt die zugeordneten Ressourcen frei.

Aufruf des Client:

```
java -cp bin;<Pfad>/tyrus-standalone-client-1.10.jar EchoClient
```

7.3 Server Push

Das Beispiel in diesem Kapitel demonstriert ein serverseitiges Push-Verfahren. Clients nehmen Verbindung mit dem Server auf und erhalten vom Server automatisch die Nachrichten, die beim Server eingegangen sind.

Ablagesystematik und Deployment sind analog zum letzten Abschnitt.

```
package websocket.news;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
```

```
import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/news")
public class News {
   private static List<Session> connections =
                     new CopyOnWriteArrayList<Session>();

   static {
      new NewsThread().start();
   }

   @OnOpen
   public void open(Session session) {
      System.out.println(session.getId() + ": onOpen");
      connections.add(session);
   }

   @OnClose
   public void close(Session session) {
      System.out.println(session.getId() + ": onClose");
      connections.remove(session);
   }

   @OnError
   public void error(Session session, Throwable error) {
      System.out.println(session.getId() + ": onError");
   }

   private static class NewsThread extends Thread {
      @Override
      public void run() {
         int counter = 0;
         SimpleDateFormat dateFormat = new SimpleDateFormat(
               "dd.MM.yyyy HH:mm:ss");

         System.out.println("NewsThread gestartet.");

         while (true) {
            int z = 5 + (int) (Math.random() * 6);
            try {
               Thread.sleep(1000 * z);
            } catch (InterruptedException e) {
            }

            broadcast("[" + dateFormat.format(new Date()) + "] News "
                  + ++counter);
         }
      }

      private void broadcast(String msg) {
         for (Session session : connections) {
            try {
               synchronized (session) {
                  session.getBasicRemote().sendText(msg);
               }
            } catch (IOException e) {
```

```
                connections.remove(session);
                try {
                   session.close();
                } catch (IOException e1) {
                }
             }
          }
       }
    }
}
```

Mit dem Laden der Klasse `News` wird ein Thread gestartet, der in zufälligen Zeitabständen neue Nachrichten generiert. Die bei Herstellung der Verbindung mit dem Server erzeugten Session-Instanzen werden in einer (statischen) Liste vom Typ `CopyOnWriteArrayList` gespeichert. `CopyOnWriteArrayList` verhindert Ausnahmen vom Typ `ConcurrentModificationException` während der Iteration über die Elemente der Liste. Jeder einzelne Client ist durch eine `Session`-Instanz repräsentiert. Der Server sendet bei Eintreffen einer neuen Nachricht, diese an alle Clients (Methode `broadcast`).

```
<html>
<head>
<title>NewsClient</title>
<script>
   var uri = 'ws://' + window.location.host + '/websocket/news';
   var socket;

   function $(a) {
      return document.getElementById(a);
   }

   function showMessage(text) {
      var element = $('message');
      element.innerHTML = element.innerHTML + '<br />' + text;
   }

   window.onload = function() {
      if (! 'WebSocket' in window) {
         showMessage('Der Browser unterstützt keine WebSockets.');
         return;
      }

      socket = new WebSocket(uri);

      socket.onopen = function() {
         showMessage('onOpen');
      };

      socket.onclose = function() {
         showMessage('onClose');
      };

      socket.onerror = function(error) {
         showMessage(error);
      };
```

```
      socket.onmessage = function(message) {
         showMessage(message.data);
      };
   }

   window.onunload = function() {
      socket.close();
   }
</script>
</head>

<body>
   <div id="message"></div>
</body>
</html>
```

```
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;

import javax.websocket.ClientEndpoint;
import javax.websocket.DeploymentException;
import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;

import org.glassfish.tyrus.client.ClientManager;

@ClientEndpoint
public class NewsClient {
   private static final String URI = "ws://localhost:8080/websocket/news";

   @OnOpen
   public void open() {
      System.out.println("onOpen");
   }

   @OnClose
   public void close() {
      System.out.println("onClose");
   }

   @OnMessage
   public void message(String message) {
      System.out.println(message);
   }

   public static void main(String[] args) {
      ClientManager client = ClientManager.createClient();
      NewsClient endpoint = new NewsClient();

      try {
         Session session = client.connectToServer(endpoint, new URI(URI));
         System.out.println("Eingabe von RETURN beendet den Client.");
         System.in.read();
         session.close();
```

```
            client.shutdown();
        } catch (DeploymentException | IOException | URISyntaxException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Aufruf des Client:

```
java -cp bin;<Pfad>/tyrus-standalone-client-1.10.jar NewsClient
```

7.4 Eine Chat-Anwendung

Mit dem Chat-Client (Browser) ist das so genannte "Chatten" mit mehreren Teilnehmern im Netz möglich. Der Teilnehmer kann sich an- und abmelden und Textzeilen an alle anderen aktiven Teilnehmer senden. Der Chat-Server registriert die angemeldeten Teilnehmer, repräsentiert durch Session-Instanzen, und verteilt eingehende Nachrichten (siehe Methode `message`) an alle registrierten Teilnehmer (Methode `broadcast`). Die Session-Instanzen werden wieder wie im letzten Beispiel in einer (statischen) Liste vom Typ `CopyOnWriteArrayList` gespeichert.

Ablagesystematik und Deployment sind wieder wie bisher.

```
package websocket.chat;

import java.io.IOException;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

import javax.websocket.OnClose;
import javax.websocket.OnError;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/chat")
public class Chat {
   private static List<Session> connections =
                       new CopyOnWriteArrayList<Session>();
   private static int id;
   private Session session;
   private String user;

   public Chat() {
      synchronized (getClass()) {
         user = "User " + ++id;
      }
   }
```

```
    @OnOpen
    public void open(Session session) {
       this.session = session;
       connections.add(session);
       login();
    }

    @OnClose
    public void close() {
       connections.remove(session);
       logout();
    }

    @OnError
    public void error(Throwable error) {
       System.out.println(error.getMessage());
    }

    @OnMessage
    public void message(String msg) {
       broadcast(user + ": " + msg);
    }

    private void login() {
       broadcast("Login: " + user);
       System.out.println("Login: " + user);
       System.out.println("Anzahl User: " + connections.size());
    }

    private void logout() {
       broadcast("Logout: " + user);
       System.out.println("Logout: " + user);
       System.out.println("Anzahl User: " + connections.size());
    }

    private void broadcast(String msg) {
       for (Session session : connections) {
          if (session.isOpen()) {
             try {
                synchronized (session) {
                   session.getBasicRemote().sendText(msg);
                }
             } catch (IOException e) {
                connections.remove(session);

                try {
                   session.close();
                } catch (IOException e1) {
                }

                logout();
             }
          }
       }
    }
}
```

```
<html>
<head>
<title>ChatClient</title>
<script>
   var uri = 'ws://' + window.location.host + '/websocket/chat';
   var socket;

   function $(a) {
      return document.getElementById(a);
   }

   function showMessage(text) {
      var element = $('message');
      element.innerHTML = element.innerHTML + '<br />' + text;
   }

   function sendMessage() {
      if ($('userInput').value.trim().length == 0)
         return;

      socket.send($('userInput').value);
   }

   window.onload = function() {
      if (! 'WebSocket' in window) {
         showMessage('Der Browser unterstützt keine WebSockets.');
         return;
      }

      socket = new WebSocket(uri);

      socket.onopen = function() {
         showMessage('Sitzung gestartet');
      };

      socket.onclose = function() {
         showMessage('Sitzung beendet');
      };

      socket.onerror = function(error) {
         showMessage(error);
      };

      socket.onmessage = function(message) {
         showMessage(message.data);
      };
   }

   window.onunload = function() {
      socket.close();
   }
</script>
</head>

<body>
   <input id="userInput" type="text" size="80" />
   <button onclick="sendMessage();">Senden</button>
   <p><div id="message"></div></p>
</body>
</html>
```

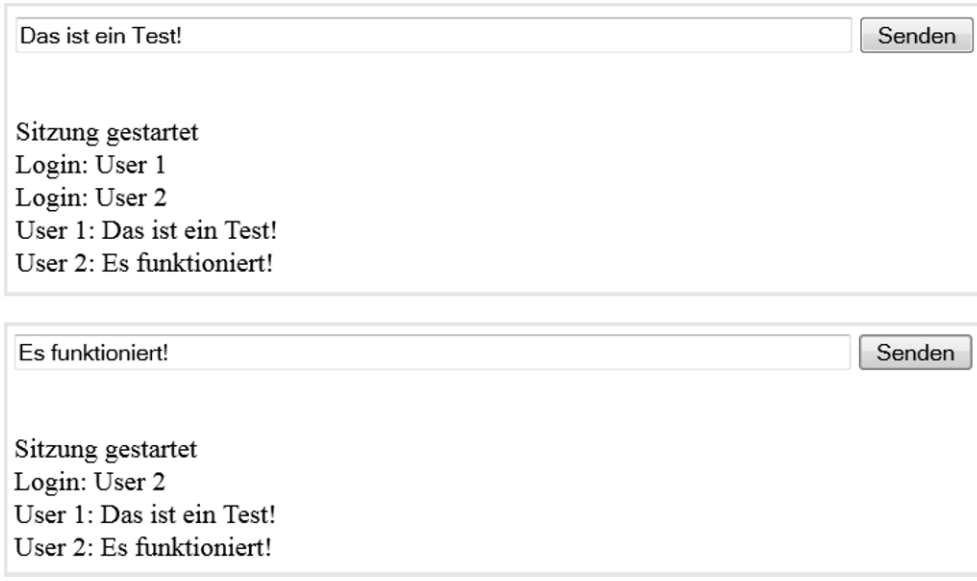

Abbildung 7-3: Zwei Verbindungen zum WebSocket-Chat-Server

7.5 Aufgaben

1. Entwickeln Sie für den Chat-Server in Kapitel 7.4 einen Java-Client mit grafischer Oberfläche (siehe hierzu auch Kapitel 3.8).

2. Entwickeln Sie analog zu Kapitel 7.3 einen Push-Dienst mit Hilfe von *Apache Tomcat* und *ActiveMQ* (siehe Kapitel 4). Nachrichten werden vom Server über JMS empfangen (Nachrichtenkanal: Topic) und sofort an die aktiven Clients übertragen. Implementieren Sie einen Topic-Publisher, der Nachrichten über JMS an den MOM-Server (ActiveMQ) sendet sowie eine HTML-Seite mit JavaScript und einen Java-Client, die die erhaltenen Nachrichten anzeigen.

   Die `ServerEndpoint`-Klasse `Push` speichert die aktiven Sessions, bietet die Callback-Methoden und die statische Methode `broadcast` analog zum Beispiel in Kapitel 7.3 an.

   Die Klasse `Consumer` implementiert das Interface `javax.jms.MessageListener`, dessen Methode `onMessage` die erhaltene Nachricht mit `Push.broadcast` verteilt. `Consumer` ist ein Thread, in dessen `run`-Methode die Verbindung zum MOM-Server gestartet wird. Der Thread wartet dann solange, bis er beim Herunterfahren per `notify` aufgeweckt wird und sich beendet:

```
synchronized (this) {
   try {
       wait();
   } catch (InterruptedException e) {
   }
}
```

Die Klasse `PushContextListener` implementiert das Interface `javax.servlet.ServletContextListener`.

In der Methode `contextInitialized` werden die Verbindungsparameter aus der Datei `web.xml` eingelesen, der `Consumer` erzeugt, die `Consumer`-Methode `init` für den JNDI-Lookup aufgerufen sowie schließlich der `Consumer` als Thread gestartet.

In der Methode `contextDestroyed` wird die `Consumer`-Methode `shutdown` aufgerufen, die den Thread beendet (Aufruf von `notify`) und die JMS-Ressourcen schließt.

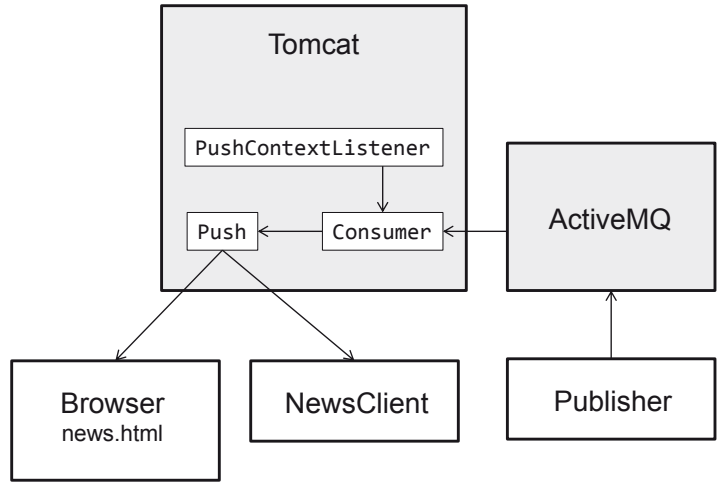

Abbildung 7-4: Integration von WebSocket-Server und MOM-Server

Die Datei `WebContent/WEB-INF/web.xml` hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">

   <context-param>
    <param-name>java.naming.factory.initial</param-name>
       <param-value>
          org.apache.activemq.jndi.ActiveMQInitialContextFactory
```

```
        </param-value>
    </context-param>

    <context-param>
        <param-name>java.naming.provider.url</param-name>
        <param-value>tcp://localhost:61616</param-value>
    </context-param>

    <context-param>
     <param-name>topic.myTopic</param-name>
        <param-value>demo.wstopic</param-value>
    </context-param>

    <listener>
        <listener-class>websocket.push.PushContextListener</listener-class>
    </listener>
</web-app>
```

Für die Compilierung sind die folgenden jar-Dateien in den Klassenpfad einzubinden:

`activemq-all-5.11.1.jar` (aus ActiveMQ)
`websocket-api.jar` (aus Tomcat)
`servlet-api.jar` (aus Tomcat)

Das Verzeichnis `WebContent/WEB-INF/lib` muss `activemq-all-5.11.1.jar` enthalten.

8 XML Remote Procedure Call (XML-RPC)

Eine bewährte und sehr verbreitete Technik zur Entwicklung von Client-Server-Anwendungen ist der so genannte *Remote Procedure Call* (RPC). Der Client ruft mit einem lokalen Prozeduraufruf einen Dienst auf, der in der Regel auf einem anderen Rechner angeboten wird (siehe auch Kapitel 5). Implementierungen des RPC-Mechanismus unterscheiden sich für verschiedene Programmiersprachen.

In diesem Kapitel wird am Beispiel von *XML-RPC* gezeigt, wie das Protokoll HTTP und XML als Kommunikationsformat für den entfernten Prozeduraufruf eingesetzt werden können. Vom Konzept her ist diese Technik programmiersprachenunabhängig.

XML-RPC ist der Vorläufer von *SOAP*, das vom World Wide Web Consortium (W3C) definiert wird und neben anderen Standardtechnologien zur Entwicklung von so genannten *Web Services* genutzt wird (siehe Kapitel 9). SOAP war ehemals Abkürzung für "Simple Object Access Protocol" und ist heute ein eigenständiger Name.

Im Vergleich zu SOAP hat XML-RPC einen geringeren Leistungsumfang, ist aber auch wesentlich einfacher in der Handhabung.

8.1 Grundkonzept und erstes Beispiel

XML-RPC ist ein einfaches Protokoll für den entfernten Prozeduraufruf (RPC). XML-RPC verwendet XML als Datenaustauschformat. *Anfrage* (Methodenname, Parameter) und *Antwort* (Rückgabewert des Methodenaufrufs) werden als XML-Dokumente mit HTTP übermittelt.

Abbildung 8.1 zeigt den Nutzdatenteil der HTTP-Anfrage (HTTP-Methode ist POST) und den Nutzdatenteil der HTTP-Antwort beim Aufruf der entfernten Methode `getEcho`.

Die Nutzdatenteile werden auch als *XML-RPC-Anfrage* bzw. *-Antwort* bezeichnet.

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_8, © Springer Fachmedien Wiesbaden 2015

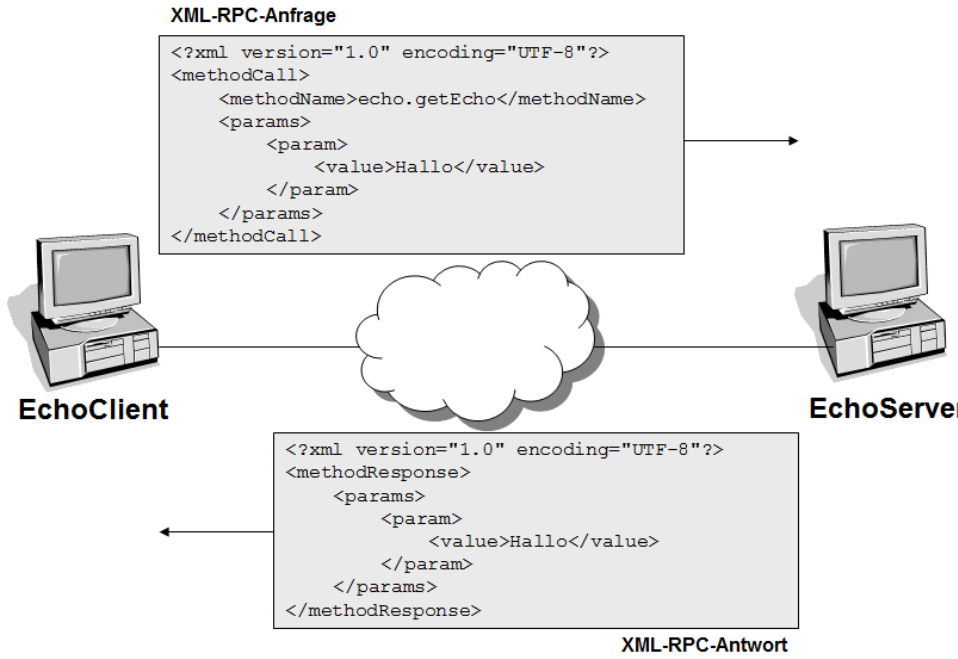

Abbildung 8-1: XML-RPC-Anfrage und -Antwort

Eine XML-RPC-Anfrage kann *mehrere* Parameterwerte enthalten (gemäß der Signatur der entfernten Methode). Eine XML-RPC-Antwort enthält jedoch *genau einen* Wert.

Struktur der XML-RPC-Anfrage:

```
<methodCall>
  <methodName>Dienstname.Methodenname</methodName>
  <params>
    <param>
      <value>Parameterwert</value>
    </param>
    ...
  </params>
</methodCall>
```

Struktur der XML-RPC-Antwort:

```
<methodResponse>
  <params>
    <param>
      <value>Rückgabewert</value>
    </param>
  </params>
</methodResponse>
```

Parameter- und Rückgabewerte haben jeweils einen bestimmten Datentyp. XML-RPC unterstützt insgesamt acht Datentypen: *einfache Datentypen* (z. B. Zeichenketten, Zahlen) und *zusammengesetzte Datentypen* (z. B. Arrays). Die Datentypen werden im nächsten Abschnitt einzeln anhand von Programmbeispielen vorgestellt.

Die *Spezifikation von XML-RPC* wurde 1999 von Dave Winer veröffentlicht. [1]

Es existieren zahlreiche Implementierungen des Verfahrens für diverse Programmiersprachen. Client und Server, in unterschiedlichen Sprachen implementiert, können miteinander kommunizieren.

Wir nutzen im Folgenden die Java-Implementierung *Apache XML-RPC* der Apache Software Foundation in der Version 3.1.3. [2]

WebServer und XmlRpcServer

Zur Implementierung eines *XML-RPC-Servers* stellt Apache XML-RPC die Klassen

`org.apache.xmlrpc.webserver.WebServer` und
`org.apache.xmlrpc.server.XmlRpcServer`

zur Verfügung.

`WebServer` implementiert einen speziell für die Behandlung von XML-RPC-Anfragen geeigneten HTTP-Server, der in eigene Applikationen eingebettet werden kann. Dieser ist zu Testzwecken sehr gut geeignet. Für höhere Ansprüche in Bezug auf Performance und Stabilität sind ausgereifte Servlet-Container wie beispielsweise *Apache Tomcat* besser geeignet.

Die Klasse `XmlRpcServer` verarbeitet die XML-RPC-Anfragen.

`WebServer(int port)`
erzeugt einen Server mit der Portnummer `port`.

`WebServer`-Methoden:

`void start() throws java.io.IOException`
startet den Server.

`void shutdown()`
stoppt den Server.

Die Methode `getXmlRpcServer()` liefert ein `XmlRpcServer`-Objekt.

---

[1] `http://www.xmlrpc.com`

[2] `http://archive.apache.org/dist/ws/xmlrpc/binaries/`

Handler

Ein so genannter *Handler* ist eine entfernt aufrufbare Methode. Folgende Bedingungen müssen erfüllt sein:

- Die Methode muss `public` sein, nicht `static` sein und darf nicht den Rückgabetyp `void` haben.
- In der Klasse, die den Handler implementiert, muss der parameterlose Standardkonstruktor implizit oder explizit vorhanden sein.

Die Klasse `org.apache.xmlrpc.server.PropertyHandlerMapping` kann mehrere Handler verwalten.

Ihre Methode

```
void addHandler(String key, Class typ) throws XmlRpcException
```

fügt die Handler-Klasse `typ` mit dem Namen `key` hinzu. Hierzu wird das `Class`-Objekt der Handler-Klasse angegeben: *`Klassenname`*`.class`.

Eine Ausnahme vom Typ `org.apache.xmlrpc.XmlRpcException` wird generell ausgelöst, wenn der Server einen Fehler meldet.

`void removeHandler(String key)`
entfernt alle Handler mit dem Namen key.

Mit dem Aufruf der `XmlRpcServer`-Methode `setHandlerMapping` werden die Handler für das `XmlRpcServer`-Objekt registriert:

```
setHandlerMapping(propertyHandlerMapping)
```

Das folgende Beispiel demonstriert eine einfache XML-RPC-Anwendung. Die Klasse `Echo` implementiert die Methoden `getEcho` und `getEchoWithDate`, die ein "Echo" ohne bzw. mit Serverdatum zurückgeben.

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class Echo {
   public String getEcho(String s) {
      return s;
   }

   public String getEchoWithDate(String s) {
      SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      return "[" + f.format(new Date()) + "] " + s;
   }
}
```

Die Klasse `EchoServer` erzeugt eine Instanz der Klasse `WebServer`, registriert den Echo-Dienst mit dem Namen "echo" und startet dann den Server.

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;

public class EchoServer {
   public static void main(String[] args) throws Exception {
      int port = 8080;

      PropertyHandlerMapping phm = new PropertyHandlerMapping();
      phm.addHandler("echo", Echo.class);

      WebServer webServer = new WebServer(port);
      XmlRpcServer server = webServer.getXmlRpcServer();
      server.setHandlerMapping(phm);
      webServer.start();
   }
}
```

Zur Implementierung eines *XML-RPC-Client* stellt Apache XML-RPC die Klasse

```
org.apache.xmlrpc.client.XmlRpcClient
```

zur Verfügung.

Mit Hilfe der Klasse

```
org.apache.xmlrpc.client.XmlRpcClientConfigImpl
```

kann ein `XmlRpcClient`-Objekt konfiguriert werden.

Die `XmlRpcClientConfigImpl`-Methode

```
void setServerURL(java.net.URL url)
```

legt den URL des Servers fest. Im Beispiel: `http://localhost:8080`.

Mit dem Aufruf der `XmlRpcClient`-Methode `setConfig` wird die Konfiguration `config` für den Client gesetzt:

```
setConfig(config)
```

Die `XmlRpcClient`-Methode

```
Object execute(String method, Object[] params) throws XmlRpcException
```

erzeugt eine XML-RPC-Anfrage und sendet sie mittels HTTP zum Server. Die zurückgeschickte XML-RPC-Antwort wird geparst und als Objekt vom Typ `Object` zurückgegeben.

Der String `method` hat den Aufbau:

```
Dienstname.Methodenname
```

Dienstname ist der Name, unter dem der Dienst auf der Serverseite registriert ist. *Methodenname* ist der Name der Methode, die der Dienst implementiert hat. Das Array `params` enthält die erforderlichen Parameter.

Wie die Parameter passend zur Signatur der Methode erzeugt werden müssen, zeigen die nächsten Programmbeispiele.

`EchoClient` ruft beide Methoden des Echo-Dienstes mit dem Argument "Hallo" auf.

```
import java.net.URL;

import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      URL url = new URL("http://localhost:8080");

      XmlRpcClient client = new XmlRpcClient();
      XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
      config.setServerURL(url);
      client.setConfig(config);

      Object[] params = { "Hallo" };
      String s = (String) client.execute("echo.getEcho", params);
      System.out.println(s);

      String t = (String) client.execute("echo.getEchoWithDate", params);
      System.out.println(t);
   }
}
```

Die einzelnen Schritte beim entfernten Methodenaufruf sind:

1. Das Client-Programm erzeugt eine `XmlRpcClient`-Instanz, konfiguriert sie und ruft dann die Methode `execute` mit Angabe des Dienstnamens, des Methodennamens und der Parameter auf.
2. Diese Angaben werden in ein XML-Dokument verpackt und per HTTP-POST an den Server geschickt.
3. Der Server empfängt die HTTP-Anfrage und leitet die Verarbeitung des XML-Dokuments ein.
4. Das XML-Dokument wird geparst und anschließend die angegebene Methode des Dienstes aufgerufen.
5. Die Methode übergibt das Ergebnis an den XML-RPC-Verarbeitungsprozess, der dann das Ergebnis in ein XML-Dokument verpackt.

6. Der Server schickt das XML-Dokument als Antwort auf die HTTP-Anfrage zurück.
7. Der XML-RPC-Client parst das XML-Dokument, extrahiert den Rückgabewert und übergibt diesen als Objekt an das Client-Programm.

Eclipse IDE for Java EE Developers enthält einen *TCP/IP-Monitor*, der zwischen Client und Server platziert werden kann und die Daten, die über eine TCP/IP-Verbindung gesendet werden, anzeigt. Der Client verbindet sich mit dem Monitor. Dieser leitet die Daten weiter an den Server und protokolliert die Ein- und Ausgabe.

Der Monitor kann über

```
Window > Show View > Other... > Debug > TCP/IP-Monitor
```

eingerichtet und gestartet werden.

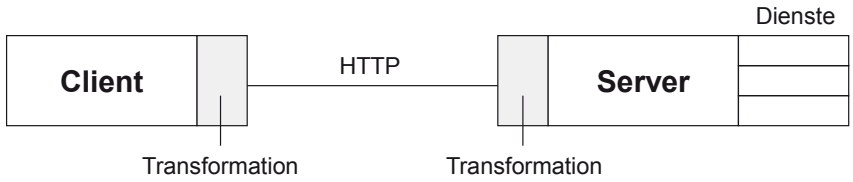

Abbildung 8-2: Kommunikation zwischen Client und Server

8.2 XML-RPC-Datentypen

Zu den einfachen Datentypen gehören:

- Ganzzahl,
- Gleitkommazahl,
- Wahrheitswert,
- Zeichenkette,
- Datum/Zeit,
- Binärdaten.

Zusammengesetzte Datentypen sind:

- Array und
- Struktur.

Ein *Array-Element* kann einen Wert vom einfachen oder zusammengesetzten Datentyp enthalten.

Eine *Struktur* ist eine Folge von Elementen, die jeweils aus einem Namen und einem Wert bestehen. Der Name muss eine Zeichenkette sein, der Wert kann vom einfachen oder zusammengesetzten Datentyp sein.

Die folgende Tabelle gibt eine Übersicht.

XML-Tag-Name	Java-Typ für execute	Java-Typ für Handler
`i4`	`java.lang.Integer`	int
`double`	java.lang.Double	`double`
`boolean`	`java.lang.Boolean`	`boolean`
`string`	`java.lang.String`	`java.lang.String`
`dateTime.iso8601`	`java.util.Date`	`java.util.Date`
`base64`	`byte[]`	`byte[]`
`array`	`java.lang.Object[]`	`java.lang.Object[]`
`struct`	`java.util.Map`	`java.util.Map`

Die erste Tabellenspalte enthält die Namen der Tags für das vom XML-RPC-Protokoll verwendete XML-Dokument. Diese Tags markieren den Datenwert zum entsprechenden Datentyp. Die zweite Spalte enthält die Entsprechungen in Java für den Aufruf der `XmlRpcClient`-Methode `execute`. Die Datentypen der dritten Spalte werden als Parameter- bzw. Rückgabetypen der Handler benutzt. `void`-Methoden sind als Handler-Methoden nicht erlaubt. Der Wert `null` ist weder als Argumentwert noch als Rückgabewert erlaubt.

Das folgende Beispiel testet die verschiedenen Datentypen.

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;

public class DatentypTestServer {
   public static void main(String[] args) throws Exception {
      int port = 8080;

      PropertyHandlerMapping phm = new PropertyHandlerMapping();
      phm.addHandler("test", DatentypTest.class);

      WebServer webServer = new WebServer(port);
      XmlRpcServer server = webServer.getXmlRpcServer();
      server.setHandlerMapping(phm);
      webServer.start();
   }
}
```

Die Handler-Klasse `DatentypTest` enthält für jeden XML-RPC-Datentyp eine Testmethode.

```
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Date;
import java.util.Map;

public class DatentypTest {
   public int testInt(int x) {
      return x;
   }

   public double testDouble(double x) {
      return x;
   }

   public boolean testBoolean(boolean x) {
      return x;
   }

   public String testString(String x) {
      return x;
   }

   public Date testDateTime(Date x) {
      return x;
   }

   public byte[] testBase64() throws IOException {
      InputStream in = new FileInputStream("java.gif");
      ByteArrayOutputStream out = new ByteArrayOutputStream();
      int c;
      while ((c = in.read()) != -1) {
         out.write(c);
      }
      in.close();
      return out.toByteArray();
   }

   public Object[] testArray(Object[] x) {
      return x;
   }

   public Map<String, String> testStruct(Map<String, String> x) {
      return x;
   }
}
```

Die Klasse `DatentypTestClient` enthält für jeden XML-RPC-Datentyp den Aufruf der zugehörigen Testmethode.

```
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.net.URL;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class DatentypTestClient {
   public static void main(String args[]) throws Exception {
      URL url = new URL("http://localhost:8080");
      XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
      config.setServerURL(url);
      XmlRpcClient client = new XmlRpcClient();
      client.setConfig(config);
      Object[] params = new Object[1];

      System.out.println("Datentyp: i4");
      params[0] = 1234;
      int i = (Integer) client.execute("test.testInt", params);
      System.out.println(i);
      System.out.println();

      System.out.println("Datentyp: double");
      params[0] = 123.456;
      double d = (Double) client.execute("test.testDouble", params);
      System.out.println(d);
      System.out.println();

      System.out.println("Datentyp: boolean");
      params[0] = true;
      boolean b = (Boolean) client.execute("test.testBoolean", params);
      System.out.println(b);
      System.out.println();

      System.out.println("Datentyp: string");
      params[0] = "<-- A & O -->";
      String s = (String) client.execute("test.testString", params);
      System.out.println(s);
      System.out.println();

      System.out.println("Datentyp: dateTime.iso8601");
      params[0] = new Date();
      Date date = (Date) client.execute("test.testDateTime", params);
      System.out.println(date);
      System.out.println();

      System.out.println("Datentyp: base64");
      byte[] bytes = (byte[]) client.execute("test.testBase64",
            new Object[] {});
      OutputStream out = new FileOutputStream("test.gif");
      for (int j = 0; j < bytes.length; j++) {
         out.write(bytes[j]);
      }
      out.flush();
      out.close();
      System.out.println();
```

```
      System.out.println("Datentyp: array");
      Object[] array = { "Das ist ein String", 4711 };
      params[0] = array;
      Object[] arr = (Object[]) client.execute("test.testArray", params);
      for (Object obj : arr) {
         System.out.println(obj);
      }
      System.out.println();

      System.out.println("Datentyp: struct");
      Map<String, String> map = new HashMap<String, String>();
      map.put("Vorname", "Hugo");
      map.put("Nachname", "Meier");
      params[0] = map;
      @SuppressWarnings("unchecked")
      Map<String, String> struct = (Map<String, String>) client.execute(
            "test.testStruct", params);
      System.out.println(struct.get("Vorname") + " " +
            struct.get("Nachname"));
      System.out.println();

      System.out.println("Datentyp: fault");
      try {
         client.execute("test.testFault", new Object[] {});
      } catch (XmlRpcException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Für jeden XML-RPC-Datentyp werden nun im Folgenden

- der Parameterwert im XML-Dokument der XML-RPC-Anfrage (`<value>`-Tag),
- der Rückgabewert im XML-Dokument der XML-RPC-Antwort (`<value>`-Tag) und
- die Ausgabe des Client

aufgeführt.

Ganzzahl

XML-RPC-Anfrage:

```
<i4>1234</i4>
```

XML-RPC-Antwort:

```
<i4>1234</i4>
```

Ausgabe des Client-Programms:

```
1234
```

Gleitkommazahl

XML-RPC-Anfrage:

```
<double>123.456</double>
```

XML-RPC-Antwort:

```
<double>123.456</double>
```

Ausgabe des Client-Programms:

```
123.456
```

Wahrheitswert

XML-RPC-Anfrage:

```
<boolean>1</boolean>
```

XML-RPC-Antwort:

```
<boolean>1</boolean>
```

Ausgabe des Client-Programms:

```
true
```

Zeichenkette

XML-RPC-Anfrage:

```
&lt;-- A &amp; O --&gt;
```

XML-RPC-Antwort:

```
&lt;-- A &amp; O --&gt;
```

Ausgabe des Client-Programms:

```
<-- A & O -->
```

Wenn wie hier kein Datentyp angegeben ist, wird `<string>` unterstellt. Die Zeichen `<`, `>` und `&` haben in XML eine Sonderrolle und werden daher umcodiert.

Datum/Zeit

XML-RPC-Anfrage:

```
<dateTime.iso8601>20150302T12:22:45</dateTime.iso8601>
```

XML-RPC-Antwort:

```
<dateTime.iso8601>20150302T12:22:45</dateTime.iso8601>
```

Ausgabe des Client-Programms:

```
Mon Mar 02 12:22:45 CET 2015
```

Binärdaten

Die XML-RPC-Anfrage enthält kein `<value>`-Tag.

XML-RPC-Antwort:

```
<base64>R0lGODlhNABYAPcAAP...</base64>
```

XML-RPC nutzt das Codierungsverfahren *Base64*, um Binärdaten technisch gesichert in XML-Strukturen zu übertragen. 24 Bit lange Gruppen der Binärdaten werden in vier Bitfolgen von jeweils 6 Bit zerlegt. Diese 6 Bit werden mit Hilfe der US-ASCII-Zeichen `A – Z`, `a – z`, `0 – 9`, `+`, `/` und `=` codiert. [3]

Ausgabe des Client-Programms:

Der Client speichert die übertragenen Daten in der Datei *test.gif*.

Array

XML-RPC-Anfrage:

```
<array>
  <data>
    <value>Das ist ein String</value>
    <value><i4>4711</i4></value>
  </data>
</array>
```

XML-RPC-Antwort:

```
<array>
  <data>
    <value>Das ist ein String</value>
    <value><int>4711</int></value>
  </data>
</array>
```

Ausgabe des Client-Programms:

```
Das ist ein String
4711
```

Struktur

XML-RPC-Anfrage:

---

[3] `http://www.ietf.org/rfc/rfc2045.txt`

```
<struct>
  <member>
    <name>Nachname</name>
    <value>Meier</value>
  </member>
  <member>
    <name>Vorname</name>
    <value>Hugo</value>
  </member>
</struct>
```

XML-RPC-Antwort:

```
<struct>
  <member>
    <name>Nachname</name>
    <value>Meier</value>
  </member>
  <member>
    <name>Vorname</name>
    <value>Hugo</value>
  </member>
</struct>
```

Ausgabe des Client-Programms:

```
Hugo Meier
```

Ausnahme

Die XML-RPC-Anfrage enthält kein <value>-Tag. Es wird eine Ausnahme vom Typ `XmlRpcException` ausgelöst, da die Methode `testFault` nicht existiert.

Die komplette XML-RPC-Antwort:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodRespose>
  <fault><value><struct>
    <member>
      <name>faultCode</name>
      <value><i4>0</i4></value>
    </member>
    <member>
      <name>faultString</name>
      <value>No such handler: test.testFault</value>
    </member>
  </struct></value></fault>
</methodResponse>
```

Ausgabe:

```
No such handler: test.testFault
```

Der TCP/IP-Monitor in Eclipse (siehe Kapitel 8.1) kann genutzt werden, um die zwischen Client und Server ausgetauschten Daten aufzuzeichnen. Der Server läuft auf Port 8080. Ist der lokale Port des Monitors z. B. auf 80 eingestellt, so muss in `DatentypTestClient` der Port ebenso den Wert 80 haben.

8.3 Dynamische Proxies

Mit Hilfe so genannter *dynamischer Proxies* kann die Programmierung des Clients vereinfacht werden. Ein *dynamischer Proxy* ist eine Klasse, die erst zur Laufzeit dynamisch generiert wird.

Ziel ist es, die entfernte, vom Server implementierte Methode wie eine "normale" lokale Methode auf der Clientseite aufzurufen.

Dazu muss zunächst die Handler-Klasse ein *Interface* implementieren. Dasselbe Interface wird auch vom Client verwendet.

Wir erläutern die Vorgehensweise anhand des Echo-Dienstes aus Kapitel 8.1.

```
public interface Echo {
   String getEcho(String s);
   String getEchoWithDate(String s);
}
```

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class EchoImpl implements Echo {
   public String getEcho(String s) {
      return s;
   }

   public String getEchoWithDate(String s) {
      SimpleDateFormat f = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      return "[" + f.format(new Date()) + "] " + s;
   }
}
```

Der beim Aufruf der Methode `addHandler` benutzte Key für die Handler-Klasse muss mit dem voll qualifizierten Namen des *Interfaces* (`Paket.Interface`) übereinstimmen.

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.webserver.WebServer;

public class EchoServer {
   public static void main(String[] args) throws Exception {
      int port = 8080;

      PropertyHandlerMapping phm = new PropertyHandlerMapping();
      phm.addHandler("Echo", EchoImpl.class);

      WebServer webServer = new WebServer(port);
      XmlRpcServer server = webServer.getXmlRpcServer();
      server.setHandlerMapping(phm);
      webServer.start();
   }
}
```

Auf der Clientseite wird nun nicht das `XmlRpcClient`-Objekt `client` direkt zum Aufruf der entfernten Methode benutzt, sondern zunächst eine Instanz der Klasse

```
org.apache.xmlrpc.client.util.ClientFactory
```

wie folgt erzeugt:

```
ClientFactory factory = new ClientFactory(client);
```

Die `ClientFactory`-Methode

```
Object newInstance(Class c)
```

wird mit dem Interface `Echo.class` als Argument aufgerufen. Zurückgeliefert wird eine Implementierung dieses Interfaces, die intern `client` nutzt, um die entfernte Methode aufzurufen:

```
Echo echo = (Echo) factory.newInstance(Echo.class);
```

Nun können die `Echo`-Methoden aufgerufen werden, z. B.

```
String s = echo.getEcho("Hallo");
```

```
import java.net.URL;

import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;
import org.apache.xmlrpc.client.util.ClientFactory;

public class EchoClient {
   public static void main(String args[]) throws Exception {
      URL url = new URL("http://localhost:8080");
      XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
      config.setServerURL(url);
      XmlRpcClient client = new XmlRpcClient();
      client.setConfig(config);

      ClientFactory factory = new ClientFactory(client);
      Echo echo = (Echo) factory.newInstance(Echo.class);
```

```
        String s = echo.getEcho("Hallo");
        System.out.println(s);

        String t = echo.getEchoWithDate("Hallo");
        System.out.println(t);
    }
}
```

8.4 XML-RPC mit Apache Tomcat nutzen

Bisher wurde in allen Beispielen die Klasse `WebServer` benutzt. Sie implementiert einen einfachen HTTP-Server, der zudem XML-RPC-Anfragen verarbeiten kann. Bisweilen ist es aus Performance- und Sicherheitsgründen erforderlich, die XML-RPC-Verarbeitung in einen anderen, marktgängigen Webserver zu integrieren.

Wir zeigen diese Integration am Beispiel von *Apache Tomcat*, der einen *Servlet-Container* als Ablaufumgebung für Webanwendungen bietet. Im Folgenden setzen wir den Umgang mit *Tomcat* und Grundlagen zur *Servlet-Technologie* voraus.

Die Klasse

```
org.apache.xmlrpc.webserver.XmlRpcServletServer
```

ist Subklasse von `XmlRpcServer` und besitzt die folgende Methode:

```
void execute(javax.servlet.http.HttpServletRequest request,
    javax.servlet.http.HttpServletResponse response)
    throws javax.servlet.ServletException, java.io.IOException
```

Sie verarbeitet die XML-RPC-Anfrage und liefert die XML-RPC-Antwort zurück. Diese Methode kann nun sehr einfach in der `doPost`-Methode eines Servlets genutzt werden.

Die folgende abstrakte Klasse `xmlrpc.AbstractXmlRpcServlet` dient als universelle Basisklasse für eigene Servlets. Zusätzlich zu den jar-Dateien von Apache XML-RPC wird aus dem Verzeichnis `lib` der Installation von Apache Tomcat die Datei `servlet-api.jar` benötigt.

```
package xmlrpc;

import java.io.IOException;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.webserver.XmlRpcServletServer;

@SuppressWarnings("serial")
public abstract class AbstractXmlRpcServlet extends HttpServlet {
   private XmlRpcServletServer server;

   public final void init(ServletConfig config) throws ServletException {
      super.init(config);
      try {
         PropertyHandlerMapping phm = new PropertyHandlerMapping();
         addHandlers(phm);
         server = new XmlRpcServletServer();
         server.setHandlerMapping(phm);
      } catch (XmlRpcException e) {
         throw new ServletException(e);
      }
   }

   public abstract void addHandlers(PropertyHandlerMapping phm)
         throws XmlRpcException;

   public final void doPost(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException {

      server.execute(request, response);
   }
}
```

Zur Demonstration ist die Klasse `echo.EchoServlet` von dieser abstrakten Klasse abgeleitet. Sie implementiert die Methode `addHandlers`. Somit haben Subklassen nur noch die Aufgabe, die geforderten Handler zu registrieren.

```
package echo;

import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.server.PropertyHandlerMapping;

import xmlrpc.AbstractXmlRpcServlet;

@SuppressWarnings("serial")
public class EchoServlet extends AbstractXmlRpcServlet {
   public void addHandlers(PropertyHandlerMapping phm) throws XmlRpcException {
      phm.addHandler("echo", Echo.class);
   }
}
```

Die hier verwendete Klasse `echo.Echo` entspricht der in Kapitel 8.1 verwendeten Klasse. Der einzige Unterschied ist, dass eine `package`-Klausel hinzugekommen ist.

Die übersetzten Klassen `AbstractXmlRpcServlet`, `EchoServlet` und `Echo` gehören gemäß ihrer Paketstruktur in das Verzeichnis `WebContent/WEB-INF/classes`. Die für

XML-RPC erforderlichen jar-Dateien müssen in das Verzeichnis `lib` unter `WEB-INF` kopiert werden.

Die Deskriptordatei `web.xml` hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
   <servlet>
      <servlet-name>EchoServlet</servlet-name>
      <servlet-class>echo.EchoServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>EchoServlet</servlet-name>
      <url-pattern>/echo</url-pattern>
   </servlet-mapping>
</web-app>
```

Zum Schluss muss diese Webanwendung Tomcat bekannt gemacht werden. Wir zeigen eine von mehreren Möglichkeiten.

Die Datei `xmlrpc.xml` mit dem Inhalt

```
<Context docBase="<Pfad-zum-Projekt>/tomcat/WebContent"
            reloadable="true" />
```

wird in das Verzeichnis

```
<Tomcat-Installationsort>/conf/Catalina/localhost
```

kopiert.

Nun kann Tomcat gestartet werden (vgl. auch Kapitel 7.2).

Der Client `EchoClient` entspricht der gleichnamigen Klasse in Kapitel 8.1.

URL ist hier:

```
http://localhost:8080/xmlrpc/echo
```

8.5 Aufgaben

1. Entwickeln Sie einen XML-RPC-Server, der an ihn übermittelte Nachrichten (Username, Text) in einer Datei speichert.

2. Entwickeln Sie einen XML-RPC-Server, der Adressen in einer Datenbank verwaltet:

```
id   nachname   vorname   telefon        email
 1   Meier      Hugo      02102/112233   h.meier@xyz.de
 ...
```

Der Client kann

- einen Eintrag erfassen:
  `boolean add(String nachname, String vorname, String telefon, String email)`
- einen Eintrag löschen:
  `boolean remove(int id)`
- alle Einträge auflisten:
  `Object[] getAll()`
- Einträge suchen:
  `Object[] lookup(String spalte, String muster)`

3. Ein XML-RPC-Server soll Eintritts- und Austrittszeiten in einer Datenbank verwalten:

```
id      beginn               ende
hugo    2014-11-30 08:01:43  2014-11-30 12:15:43
hugo    2014-11-30 13:10:12  2014-11-30 15:45:02
hugo    2014-11-30 16:25:11
...
```

Der Client kann

- Beginn und Ende zu einer ID erfassen:
  `String setTime(String id)`
- die Gesamtzeit zu einer ID abfragen:
  `String getTotalTime(String id)`

Zur Lösung der Aufgaben 2 und 3 sind JDBC-Kenntnisse erforderlich (siehe auch Literaturhinweise am Ende des Buches). Die Lösungen im Begleitmaterial verwenden das relationale Datenbanksystem H2 (siehe hierzu auch Kapitel 10.9).

9 SOAP-basierte Web Services mit JAX-WS

Verfahren für den Zugriff auf entfernte Dienste gibt es schon lange: DCOM (*Distributed Component Object Model*) für Windows-Systeme, CORBA (*Common Objekt Request Broker Architecture*) für nahezu alle Plattformen und RMI (*Remote Method Invocation*) für Java-Anwendungen. Allerdings sind die Formate der zu übertragenden Daten nur innerhalb eines Modells standardisiert. Die Zusammenarbeit von Anwendungen, die auf unterschiedlichen Verfahren basieren, kann nur mit Hilfe von proprietären Brücken-Programmen sichergestellt werden. Web Services sollen durch Nutzung von Internet-Standards auf eine einfache Art und Weise eine bessere Interoperabilität heterogener Anwendungen bieten.

Anforderungen an Services

Services implementieren Teile eines Geschäftsprozesses aus Verarbeitungssicht. Die folgende Aufzählung enthält allgemeine Anforderungen an Services: [1]

- *Verteilung*
  Ein Service steht in einem Netzwerk zur Verfügung und hat ggf. Zugriff auf weitere Services.
- *Geschlossenheit*
  Ein Service implementiert eine in sich geschlossene Funktion und kann unabhängig von anderen Services aufgerufen werden.
- *Zustandslosigkeit*
  Ein Service kann unabhängig von früheren Aufrufen genutzt werden. Er startet bei jedem Aufruf im gleichen Zustand.
- *Lose Kopplung*
  Abhängigkeiten zwischen Services sind auf ein Minimum reduziert. Ein Service kann dynamisch zur Laufzeit vom Nutzer gesucht und aufgerufen werden.
- *Austauschbarkeit*
  Ein Service kann relativ leicht gegen einen anderen Service mit gleicher Schnittstelle ausgetauscht werden.
- *Ortstransparenz*
  Auf welchem Rechner ein Service läuft, ist für den Nutzer nicht von Belang.

---

[1] Siehe: Goll, J: Architektur- und Entwurfsmuster der Softwaretechnik. Springer Vieweg, 2. Auflage 2014

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_9, © Springer Fachmedien Wiesbaden 2015

- *Plattformunabhängigkeit*
  Serviceanbieter und Servicenutzer können verschiedene Systeme (Rechner, Betriebssysteme) verwenden, ebenso kann sich die eingesetzte Programmiersprache unterscheiden.
- *Zugriff über eine Schnittstelle*
  Die Funktionalität eines Service wird den Clients über eine Schnittstelle bereitgestellt. Die Implementierung bleibt verborgen.
- *Verzeichnisdient*
  Services sowie fachliche und technische Informationen zu Services können in einem Verzeichnis registriert sein.

Definition Web Service

Eine technische Erläuterung des Begriffs *Web Service* stellt den Zusammenhang zu Schnittstellen, Programmierung und Standards her:

"Web-Services sind selbstbeschreibende, gekapselte Software-Komponenten, die eine Schnittstelle anbieten, über die ihre Funktionen entfernt aufgerufen und die lose durch den Austausch von Nachrichten miteinander gekoppelt werden können. Zur Erreichung universeller Interoperabilität werden für die Kommunikation die herkömmlichen Kanäle des Internets verwendet." [2]

"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." [3]

Die zuletzt aufgeführte Definition weist auf die wichtigsten Technologien für Web Services hin: HTTP, SOAP und WSDL.

9.1 Basiskonzepte von Web Services

Ein Web Service wird innerhalb einer speziellen Laufzeitumgebung (Applikationsserver) ausgeführt, die die notwendigen Protokolle unterstützt und die Nutzung weiterer Dienste wie beispielsweise Authentifizierung und Anbindung an Datenbanken erlaubt.

---

[2] Gesellschaft für Informatik: Symposium auf der 33. GI-Jahrestagung, August 2003

[3] `http://www.w3.org/TR/ws-gloss/`

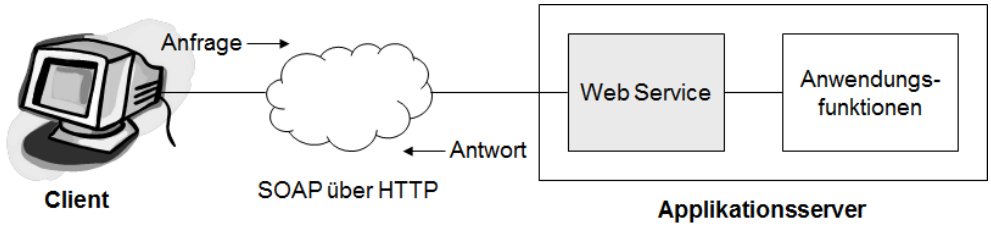

Abbildung 9-1: Aufruf eines Web Service

Die Nutzung eines Web Service soll anhand des einfachen Beispiels "Ermittlung des aktuellen Artikelpreises" verdeutlicht werden. Die in Java implementierte Client-Anwendung enthält den an den Web Service gerichteten Aufruf der Methode `getPreis` mit der Artikelnummer 4711 als Argument:

```
PreisImplService service = new PreisImplService();
Preis port = service.getPreisImplPort();
double preis = port.getPreis("4711");
```

Der Server enthält die Implementierung der Methode:

```
public double getPreis(String artikelnummer) { ... }
```

Die Methode sucht nach der vorgegebenen Artikelnummer und liefert als Ergebnis den zugehörigen Artikelpreis bzw. `-1`, falls der Artikel nicht gefunden wurde.

Analog zum Abruf einer Webseite im Webbrowser sendet der Client, der die Leistung des Web Service nutzen will, eine Anfrage via HTTP an die Adresse des Web Service (im Beispiel `http://localhost:8080/ws/preis`). Diese Anfrage wird vom Web Service verarbeitet und das Ergebnis wird als HTTP-Antwort an den Client zurückgeschickt.

Im Unterschied zum Abruf einer Webseite enthalten Anfrage und Antwort XML-Dokumente. Die Auszeichnungssprache XML (Extensible Markup Language) ist in idealer Weise dazu geeignet, die zu übertragenden Daten exakt zu beschreiben. SOAP definiert die Struktur dieser XML-Dokumente.

SOAP

Eine *SOAP-Nachricht* besteht aus einem Umschlag (*Envelope*), der sich aus einem optionalen *Header* und einem notwendigen *Body* zusammensetzt. Der Body enthält die eigentliche Nachricht für den Empfänger, der Header zusätzliche Verarbeitungsinformationen.

SOAP-Anfrage:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getPreis xmlns:ns2="http://artikel.demo/">
         <artikelnummer>4711</artikelnummer>
      </ns2:getPreis>
   </S:Body>
</S:Envelope>
```

SOAP-Antwort:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:getPreisResponse xmlns:ns2="http://artikel.demo/">
         <return>110.0</return>
      </ns2:getPreisResponse>
   </S:Body>
</S:Envelope>
```

Die Verwendung von HTTP als Transportprotokoll zur Übertragung der SOAP-Nachrichten ist am weitesten verbreitet. Gründe hierfür sind:

- HTTP-Server sind ausgereift und weit verbreitet,
- HTTP ermöglicht Verbindungen über Firewalls (Port 80),
- SOAP-Nachrichten können einfach in HTTP-Aufrufe eingebunden werden,
- Sicherheitsmechanismen für HTTP können auf SOAP-Nachrichten angewendet werden.

SOAP-Nachrichten können auch mit anderen Protokollen (z. B. SMTP, FTP, JMS) übertragen werden. Das ermöglicht einen zeitversetzten (asynchronen) Datenaustausch zwischen Client und Web Service.

Aufgrund der hier eingesetzten Technologien sind Web Services unabhängig vom Betriebssystem und von der für die Implementierung gewählten Programmiersprache.

WSDL

Mit Hilfe der XML-basierten Sprache *WSDL* (*Web Service Description Language*) werden Web Services beschrieben. WSDL-Dokumente enthalten eine Spezifikation der Funktionsschnittstellen sowie technische Details zum Web Service, z. B. das verwendete Transportprotokoll und die Adresse, unter der der Web Service

erreicht werden kann. Nachfolgend ist ein Ausschnitt des WSDL-Dokuments des hier benutzten Beispiels aufgeführt.

WSDL-Dokumente können von Client-Anwendungen automatisch ausgewertet werden, um die genauen Aufrufmodalitäten eines Web Service zu erfahren. Auf dieser Basis können Clients dann zur Laufzeit den Programmcode zum Aufruf des gerade benötigten Web Service dynamisch einbinden. Damit können die Abhängigkeiten zwischen Service-Anbietern und Service-Nutzern auf ein notwendiges Minimum reduziert werden (*lose Kopplung*).

```
<definitions targetNamespace="http://artikel.demo/"
        name="PreisImplService" ... >

   <types>

      (siehe unten)

   </types>

   <message name="getPreis">
      <part name="parameters" element="tns:getPreis" />
   </message>

   <message name="getPreisResponse">
      <part name="parameters" element="tns:getPreisResponse" />
   </message>

   <portType name="Preis">
      <operation name="getPreis">
         <input ... message="tns:getPreis" />
         <output ... message="tns:getPreisResponse" />
      </operation>
   </portType>

   <binding name="PreisImplPortBinding" type="tns:Preis">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
         style="document" />
      <operation name="getPreis">
         <soap:operation soapAction="" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>

   <service name="PreisImplService">
      <port name="PreisImplPort" binding="tns:PreisImplPortBinding">
         <soap:address location="http://localhost:8080/ws/preis" />
      </port>
   </service>
</definitions>
```

`types`-Element:

```
<xsd:schema version="1.0" targetNamespace="http://artikel.demo/">
   <xsd:element name="getPreis" type="tns:getPreis" />
   <xsd:element name="getPreisResponse" type="tns:getPreisResponse" />

   <xsd:complexType name="getPreis">
      <xsd:sequence>
         <xsd:element name="artikelnummer" type="xsd:string"
            minOccurs="0" />
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="getPreisResponse">
      <xsd:sequence>
         <xsd:element name="return" type="xsd:double" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Abbildung 9-2 zeigt, wie ein WSDL-Dokument strukturiert ist.

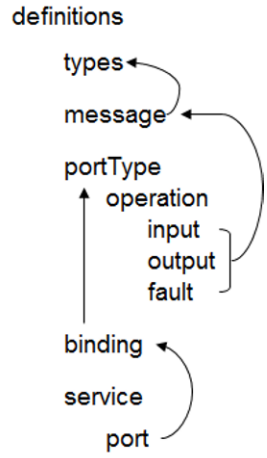

Abbildung 9-2: Aufbau eines WSDL-Dokuments (WSDL 1.1)

Das `port`-Element enthält die Adresse des Web Service und einen Verweis auf das `binding`-Element. Dort ist definiert, mit welchem Protokoll und in welcher Form Daten übertragen werden. `binding` verweist auf das `portType`-Element. Dieses definiert die Operationen des Service. Jedes `operation`-Element enthält ein `input`-Element und ggf. ein `output`- und `fault`-Element. Alle drei Elemente werden über `message`-Elemente genauer spezifiziert. Das `types`-Element enthält die Datentypen. Sie werden durch ein XML-Schema beschrieben.

SOA

Das Konzept der Web Services bietet die Möglichkeit, die Fähigkeiten einer so genannten *Service-orientierten Architektur* (*SOA*) in Anwendungen zu nutzen.

In einer Service-orientierten Architektur können drei Rollen unterschieden werden:

- Service-Anbieter,
- Service-Verzeichnis und
- Service-Nutzer.

Der *Service-Anbieter* stellt einen Web Service bereit und veröffentlicht seine Verfügbarkeit in einem Verzeichnis.

Das *Service-Verzeichnis* (Registry) stellt Beschreibungen der Web Services (WSDL) bereit und unterstützt das Veröffentlichen und Auffinden von Web Services.

Der *Service-Nutzer* sucht und findet den gewünschten Web Service im Verzeichnis und nutzt ihn, indem er sich mit dem Anbieter verbindet.

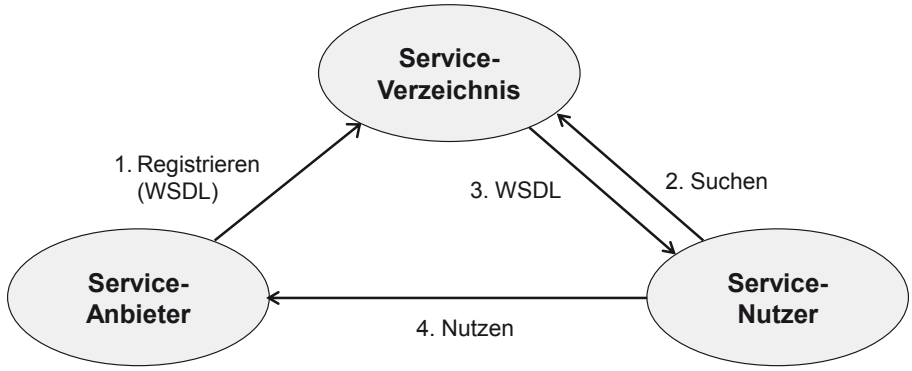

Abbildung 9-3: Rollen einer Service-orientierten Architektur

9.2 Entwicklungsplattform und API

Für die Entwicklung von Web Services stehen vor allem zwei Plattformen zur Verfügung: das *.NET-Framework* von Microsoft und die *Java Enterprise Edition* (Java EE). Während man bei .NET auf die Programmierplattform eines einzigen Herstellers beschränkt ist, ist man bei Java EE an die Sprache Java gebunden. Implementierungen von Java EE sind sowohl von Oracle als auch von zahlreichen Drittanbietern auf den meisten Betriebssystemplattformen verfügbar. Aufgrund der standardisierten Schnittstellen interagieren Web Services auf Basis von .NET und Java EE (fast) problemlos miteinander.

Es existieren mehrere Open Source Frameworks zur Entwicklung von Web Services mit Java: z. B. *Apache CXF* und *Apache Axis2*.

JAX-WS

Wir nutzen das im Rahmen des *Java Community Process* entwickelte *Java API for XML Web Services* (*JAX-WS*).

Mit diesem API können Web Services entwickelt und genutzt werden. Vor allem werden geeignete Annotationen verwendet, um normale Klassen zu Web Services zu erweitern.

Die Referenzimplementierung der JAX-WS-Spezifikation (JAX-WS RI) ist Bestandteil von Java SE und des *Metro Web Service Stack*.[4] Wir verwenden die Version *Metro 2.3*.[5]

Installation, Konfiguration

Alle jar-Dateien des Metro-Release müssen für die Java-Entwicklung in den Klassenpfad eingebunden werden. Am besten richtet man in der Entwicklungsumgebung Eclipse eine User Library *metro* ein, die alle diese jar-Dateien enthält und in jedes Projekt eingebunden wird.

Ab Java SE 6 enthält das JDK (Java Development Kit) und auch JRE (Java Runtime Environment) die JAX-WS-Referenzimplementierung. Ihre Version ist allerdings in der Regel älter als die des Metro-Release.

Um hier die Java-Plattform bzgl. JAX-WS RI auf den neuesten Stand zu bringen, kann der *Java Endorsed Standards Override Mechanism* eingesetzt werden.

Zu diesem Zweck muss das folgende Verzeichnis ermittelt bzw. erstellt werden:

```
<java-home>/lib/endorsed
```

`<java-home>` bezeichnet das Unterverzeichnis `jre` des JDK bzw. das Hauptverzeichnis der JRE.

Die Datei `webservices-api.jar` des Metro-Release muss in das Verzeichnis `endorsed` kopiert werden. Zur Laufzeit einer Java-Anwendung wird dann die Implementierung des Metro-Release und nicht diejenige der Java Runtime Environment verwendet.

---

[4] `http://metro.java.net`

[5] `metro-standalone-2.3.1.zip`

9.3 Ein erstes Beispiel

Als erstes Beispiel soll ein Web Service entwickelt werden, der den *Body-Mass-Index* (BMI) für die Bewertung der Körpermasse eines Menschen berechnet.

Die hierfür zu implementierende Methode

```
String bmi(double gewicht, double groesse)
```

wird im *Service Endpoint Interface* (*SEI*) deklariert.

Service Endpoint Interface (SEI)

```
package demo.bmi;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Bmi {
   @WebMethod
   String bmi(@WebParam(name = "gewicht") double gewicht,
         @WebParam(name = "groesse") double groesse);
}
```

Die Annotation `@WebService` gibt an, dass es sich um ein SEI für einen Web Service handelt. Mit `@WebMethod` wird die nachfolgende Methode als Service Operation gekennzeichnet. `@WebParam` sorgt dafür, dass der Wert des Attributs `name` als Name des XML-Elements, das den Parameter beschreibt, verwendet wird (siehe WSDL).

Service Implementation Bean (SIB)

Die folgende Klasse implementiert das Interface.

```
package demo.bmi;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.bmi.Bmi")
public class BmiImpl implements Bmi {
   public String bmi(double gewicht, double groesse) {
      double bmi = gewicht / (groesse * groesse);

      if (bmi < 18.5) {
         return "Untergewicht: " + bmi;
      } else if ((bmi >= 18.5) && (bmi < 25)) {
         return "Normalgewicht: " + bmi;
      } else if ((bmi >= 25) && (bmi < 30)) {
         return "Uebergewicht: " + bmi;
      } else if (bmi >= 30) {
         return "Adipositas: " + bmi;
      }
```

```
        return null;
    }
}
```

Das Attribut `endpointInterface` der Annotation `@WebService` verbindet die Implementierung mit dem SEI. Das Gewicht ist in Kilogramm, die Größe in Meter anzugeben.

Tool wsgen

Mit Hilfe von `wsgen` [6] werden die WSDL sowie verschiedene Artefakte für die Ausführung des Service generiert. Das Kommando kann in der Eingabekonsole eingegeben werden (eine einzige Zeile):

```
%METRO_HOME%/bin/wsgen -cp bin -keep -s src -d bin -wsdl -r wsdl
demo.bmi.BmiImpl
```

Die Umgebungsvariable `METRO_HOME` enthält den Namen des Installationsverzeichnisses von Metro.

Die generierten Artefakte werden als Quellcode im Verzeichnis `src` und als Bytecode in `bin` abgelegt. `keep` gibt an, ob der Quellcode der Artefakte erhalten bleibt. Das generierte WSDL-Dokument wird im Verzeichnis `wsdl` abgelegt.

Nach Ausführung von `wsgen` liegen die generierten Klassen `Bmi` und `BmiResponse` im Paket `demo.bmi.jaxws` und das WSDL-Dokument (`BmiImplService.wsdl`, `BmiImplService_schema1.xsd`) im Verzeichnis `wsdl`.

Eclipse (Java EE IDE for Web Developers) enthält einen WSDL- und XML-Schema-Editor, mit dem die Strukturen gut analysiert werden können (siehe Abbildung 9-4).

[6] Siehe `<Metro-Installationsort>/metro/bin`

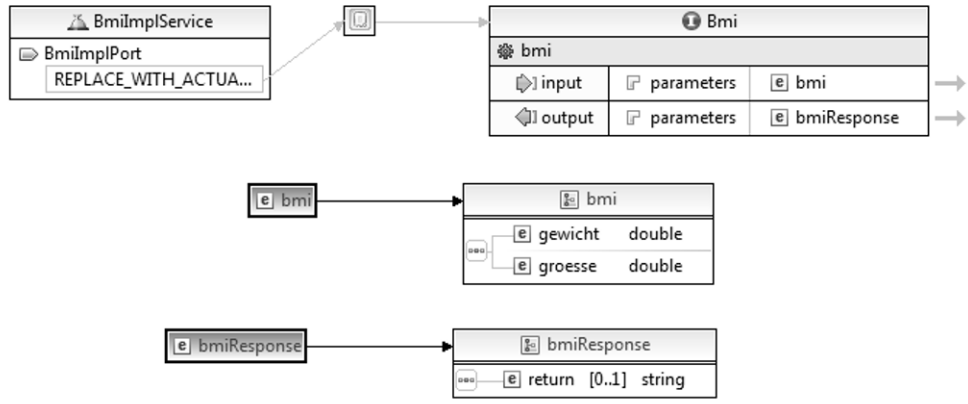

Abbildung 9-4: WSDL Editor und XML Schema Editor

Web Service starten

Die Klasse `javax.xml.ws.Endpoint` enthält die Methode `publish`, mit der ein Web Service gestartet werden kann:

```
static Endpoint publish(String address, Object implementor)
```

`address` ist der URL, unter dem der Web Service erreicht werden kann, `implementor` ist eine Instanz der Implementierung des Web Service.

```
package server;

import javax.xml.ws.Endpoint;

import demo.bmi.BmiImpl;

public class BmiPublisher {
   public static void main(String[] args) {
      String address = "http://localhost:8080/ws/bmi";
      Object implementor = new BmiImpl();

      Endpoint.publish(address, implementor);
      System.out.println("Service gestartet: " + address);
   }
}
```

Nachdem dieses Programm gestartet ist[7], kann im Browser das WSDL-Dokument abgerufen werden:

```
http://localhost:8080/ws/bmi?wsdl
```

Das zugehörige XML-Schema ist ausgelagert und kann unter der Adresse

```
http://localhost:8080/ws/bmi?xsd=1
```

erreicht werden.

Tool wsimport

Mit `wsimport` werden Klassen und Interfaces zur Unterstützung der Entwicklung des Service Client auf der Basis des WSDL-Dokuments generiert. [8] Das Kommando muss in einer einzigen Zeile eingegeben werden:

```
%METRO_HOME%/bin/wsimport.bat -keep -s src -d bin -p client
http://localhost:8080/ws/bmi?wsdl
```

Der Web Service muss gestartet sein, damit das WSDL-Dokument unter der oben genannten Adresse abgerufen werden kann. Die erzeugten Artefakte werden dem Paket `client` zugeordnet. Die übrigen Optionen sind aus `wsgen` bekannt.

Web Service Client

```
import client.Bmi;
import client.BmiImplService;

public class BmiClient {
   public static void main(String[] args) {
      BmiImplService service = new BmiImplService();
      Bmi port = service.getBmiImplPort();

      System.out.println(port.bmi(46, 1.7));
      System.out.println(port.bmi(56, 1.7));
      System.out.println(port.bmi(78, 1.76));
   }
}
```

Bei `client.Bmi` und `client.BmiImplService` handelt es sich um von `wsimport` generierte Artefakte.

---

[7] Damit eine unschöne Warnung nicht angezeigt wird, sollte aus der Installation von Apache Tomcat die Datei `servlet-api.jar` in den Klassenpfad mit aufgenommen werden.

[8] Ein evtl. bei der Ausführung von `wsimport` mit Java SE 8 auftretendes Problem kann dadurch behoben werden, dass in der Datei `wsimport.bat` bzw. `wsimport.sh` `java` mit der Systemeigenschaft `-Djavax.xml.accessExternalSchema=all` aufgerufen wird.

Die Ausführung des Programms `BmiClient` liefert die folgende Ausgabe:

```
Untergewicht: 15.91695501730104
Normalgewicht: 19.377162629757787
Uebergewicht: 25.180785123966942
```

Ruft man das Programm mit der System Property

```
-Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true
```

auf, so werden die ein- und ausgehenden SOAP-Nachrichten mit ausgegeben.

Hier ein Auszug aus dem Protokoll:

```
---[HTTP request - http://localhost:8080/ws/bmi]---
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
   <S:Body>
      <ns2:bmi xmlns:ns2="http://bmi.demo/">
         <gewicht>46.0</gewicht>
         <groesse>1.7</groesse>
      </ns2:bmi>
   </S:Body>
</S:Envelope>

---[HTTP response - http://localhost:8080/ws/bmi - 200]---
<?xml version='1.0' encoding='UTF-8'?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:bmiResponse xmlns:ns2="http://bmi.demo/">
         <return>Untergewicht: 15.91695501730104</return>
      </ns2:bmiResponse>
   </S:Body>
</S:Envelope>
```

Die Protokollierung kann auch Server-seitig eingestellt werden. Hierzu muss das Programm `BmiPublisher` mit der System Property

```
-Dcom.sun.xml.ws.transport.http.HttpAdapter.dump=true
```

aufgerufen werden.

Abbildung 9-5 fasst das Vorgehen zusammen.

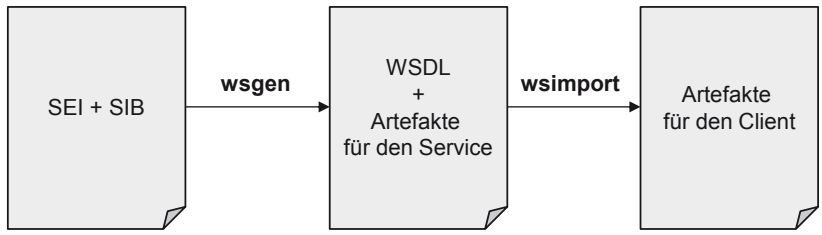

Abbildung 9-5: Vorgehensweise

9.4 Ein Web Service zur Artikelverwaltung

Dieser Abschnitt zeigt, dass nicht nur einfache Datentypen als Argument- oder Rückgabetypen genutzt werden können. Außerdem erfahren Sie, wie Ausnahmen behandelt werden.

Der folgende Web Service ermöglicht die Verwaltung von Artikeln: Speichern eines neuen Artikels, Suche nach einem Artikel, Ausgabe aller Artikel, Änderung eines Artikels.

Ein Objekt vom Typ Artikel enthält Artikel-ID, -bezeichnung und -preis. JAX-WS verwendet intern das *Java API for XML-Binding* (JAXB), um Java-Objekte in XML-Dokumente bzw. Java-Klassen in XML-Schema-Dokumente zu transformieren. In der folgenden Klasse `Artikel` wird die Annotation genutzt, um die Reihenfolge der Elemente in einem komplexen XML-Schema-Typ festzulegen.

```
package demo.artikel;

import javax.xml.bind.annotation.XmlType;

@XmlType(propOrder = { "id", "bezeichnung", "preis" })
public class Artikel {
   private int id;
   private String bezeichnung;
   private double preis;

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getBezeichnung() {
      return bezeichnung;
   }

   public void setBezeichnung(String bezeichnung) {
      this.bezeichnung = bezeichnung;
   }

   public double getPreis() {
      return preis;
   }

   public void setPreis(double preis) {
      this.preis = preis;
   }
}
```

Der Web Service soll die folgenden Methoden anbieten (SEI):

```
package demo.artikel;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface ArtikelManager {
   @WebMethod
   void createArtikel(@WebParam(name = "artikel") Artikel artikel)
         throws ArtikelFault;

   @WebMethod
   Artikel getArtikel(@WebParam(name = "id") int id) throws ArtikelFault;

   @WebMethod
   List<Artikel> getArtikelListe();

   @WebMethod
   void updateArtikel(@WebParam(name = "artikel") Artikel artikel)
         throws ArtikelFault;
}
```

```
package demo.artikel;

@SuppressWarnings("serial")
public class ArtikelFault extends Exception {
   public ArtikelFault() {
   }

   public ArtikelFault(String msg) {
      super(msg);
   }
}
```

Hier folgt die Implementierung (SIB). Aus Vereinfachungsgründen werden die Artikel nur im Hauptspeicher in einer Map gespeichert.

```
package demo.artikel;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.artikel.ArtikelManager")
public class ArtikelManagerImpl implements ArtikelManager {
   private Map<Integer, Artikel> map;
```

```
    public ArtikelManagerImpl() {
       map = new Hashtable<Integer, Artikel>();
    }

    public void createArtikel(Artikel artikel) throws ArtikelFault {
       int id = artikel.getId();
       if (find(id) == null) {
          map.put(id, artikel);
       } else
          throw new ArtikelFault("Artikel " + id + " bereits vorhanden");
    }

    public Artikel getArtikel(int id) throws ArtikelFault {
       Artikel artikel = find(id);
       if (artikel == null)
          throw new ArtikelFault("Artikel " + id + " nicht vorhanden");
       return artikel;
    }

    public List<Artikel> getArtikelListe() {
       List<Artikel> list = new ArrayList<Artikel>();
       Object[] keys = map.keySet().toArray();
       Arrays.sort(keys);
       for (int i = 0; i < keys.length; i++) {
          Integer key = (Integer) keys[i];
          list.add(map.get(key));
       }
       return list;
    }

    public void updateArtikel(Artikel artikel) throws ArtikelFault {
       int id = artikel.getId();
       Artikel a = find(id);
       if (a == null)
          throw new ArtikelFault("Artikel " + id + " nicht vorhanden");
       else
          map.put(id, artikel);
    }

    private Artikel find(int id) {
       Artikel artikel = map.get(id);
       return artikel;
    }
}
```

Die Vorgehensweise zur Entwicklung des Web Service entspricht der des vorhergehenden Abschnitts.

Im WSDL-Dokument ist für jede Operation, die eine Ausnahme (`ArtikelFault`) auslösen kann, eine Fehlernachricht vorgesehen, z. B.

```
<operation name="createArtikel">
    <input wsam:Action="..." message="tns:createArtikel"/>
    <output wsam:Action="..." message="tns:createArtikelResponse"/>
    <fault message="tns:ArtikelFault" name="ArtikelFault" wsam:Action="..."/>
</operation>
```

Hier nun der Client:

```
import java.util.List;

import client.Artikel;
import client.ArtikelFault_Exception;
import client.ArtikelManager;
import client.ArtikelManagerImplService;

public class ArtikelManagerClient {
   private static ArtikelManager port;

   public static void main(String[] args) {
      ArtikelManagerImplService service = new ArtikelManagerImplService();
      port = service.getArtikelManagerImplPort();

      try {
         Artikel artikel = new Artikel();
         artikel.setId(4711);
         artikel.setBezeichnung("Hammer");
         artikel.setPreis(2.99);
         port.createArtikel(artikel);

         artikel.setId(4712);
         artikel.setBezeichnung("Zange");
         artikel.setPreis(3.50);
         port.createArtikel(artikel);

         artikel = port.getArtikel(4711);
         System.out.println(artikel.getId() + ", "
               + artikel.getBezeichnung() + ", " + artikel.getPreis());

         artikel.setPreis(2.00);
         port.updateArtikel(artikel);

         getArtikelListe();

         artikel = port.getArtikel(5000);
         System.out.println(artikel.getId() + ", "
               + artikel.getBezeichnung() + ", " + artikel.getPreis());
      } catch (ArtikelFault_Exception e) {
         System.err.println(e.getMessage());
      }
   }

   private static void getArtikelListe() {
      List<Artikel> liste = port.getArtikelListe();
      for (Artikel a : liste) {
         System.out.println(a.getId() + "\t" + a.getBezeichnung() + "\t"
               + a.getPreis());
      }
   }
}
```

Da der Artikel 5000 nicht vorhanden ist, wird eine Ausnahme ausgelöst. `wsimport` hat hierzu die `Exception`-Klasse

```
ArtikelFault_Exception
```

generiert.

Die folgende SOAP-Nachricht wird in diesem Fall zum Client geschickt:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Artikel 5000 nicht vorhanden</faultstring>
         <detail>
            <ns2:ArtikelFault xmlns:ns2="http://artikel.demo/">
               <message>Artikel 5000 nicht vorhanden</message>
            </ns2:ArtikelFault>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

9.5 Web Services mit Apache Tomcat veröffentlichen

Mit jedem HTTP-Server, der die Servlet-Technologie unterstützt, kann ein JAX-WS Web Service veröffentlicht werden, z. B. mit *Apache Tomcat* (Version 8.x).[9] Gegenüber der bisher genutzten Möglichkeit, einen Web Service zu starten, hat der Einsatz von Tomcat mehrere Vorteile:

- Veröffentlichung mehrerer Web Services innerhalb derselben Webanwendung,
- Nutzung von Container-gesteuerten Sicherheitsfunktionen (Authentifizierung),
- höhere Performance und Skalierbarkeit.

Tomcat konfigurieren

Zusätzlich zu der in Kapitel 9.2 beschriebenen allgemeinen Konfiguration ist in der Datei `<Tomcat-Installationsort>/conf/catalina.properties` die Zeile, die mit `shared.loader` beginnt, wie folgt anzupassen:

```
shared.loader=<Metro-Installationsort>/lib/*.jar
```

Wichtig ist, dass hier die "normalen" Schrägstriche verwendet werden.

[9] `http://tomcat.apache.org`

Einrichten der Webanwendung

`WebContent/WEB-INF/web.xml` hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <listener>
      <listener-class>
         com.sun.xml.ws.transport.http.servlet.WSServletContextListener
      </listener-class>
   </listener>
   <servlet>
      <servlet-name>jaxws</servlet-name>
      <servlet-class>
         com.sun.xml.ws.transport.http.servlet.WSServlet
      </servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>jaxws</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Die Datei `WebContent/WEB-INF/sun-jaxws.xml` enthält die Endpoint-Beschreibungen der zu veröffentlichenden Web Services:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
   version="2.0">
   <endpoint name="bmi" implementation="demo.bmi.BmiImpl"
      url-pattern="/bmi" />
   <endpoint name="artikel" implementation="demo.artikel.ArtikelManagerImpl"
      url-pattern="/artikel" />
</endpoints>
```

Die Bytecodes der Web Services aus Kapitel 9.3 und 9.4 können jeweils in einer jar-Datei zusammengefasst werden.

```
jar cf WebContent/WEB-INF/lib/bmi.jar -C ../bmi/bin demo/bmi/
jar cf WebContent/WEB-INF/lib/artikel.jar -C ../artikel/bin demo/artikel/
```

Die Context-Datei `ws.xml` der Webanwendung `ws` hat den folgenden Inhalt:

```
<Context
   docBase="D:\Buchprojekte\neu\MKJava4\ws\workspace\tomcat\WebContent"
   reloadable="true" />
```

`ws.xml` muss anschließend nach

```
<Tomcat-Installaionsort>/conf/Catalina/localhost
```

kopiert werden.

Nun kann Tomcat mit `startup.bat` bzw. `startup.sh` gestartet werden.

Test mit Web Browser:

`http://localhost:8080/ws/bmi?wsdl`

`http://localhost:8080/ws/artikel?wsdl`

9.6 Oneway-Operationen

Mit der Annotation `@Oneway` wird für eine `void`-Methode festgelegt, dass die Operation *keine Nachricht* an den Aufrufer als Antwort sendet. Beim Aufruf einer solchen Operation wird die Kontrolle an den Aufrufer zurückgegeben, bevor die Methode ausgeführt wird.

Das folgende Beispiel zeigt den Unterschied zwischen dem Aufruf einer normalen `void`-Methode und einer Oneway-Operation.

```
package demo.oneway;

import javax.jws.Oneway;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Send {
   @WebMethod
   void send1(@WebParam(name = "msg") String msg);

   @WebMethod
   @Oneway
   void send2(@WebParam(name = "msg") String msg);
}
```

```
package demo.oneway;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.oneway.Send")
public class SendImpl implements Send {
```

```
   public void send1(String msg) {
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      System.out.println(msg);
   }

   public void send2(String msg) {
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }

      System.out.println(msg);
   }
}
```

Die Ausgabe der Nachricht `msg` auf der Server-Seite wird in beiden Fällen um 10 Sekunden verzögert.

```
import client.Send;
import client.SendImplService;

public class SendClient {
   public static void main(String[] args) {
      SendImplService service = new SendImplService();
      Send port = service.getSendImplPort();

      System.out.println("--- send1 ---");
      port.send1("Hallo");

      System.out.println("--- send2 ---");
      port.send2("Auf Wiedersehen");

      System.out.println("--- Ende ---");
   }
}
```

Ruft man das Client-Programm auf, sieht man deutlich den Unterschied: `send1` kehrt erst nach 10 Sekunden zurück, `send2` praktisch sofort.

SOAP-Antwort von `send1`:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:send1Response xmlns:ns2="http://oneway.demo/" />
   </S:Body>
</S:Envelope>
```

Bei der Ausführung von `send2` wird lediglich der HTTP-Header

```
HTTP/1.1 202 Accepted
```

als Antwort übertragen, keine SOAP-Nachricht.

9.7 Asynchrone Aufrufe

Der Client kann eine Methode so aufrufen, dass der Aufruf nicht so lange blockiert, bis das Ergebnis vorliegt. Vielmehr kann der Client weiter arbeiten. Er wartet *asynchron* auf das Ergebnis.

Polling oder Callback

Hierzu gibt es zwei Möglichkeiten:

- Der Client fragt von Zeit zu Zeit nach, ob die Antwort da ist (*polling*, *Pull-Prinzip*), oder
- er stellt eine *Callback*-Methode bereit, die dann automatisch aufgerufen wird, wenn die Antwort vorliegt (*Push-Prinzip*).

Wir demonstrieren diese beiden Möglichkeiten anhand eines einfachen Beispiels. Der Web Service ermittelt, ob eine bestimmte Menge eines Produkts verfügbar ist. Die Bearbeitung kann unterschiedlich lange dauern (3 bis 8 Sekunden).

```
package demo.product;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Product {
   @WebMethod
   boolean isAvailable(@WebParam(name = "id") int id,
         @WebParam(name = "amount") int amount);
}
```

```
package demo.product;

import java.util.Random;

import javax.jws.WebService;

@WebService(endpointInterface = "demo.product.Product")
public class ProductImpl implements Product {
   public boolean isAvailable(int id, int amount) {
      Random r = new Random();
      int delay = 3 + r.nextInt(6);
```

```
      try {
         Thread.sleep(delay * 1000);
      } catch (InterruptedException e) {
      }

      return r.nextBoolean();
   }
}
```

`wsimport` erzeugt aus dem WSDL-Dokument die für die Entwicklung des Clients nötigen Artefakte. Damit Code für den asynchronen Aufruf (Polling bzw. Callback) generiert wird, muss das folgende XML-Dokument erstellt werden und beim Aufruf von `wsimport` berücksichtigt werden.

Bindings-Dokument async.xml für wsimport

```
<bindings wsdlLocation="http://localhost:8080/ws/product?wsdl"
   xmlns="http://java.sun.com/xml/ns/jaxws">
   <enableAsyncMapping>true</enableAsyncMapping>
</bindings>
```

Das Kommando muss in einer einzigen Zeile eingegeben werden:

```
%METRO_HOME%/bin/wsimport.bat -keep -s src -d bin -p client -b async.xml
http://localhost:8080/ws/product?wsdl
```

Es folgt der Quellcode des Client-Programms, der im Anschluss erläutert wird.

```
import java.util.concurrent.ExecutionException;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import client.IsAvailableResponse;
import client.Product;
import client.ProductImplService;

public class ProductClient {
   private Product port;

   public ProductClient() {
      ProductImplService service = new ProductImplService();
      port = service.getProductImplPort();
   }

   public static void main(String[] args) {
      ProductClient client = new ProductClient();

      client.invokeBlocking();

      client.invokeNonblockingPoll();
```

```
      client.invokeNonblockingCallback();
      System.out.println("Warte auf Antwort");
      try {
         Thread.sleep(10000);
      } catch (InterruptedException e) {
      }
   }

   private void invokeBlocking() {
      System.out.println("--- blocking ---");
      boolean isAvailable = port.isAvailable(4711, 10);
      System.out.println("isAvailable: " + isAvailable);
   }

   private void invokeNonblockingPoll() {
      System.out.println("--- nonblocking, polling ---");
      Response<IsAvailableResponse> res = port.isAvailableAsync(4711, 10);

      while (!res.isDone()) {
         System.out.println("Warte auf Antwort");
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
         }
      }

      try {
         IsAvailableResponse response = res.get();
         boolean isAvailable = response.isReturn();
         System.out.println("isAvailable: " + isAvailable);
      } catch (InterruptedException e) {
         System.err.println(e.getMessage());
      } catch (ExecutionException e) {
         System.err.println(e.getMessage());
      }
   }

   private void invokeNonblockingCallback() {
      System.out.println("--- nonblocking, callback ---");
      port.isAvailableAsync(4711, 10, new MyHandler());
   }

   private class MyHandler implements AsyncHandler<IsAvailableResponse> {
      public void handleResponse(Response<IsAvailableResponse> res) {
         try {
            IsAvailableResponse response = res.get();
            boolean isAvailable = response.isReturn();
            System.out.println("isAvailable: " + isAvailable);
         } catch (InterruptedException e) {
            System.err.println(e.getMessage());
         } catch (ExecutionException e) {
            System.err.println(e.getMessage());
         }
      }
   }
}
```

Nicht-blockierend mit Polling

Die generierte Klasse `client.Product` enthält die Methode

```
Response<IsAvailableResponse> isAvailableAsync(int id, int amount)
```

Das Interface `javax.xml.ws.Response<T>` repräsentiert die Antwort des Aufrufs. Es enthält u. a. die beiden folgenden Methoden:

```
boolean isDone()
T get() throws InterruptedException,
       java.util.concurrent.ExecutionException
```

`isDone` liefert `true`, sobald die Antwort vorliegt, sonst `false`. `get` liefert die Antwort, hier ein Objekt vom Typ `IsAvailableResponse`. Diese Klasse enthält die `boolean`-Property `return`.

Innerhalb einer Schleife wird im Abstand von einer Sekunde nachgefragt, ob die Antwort vorliegt (polling). Ist die Schleife beendet, so kann die Antwort mit `get` geholt werden.

Nicht-blockierend mit Callback

`client.Product` enthält ebenfalls die Methode

```
Future<?> isAvailableAsync(int id, int amount,
       AsyncHandler<IsAvailableResponse> asyncHandler)
```

`Response` ist ein Subinterface von `Future`. Das Interface `javax.xml.ws.AsyncHandler` deklariert die Methode

```
void handleResponse(Response<T> res),
```

die automatisch aufgerufen wird, sobald die Antwort vorliegt.

Die innere Klasse `MyHandler` implementiert diese Methode für den Typparameter `IsAvailableResponse`.

Nach dem Aufruf von `invokeNonblockingCallback` wartet das Programm 10 Sekunden, damit die `main`-Methode nicht endet, bevor der Rückruf per Callback erfolgt ist.

Beispielhafte Ausgabe des Client-Programms:

```
--- blocking ---
isAvailable: false
```

```
--- nonblocking, polling ---
Warte auf Antwort
Warte auf Antwort
Warte auf Antwort
Warte auf Antwort
isAvailable: false
--- nonblocking, callback ---
Warte auf Antwort
isAvailable: true
```

9.8 Transport von Binärdaten

Bei SOAP-Nachrichten handelt es sich um XML-Dokumente, die ausschließlich Textzeichen enthalten dürfen. Will man nun Nachrichten mit binärem Inhalt übertragen, so kann man die binären Daten vor der Übertragung in eine Textform überführen (Codierung) und beim Empfänger wieder dekodieren.

Wir demonstrieren diesen Sachverhalt anhand eines Web Service, der den Upload und Download von Dateien anbietet.

```
package demo.transfer;

import java.io.FileNotFoundException;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Transfer {
   @WebMethod
   void upload(@WebParam(name = "name") String name,
         @WebParam(name = "data") byte[] data);

   @WebMethod
   byte[] download(@WebParam(name = "name") String name)
         throws FileNotFoundException;
}
```

Die `byte`-Arrays enthalten den Inhalt der Datei, `name` enthält den Dateinamen.

```
package demo.transfer;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
import javax.jws.WebService;
import javax.xml.ws.WebServiceException;

@WebService(endpointInterface = "demo.transfer.Transfer")
public class TransferImpl implements Transfer {
   private final static String DIR = "D:/temp";

   public void upload(String name, byte[] data) {
      String path = DIR + "/" + name;
      try (BufferedOutputStream out = new BufferedOutputStream(
            new FileOutputStream(path))) {

         out.write(data);
         out.flush();
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Upload");
      }
   }

   public byte[] download(String name) throws FileNotFoundException {
      String path = DIR + "/" + name;
      File file = new File(path);
      if (!file.exists())
         throw new FileNotFoundException("Datei existiert nicht.");

      try (ByteArrayOutputStream out = new ByteArrayOutputStream();
            BufferedInputStream in = new BufferedInputStream(
                  new FileInputStream(path))) {

         byte[] buffer = new byte[2048];
         int n = 0;
         while ((n = in.read(buffer)) != -1) {
            out.write(buffer, 0, n);
         }

         return out.toByteArray();
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Download.");
      }
   }
}
```

`javax.xml.ws.WebServiceException` ist die Basisklasse aller JAX-WS-Laufzeitfehler (`RuntimeException`).

Im generierten XML-Schema ist für den Parameter `data` der XML-Schema-Datentyp `base64Binary` eingetragen.

Der `UploadClient` überträgt die Datei `duke.png` an den Web Service, der sie dann im Verzeichnis `D:/temp` speichert.

```
import java.io.BufferedInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

```
import client.Transfer;
import client.TransferImplService;

public class UploadClient {
   public static void main(String[] args) {
      TransferImplService service = new TransferImplService();
      Transfer port = service.getTransferImplPort();

      String name = "duke.png";
      try (ByteArrayOutputStream out = new ByteArrayOutputStream();
            BufferedInputStream in = new BufferedInputStream(
                  new FileInputStream(name))) {
         byte[] buffer = new byte[2048];
         int n = 0;
         while ((n = in.read(buffer)) != -1) {
            out.write(buffer, 0, n);
         }

         port.upload(name, out.toByteArray());

         System.out.println(name + " wurde hochgeladen.");
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Die Binärdaten werden nach dem *Base64-Verfahren* codiert. Dabei werden Bytes in eine Folge von ASCII-Zeichen umgewandelt.

SOAP-Request

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:upload xmlns:ns2="http://transfer.demo/">
         <name>duke.png</name>
         <data>iVBORw0KGgoAA ... SUVORK5CYII=</data>
      </ns2:upload>
   </S:Body>
</S:Envelope>
```

Diese Codierung führt zu einer Vergrößerung des Datenvolumens um ca. ein Drittel.

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;

import client.Transfer;
import client.TransferImplService;

public class DownloadClient {
   public static void main(String[] args) {
      TransferImplService service = new TransferImplService();
      Transfer port = service.getTransferImplPort();
```

```
      String name = "duke.png";
      try {
         byte[] data = port.download(name);

         try (BufferedOutputStream out = new BufferedOutputStream(
               new FileOutputStream(name))) {
            out.write(data);
            out.flush();
         }

         System.out.println(name + " wurde heruntergeladen.");
      } catch (Exception e) {
         System.err.println(e.getMessage());
      }
   }
}
```

MTOM

SOAP Message Transmission Optimization Mechanism (*MTOM*) ist ein Verfahren zur Optimierung der Übertragung binärer Daten. Der in der SOAP-Nachricht im Base64-Format enthaltene Inhalt wird aus der Nachricht entfernt und in den Anhang verschoben. In der Nachricht selbst befindet sich dann eine *Referenz* auf die Originaldaten (*raw bytes*) als HTTP-Attachment.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:upload xmlns:ns2="http://transfer.demo/">
         <name>duke.png</name>
         <data>
            <xop:Include xmlns:xop="http://www.w3.org/2004/08/xop/include"
                href="cid:34585f69-58e2-4e8c-bb ... 1ef@example.jaxws.sun.com" />
         </data>
      </ns2:upload>
   </S:Body>
</S:Envelope>
```

Mit der Annotation `@javax.xml.ws.soap.MTOM` in der Implementierung des Web Service wird die Optimierung eingeschaltet.

```
@WebService(endpointInterface = "demo.transfer.Transfer")
@MTOM
public class TransferImpl implements Transfer {
   ...
}
```

Das generierte WSDL-Dokument enthält einen *Policy-Eintrag*, der aussagt, dass MTOM verwendet wird.

```
<wsp:Policy wsu:Id="TransferImplPortBinding_MTOM_Policy-...">
   <ns1:OptimizedMimeSerialization wsp:Optional="true"
      xmlns:ns1="http://.../policy/optimizedmimeserialization"/>
</wsp:Policy>
```

Im `binding`-Element des WSDL-Dokuments wird auf diesen Eintrag verwiesen:

```
<binding name="TransferImplPortBinding" type="tns:Transfer">
    <wsp:PolicyReference URI="#TransferImplPortBinding_MTOM_Policy-..."/>
    ...
</binding>
```

Sollen sehr große Datenmengen ausgetauscht werden, können diese nicht mehr komplett im Hauptspeicher gehalten werden, sodass `byte[]` als Datentyp ausscheidet. Mit Hilfe eines `javax.activation.DataHandler` können Datenströme verarbeitet werden.

```
package demo.transfer;

import java.io.FileNotFoundException;

import javax.activation.DataHandler;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.xml.bind.annotation.XmlMimeType;

@WebService
public interface Transfer {
   @WebMethod
   void upload(
         @WebParam(name = "name") String name,
         @WebParam(name = "data") @XmlMimeType("application/octet-stream")
                     DataHandler data);

   @WebMethod
   @XmlMimeType("application/octet-stream")
   DataHandler download(@WebParam(name = "name") String name)
         throws FileNotFoundException;
}
```

Die Annotation `@XmlMimeType("application/octet-stream")` sorgt dafür, dass im WSDL-Dokument (XML-Schema) für die Elemente `data` und `return` das Attribut

```
xmime:expectedContentTypes="application/octet-stream"
```

erzeugt wird.

Dies ermöglicht durch entsprechende Generierung die Nutzung des `DataHandler` in den Client-Programmen.[10]

Die spezielle Implementierung `com.sun.xml.ws.developer.StreamingDataHandler` von JAX-WS RI bietet die Methode `moveTo`, um die übertragenen Daten in eine Datei zu speichern.

```
package demo.transfer;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.jws.WebService;
import javax.xml.ws.WebServiceException;
import javax.xml.ws.soap.MTOM;

import com.sun.xml.ws.developer.StreamingDataHandler;

@WebService(endpointInterface = "demo.transfer.Transfer")
@MTOM
public class TransferImpl implements Transfer {
   private final static String DIR = "D:/temp";

   public void upload(String name, DataHandler data) {
      String path = DIR + "/" + name;
      try (StreamingDataHandler dh = (StreamingDataHandler) data) {
         dh.moveTo(new File(path));
      } catch (IOException e) {
         throw new WebServiceException("Fehler beim Upload");
      }
   }

   public DataHandler download(String name) throws FileNotFoundException {
      String path = DIR + "/" + name;
      File file = new File(path);
      if (!file.exists())
         throw new FileNotFoundException("Datei existiert nicht.");

      DataHandler dh = new DataHandler(new FileDataSource(file));
      return dh;
   }
}
```

Der `UploadClient` nutzt die Transfercodierung `chunked`, die die zu übertragenen Daten als Folge mehrerer Teile (Chunks) codiert. Der Sender kann damit bereits mit der Übertragung der Daten beginnen, ohne ihre Gesamtlänge zu kennen.

[10] Die Ausführung von wsimport erfordert hier eine Internet-Verbindung zum Zugriff auf das Schema `http://www.w3.org/2005/05/xmlmime`.

```
import java.util.Map;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.xml.ws.BindingProvider;

import client.Transfer;
import client.TransferImplService;

import com.sun.xml.ws.developer.JAXWSProperties;

public class UploadClient {
   public static void main(String[] args) {
      TransferImplService service = new TransferImplService();
      Transfer port = service.getTransferImplPort();

      // Transfer-Encoding: chunked
      Map<String, Object> ctx = ((BindingProvider) port).getRequestContext();
      ctx.put(JAXWSProperties.HTTP_CLIENT_STREAMING_CHUNK_SIZE, 8129);

      String name = "duke.png";
      DataHandler dh = new DataHandler(new FileDataSource(name));
      port.upload(name, dh);
      System.out.println(name + " wurde hochgeladen.");
   }
}
```

```
import java.io.File;

import client.Transfer;
import client.TransferImplService;

import com.sun.xml.ws.developer.StreamingDataHandler;

public class DownloadClient {
   public static void main(String[] args) {
      TransferImplService service = new TransferImplService();
      Transfer port = service.getTransferImplPort();

      String name = "duke.png";
      try (StreamingDataHandler dh = (StreamingDataHandler) port
            .download(name)) {
         dh.moveTo(new File(name));
         System.out.println(name + " wurde heruntergeladen.");
      } catch (Exception e) {
         System.err.println(e.getMessage());
      }
   }
}
```

9.9 SOAP Message Handler

JAX-WS unterstützt so genannte *Interzeptoren* (hier *Handler* genannt), die die Funktionalität auf Client- und/oder Serverseite erweitern. Hiermit können ausgehende und eingehende SOAP-Nachrichten beim Client bzw. Server ausgewertet und auch geändert werden (Nachverarbeitung, Vorverarbeitung).

So kann zum Beispiel auf der Clientseite ein Handler implementiert werden, der unabhängig von der Funktionalität des Web Service die ausgehende SOAP-Nachricht um Authentifizierungsdaten im SOAP-Header ergänzt. Auf der Serverseite kann dann die eingehende SOAP-Nachricht geprüft werden, sodass der eigentliche Web Service gar nicht aufgerufen wird, falls die Authentifizierung fehlgeschlagen ist. Solche Handler können dann in der Regel mehrfach wiederverwendet werden und die fachliche Logik muss nicht zusätzlich aufgebläht werden.

Mögliche Einsatzbereiche sind:

- Authentifizierung,
- Verschlüsselung und Entschlüsselung,
- Protokollierung,
- Auditing (wer hat zu welcher Zeit was getan?)

Mehrere Handler können in einer Kette angeordnet sein, die dann in der Reihenfolge ihres Auftretens durchlaufen werden.

Während *SOAP Handler* Zugriff auf die komplette SOAP-Nachricht (Header und Body) haben, können *Logical Handler* nur auf die Nutzdaten des SOAP Body zugreifen. Wir betrachten hier nur SOAP Handler.

Wir wollen im Folgenden drei Handler implementieren:

- Der `ClientHashHandler` schreibt den Usernamen und das Passwort des Benutzers in den SOAP-Header. Passwort ist hier der Hashwert (nach dem Verfahren SHA-256) des Klartextpassworts.
- Der `ServiceHashHandler` extrahiert den Usernamen und den Hashwert und prüft diese Angaben gegen eine Tabelle (Klasse `DataStore`), die alle registrierten User mit dem Hashwert des Klartextpassworts enthält.
- Der `LogHandler` zeichnet aus- und eingehende SOAP-Nachrichten auf. Er wird sowohl auf der Client- als auch auf der Serverseite eingesetzt.

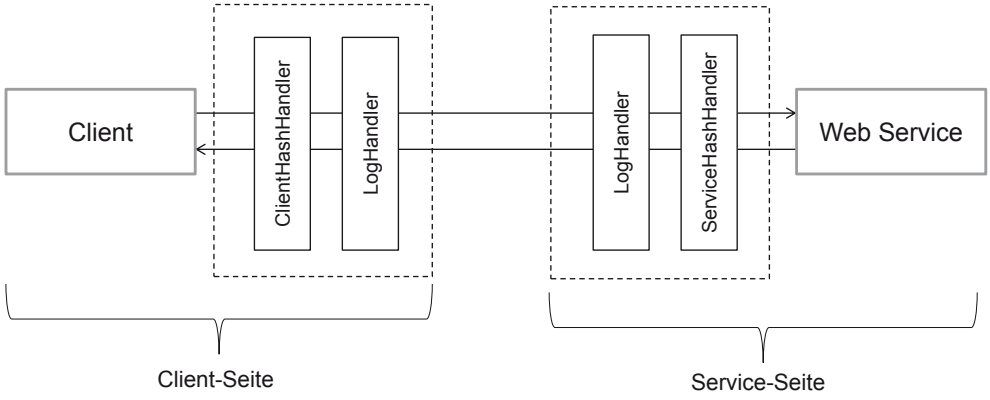

Abbildung 9-6: SOAP Handler des Beispiels

Wir demonstrieren die Wirkungsweise der Handler anhand eines sehr einfachen Web Service. Hier das SEI:

```
package demo.echo;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public interface Echo {
   @WebMethod
   String echo(@WebParam(name = "text") String text);
}
```

Der `ClientHashHandler` implementiert das Interface `SOAPHandler` mit vier Methoden, wobei hier nur die Methode `handleMessage` relevant ist.

`handleMessage` und `handleFault` liefern einen booleschen Wert zurück. Bei `true` wird die Verarbeitung mit evtl. weiteren Handlern fortgesetzt, bei `false` werden andere Handler nicht mehr ausgeführt.

`handleMessage` prüft zunächst, ob es sich um eine ausgehende (Request) oder eine eingehende Nachricht (Response) handelt. In diesem Beispiel muss die ausgehende SOAP-Nachricht geändert werden.

Der Hashwert wird mit Hilfe der Methoden der Klasse `java.security.MessageDigest` berechnet.

Der von `ClientHashHandler` erzeugte SOAP-Header-Element sieht wie folgt aus:

```
<credentials xmlns="http://handler.demo">
   <username>username</username>
   <password>Hashwert des Passworts (hexadezimal)</password>
</credentials>
```

```
package demo.handler;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class ClientHashHandler implements SOAPHandler<SOAPMessageContext> {
   private static final String ALGORITHM = "SHA-256";
   private String username;
   private String password;

   public ClientHashHandler(String username, String password) {
      this.username = username;
      this.password = password;
   }

   @Override
   public void close(MessageContext ctx) {
   }

   @Override
   public boolean handleFault(SOAPMessageContext ctx) {
      return true;
   }

   @Override
   public boolean handleMessage(SOAPMessageContext ctx) {
      Boolean outbound = (Boolean) ctx
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

      if (outbound) {
         try {
            SOAPMessage msg = ctx.getMessage();
            SOAPEnvelope env = msg.getSOAPPart().getEnvelope();
            if (env.getHeader() == null)
               env.addHeader();
            SOAPHeader hdr = env.getHeader();
            SOAPHeaderElement elem = hdr.addHeaderElement(new QName(
                  "http://handler.demo", "credentials"));
            elem.addChildElement(
                  new QName("http://handler.demo", "username"))
                  .addTextNode(username);
            elem.addChildElement(
                  new QName("http://handler.demo", "password"))
                  .addTextNode(digest(password));
            msg.saveChanges();
```

```
            } catch (Exception e) {
               throw new RuntimeException(e.getMessage());
            }
         }

         return true;
      }

      @Override
      public Set<QName> getHeaders() {
         return null;
      }

      private String digest(String text) throws NoSuchAlgorithmException {
         MessageDigest md = MessageDigest.getInstance(ALGORITHM);
         md.update(text.getBytes());
         byte[] digest = md.digest();
         StringBuilder sb = new StringBuilder();
         for (byte b : digest) {
            sb.append(String.format("%02x", b));
         }
         return sb.toString();
      }
   }
```

Analog arbeitet der `ServiceHashHandler`. Aus einer eingehenden SOAP-Nachricht (Request) werden der Username und der Hashwert des Passworts extrahiert und geprüft, ob der Username registriert ist und ob der gesendete Hashwert mit dem beim Server gespeicherten Hashwert übereinstimmt. Diverse Fehler, die hier auftreten können, werden als SOAP Fault an den Client geschickt.

```
package demo.handler;

import java.util.Iterator;
import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;
import javax.xml.ws.soap.SOAPFaultException;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class ServiceHashHandler implements SOAPHandler<SOAPMessageContext> {
   @Override
   public void close(MessageContext ctx) {
   }
```

```
@Override
public boolean handleFault(SOAPMessageContext ctx) {
   return true;
}

@Override
public boolean handleMessage(SOAPMessageContext ctx) {
   Boolean outbound = (Boolean) ctx
         .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

   if (!outbound) {
      SOAPMessage msg = ctx.getMessage();
      SOAPHeader hdr = null;

      try {
         hdr = msg.getSOAPHeader();
      } catch (SOAPException e) {
      }

      if (hdr == null) {
         generateSOAPFault(msg, "Kein Header vorhanden.");
      }

      QName qn = new QName("http://handler.demo", "credentials");
      @SuppressWarnings("rawtypes")
      Iterator it = hdr.getChildElements(qn);

      if (!it.hasNext()) {
         generateSOAPFault(msg,
               "Header 'credentials' ist nicht vorhanden.");
      }

      Node node = (Node) it.next();

      NodeList list = node.getChildNodes();
      if (list.getLength() != 2) {
         generateSOAPFault(msg, "Header 'credentials' ist fehlerhaft.");
      }

      String username = list.item(0).getTextContent();
      String password = list.item(1).getTextContent();

      if (username == null || password == null
            || username.trim().length() == 0
            || password.trim().length() == 0) {
         generateSOAPFault(msg, "Username bzw. Passwort fehlt.");
      }

      String digest = DataStore.get(username);
      if (digest == null) {
         generateSOAPFault(msg, username + " ist nicht registriert.");
      }

      if (!digest.equals(password)) {
         generateSOAPFault(msg, "Passwort ist falsch.");
      }
   }

   return true;
}
```

```
   @Override
   public Set<QName> getHeaders() {
      return null;
   }

   private void generateSOAPFault(SOAPMessage msg, String reason) {
      try {
         SOAPBody body = msg.getSOAPBody();
         SOAPFault fault = body.addFault();
         fault.setFaultString(reason);
         throw new SOAPFaultException(fault);
      } catch (SOAPException e) {
      }
   }
}
```

`DataStore` enthält die registrierten User mit den Hashwerten ihrer Passwörter.[11] Diese In-Memory-Lösung simuliert eine Datenbanktabelle.

```
package demo.handler;

import java.util.Hashtable;

public class DataStore {
   private static final Hashtable<String, String> map;

   static {
      map = new Hashtable<String, String>();
      map.put("hugo",
            "5c479f371e72181f28fc221d354a555cd2a070d68b04527d970322cc600b54d8");
      map.put("emil",
            "efbaa8cbfffc1af3afcf8082a3732e4d5111104ae3f6ef8b2545975e50497505");
   }

   public static String get(String key) {
      return map.get(key);
   }
}
```

Auch beim `LogHandler` ist nur die Methode `handleMessage` relevant. Hier wird aber sowohl auf ausgehende als auch eingehende Nachrichten reagiert. Wir nutzen das *Transformer API for XML*, um die Ausgabe des XML-Dokuments mit Zeilenumbrüchen und Einrückungen aufzubereiten, da sonst das gesamte Dokument in einer einzigen Zeile erscheinen würde.

[11] Die Seite `http://www.freeformatter.com/message-digest.html` bietet die Möglichkeit, Hashwerte mit verschiedenen Algorithmen berechnen zu lassen.

```
package demo.handler;

import java.io.ByteArrayOutputStream;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Set;

import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

public class LogHandler implements SOAPHandler<SOAPMessageContext> {
   @Override
   public void close(MessageContext ctx) {
   }

   @Override
   public boolean handleFault(SOAPMessageContext ctx) {
      log(ctx);
      return true;
   }

   @Override
   public boolean handleMessage(SOAPMessageContext ctx) {
      log(ctx);
      return true;
   }

   private static String prettyPrint(SOAPMessage msg) {
      try {
         TransformerFactory factory = TransformerFactory.newInstance();
         Transformer transformer = factory.newTransformer();

         transformer.setOutputProperty(OutputKeys.INDENT, "yes");
         transformer.setOutputProperty(
               "{http://xml.apache.org/xslt}indent-amount", "3");

         Source source = msg.getSOAPPart().getContent();

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         StreamResult result = new StreamResult(out);
         transformer.transform(source, result);

         return out.toString();
      } catch (Exception e) {
         throw new RuntimeException(e.getMessage());
      }
   }

   @Override
   public Set<QName> getHeaders() {
      return null;
   }
```

```
   private void log(SOAPMessageContext ctx) {
      SimpleDateFormat simpleFormat = new SimpleDateFormat(
            "yyyy-MM-dd HH:mm:ss.SSS");

      Boolean outbound = (Boolean) ctx
            .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);

      if (outbound) {
         System.out.println(simpleFormat.format(new Date())
               + " --- outgoing ---");
      } else {
         System.out.println(simpleFormat.format(new Date())
               + " --- incoming ---");
      }

      try {
         SOAPMessage msg = ctx.getMessage();
         String strMsg = prettyPrint(msg);
         System.out.println(strMsg);
      } catch (Exception e) {
         throw new RuntimeException(e.getMessage());
      }
   }
}
```

Nun müssen die Handler beim Client und beim Web Service registriert werden.

Handler registrieren

Die Implementierung des Web Service enthält die Annotation `@HandlerChain`. Die Datei `server-handler-chain.xml` liegt in dem Verzeichnis, in dem auch `EchoImpl.class` liegt.

```
package demo.echo;

import javax.jws.HandlerChain;
import javax.jws.WebService;

@WebService(endpointInterface = "demo.echo.Echo")
@HandlerChain(file = "server-handler-chain.xml")
public class EchoImpl implements Echo {
   public String echo(String text) {
      return text;
   }
}
```

Inhalt der Datei `server-handler-chain.xml`:

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
   <handler-chain>
      <handler>
         <handler-class>demo.handler.ServiceHashHandler</handler-class>
      </handler>
      <handler>
```

```
        <handler-class>demo.handler.LogHandler</handler-class>
      </handler>
   </handler-chain>
</handler-chains>
```

Beim Client wird die Handler-Kette durch Aufruf der `Service`-Methode

```
void setHandlerResolver(HandlerResolver handlerResolver)
```

registriert.

Die Ausführungsreihenfolge in der Handler-Kette ist wie folgt festgelegt: Bei ausgehenden Nachrichten startet die Verarbeitung mit dem ersten Handler in der Kette. Bei eingehenden Nachrichten ist die Reihenfolge umgekehrt.

```
import javax.xml.ws.soap.SOAPFaultException;

import client.Echo;
import client.EchoImplService;

public class EchoClient {
   public static void main(String[] args) {
      String username = "hugo";
      String password = "oguh";

      EchoImplService service = new EchoImplService();
      service.setHandlerResolver(new ClientHandlerResolver(username, password));
      Echo port = service.getEchoImplPort();

      try {
         String echo = port.echo("Hallo");
         System.out.println(echo);
      } catch (SOAPFaultException e) {
         System.err.println(e.getFault().getFaultString());
      }
   }
}
```

Der `ClientHandlerResolver` implementiert die Methode `getHandlerChain` des Interfaces `HandlerResolver`. Die Reihenfolge der `add`-Aufrufe bestimmt die Reihenfolge in der Kette.

```
import java.util.ArrayList;
import java.util.List;

import javax.xml.ws.handler.Handler;
import javax.xml.ws.handler.HandlerResolver;
import javax.xml.ws.handler.PortInfo;

import demo.handler.ClientHashHandler;
import demo.handler.LogHandler;
```

```
public class ClientHandlerResolver implements HandlerResolver {
   private String username;
   private String password;

   public ClientHandlerResolver(String username, String password) {
      this.username = username;
      this.password = password;
   }

   @SuppressWarnings("rawtypes")
   @Override
   public List<Handler> getHandlerChain(PortInfo portInfo) {
      List<Handler> handlerChain = new ArrayList<Handler>();
      handlerChain.add(new ClientHashHandler(username, password));
      handlerChain.add(new LogHandler());
      return handlerChain;
   }
}
```

Aufrufbeispiel:

Klartextpasswort für "hugo" ist "oguh", für "emil" ist "lime".

Das fettgedruckte XML-Fragment wurde vom `ClientHashHandler` eingefügt. Hier wird nur die Ausgabe beim Client gezeigt.

```
2015-03-04 16:40:05.388 --- outgoing ---
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header>
      <credentials xmlns="http://handler.demo">
         <username>hugo</username>
         <password>5c479f371e72181f28fc221d354a555 … 70322cc600b54d8</password>
      </credentials>
   </SOAP-ENV:Header>
   <S:Body>
      <ns2:echo xmlns:ns2="http://echo.demo/">
         <text>Hallo</text>
      </ns2:echo>
   </S:Body>
</S:Envelope>

2015-03-04 16:40:05.529 --- incoming ---
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Header/>
   <S:Body>
      <ns2:echoResponse xmlns:ns2="http://echo.demo/">
         <return>Hallo</return>
      </ns2:echoResponse>
   </S:Body>
</S:Envelope>

Hallo
```

9.10 Der Contract-First-Ansatz

Das WSDL-Dokument ist das zentrale Bindeglied zwischen Service-Nutzer und Service-Anbieter. Es beschreibt den Web Service plattform- und programmiersprachenunabhängig.

Das WSDL-Dokument kann auf zwei Arten erstellt werden:

- *Code First*
  Es wird zunächst der Web Service in Java programmiert, daraus werden dann das WSDL-Dokument und andere Artefakte generiert. Diese Vorgehensweise haben wir in allen Beispielen bisher gewählt.
- *Contract First*
  Es wird zuerst das WSDL-Dokument (z. B. von Hand oder mit geeigneten Tools) erstellt und hieraus der notwendige Java-Code generiert. Auf dieser Basis wird dann die Anwendungslogik des Web Service implementiert.

Bei beiden Vorgehensweisen werden die Artefakte zur Entwicklung des Clients aus dem WSDL-Dokument generiert.

Nachteile des Code-First-Ansatzes

Ein WSDL-Dokument manuell zu erstellen, ist relativ aufwändig. Warum dieser Weg aber doch sinnvoll ist, zeigen die folgenden Gefahren des Code-First-Ansatzes:

- Mit der Änderung des Web Service ändert sich automatisch das WSDL-Dokument. Die Gefahr ist vorhanden, dass sich dann auch Teile der Client-Artefakte mit ändern. Eine einmal erstellte Schnittstellenbeschreibung sollte aber stabil sein. Client-Anwendungen, die gegen die veröffentlichte Schnittstelle programmiert sind, sollten nicht geändert werden müssen.
- Auch beim Wechsel des Frameworks zur Entwicklung von Web Services (Einsatz anderer Generierungstools) kann sich das WSDL-Dokument ändern. Getroffene Annahmen bei der Generierung aus der Java-Implementierung können leicht zu Inkompatibilitäten führen.
- Eine komplizierte, ungeschickte Implementierung des Web Service wirkt sich möglicherweise auf das generierte WSDL-Dokument aus, sodass dieses auch nur schwer verständlich ist.

Dem Aufwand bei der manuellen Erstellung steht die Einfachheit und Schnelligkeit der automatischen WSDL-Generierung gegenüber.

Ein *Kompromiss* wäre, den Code-First-Ansatz für die erstmalige Erstellung des WSDL-Dokuments zu verwenden und bei späteren Iterationen zum Contract-First-

Ansatz zu wechseln. Darüber hinaus bietet JAX-WS eine Reihe von Möglichkeiten, mit Annotationen Inhalte des WSDL-Dokuments festzulegen.

Im Folgenden wird das Vorgehen beim *Contract-First-Ansatz* demonstriert. Ausgangsbasis ist das folgende WSDL-Dokument `TimeService.wsdl`. Es wurde mit dem *WSDL-Editor* der Eclipse Java EE IDE erstellt. Die Datei liegt im Verzeichnis `src/WEB-INF/wsdl` bzw. `bin/WEB-INF/wsdl`. [12]

WSDL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://time.demo/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="TimeService"
   targetNamespace="http://time.demo/">
   <wsdl:documentation>Test Contract First</wsdl:documentation>
   <wsdl:types>
      <xsd:schema targetNamespace="http://time.demo/">
         <xsd:element name="getTime">
            <xsd:complexType>
               <xsd:sequence />
            </xsd:complexType>
         </xsd:element>
         <xsd:element name="getTimeResponse">
            <xsd:complexType>
               <xsd:sequence>
                  <xsd:element name="time" minOccurs="1" maxOccurs="1">
                     <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                           <xsd:length value="19"></xsd:length>
                        </xsd:restriction>
                     </xsd:simpleType>
                  </xsd:element>
               </xsd:sequence>
            </xsd:complexType>
         </xsd:element>
      </xsd:schema>
   </wsdl:types>
   <wsdl:message name="getTimeRequest">
      <wsdl:part element="tns:getTime" name="parameters" />
   </wsdl:message>
   <wsdl:message name="getTimeResponse">
      <wsdl:part element="tns:getTimeResponse" name="parameters" />
   </wsdl:message>
   <wsdl:portType name="Time">
      <wsdl:operation name="getTime">
         <wsdl:input message="tns:getTimeRequest" />
         <wsdl:output message="tns:getTimeResponse" />
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="TimeServiceSOAP" type="tns:Time">
      <soap:binding style="document"
```

[12] Die Datei ist dort abgelegt, um kompatibel mit den Anforderungen von Tomcat zu sein.

```
        transport="http://schemas.xmlsoap.org/soap/http" />
      <wsdl:operation name="getTime">
         <soap:operation soapAction="http://time.demo/getTime" />
         <wsdl:input>
            <soap:body use="literal" />
         </wsdl:input>
         <wsdl:output>
            <soap:body use="literal" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="TimeService">
      <wsdl:port binding="tns:TimeServiceSOAP" name="TimeServiceSOAP">
         <soap:address location="http://localhost:8080/ws/time" />
      </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Für das Weitere können wir dieser Beschreibung entnehmen:

Element	Name	Artefakt
`targetNamespace`	`http://time.demo/`	Paket: `demo.time`
`portType`	`Time`	SEI
`operation`	`getTime`	`Time`-Methode: `getTime`
`service`	`TimeService`	Klasse: `TimeService` (Client)
`port`	`TimeServiceSOAP`	`TimeService`-Methode: `getTimeServiceSOAP` (Client)

Der fett gedruckte Teil des obigen XML-Schemas zeigt, dass der Datentyp des Rückgabewerts der Operation `getTime` gegenüber den Möglichkeiten der automatischen Generierung weiter eingeschränkt ist: `getTime` liefert eine von `null` verschiedene Zeichenkette der Länge 19.

Das WSL-Dokument als *Contract* kann also Service-Anforderungen spezifizieren, die mit der Angabe der Java-Datentypen alleine nicht ausgedrückt werden können.

Implementierung des Web Service

Zunächst werden die Java-Artefakte zur Implementierung des Web Service generiert. Der generiert Quellcode befindet sich im Paket `demo.time.gen`. Das Kommando muss in einer einzigen Zeile eingegeben werden:

```
%METRO_HOME%/bin/wsimport.bat -keep -s src -d bin -p demo.time.gen
src/WEB-INF/wsdl/TimeService.wsdl
```

```
package demo.time;

import java.text.SimpleDateFormat;
import java.util.Date;

import javax.jws.WebService;

import demo.time.gen.Time;

@WebService(endpointInterface = "demo.time.gen.Time",
   serviceName = "TimeService",
   portName = "TimeServiceSOAP",
   wsdlLocation = "WEB-INF/wsdl/TimeService.wsdl")
public class TimeServiceImpl implements Time {
   private static final SimpleDateFormat SIMPLE_FORMAT = new SimpleDateFormat(
         "yyyy-MM-dd HH:mm:ss");

   @Override
   public String getTime() {
      return SIMPLE_FORMAT.format(new Date());
   }
}
```

Der Web Service kann nun wie bekannt veröffentlicht werden: Entweder mit der `Endpoint`-Methode `publish` oder mit Apache Tomcat.

Soll Tomcat benutzt werden, muss das Verzeichnis `wsdl` mit `TimeService.wsdl` in das `WEB-INF`-Verzeichnis von Tomcat kopiert werden. [13]

Die Attribute der Annotation `@WebService` in `TimeServiceImpl` sorgen dafür, dass nicht das aus den Artefakten generierte, sondern das selbst erstellte WSDL-Dokument über `http://localhost:8080/ws/time?wsdl` zur Verfügung gestellt wird.

Implementierung des Client

Die Generierung der Artefakte für den Client erfolgt mit:

```
%METRO_HOME%/bin/wsimport.bat -keep -s src -d bin -p client
http://localhost:8080/ws/time?wsdl
```

Das Kommando muss in einer einzigen Zeile eingegeben werden.

```
import client.Time;
import client.TimeService;

public class TimeServiceClient1 {
   public static void main(String[] args)  {
      TimeService service = new TimeService();
```

[13] Weiteres hierzu in Kapitel 9.5

```
      Time port = service.getTimeServiceSOAP();
      String time = port.getTime();
      System.out.println(time);
   }
}
```

Die WSDL-Adresse und der qualifizierte Service-Name sind in der Klasse `client.TimeService` codiert.

Um den Service-Aufruf flexibler zu gestalten, können die Parameter zur Herstellung der Verbindung mit dem Web Service explizit und erst zur Laufzeit angegeben werden.

```
import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import client.Time;

public class TimeServiceClient2 {
   public static void main(String[] args) throws MalformedURLException {
      Service service = Service.create(new URL(
            "http://localhost:8080/ws/time?wsdl"), new QName(
            "http://time.demo/", "TimeService"));

      Time port = service.getPort(Time.class);
      String time = port.getTime();
      System.out.println(time);
   }
}
```

```
static Service create(java.net.URL wsdlDocumentLocation,
       javax.xml.namespace.QName serviceName)
```

erzeugt eine `Service`-Instanz.

```
<T> T getPort(Class<T> sei)
```

liefert zum vorgegebenen SEI (Service Endpoint Interface) einen Proxy.

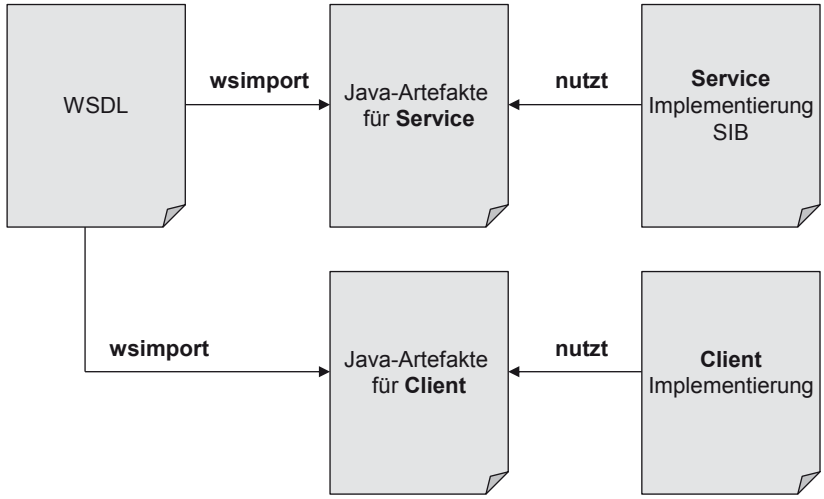

Abbildung 9-7: Vorgehen bei Contract First

9.11 Aufgaben

1. Die Mosteller-Formel ist eine mathematische Formel zur Abschätzung der Körperoberfläche (m^2) aus der Körpergröße (cm) und dem Körpergewicht (kg) beim Menschen:

   ```
   Oberfläche = Math.sqrt(Größe * Gewicht) / 60.
   ```
   [14]

   Erzeugen Sie nach dem Code-First-Ansatz einen Web Service und einen Client.

2. Durch eine Hash-Funktion wird eine Nachricht auf einen Hashwert mit einer festen Länge abgebildet. In Java steht hierfür die Klasse `java.security.MessageDigest` zur Verfügung.

   - Entwickeln Sie ein Testprogramm, das zu einer Textnachricht den Hashwert mit Hilfe der Algorithmen MD5, SHA-1, SHA-256, SHA-512 berechnet und diesen im Hexadezimalformat ausgibt.
   - Entwickeln Sie einen Web Service nach dem Code-First-Ansatz mit der Operation

     ```
     String digest(String text, String algorithm)
     ```
   - Entwickeln Sie dann einen Client, der diesen Web Service aufruft.

---

[14] Siehe `http://de.wikipedia.org/wiki/Mosteller-Formel`

3. Recherchieren Sie im Internet nach frei zugänglichen Web Services (z. B. `http://www.service-repository.com`) und erzeugen Sie einen Client für einen gefundenen Web Service.
4. Veröffentlichen Sie die Web Services aus den Aufgaben 1 und 2 mit Apache Tomcat.
5. Erstellen Sie einen Web Service mit der `Oneway`-Operation

   ```
   void send(Message message),
   ```

   die eine Nachricht auf der Server-Konsole ausgibt. `message` enthält den Zeitstempel, den Usernamen und den eigentlichen Text. Implementieren Sie auch einen Client.
6. Erstellen Sie einen Client, der den Web Service aus Aufgabe 2 asynchron aufruft und für die Antwort eine Callback-Methode zur Verfügung stellt.
7. Ein Webservice soll die Operation

   ```
   DataHandler download()
   ```

   anbieten, mit der ein bestimmtes PDF-Dokument (z. B. Auftragsformular) heruntergeladen werden kann. Nutzen Sie MTOM.
8. Entwickeln Sie einen Handler für einen Service, der die aktuelle Zeit (mit Angabe von Millisekunden) des Sendens und Empfangens einer Nachricht auf der Server-Seite ausgibt. Testen Sie den Handler mit einem in den vorhergehenden Aufgaben erstellten Web Service.
9. Erstellen Sie nach dem Contract-First-Ansatz einen Web Service, der die folgende Operation anbietet:

   ```
   List<Integer> rand(int bound, int length)
   ```

   Hier das zugehörige XML-Schema:

```
<xsd:schema targetNamespace="http://rand.demo/">
   <xsd:element name="rand" type="tns:rand" />

   <xsd:element name="randResponse" type="tns:randResponse" />

   <xsd:complexType name="rand">
      <xsd:sequence>
         <xsd:element name="bound" type="xsd:int" />
         <xsd:element name="length" type="xsd:int" />
      </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="randResponse">
      <xsd:sequence>
         <xsd:element name="return" type="xsd:int" minOccurs="0"
            maxOccurs="unbounded" />
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Die Operation soll `length` Zufallszahlen liefern, die zwischen `0` und `bound` liegen. Entwickeln Sie auch einen Client hierzu.

10. Recherchieren Sie im Internet nach frei zugänglichen Web Services (z. B. `http://www.service-repository.com`). Laden Sie das WSDL-Dokument zu einem Web Service herunter und entwickeln Sie hierzu nach dem Contract-First-Ansatz einen eigenen Web Service.

    Dabei sollen die Methoden nur soweit implementiert werden, dass der Aufruf getestet werden kann (Dummy).

10 REST-basierte Web Services mit JAX-RS

Wie wir im vorhergehenden Kapitel gesehen haben, sind SOAP-basierte Web Services und die im Umfeld vorhandenen Standards gut geeignet, die Implementierung von SOA zu ermöglichen. Allerdings haben die komplizierten XML-Nachrichten einen recht großen Overhead: viele Metadaten, wenig Nutzdaten, was u. a. Analyse und Synthese solcher Nachrichten aufwändig macht. Zudem machen Hunderte von so genannten "WS-*"-Standards (Spezifikationen im Kontext von Web Services mit SOAP/WSDL) einen guten Überblick fast unmöglich.

In den letzten Jahren ist eine leichtgewichtige und datenformatunabhängige Alternative zur Realisierung von servicebasierten Anwendungen hervorgetreten: REST (Representational State Transfer). Im Folgenden werden die Grundlagen von REST erläutert und die Implementierung von REST-basierten Web Services (hier kurz *REST-Services* genannt) auf der Basis von JAX-RS vorgestellt.

10.1 Grundprinzipien von REST-Architekturen

REST (*Representational State Transfer*) ist ein Architekturstil, dessen Prinzipien wichtige Eigenschaften des Web – abstrahiert von der konkreten Architektur, die HTTP zugrunde liegt – zusammenfassen.[1] Das Web ist demgemäß eine Ausprägung des REST-Architekturstils.

Ressourcenorientierung

Jede Client-Anfrage ist auf eine *Ressource* bezogen. Ressource steht hier für eine Informationseinheit, z. B. ein Textdokument, ein Bild, ein Auftrag in einem Auftragssystem usw., kann aber auch eine Sammlung von mehreren anderen Ressourcen sein.

Adressierbarkeit

Jede Ressource wird über einen eindeutigen *URI* (*Uniform Resource Identifier*) identifiziert.

---

[1] Der Begriff REST taucht erstmalig in der im Jahr 2000 beendeten Dissertation von Roy Thomas Fielding auf. R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD thesis, University of California 2000, `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3_10, © Springer Fachmedien Wiesbaden 2015

Beispiel: `http://www.test.de/customers/4711` identifiziert den Kunden mit dem Schlüssel 4711.

Unterschiedliche Repräsentationen

Es können verschiedene *Repräsentationen* ein- und derselben Ressource existieren. So kann ein Client eine Ressource explizit beispielsweise im XML-, HTML- oder JSON-Format anfordern.

Verknüpfungen

Jede Repräsentation einer Ressource versetzt den Client in einen neuen Zustand (*State*). Die Repräsentation kann einen Hyperlink enthalten, der mitteilt, welche Aktion er als nächste ausführen kann. Die Verwendung des Hyperlinks gibt eine weitere Repräsentation einer anderen Ressource zurück und der Client wird in einen neuen Zustand versetzt (Zustandsübergang, *State Transfer*). Diese Eigenschaft wird auch als "Hypermedia As The Engine Of Application State" (*HATEOAS*) bezeichnet.

Statuslose Kommunikation

Um die Skalierbarkeit der Anwendung zu erhöhen und die Kopplung des Clients an den Server zu verringern, wird auf einen clientspezifischen Sitzungsstatus verzichtet. Es existieren also keine Benutzersitzungen in Form von Sessions oder Cookies. Bei jeder Anfrage werden alle notwendigen Informationen wieder neu mitgeschickt, der Zustand wird also vom Client gehalten oder der Server erstellt für den Status eine eigene Ressource (Beispiel: Warenkorb).

Einheitlicher Zugriff auf Ressourcen

REST definiert einen einheitlichen Satz von Standardmethoden, die auf Ressourcen angewendet werden können: GET, POST, PUT, DELETE, HEAD und OPTIONS.

GET

Mit GET wird die Repräsentation einer Ressource abgefragt. Da nur lesend zugegriffen wird, kann eine Ressource auf dem Server nicht verändert werden.

POST

Mit POST kann eine neue Ressource mit einem vom Server bestimmten URI erstellt werden. Mit POST kann aber auch eine beliebige Verarbeitung auf dem Server angestoßen werden.

PUT

PUT wird verwendet, um eine Ressource, deren URI bereits bekannt ist, zu erstellen oder zu ändern.

DELETE

Mit DELETE wird eine Ressource gelöscht.

HEAD

Im Unterschied zu GET wird mit HEAD keine Repräsentation zurückgeliefert, sondern nur Metadaten über die Ressource (Status-Code, Header). Hiermit kann z. B. die Existenz einer Ressource geprüft werden.

OPTIONS

OPTIONS liefert Informationen über eine Ressource, z. B. welche Repräsentationsformate unterstützt werden und welche Methoden erlaubt sind.

Sicher

Eine Methode heißt *sicher* (*safe*), wenn der Aufruf dieser Methode den Zustand der Ressource, auf die zugegriffen wird, nicht verändern kann. In diesem Sinne sind nur die Methoden GET, HEAD und OPTIONS sicher.

Idempotent

Eine Methode heißt *idempotent*, wenn ein mehrmaliges Aufrufen der Methode den gleichen Effekt bewirkt wie ein einmaliges. In diesem Sinne sind alle Methoden mit Ausnahme von POST idempotent.

Die HTTP-Methode GET unterstützt ein sehr effizientes *Caching*. Ein REST-Service kann festlegen, unter welchen Bedingungen und für welchen Zeitraum ein Client die Antworten zwischenspeichern kann. Es existieren zwei Modelle.[2]

Expirationsmodell: Vom Server gesendete HTTP-Header enthalten Informationen über die Gültigkeitsdauer einer Antwort. Innerhalb dieses Zeitraums kann der Client die zwischengespeicherte Antwort wiederverwenden, ohne mit dem Server erneut zu kommunizieren.

Validierungsmodell: Der Client fragt beim Server an, ob eine bereits gelieferte Antwort wiederverwendet werden kann. Hierdurch wird dann ggf. eine Übertragung der Antwortdaten vermieden.

Im Gegensatz zu REST sind SOAP-basierte Web Services *operationsorientiert*, d. h. für jede Operation ist die Zugriffsschnittstelle operationsspezifisch. REST dagegen definiert immer dieselben oben aufgeführten Standardmethoden, um auf einer Ressource operieren zu können. REST ist *ressourcenorientiert*.

---

[2] Siehe Tilkov, S.: REST und HTTP. dpunkt.verlag, 2. Auflage 2011, Kapitel 10

Ein Beispiel

Ein Service verwaltet Textnachrichten, die vom Client erzeugt und abgefragt werden können.

Durch eine POST-Anfrage an den URL `http://www.test.de/messages` kann eine neue Nachricht erstellt werden:

```
POST /messages HTTP/1.1
User-Agent: curl/7.39.0
Content-Type: application/xml
Content-Length: 43

<message><text>Nachricht A</text></message>
```

Service-Antwort:

```
HTTP/1.1 201 Created
Location: http://www.test.de/messages/1
```

Der Aufruf von `http://www.test.de/messages/1` führt zur GET-Anfrage

```
GET /messages/1 HTTP/1.1
```

mit der Antwort

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<message id="1">
    <timestamp>2015-01-11 15:02:05.336</timestamp>
    <text>Nachricht A</text>
</message>
```

10.2 Entwicklungsplattform und API

Die Implementierung REST-basierter Web Services wird von zahlreichen Frameworks unterstützt.

JAX-RS

Wir nutzen das im Rahmen des Java Community Process entwickelte *Java API for RESTful Web Services* (*JAX-RS*). Das Open-Source-Projekt *Jersey*[3] stellt die Referenzimplementierung für JAX-RS bereit. Im Internet sind die API-Dokumentation und ein User Guide[4] verfügbar. Wir nutzen hier die Version 2.16.

---

[3] `https://jersey.java.net`

[4] `https://jersey.java.net/apidocs/latest/jersey/index.html`
`https://jersey.java.net/documentation/latest/user-guide.html`

Installation

Die ZIP-Datei `jaxrs-ri-2.16.zip` enthält das JAX-RS 2.0 API, die Kernmodule von Jersey sowie die weiteren benötigten jar-Dateien.

Alle jar-Dateien aus den Unterverzeichnissen `api`, `ext` und `lib` in `jaxrs-ri` müssen sich für die Java-Entwicklung im Klassenpfad der Anwendung befinden. Am besten richtet man in der Entwicklungsumgebung Eclipse eine User Library *jersey* ein, die alle diese jar-Dateien enthält und in jedes Projekt eingebunden wird.

Zur Bereitstellung von REST-Services verwenden wir zum Testen den JDK HTTP Server mit der Erweiterung durch den Jersey-Container `jersey-container-jdk-http-2.16.jar`.[5]

Um in einfachen Fällen Services testen zu können, ohne eigens einen Java-Client zu realisieren, benutzen wir das Kommandozeilenprogramm `cURL` (Client for URLs).[6]

10.3 Ein erstes Beispiel

Wir beginnen mit einem einfachen REST-Service, der auf eine GET-Anfrage einen Willkommensgruß sendet. Dieser kann in unterschiedlichen Formaten geliefert werden: als einfacher Text, im HTML-Format oder im XML-Format.

Der Service

```
package demo.hello;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class HelloResource {
   public HelloResource() {
      System.out.println("Neue Instanz von HelloResource: " + this);
   }

   @GET
   @Produces(MediaType.TEXT_PLAIN)
   public String getPlainMessage() {
      return "Herzlich Willkommen!";
   }
```

[5] http://mvnrepository.com/artifact/org.glassfish.jersey.containers/jersey-container-jdk-http/2.16

[6] http://curl.haxx.se

```
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String getHtmlMessage() {
       return "<html><body><h1>Herzlich Willkommen!<h1></body></html>";
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public String getXmlMessage() {
       return "<?xml version=\"1.0\"?><hello>Herzlich Willkommen!</hello>";
    }

    @Path("/extended")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getPlainMessageExtended() {
       return "Herzlich Willkommen! Und gute Unterhaltung.";
    }
}
```

Die Annotation `@Path("/hello")` vor der Definition der Klasse legt den Ressourcenpfad fest. So wenden sich in unserem Beispiel alle HTTP-Anfragen `http://localhost:8080/rest/hello` an diese Klasse. Hier ist der *Context Root* der Applikation `rest`.

Die Annotation `@GET` vor der Methode bestimmt, dass eine GET-Anfrage zur Ausführung dieser Methode führt. Hier stehen aber drei verschiedene Methoden zur Auswahl. Welche Methode wird gewählt?

Die Annotation `@Produces` legt fest, welchen HTTP-Content-Type die GET-Antwort hat. Die Klasse `javax.ws.rs.core.MediaType` enthält String-Konstanten zur Bezeichnung von Medientypen. Der Client kann die gewünschte Repräsentationsform über den Wert des HTTP-Header `Accept` bestimmen. In unserem Beispiel kommt als Wert `text/plain`, `text/html` oder `application/xml` infrage. Hiermit ist dann die geeignete Methode eindeutig bestimmt. Diese Form der Anforderung einer Repräsentation vonseiten des Clients wird als *Content Negotiation* bezeichnet.

Eine Methode kann auch mit `@Path` annotiert werden.

Beispiel: `@Path("/extended")`

Die GET-Anfrage `http://localhost:8080/rest/hello/extended` führt dann zur Ausführung dieser Methode.

Die von `javax.ws.rs.core.Application` abgeleitete Klasse `MyApplication` sorgt für die Registrierung der JAX-RS-Ressourcen beim Server.

```
package demo.hello;

import java.util.HashSet;
import java.util.Set;

import javax.ws.rs.core.Application;

public class MyApplication extends Application {
   private Set<Object> singletons = new HashSet<Object>();
   private Set<Class<?>> classes = new HashSet<Class<?>>();

   public MyApplication() {
      singletons.add(new HelloResource());
   }

   @Override
   public Set<Class<?>> getClasses() {
      return classes;
   }

   @Override
   public Set<Object> getSingletons() {
      return singletons;
   }
}
```

Die Methode `getClasses()` liefert eine Liste von JAX-RS-Ressourcenklassen. Für jede HTTP-Anfrage, die sich an eine dieser Klassen richtet, wird für jede Anfrage eine neue Instanz dieser Klasse vom Laufzeitsystem erzeugt.

Die alternative Methode `getSingletons()` liefert eine Liste von bereits erstellten Instanzen. Diese Instanzen werden im Gegensatz zum oben beschriebenen Verhalten für alle Anfragen an die jeweilige Klasse beibehalten.

Zur Bereitstellung des Service benutzen wir den erweiterten JDK HTTP Server (siehe Kapitel 10.2).

```
import java.net.URI;

import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

import demo.hello.MyApplication;

public class Server {
   public static void main(String[] args) throws Exception {
      final String BASE_URL = "http://localhost:8080/rest";

      URI endpoint = new URI(BASE_URL);
      ResourceConfig rc = ResourceConfig
            .forApplicationClass(MyApplication.class);
      JdkHttpServerFactory.createHttpServer(endpoint, rc);
      System.out.println("Server gestartet: " + BASE_URL);
   }
}
```

Basis-URL mit Kontext `rest` ist `http://localhost:8080/rest`.

Der Client

Wir nutzen zunächst das Kommandozeilenprogramm `cURL`:

```
curl -i -H "Accept: text/plain" http://localhost:8080/rest/hello
curl -i -H "Accept: text/html" http://localhost:8080/rest/hello
curl -i -H "Accept: application/xml" http://localhost:8080/rest/hello
curl -i -H "Accept: text/plain" http://localhost:8080/rest/hello/extended
```

Das letzte dieser vier Kommandos liefert beispielsweise die Ausgabe:

```
HTTP/1.1 200 OK
Content-type: text/plain
Content-length: 43
Date: Thu, 05 Mar 2015 12:53:17 GMT

Herzlich willkommen! Und gute Unterhaltung.
```

Mit Java-Bordmitteln können wir einen Java-Client realisieren. Die hier zentral verwendete Klasse ist `java.net.HttpURLConnection`.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class Client1 {
   private static final String BASE_URL = "http://localhost:8080/rest";

   public static void main(String[] args) throws IOException {
      String uri = BASE_URL + "/hello";

      get(uri, "text/plain");
      get(uri, "text/html");
      get(uri, "application/xml");
      get(uri + "/extended", "text/plain");
   }

   private static void get(String uri, String mediaType) throws IOException {
      URL url = new URL(uri);
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setRequestMethod("GET");
      connection.setRequestProperty("Accept", mediaType);

      BufferedReader reader = new BufferedReader(new InputStreamReader(
            connection.getInputStream()));
      String line = reader.readLine();
      while (line != null) {
         System.out.println(line);
         line = reader.readLine();
      }
      reader.close();
   }
}
```

Im Folgenden nutzen wir das JAX-RS 2.0 Client API, um auf den Service zuzugreifen.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class Client2 {
   private static final String BASE_URL = "http://localhost:8080/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();
      String uri = BASE_URL + "/hello";

      get(uri, MediaType.TEXT_PLAIN);
      get(uri, MediaType.TEXT_HTML);
      get(uri, MediaType.APPLICATION_XML);
      get(uri + "/extended", MediaType.TEXT_PLAIN);
   }

   private static void get(String uri, String mediaType) {
      WebTarget target = client.target(uri);
      Response result = target.request().accept(mediaType).get();
      System.out.println(result.readEntity(String.class));
   }
}
```

Mit Hilfe der Klassenmethode `javax.ws.rs.client.ClientBuilder.newClient()` wird eine Instanz vom Typ des Interface `javax.ws.rs.client.Client` erzeugt.

Die `Client`-Methode

```
WebTarget target(String uri)
```

liefert zum vorgegebenen URI `uri` ein Ressourcenziel.

Das Interface `javax.ws.rs.client.WebTarget` enthält u. a. die Methode

```
Invocation.Builder request(),
```

die den Aufbau einer Anfrage startet.

Das Interface `javax.ws.rs.client.Invocation.Builder` bietet diverse Methoden. Hier nutzen wir:

`Invocation.Builder accept(String... mediaTypes)`
: legt den gewünschten Medientyp fest.

`Response get()`
: ruft die GET-Methode auf und liefert die Antwort zurück.

Die Klasse `javax.ws.rs.core.Response` repräsentiert die HTTP-Antwort.

Die `Response`-Methode

```
<T> T readEntity(Class<T> entityType)
```

stellt den Inhalt im entsprechenden Java-Typ bereit.

Eclipse IDE for Java EE Developers enthält einen *TCP/IP-Monitor*, der zwischen Client und Server platziert werden kann und die Daten, die über eine TCP/IP-Verbindung gesendet werden, anzeigt. Der Client verbindet sich mit dem Monitor. Dieser leitet die Daten weiter an den Server und protokolliert die Ein- und Ausgabe.

Der Monitor kann über

```
Window > Show View > Other... > Debug > TCP/IP-Monitor
```

eingerichtet und gestartet werden.

Läuft der Server auf der Portnummer 8080 und der TCP/IP-Monitor auf der Portnummer 80, so muss in den Client-Programmen die Portnummer 80 anstellen von 8080 eingestellt werden.

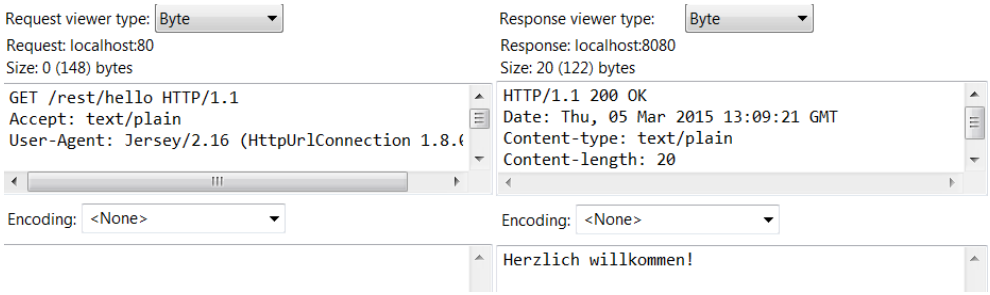

Abbildung 10-1: TCP/IP-Monitor

10.4 CRUD-Operationen

Im Folgenden nutzen wir alle vier HTTP-Methoden POST, GET, PUT und DELETE, um die *CRUD-Operationen* Create, Read, Update und Delete auszuführen.

Der Service

Der Service `MessagesResource` verwaltet Nachrichten vom Typ `Message` zur Vereinfachung in einer `Map` im Hauptspeicher. Eine Message besteht aus einer eindeutigen Id, einem Zeitstempel und der eigentlichen Textnachricht. Messages können anhand ihrer Id verglichen werden.

```
package demo.messages;

public class Message implements Comparable<Message> {
   private int id;
   private String timestamp;
   private String text;

   public Message() {
   }

   public Message(String text) {
      this.text = text;
   }

   public Message(int id, String timestamp, String message) {
      this.id = id;
      this.timestamp = timestamp;
      this.text = message;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getTimestamp() {
      return timestamp;
   }

   public void setTimestamp(String timestamp) {
      this.timestamp = timestamp;
   }

   public String getText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }

   @Override
   public String toString() {
      return "Message [id=" + id + ", timestamp=" + timestamp + ", text="
            + text + "]";
   }

   @Override
   public int compareTo(Message message) {
      return Integer.valueOf(id)
            .compareTo(Integer.valueOf(message.id));
   }
}
```

```
package demo.messages;

import java.net.URI;
import java.sql.Timestamp;
import java.util.Arrays;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("/messages")
public class MessagesResource {
   private static Map<Integer, Message> messages;
   private static int count;

   static {
      messages = new ConcurrentHashMap<Integer, Message>();
   }

   @POST
   @Consumes(MediaType.TEXT_PLAIN)
   public Response newMessage1(@Context UriInfo info, String text) {
      int id = getId();
      Message message = new Message(id, getTimestamp(), text);
      messages.put(id, message);
      URI location = info.getAbsolutePathBuilder().path(String.valueOf(id))
            .build();
      return Response.created(location).build();
   }

   @POST
   @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public Response newMessage2(@Context UriInfo info,
         @FormParam("text") String text) {
      int id = getId();
      Message message = new Message(id, getTimestamp(), text);
      messages.put(id, message);
      URI location = info.getAbsolutePathBuilder().path(String.valueOf(id))
            .build();
      return Response.created(location)
            .status(Response.Status.MOVED_PERMANENTLY).build();
   }

   @PUT
   @Path("{id}")
   @Consumes(MediaType.TEXT_PLAIN)
   public Response updateMessage(@PathParam("id") int id, String text) {
      Message message = messages.get(id);
      if (message == null) {
```

```
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        } else {
            message.setTimestamp(getTimestamp());
            message.setText(text);
            messages.put(id, message);
            return Response.noContent().status(Response.Status.OK).build();
        }
    }

    @DELETE
    @Path("{id}")
    public Response deleteMessage(@PathParam("id") int id) {
        if (messages.get(id) != null) {
            messages.remove(id);
            return Response.noContent().status(Response.Status.OK).build();
        } else {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        }
    }

    @DELETE
    public void deleteAllMessages() {
        messages.clear();
    }

    @GET
    @Path("{id}")
    @Produces(MediaType.APPLICATION_XML)
    public Response getMessage(@PathParam("id") int id) {
        Message message = messages.get(id);
        if (message == null) {
            return Response.noContent().status(Response.Status.NOT_FOUND).build();
        }

        StringBuilder sb = new StringBuilder();
        sb.append("<?xml version=\"1.0\"?>");
        sb.append("<message id=\"" + message.getId() + "\">");
        sb.append("<timestamp>" + message.getTimestamp() + "</timestamp>");
        sb.append("<text>" + message.getText() + "</text>");
        sb.append("</message>");
        return Response.ok(sb.toString()).build();
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Response getAllMessages() {
        Object[] values = messages.values().toArray();
        Arrays.sort(values);
        StringBuilder sb = new StringBuilder();
        sb.append("<?xml version=\"1.0\"?>");
        sb.append("<messages>");
        for (int i = 0; i < values.length; i++) {
            Message message = (Message) values[i];
            sb.append("<message id=\"" + message.getId() + "\">");
            sb.append("<timestamp>" + message.getTimestamp() + "</timestamp>");
            sb.append("<text>" + message.getText() + "</text>");
            sb.append("</message>");
        }
        sb.append("</messages>");
        return Response.ok(sb.toString()).build();
    }
```

```
    private synchronized static int getId() {
        return ++count;
    }

    private static String getTimestamp() {
        return new Timestamp(System.currentTimeMillis()).toString();
    }
}
```

`MessagesResource` verwendet zur Kennzeichnung der HTTP-Methoden die Annotationen `@POST`, `@GET` (siehe Kapitel 10.3), `@PUT` und `@DELETE`.

Die Annotation `@Consumes` legt fest, welchen Medientyp die so annotierte Methode für die Daten in der HTTP-Anfrage erwartet. In unserem Beispiel:

`MediaType.TEXT_PLAIN` (unformatierter Text),
`MediaType.APPLICATION_FORM_URLENCODED` (URL-codierte Daten).

Die Annotation `@Context` zu einem Methodenparameter (hier bei `newMessage1` und `newMessage2`) sorgt dafür, dass das Laufzeitsystem bestimmte Objekte injiziert.

Das Interface `javax.ws.rs.core.UriInfo` bietet verschiedene Methoden zur Abfrage von Informationen über den URI der Anfrage.

Die Anweisung

```
URI location =
    info.getAbsolutePathBuilder().path(String.valueOf(id)).build();
```

ermittelt den absoluten Pfad der Anfrage in einem `UriBuilder`, erweitert diesen um den Pfad mit der Id der Message und baut damit den kompletten Pfad auf.

`static Response.ResponseBuilder created(URI location)`
erzeugt einen `ResponseBuilder` für die neu angelegte Ressource und setzt den Location-Header für die HTTP-Antwort.

`static Response.ResponseBuilder noContent()`
erzeugt einen `ResponseBuilder` für eine "leere" Antwort.

`static Response.ResponseBuilder ok(Object entity)`
erzeugt einen `ResponseBuilder`, der eine Repräsentation enthält.

`Response.ResponseBuilder status(Response.Status status)`
setzt den HTTP-Status für die Antwort, hier: 301 (Moved Permanently).

Ist der Client ein Web Browser so führt der Status 301 zur sofortigen Weiterleitung (Redirect mit GET) an die im Location-Header angegebene Adresse.

Die in der Methode `newMessage2` verwendete Annotation `@FormParam` erlaubt die Extraktion von per POST gesendeten Feldwerten eines HTML-Formulars.

Die Methoden `updateMessage`, `deleteMessage` und `getMessage` sind jeweils mit der zusätzlichen Annotation `@Path({id})` versehen. `{id}` ist ein so genannter *Template-Parameter*.

Lautet der konkrete Ressourcenpfad der Anfrage beispielsweise

`http://localhost:8080/rest/messages/123`,

so ist 123 dem Parameter `id` zugeordnet.

Die Annotation `@PathParam` bei einem Methodenparameter injiziert den Wert des entsprechenden Template-Parameters. Im obigen Beispiel erhält der Parameter `id` den Wert 123.

WADL

Der Aufruf des URL `http://localhost:8080/rest/application.wadl` im Web Browser liefert die Beschreibung des REST-Service in einem XML-basierten Format: *Web Application Description Language* (*WADL*). Diese Beschreibung wird aus der Implementierung des Service automatisch generiert.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc xmlns:jersey="http://jersey.java.net/"
        jersey:generatedBy="Jersey: 2.16 2015-02-11 11:28:50"/>
    <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="..."/>
    <grammars/>
    <resources base="http://localhost:8080/rest/">
        <resource path="/messages">
            <method id="deleteAllMessages" name="DELETE"/>
            <method id="getAllMessages" name="GET">
                <response>
                    <representation mediaType="application/xml"/>
                </response>
            </method>
            <method id="newMessage1" name="POST">
                <request>
                    <representation mediaType="text/plain"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
            <method id="newMessage2" name="POST">
                <request>
                    <representation mediaType="application/x-www-form-urlencoded">
                        <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                                name="text" style="query" type="xs:string"/>
                    </representation>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
```

```
            <resource path="{id}">
                <param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="id"
                        style="template" type="xs:int"/>
                <method id="getMessage" name="GET">
                    <response>
                        <representation mediaType="application/xml"/>
                    </response>
                </method>
                <method id="deleteMessage" name="DELETE">
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
                <method id="updateMessage" name="PUT">
                    <request>
                        <representation mediaType="text/plain"/>
                    </request>
                    <response>
                        <representation mediaType="*/*"/>
                    </response>
                </method>
            </resource>
        </resource>
    </resources>
</application>
```

Der Client

Kommandozeilenprogramm `cURL`:

```
curl -i -H "Content-Type: text/plain" -X POST
http://localhost:8080/rest/messages -d "Nachricht A"

curl -i -H "Content-Type: text/plain" -X POST
http://localhost:8080/rest/messages -d "Nachricht B"

curl -i -H "Content-Type: text/plain" -X POST
http://localhost:8080/rest/messages -d "Nachricht C"

curl -i http://localhost:8080/rest/messages
curl -i http://localhost:8080/rest/messages/1

curl -i -X HEAD http://localhost:8080/rest/messages
curl -i -X HEAD http://localhost:8080/rest/messages/1

curl -i -H "Content-Type: text/plain" -X PUT
http://localhost:8080/rest/messages/1 -d "Nachricht AA"
curl -i http://localhost:8080/rest/messages/1

curl -i -X DELETE http://localhost:8080/rest/messages/2
curl -i http://localhost:8080/rest/messages

curl -i -X DELETE http://localhost:8080/rest/messages
curl -i http://localhost:8080/rest/messages

curl -i -X OPTIONS http://localhost:8080/rest/messages
```

```
curl -i -X OPTIONS http://localhost:8080/rest/messages/1
```

Die Kommandos müssen jeweils in einer einzigen Zeile eingegeben werden.

Die mit den HTTP-Methoden HEAD und OPTIONS verbundene Funktionalität wird vom Laufzeitsystem automatisch bereitgestellt und muss nicht vom Entwickler implementiert werden.

Wir behandeln hier nur den mit dem JAX-RS 2.0 Client API entwickelten Client. Der auf `HttpURLConnection` basierende Client ist im Begleitmaterial vorhanden.

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class Client2 {
   private static final String BASE_URL = "http://localhost:8080/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      post(BASE_URL + "/messages", "Nachricht A");
      post(BASE_URL + "/messages", "Nachricht B");
      post(BASE_URL + "/messages", "Nachricht C");

      get(BASE_URL + "/messages");
      get(BASE_URL + "/messages/1");

      head(BASE_URL + "/messages");
      head(BASE_URL + "/messages/1");

      put(BASE_URL + "/messages/1", "Nachricht AA");
      get(BASE_URL + "/messages/1");

      delete(BASE_URL + "/messages/2");
      get(BASE_URL + "/messages");

      delete(BASE_URL + "/messages");
      get(BASE_URL + "/messages");

      options(BASE_URL + "/messages");
      options(BASE_URL + "/messages/1");
   }

   private static void options(String uri) {
      System.out.println("\n--- OPTIONS " + uri);
      WebTarget target = client.target(uri);
      Response response = target.request().accept(MediaType.WILDCARD_TYPE)
            .options();
      System.out.println(response.readEntity(String.class));
   }
```

```
    private static void head(String uri) {
       System.out.println("\n--- HEAD " + uri);
       WebTarget target = client.target(uri);
       Response response = target.request().head();
       status(response);
    }

    private static void get(String uri) {
       System.out.println("\n--- GET " + uri);
       WebTarget target = client.target(uri);
       Response response = target.request().get();
       if (status(response) == 200)
          System.out.println(response.readEntity(String.class));
    }

    private static void post(String uri, String text) {
       System.out.println("\n--- POST " + uri);
       WebTarget target = client.target(uri);
       Entity<String> entity = Entity.entity(text, MediaType.TEXT_PLAIN);
       Response response = target.request().post(entity);
       if (status(response) == 201) {
          String location = response.getLocation().toString();
          System.out.println("Location: " + location);
       }
    }

    private static void put(String uri, String text) {
       System.out.println("\n--- PUT " + uri);
       WebTarget target = client.target(uri);
       Entity<String> entity = Entity.entity(text, MediaType.TEXT_PLAIN);
       Response response = target.request().put(entity);
       status(response);
    }

    private static void delete(String uri) {
       System.out.println("\n--- DELETE " + uri);
       WebTarget target = client.target(uri);
       Response response = target.request().delete();
       status(response);
    }

    private static int status(Response response) {
       int code = response.getStatus();
       String message = response.getStatusInfo().toString();
       System.out.println("Status: " + code + " " + message);
       return code;
    }
}
```

Weitere `Invocation.Builder`-Methoden:

```
Response options()
Response head()
Response post(Entity<?> entity)
Response put(Entity<?> entity)
Response delete()
```

Diese Methoden rufen die entsprechenden HTTP-Methoden auf.

Die Klasse `javax.ws.rs.client.Entity<T>` repräsentiert die Nutzdaten, die gesendet werden.

`static <T> Entity<T> entity(T entity, MediaType mediaType)`
erzeugt ein `Entity`-Objekt mit den Daten `entity` und dem Content-Type `mediaType`.

Weitere `Response`-Methoden:

`URI getLocation()`
liefert den Wert des Location-Headers.

`int getStatus()`
liefert den HTTP-Status der Antwort.

`Response.StatusType getStatusInfo()`
liefert die komplette Status-Information der Antwort.

Ausgabe:

```
--- POST http://localhost:8080/rest/messages
Status: 201 Created
Location: http://localhost:8080/rest/messages/1

--- POST http://localhost:8080/rest/messages
Status: 201 Created
Location: http://localhost:8080/rest/messages/2

--- POST http://localhost:8080/rest/messages
Status: 201 Created
Location: http://localhost:8080/rest/messages/3

--- GET http://localhost:8080/rest/messages
Status: 200 OK
<?xml version="1.0"?><messages><message id="1"><timestamp>2015-03-05
14:40:42.676</timestamp><text>Nachricht A</text></message><message
id="2"><timestamp>2015-03-05 14:40:42.754</timestamp><text>Nachricht
B</text></message><message id="3"><timestamp>2015-03-05
14:40:42.77</timestamp><text>Nachricht C</text></message></messages>

--- GET http://localhost:8080/rest/messages/1
Status: 200 OK
<?xml version="1.0"?><message id="1"><timestamp>2015-03-05
14:40:42.676</timestamp><text>Nachricht A</text></message>

--- HEAD http://localhost:8080/rest/messages
Status: 200 OK

--- HEAD http://localhost:8080/rest/messages/1
Status: 200 OK

--- PUT http://localhost:8080/rest/messages/1
Status: 200 OK

--- GET http://localhost:8080/rest/messages/1
Status: 200 OK
```

```
<?xml version="1.0"?><message id="1"><timestamp>2015-03-05
14:40:42.865</timestamp><text>Nachricht AA</text></message>

--- DELETE http://localhost:8080/rest/messages/2
Status: 200 OK

--- GET http://localhost:8080/rest/messages
Status: 200 OK
<?xml version="1.0"?><messages><message id="1"><timestamp>2015-03-05
14:40:42.865</timestamp><text>Nachricht AA</text></message><message
id="3"><timestamp>2015-03-05 14:40:42.77</timestamp><text>Nachricht
C</text></message></messages>

--- DELETE http://localhost:8080/rest/messages
Status: 204 No Content

--- GET http://localhost:8080/rest/messages
Status: 200 OK
<?xml version="1.0"?><messages></messages>

--- OPTIONS http://localhost:8080/rest/messages
```

Die Ausgabe entspricht der WADL-Ausgabe für diese Ressource.

```
--- OPTIONS http://localhost:8080/rest/messages/1
```

Die Ausgabe entspricht der WADL-Ausgabe für diese Ressource.

Die HTML-Datei `createMessage.html` enthält ein Formular zur Erfassung einer neuen Nachricht:

```
<html>
<head>
<title>Neue Nachricht</title>
</head>
<body>
   <form action="http://localhost:8080/rest/messages" method="POST"
        accept-charset="UTF-8">
      Text: <input type="text" name="text" size="80" /> <input type="submit"
         value="Senden" />
   </form>
</body>
</html>
```

Diese Datei kann im Web Browser als lokale Datei geöffnet werden.

Abbildung 10-2: Formulareingabe und Ergebnis

10.5 REST-Services mit Apache Tomcat veröffentlichen

Mit jedem HTTP-Server, der die Servlet-Technologie unterstützt, kann ein JAX-RS Service veröffentlicht werden, z. B. mit *Apache Tomcat* (Version 8.x). Gegenüber der bisher genutzten Möglichkeit, einen Web Service zu starten, hat der Einsatz von Tomcat mehrere Vorteile:

- Veröffentlichung mehrerer Web Services innerhalb derselben Webanwendung,
- Nutzung von Container-gesteuerten Sicherheitsfunktionen (Authentifizierung),
- höhere Performance und Skalierbarkeit.

Tomcat konfigurieren

Tomcat muss Zugriff auf die JAX-RS-Implementierung haben.

In der Datei `<Tomcat-Installationsort>/conf/catalina.properties` wird dazu die Zeile, die mit `shared.loader` beginnt, wie folgt angepasst:

```
shared.loader=<Jersey-Installationsort>/api/*.jar,<Jersey-
      Installationsort>/ext/*.jar,<Jersey-Installationsort>/lib/*.jar
```

Wichtig ist, dass hier die "normalen" Schrägstriche verwendet werden.

Die Bereitstellung von REST-Services mittels Tomcat kann auf unterschiedliche Weise erfolgen. REST-Services werden generell als Webanwendung konfiguriert. Zu jeder Webanwendung gehört ein Kontextpfad. Wir zeigen beispielhaft drei Varianten.

Variante 1

Ein REST-Service, ein Deployment Descriptor `web.xml`

`WebContent/WEB-INF/web.xml` hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <servlet>
      <servlet-name>demo.hello.MyApplication</servlet-name>
   </servlet>

   <servlet-mapping>
      <servlet-name>demo.hello.MyApplication</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Die Bytecodes des Service können in einer jar-Datei zusammengefasst werden.

Im Verzeichnis `tomcat_v1` (siehe Begleitmaterial) ausführen:

```
jar cf WebContent/WEB-INF/lib/hello.jar -C ../hello/bin demo/
```

Die Context-Datei `rest.xml` der Webanwendung `rest` hat den Inhalt

```
<Context docBase="<Pfad-zum-Projekt>/tomcat_v1/WebContent"
       reloadable="true" />
```

`rest.xml` muss anschließend nach

```
   <Tomcat-Installationsort>/conf/Catalina/localhost
```

kopiert werden.

Nun kann Tomcat mit `startup.bat` bzw. `startup.sh` gestartet werden.

Basis-URL des REST-Service ist: `http://localhost:8080/rest`.

Variante 2

Mehrere REST-Services, ein Deployment Descriptor `web.xml`

Hier müssen die Services unterschieden werden (`demo1`, `demo2`).

`WebContent/WEB-INF/web.xml` hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <servlet>
      <servlet-name>demo.hello.MyApplication</servlet-name>
   </servlet>

   <servlet>
      <servlet-name>demo.messages.MyApplication</servlet-name>
   </servlet>

   <servlet-mapping>
      <servlet-name>demo.hello.MyApplication</servlet-name>
      <url-pattern>/demo1/*</url-pattern>
   </servlet-mapping>

   <servlet-mapping>
      <servlet-name>demo.messages.MyApplication</servlet-name>
      <url-pattern>/demo2/*</url-pattern>
   </servlet-mapping>
</web-app>
```

Das Weitere entspricht Variante 1.

Basis-URL der REST-Services:

```
http://localhost:8080/rest/demo1
http://localhost:8080/rest/demo2
```

Variante 3

Mehrere REST-Services, kein Deployment Descriptor

Hier fehlt die Datei web.xml. Stattdessen müssen die Kontextpfade `demo1` und `demo2` als Annotation für die Klasse `MyApplication` angegeben werden:

```
@ApplicationPath("demo1")
public class MyApplication extends Application { ... }

@ApplicationPath("demo2")
public class MyApplication extends Application { ... }
```

Das Weitere entspricht Variante 1.

Basis-URL der REST-Services:

```
http://localhost:8080/rest/demo1
http://localhost:8080/rest/demo2
```

10.6 JAXB

JAXB (*Java Architecture for XML Binding*) ist ein API, um Java-Objekte in XML-Dokumente und umgekehrt zu transformieren. JAXB ist Teil von Java SE.

Die Umwandlung von Objekten in XML-Dokumente wird als *Marshalling* bezeichnet, der umgekehrte Weg als *Unmarshalling*.

Die Umwandlung kann mit Annotationen in der Java-Klasse beeinflusst werden.

Das folgende Beispiel greift die Klasse `Message` aus dem Kapitel 10.4 wieder auf.

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "message")
@XmlType(propOrder = { "id", "timestamp", "text" })
public class Message implements Comparable<Message> {
   private int id;
   private String timestamp;
   private String text;
```

```
    public Message() {
    }

    public Message(String text) {
       this.text = text;
    }

    public Message(int id, String timestamp, String message) {
       this.id = id;
       this.timestamp = timestamp;
       this.text = message;
    }

    @XmlAttribute
    public int getId() {
       return id;
    }

    public void setId(int id) {
       this.id = id;
    }

    public String getTimestamp() {
       return timestamp;
    }

    public void setTimestamp(String timestamp) {
       this.timestamp = timestamp;
    }

    public String getText() {
       return text;
    }

    public void setText(String text) {
       this.text = text;
    }

    @Override
    public String toString() {
       return "Message [id=" + id + ", timestamp=" + timestamp + ", text="
             + text + "]";
    }

    @Override
    public int compareTo(Message message) {
       return Integer.valueOf(id).compareTo(Integer.valueOf(message.id));
    }
}
```

JAXB liest die Werte standardmäßig über die get- und set-Methoden.

Die Annotation `@XmlRootElement` ist an der Klasse nötig, wenn sie das Wurzelelement des XML-Baumes bildet. Standardmäßig gilt der Klassenname als Tag-Name, ansonsten wird der Wert des Attributs `name` genutzt.

Die Annotation `@XmlType` mit dem Attribut `propOrder` wird genutzt, um die Reihenfolge der Elemente in einem komplexen XML-Schema-Typ festzulegen.

Die Annotation `@XmlAttribute` an einer get-Methode legt fest, dass der Wert als Attribut in XML geschrieben werden soll.

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "messages")
public class Messages {
   private Date date = new Date();
   private List<Message> entry;

   public Messages() {
      entry = new ArrayList<Message>();
   }

   public Messages(List<Message> entry) {
      this.entry = entry;
   }

   @XmlAttribute
   public Date getDate() {
      return date;
   }

   public void setDate(Date date) {
      this.date = date;
   }

   @XmlElement(name = "message")
   public List<Message> getEntry() {
      return entry;
   }

   public void setEntry(List<Message> entry) {
      this.entry = entry;
   }

   @Override
   public String toString() {
      return "Messages [date=" + date + ", entry=" + entry + "]";
   }
}
```

Die Wert des Attributs `name` der Annotation `@XmlElement` an einer get-Methode überschreibt den Tag-Namen des XML-Elements. Hier also `message` statt `entry`.

Es folgen zwei Programme, die das Marshalling und Unmarshalling demonstrieren.

```
import java.io.File;
import java.util.ArrayList;
import java.util.List;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

public class MarshalTest {
   public static void marshal(Object obj, String filename)
         throws JAXBException {

      JAXBContext context = JAXBContext.newInstance(obj.getClass());
      Marshaller marshaller = context.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
      File xmlFile = new File(filename);
      marshaller.marshal(obj, xmlFile);
   }

   public static void main(String[] args) throws JAXBException {
      Message message1 = new Message(1, "2015-01-15 13:41:05", "Nachricht A");
      Message message2 = new Message(2, "2015-01-15 14:00:00", "Nachricht B");

      List<Message> list = new ArrayList<Message>();
      list.add(message1);
      list.add(message2);

      Messages messages = new Messages(list);
      System.out.println(messages);

      marshal(messages, "messages.xml");
      marshal(message1, "message.xml");
   }
}
```

```
import java.io.File;

import javax.xml.bind.JAXB;
import javax.xml.bind.JAXBException;

public class UnmarshalTest {
   public static void main(String[] args) throws JAXBException {
      File xmlFile1 = new File("messages.xml");
       Messages messages = JAXB.unmarshal(xmlFile1, Messages.class);
       System.out.println(messages);

       File xmlFile2 = new File("message.xml");
       Message message = JAXB.unmarshal(xmlFile2, Message.class);
       System.out.println(message);
   }
}
```

JSON statt XML

Dieselben mit JABX-Annotationen versehen Klassen können auch verwendet werden, um statt XML-Dokumente JSON-Dokumente zu verarbeiten.

JSON (*JavaScript Object Notation*) ist ein sehr häufig verwendetes, platzsparendes Austauschformat insbesondere in Webanwendungen. JSON-Dokumente können direkt in JavaScript interpretiert werden. Daten werden als Liste von Schlüssel/Wert-Paaren dargestellt. Die Daten können verschachtelt werden, beispielsweise ist ein Array von JSON-Dokumenten möglich.

Beispiel:

```
{
  "date" : 1425571861245,
  "message" : [ {
    "id" : 1,
    "timestamp" : "2015-01-15 13:41:05",
    "text" : "Nachricht A"
  }, {
    "id" : 2,
    "timestamp" : "2015-01-15 14:00:00",
    "text" : "Nachricht B"
  } ]
}
```

Der *Jackson JSON Processor*[7] ermöglicht zusammen mit JAXB die Umwandlung von Java-Objekten nach und von JSON-Dokumenten. Wir nutzen die folgenden jar-Dateien:

```
jackson-core-2.5.1.jar
jackson-databind-2.5.1.jar
jackson-annotations-2.5.1.jar
jackson-module-jaxb-annotations-2.5.1.jar
```

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import com.fasterxml.jackson.core.JsonGenerationException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;
import com.fasterxml.jackson.module.jaxb.JaxbAnnotationIntrospector;
```

[7] http://wiki.fasterxml.com/JacksonHome

```
public class MarshalTest {
   public static void marshal(Object obj, String filename)
         throws JsonGenerationException, JsonMappingException, IOException {

      ObjectMapper mapper = new ObjectMapper();
      mapper.setAnnotationIntrospector(new JaxbAnnotationIntrospector(mapper
            .getTypeFactory()));
      mapper.enable(SerializationFeature.INDENT_OUTPUT);
      File jsonFile = new File(filename);
      mapper.writeValue(jsonFile, obj);
   }

   public static void main(String[] args) throws Exception {
      Message message1 = new Message(1, "2015-01-15 13:41:05", "Nachricht A");
      Message message2 = new Message(2, "2015-01-15 14:00:00", "Nachricht B");

      List<Message> list = new ArrayList<Message>();
      list.add(message1);
      list.add(message2);

      Messages messages = new Messages(list);
      System.out.println(messages);

      marshal(messages, "messages.json");
      marshal(message1, "message.json");
   }
}
```

```
import java.io.File;
import java.io.IOException;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.module.jaxb.JaxbAnnotationIntrospector;

public class UnmarshalTest {
   public static void main(String[] args) throws JsonParseException,
         JsonMappingException, IOException {

      ObjectMapper mapper = new ObjectMapper();
      mapper.setAnnotationIntrospector(new JaxbAnnotationIntrospector(mapper
            .getTypeFactory()));

      File jsonFile1 = new File("messages.json");
      Messages messages = mapper.readValue(jsonFile1, Messages.class);
      System.out.println(messages);

      File jsonFile2 = new File("message.json");
      Message message = mapper.readValue(jsonFile2, Message.class);
      System.out.println(message);
   }
}
```

10.7 JAXB in REST-Services und -Clients nutzen

JAX-RS unterstützt automatisch das Marshalling und Unmarshalling zur Umwandlung von Objekten in XML-Dokumente und umgekehrt. Das vereinfacht die Implementierung von Services und Clients, da XML-Code nicht mehr vom Entwickler selbst verarbeitet werden muss.

Das folgende Beispiel nutzt die beiden mit JAXB-Annotationen versehenen Klassen `Message` und `Messages` aus Kapitel 10.6.

Als Methodenparameter- und Rückgabetyp können die Klassen `Message` bzw. `Messages` direkt eingesetzt werden.

```
package demo.jaxb.xml;

import java.net.URI;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("/messages")
public class MessagesResource {
   private static Map<Integer, Message> messages;
   private static int count;

   static {
      messages = new ConcurrentHashMap<Integer, Message>();
   }

   @POST
   @Consumes(MediaType.APPLICATION_XML)
   public Response newMessage(@Context UriInfo uriInfo, Message message) {
      int id = getId();
      message.setId(id);
      message.setTimestamp(getTimestamp());
      messages.put(id, message);
      URI location = uriInfo.getAbsolutePathBuilder()
            .path(String.valueOf(id)).build();
      return Response.created(location).build();
   }
```

```
@POST
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public Response newMessage(@Context UriInfo uriInfo,
      @FormParam("text") String text) {
   int id = getId();
   Message message = new Message(id, getTimestamp(), text);
   messages.put(id, message);
   URI location = uriInfo.getAbsolutePathBuilder()
         .path(String.valueOf(id)).build();
   return Response.created(location)
         .status(Response.Status.MOVED_PERMANENTLY).build();
}

@PUT
@Path("{id}")
@Consumes(MediaType.APPLICATION_XML)
public Response updateMessage(@PathParam("id") int id, Message message) {
   if (messages.get(id) == null) {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   } else {
      message.setId(id);
      message.setTimestamp(getTimestamp());
      messages.put(id, message);
      return Response.noContent().status(Response.Status.OK).build();
   }
}

@DELETE
@Path("{id}")
public Response deleteMessage(@PathParam("id") int id) {
   if (messages.get(id) != null) {
      messages.remove(id);
      return Response.noContent().status(Response.Status.OK).build();
   } else {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   }
}

@DELETE
public void deleteAllMessages() {
   messages.clear();
}

@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_XML)
public Response getMessage(@PathParam("id") int id) {
   Message message = messages.get(id);
   if (message == null) {
      return Response.noContent().status(Response.Status.NOT_FOUND).build();
   }
   return Response.ok(message).build();
}

@GET
@Produces(MediaType.APPLICATION_XML)
public Messages getAllMessages() {
   Object[] values = messages.values().toArray();
   Arrays.sort(values);
   List<Message> list = new ArrayList<Message>();
```

```
      for (int i = 0; i < values.length; i++) {
         list.add((Message) values[i]);
      }
      Messages messages = new Messages(list);
      return messages;
   }

   private synchronized static int getId() {
      return ++count;
   }

   private String getTimestamp() {
      return new Timestamp(System.currentTimeMillis()).toString();
   }
}
```

Der JAX-RS Client:

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import demo.jaxb.xml.Message;
import demo.jaxb.xml.Messages;

public class Client2 {
   private static final String BASE_URL = "http://localhost:8080/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      post(BASE_URL + "/messages", "Nachricht A");
      post(BASE_URL + "/messages", "Nachricht B");
      post(BASE_URL + "/messages", "Nachricht C");

      get(BASE_URL + "/messages");
      get(BASE_URL + "/messages/1");

      head(BASE_URL + "/messages");
      head(BASE_URL + "/messages/1");

      put(BASE_URL + "/messages/1", "Nachricht AA");
      get(BASE_URL + "/messages/1");

      delete(BASE_URL + "/messages/2");
      get(BASE_URL + "/messages");

      delete(BASE_URL + "/messages");
      get(BASE_URL + "/messages");

      options(BASE_URL + "/messages");
      options(BASE_URL + "/messages/1");
   }
```

```
private static void options(String uri) {
   System.out.println("\n--- OPTIONS " + uri);
   WebTarget target = client.target(uri);
   Response response = target.request().accept(MediaType.WILDCARD_TYPE)
         .options();
   System.out.println(response.readEntity(String.class));
}

private static void head(String uri) {
   System.out.println("\n--- HEAD " + uri);
   WebTarget target = client.target(uri);
   Response response = target.request().head();
   status(response);
}

private static void get(String uri) {
   System.out.println("\n--- GET " + uri);
   WebTarget target = client.target(uri);
   Response response = target.request().get();
   if (status(response) == 200) {
      if (uri.endsWith("messages")) {
         Messages messages = response.readEntity(Messages.class);
         System.out.println(messages.getDate());
         for (Message message : messages.getEntry()) {
            System.out.println(message);
         }
      } else {
         System.out.println(response.readEntity(Message.class));
      }
   }
}

private static void post(String uri, String text) {
   System.out.println("\n--- POST " + uri);
   WebTarget target = client.target(uri);
   Message message = new Message(text);
   Entity<Message> entity = Entity.entity(message,
         MediaType.APPLICATION_XML);
   Response response = target.request().post(entity);
   if (status(response) == 201) {
      String location = response.getLocation().toString();
      System.out.println("Location: " + location);
   }
}

private static void put(String uri, String text) {
   System.out.println("\n--- PUT " + uri);
   WebTarget target = client.target(uri);
   Message message = new Message(text);
   Entity<Message> entity = Entity.entity(message,
         MediaType.APPLICATION_XML);
   Response response = target.request().put(entity);
   status(response);
}

private static void delete(String uri) {
   System.out.println("\n--- DELETE " + uri);
   WebTarget target = client.target(uri);
   Response response = target.request().delete();
   status(response);
}
```

```
    private static int status(Response response) {
       int code = response.getStatus();
       String message = response.getStatusInfo().toString();
       System.out.println("Status: " + code + " " + message);
       return code;
    }
}
```

JAXB und JSON

Wir wollen nun Jackson als JSON Provider in unserem JAX-RS Service verwenden. Hierzu benötigen wir zusätzlich zu den in Kapitel 10.6 aufgeführten jar-Dateien die folgenden Dateien:

```
jackson-jaxrs-base-2.5.1.jar
jackson-jaxrs-json-provider-2.5.1.jar
jersey-entity-filtering-2.16.jar
jersey-media-json-jackson-2.16.jar
```

Auf der Serverseite muss im Konstruktor von `MyApplication` das `JacksonFeature` registriert werden:

```
public MyApplication() {
   singletons.add(new MessagesResource());
   singletons.add(new JacksonFeature());
}
```

In `MessagesResource` muss nur

`MediaType.APPLICATION_XML` gegen `MediaType.APPLICATION_JSON`

ausgetauscht werden.

Dieser Austausch muss auch im Jersey-Client `Client2` erfolgen. Zusätzlich muss zu Beginn das `JacksonFeature` registriert werden:

```
client.register(JacksonFeature.class);
```

10.8 Upload und Download von Dateien

Die Klasse `java.io.InputStream` kann in den Zugriffsmethoden des REST-Service zum Lesen von Anfragedaten und für die Rückgabe von Antwortdaten genutzt werden.

Im folgenden Beispiel handelt es sich um einen Service zum Upload und Download von Dateien beliebigen Inhalts.

Der Service

```
package demo.files;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Response;

@Path("/files/{filename}")
public class FileResource {
   private final static String path = "D:/temp";

   @GET
   public InputStream download(@PathParam("filename") String filename) {
      try {
         return new FileInputStream(path + "/" + filename);
      } catch (FileNotFoundException e) {
         System.err.println(e.getMessage());
         throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
   }

   @PUT
   public Response upload(@PathParam("filename") String filename,
         InputStream input) {

      try (FileOutputStream output = new FileOutputStream(path + "/"
            + filename)) {
         int c;
         while ((c = input.read()) != -1) {
            output.write(c);
         }

         input.close();
         return Response.noContent().status(Response.Status.OK).build();
      } catch (IOException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }
}
```

`javax.ws.rs.WebApplicationException` ist eine von `RuntimeException` abgeleitete Klasse. Bei Auslösung einer solchen Exception wird ein `Response` mit dem angegeben Status-Code zurückgegeben. Fehlt ein solcher Parameter wird der Status-Code 500 (Internal Server Error) zurückgegeben.

Der Client

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class Client2 {
   private static final String BASE_URL = "http://localhost:8080/rest";
   private static Client client;

   public static void main(String[] args) {
      client = ClientBuilder.newClient();

      upload("test.pdf");
      download("test.pdf");
   }

   private static void upload(String filename) {
      WebTarget target = client.target(BASE_URL + "/files/" + filename);

      try {
         InputStream input = new FileInputStream(filename);
         Entity<InputStream> entity = Entity.entity(input,
               MediaType.APPLICATION_OCTET_STREAM);
         Response response = target.request().put(entity);
         status(response);
      } catch (FileNotFoundException e) {
         System.err.println(e);
      }
   }

   private static void download(String filename) {
      WebTarget target = client.target(BASE_URL + "/files/" + filename);
      Response response = target.request().get();
      if (status(response) == 200) {
         InputStream input = response.readEntity(InputStream.class);
         try (FileOutputStream output = new FileOutputStream(filename)) {
            byte[] buffer = new byte[2048];
            int n = 0;
            while ((n = input.read(buffer)) != -1) {
               output.write(buffer, 0, n);
            }
            output.flush();
            input.close();
         } catch (IOException e) {
            System.err.println(e);
         }
      }
   }

   private static int status(Response response) {
      int code = response.getStatus();
```

```
        String message = response.getStatusInfo().toString();
        System.out.println("Status: " + code + " " + message);
        return code;
    }
}
```

10.9 Verwaltung von Kontaktdaten als REST-Service

Im Folgenden entwickeln wir einen Service zur Verwaltung von Kontaktdaten (Name, Mailadresse, Kommentar) in einer Datenbank. Hierzu benutzen wir das relationale Datenbanksystem *H2*.[8]

Wir betreiben die H2-Datenbank mit dem Namen `contactsdb` im eingebetteten Modus, d. h. Datenbank und Java-Anwendung laufen im selben Prozess. Der Zugriff auf die Datenbank erfolgt über das *JDBC-API*. Der *JDBC-Treiber* von H2 `h2-x.x.x.jar` muss in den Klassenpfad der Anwendung eingebunden werden.

Der URL der Datenbank lautet:

```
jdbc:h2:~/contactsdb
```

`~/` zeigt auf das User-Verzeichnis, z. B. `C:\Users\`*`username`* bei Windows 7.

Der Service

Die Datenbankzugriffsmethoden `create`, `insert`, `update`, `delete` und verschiedene Methoden zur Auswertung (`select`) befinden sich alle in der Klasse `DBManager`.

Wir setzen hier SQL- und JDBC-Kenntnisse voraus und erläutern den Quellcode nicht.

```
package demo.contacts;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class DBManager {
   private static Connection getConnection() throws SQLException {
      return DriverManager.getConnection("jdbc:h2:~/contactsdb");
   }
```

[8] http://www.h2database.com

```
public static void create() {
   try (Connection con = getConnection()) {
      String sql = "create table if not exists contact ("
            + "id int identity,"
            + "name varchar(50),"
            + "email varchar(50),"
            + "comment varchar(100),"
            + "primary key (id))";

      Statement stmt = con.createStatement();
      stmt.executeUpdate(sql);
      stmt.close();
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
            "create: " + e.getMessage());
      throw new RuntimeException();
   }
}

public static int insert(Contact contact) {
   try (Connection con = getConnection()) {
      String sql =
          "insert into contact (name, email, comment) values (?,?,?)";
      PreparedStatement stmt = con.prepareStatement(sql,
            Statement.RETURN_GENERATED_KEYS);
      stmt.setString(1, contact.getName());
      stmt.setString(2, contact.getEmail());
      stmt.setString(3, contact.getComment());
      stmt.executeUpdate();
      ResultSet rs = stmt.getGeneratedKeys();
      int id = 0;
      if (rs.next())
         id = rs.getInt(1);
      stmt.close();
      return id;
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
            "insert: " + e.getMessage());
      throw new RuntimeException();
   }
}

public static int update(int id, Contact contact) {
   try (Connection con = getConnection()) {
      String sql =
          "update contact set name = ?, email = ?, comment = ? where id = ?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, contact.getName());
      stmt.setString(2, contact.getEmail());
      stmt.setString(3, contact.getComment());
      stmt.setInt(4, id);
      int n = stmt.executeUpdate();
      stmt.close();
      return n;
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
            "update: " + e.getMessage());
      throw new RuntimeException();
   }
}
```

```
public static int delete(int id) {
   try (Connection con = getConnection()) {
      String sql = "delete from contact where id = ?";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setInt(1, id);
      int n = stmt.executeUpdate();
      stmt.close();
      return n;
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
            "delete: " + e.getMessage());
      throw new RuntimeException();
   }
}

public static List<Contact> query() {
   try (Connection con = getConnection()) {
      String sql =
          "select id, name, email, comment from contact order by name";
      PreparedStatement stmt = con.prepareStatement(sql);
      ResultSet rs = stmt.executeQuery();
      List<Contact> list = new ArrayList<Contact>();
      while (rs.next()) {
         Contact c = new Contact();
         c.setId(rs.getInt(1));
         c.setName(rs.getString(2));
         c.setEmail(rs.getString(3));
         c.setComment(rs.getString(4));
         list.add(c);
      }
      rs.close();
      stmt.close();
      return list;
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
            "query: " + e.getMessage());
      throw new RuntimeException();
   }
}

public static List<Contact> query(String name) {
   try (Connection con = getConnection()) {
      String sql = "select id, name, email, comment from contact "
           + "where name like ? order by name";
      PreparedStatement stmt = con.prepareStatement(sql);
      stmt.setString(1, "%" + name + "%");
      ResultSet rs = stmt.executeQuery();
      List<Contact> list = new ArrayList<Contact>();
      while (rs.next()) {
         Contact c = new Contact();
         c.setId(rs.getInt(1));
         c.setName(rs.getString(2));
         c.setEmail(rs.getString(3));
         c.setComment(rs.getString(4));
         list.add(c);
      }
      rs.close();
      stmt.close();
      return list;
   } catch (Exception e) {
      Logger.getAnonymousLogger().log(Level.SEVERE,
```

```
                "query(name): " + e.getMessage());
            throw new RuntimeException();
        }
    }

    public static Contact find(int id) {
        try (Connection con = getConnection()) {
            String sql =
                "select id, name, email, comment from contact where id = ?";
            PreparedStatement stmt = con.prepareStatement(sql);
            stmt.setInt(1, id);
            ResultSet rs = stmt.executeQuery();
            Contact contact = null;
            if (rs.next()) {
                contact = new Contact();
                contact.setId(rs.getInt(1));
                contact.setName(rs.getString(2));
                contact.setEmail(rs.getString(3));
                contact.setComment(rs.getString(4));
            }
            rs.close();
            stmt.close();
            return contact;
        } catch (Exception e) {
            Logger.getAnonymousLogger().log(Level.SEVERE,
                    "find: " + e.getMessage());
            throw new RuntimeException();
        }
    }

    public static int count() {
        try (Connection con = getConnection()) {
            String sql = "select count(*) from contact";
            PreparedStatement stmt = con.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery();
            int count = 0;
            if (rs.next()) {
                count = rs.getInt(1);
            }
            rs.close();
            stmt.close();
            return count;
        } catch (Exception e) {
            Logger.getAnonymousLogger().log(Level.SEVERE,
                    "count: " + e.getMessage());
            throw new RuntimeException();
        }
    }
}
```

```
package demo.contacts;

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "contact")
@XmlType(propOrder = { "id", "name", "email", "comment" })
public class Contact {
    private int id;
```

```
    private String name;
    private String email;
    private String comment;

    public Contact() {
    }

    public Contact(String name, String email, String comment) {
       this.name = name;
       this.email = email;
       this.comment = comment;
    }

    @XmlAttribute
    public int getId() {
       return id;
    }

    public void setId(int id) {
       this.id = id;
    }

    public String getName() {
       return name;
    }

    public void setName(String name) {
       this.name = name;
    }

    public String getEmail() {
       return email;
    }

    public void setEmail(String email) {
       this.email = email;
    }

    public String getComment() {
       return comment;
    }

    public void setComment(String comment) {
       this.comment = comment;
    }

    @Override
    public String toString() {
       return "Contact [id=" + id + ", name=" + name + ", email=" + email
             + ", comment=" + comment + "]";
    }
}
```

```
package demo.contacts;

import java.util.ArrayList;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement(name = "contacts")
public class ContactList {
   private List<Contact> entry;

   public ContactList() {
      entry = new ArrayList<Contact>();
   }

   public ContactList(List<Contact> entry) {
      this.entry = entry;
   }

   @XmlElement(name = "contact")
   public List<Contact> getEntry() {
      return entry;
   }

   public void setEntry(List<Contact> entry) {
      this.entry = entry;
   }

   @Override
   public String toString() {
      return "ContactList [entry=" + entry + "]";
   }
}
```

Die in `MyApplication` registrierte Ressource `ContactListResource` enthält einen Konstruktor, der die DB-Tabelle `contact` in der Datenbank anlegt, sofern sie noch nicht existiert (siehe `DBManager.create()`). Man beachte, dass dieser Konstruktor nur ein einziges Mal mit dem Start des Servers aufgerufen wird (siehe `MyApplication`).

```
package demo.contacts;

import java.net.URI;

import javax.ws.rs.Consumes;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.UriInfo;

@Path("/contacts")
public class ContactListResource {
   public ContactListResource() {
```

```
      DBManager.create();
   }

   @POST
   @Consumes(MediaType.APPLICATION_XML)
   public Response newContact(@Context UriInfo uriInfo, Contact contact) {
      try {
         int id = DBManager.insert(contact);
         URI location = uriInfo.getAbsolutePathBuilder()
               .path(String.valueOf(id)).build();
         return Response.created(location).build();
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @POST
   @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public Response newContact(@Context UriInfo uriInfo,
         @FormParam("name") String name, @FormParam("email") String email,
         @FormParam("comment") String comment) {

      try {
         Contact contact = new Contact(name, email, comment);
         int id = DBManager.insert(contact);
         URI location = uriInfo.getAbsolutePathBuilder()
               .path(String.valueOf(id)).build();
         return Response.created(location)
               .status(Response.Status.MOVED_PERMANENTLY).build();
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @GET
   @Produces(MediaType.APPLICATION_XML)
   public ContactList getContacts() {
      try {
         return new ContactList(DBManager.query());
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @GET
   @Path("/search")
   @Produces(MediaType.APPLICATION_XML)
   public ContactList search(@DefaultValue("") @QueryParam("name") String name) {
      try {
         return new ContactList(DBManager.query(name));
      } catch (RuntimeException e) {
         System.err.println(e);
         throw new WebApplicationException();
      }
   }

   @GET
   @Path("/count")
```

```
    @Produces(MediaType.TEXT_PLAIN)
    public String getCount() {
       try {
          return String.valueOf(DBManager.count());
       } catch (RuntimeException e) {
          System.err.println(e);
          throw new WebApplicationException();
       }
    }

    @Path("{id}")
    public ContactResource getContact(@PathParam("id") int id) {
       return new ContactResource(id);
    }
}
```

Die Methode `search` enthält zwei neue Annotationen zu einem Parameter:

`@DefaultValue`

Der angegebene Wert (hier der Leerstring `""`) gilt hier, wenn ein Wert vom Client nicht mitgeliefert wurde.

`@QueryParam`

Der Wert des Query-Parameters (hier name) wird in den Java-Parameter injiziert.

Beispiel: `http://localhost:8080/rest/contacts/search?name=Musterfrau`

Der Query-Parameter `name` hat den Wert "Musterfrau".

Die mit `@Path("{id}")` annotierte Methode `getContact` erzeugt eine neue Ressource vom Typ `ContactResource` und gibt diese als Ergebnis zurück. JAX-RS nutzt diese Ressource, um die Anfrage weiter zu bearbeiten.

Eine solche Ressource, die gewissermaßen Teil einer anderen Ressource ist, wird als *Subressource* bezeichnet. Diese muss in `MyApplication` nicht registriert werden.

```
package demo.contacts;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.PUT;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class ContactResource {
   private int id;
```

```
    public ContactResource(int id) {
       this.id = id;
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Contact getContact() {
       Contact contact;
       try {
          contact = DBManager.find(id);
       } catch (RuntimeException e) {
          System.err.println(e);
          throw new WebApplicationException();
       }
       if (contact == null) {
          throw new WebApplicationException(Response.Status.NOT_FOUND);
       }
       return contact;
    }

    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public Response updateContact(Contact contact) {
       try {
          int n = DBManager.update(id, contact);
          if (n == 0) {
             return Response.noContent().status(Response.Status.NOT_FOUND)
                   .build();
          } else {
             return Response.noContent().status(Response.Status.OK).build();
          }
       } catch (RuntimeException e) {
          System.err.println(e);
          throw new WebApplicationException();
       }
    }

    @DELETE
    public Response deleteContact() {
       try {
          int n = DBManager.delete(id);
          if (n == 0) {
             return Response.noContent().status(Response.Status.NOT_FOUND)
                   .build();
          } else {
             return Response.noContent().status(Response.Status.OK).build();
          }
       } catch (RuntimeException e) {
          System.err.println(e);
          throw new WebApplicationException();
       }
    }
}
```

Der Client

```
import java.io.IOException;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

import demo.contacts.Contact;
import demo.contacts.ContactList;

public class Client2 {
   private static final String BASE_URL = "http://localhost:8080/rest";
   private static Client client;

   public static void main(String[] args) throws Exception {
      client = ClientBuilder.newClient();

      Contact contact1 = new Contact("Mustermann, Max",
            "max.mustermann@web.de", "Test");
      Contact contact2 = new Contact("Musterfrau, Maria",
            "maria.musterfrau@web.de", "Test");

      post(contact1);
      post(contact2);

      getCount();

      listContacts();

      Contact c1 = getContact(1);
      System.out.println(c1);

      Contact c2 = getContact(2);
      System.out.println(c2);

      c1.setName("Mustermann, Otto");
      c1.setEmail("otto.mustermann@web.de");
      c1.setComment("geändert");
      put(c1);

      search("Musterfrau");

      delete(2);
      listContacts();
   }

   private static void listContacts() throws IOException {
      System.out.println("\n--- listContacts");
      WebTarget target = client.target(BASE_URL + "/contacts");
      Response response = target.request().accept(MediaType.APPLICATION_XML)
            .get();
      if (status(response) == 200) {
         ContactList all = response.readEntity(ContactList.class);
         for (Contact contact : all.getEntry()) {
            System.out.println(contact);
         }
```

```
    }
}

private static Contact getContact(int id) throws IOException {
    System.out.println("\n--- getContact");
    WebTarget target = client.target(BASE_URL + "/contacts/" + id);
    Response response = target.request().accept(MediaType.APPLICATION_XML)
          .get();
    Contact contact = null;
    if (status(response) == 200) {
       contact = response.readEntity(Contact.class);
    }
    return contact;
}

private static void post(Contact contact) throws IOException {
    System.out.println("\n--- post");
    WebTarget target = client.target(BASE_URL + "/contacts");
    Entity<Contact> entity = Entity.entity(contact,
          MediaType.APPLICATION_XML);
    Response response = target.request().post(entity);
    if (status(response) == 201) {
       String location = response.getLocation().toString();
       System.out.println("Location: " + location);
    }
}

private static void getCount() throws Exception {
    System.out.println("\n--- getCount");
    WebTarget target = client.target(BASE_URL + "/contacts/count");
    Response response = target.request().get();
    if (status(response) == 200)
       System.out.println(response.readEntity(String.class));
}

private static void put(Contact contact) throws IOException {
    System.out.println("\n--- put");
    WebTarget target = client.target(BASE_URL + "/contacts/"
          + contact.getId());
    Entity<Contact> entity = Entity.entity(contact,
          MediaType.APPLICATION_XML);
    Response response = target.request().put(entity);
    status(response);
}

private static void search(String name) throws IOException {
    System.out.println("\n--- search");
    WebTarget target = client.target(BASE_URL + "/contacts/search")
          .queryParam("name", name);
    Response response = target.request().accept(MediaType.APPLICATION_XML)
          .get();
    if (status(response) == 200) {
       ContactList list = response.readEntity(ContactList.class);
       for (Contact contact : list.getEntry()) {
          System.out.println(contact);
       }
    }
}

private static void delete(int id) throws IOException {
    System.out.println("\n--- delete");
```

```
        WebTarget target = client.target(BASE_URL + "/contacts/" + id);
        Response response = target.request().delete();
        status(response);
    }

    private static int status(Response response) {
        int code = response.getStatus();
        String message = response.getStatusInfo().toString();
        System.out.println("Status: " + code + " " + message);
        return code;
    }
}
```

Die `WebTarget`-Methode

```
WebTarget queryParam(String name, Object... values)
```

erzeugt einen Query-Parameter.

10.10 HTTP-Authentifizierung und SSL

HTTP bietet Mechanismen, um den Zugriff auf Ressourcen zu sichern, z. B.

- SSL und HTTPS, um die komplette Kommunikation zwischen Client und Server zu verschlüsseln,
- Authentifizierung mit HTTP Basic Authentication.

Basic Authentication

JAX-RS unterstützt die Authentifizierung mit Benutzername und Passwort nach dem Verfahren *Basic Authentication*.[9]

Ein Client, der das JAX-RS Client API nutzt, muss dieses Feature zu Beginn registrieren:

```
HttpAuthenticationFeature feature =
HttpAuthenticationFeature.basic("hugo", "oguh");
client = ClientBuilder.newClient();
client.register(feature);
```

Benutzername und Passwort werden mit einem Doppelpunkt verkettet, dann nach dem Base64-Verfahren[10] codiert und in einem speziellen HTTP-Header an den Server geschickt.

---

[9] `http://www.ietf.org/rfc/rfc2617.txt`

[10] `http://de.wikipedia.org/wiki/Base64`

Zu beachten ist, dass es sich hierbei um keine echte Verschlüsselung von Benutzername und Passwort mit dem Ziel der Geheimhaltung handelt. Base64 bildet 8-Bit-Binärdaten auf eine Zeichenkette aus Buchstaben ohne Umlaute, Ziffern, "+" und "/" ab.

Beispiel:

Ist der Benutzername "hugo" und das Passwort "oguh", so lautet der HTTP-Header:

```
Authorization: Basic aHVnbzpvZ3Vo
```

"aHVnbzpvZ3Vo" wird gemäß Base64-Verfahren auf der Serverseite zu "hugo:oguh" decodiert.

Berechtigungsmodell in unserem Beispiel

Benutzer mit der Rolle "user" sollen nur mit den HTTP-Methoden GET und POST auf alle Ressourcen zugreifen können. Benutzer mit der Rolle "admin" unterliegen keiner Einschränkung.

Zur Bereitstellung des Service nutzen wir Tomcat. Der Server muss für die Authentifizierung konfiguriert werden.

1. Festlegung der Benutzer und Rollen

Die zugriffsberechtigten User werden der Einfachheit halber in der XML-Datei

```
<Tomcat-Installationsort>/conf/tomcat-users.xml
```

erfasst:

```
<role rolename="user"/>
<role rolename="admin"/>
<user username="hugo" password="oguh" roles="admin"/>
<user username="emil" password="lime" roles="user"/>
<user username="pit" password="tip" roles="user"/>
```

2. Ergänzungen in der Datei web.xml

Die Deskriptordatei web.xml hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <servlet>
      <servlet-name>demo.security.MyApplication</servlet-name>
   </servlet>

  <servlet-mapping>
    <servlet-name>demo.security.MyApplication</servlet-name>
       <url-pattern>/*</url-pattern>
   </servlet-mapping>
```

```
   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Nachrichten erstellen</web-resource-name>
         <url-pattern>/*</url-pattern>
         <http-method>POST</http-method>
         <http-method>GET</http-method>
      </web-resource-collection>
      <auth-constraint>
         <role-name>user</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
   </security-constraint>

   <security-constraint>
      <web-resource-collection>
         <web-resource-name>Nachrichten verwalten</web-resource-name>
         <url-pattern>/*</url-pattern>
      </web-resource-collection>
      <auth-constraint>
         <role-name>admin</role-name>
      </auth-constraint>
      <user-data-constraint>
         <transport-guarantee>NONE</transport-guarantee>
      </user-data-constraint>
   </security-constraint>

   <login-config>
      <auth-method>BASIC</auth-method>
      <realm-name>jaxrs</realm-name>
   </login-config>

   <security-role>
      <role-name>user</role-name>
   </security-role>

   <security-role>
      <role-name>admin</role-name>
   </security-role>
</web-app>
```

Das Element `<login-config>` legt das Authentifizierungsverfahren fest.

`<security-role>` legt die verwendeten Rollen fest.

`<web-resource-collection>` im Element `<security-constraint>` bestimmt, welche Ressourcen (`<url-pattern>`) für welche Zugriffsmethoden (`<http-method>`) zu schützen sind. Fehlt `<http-method>`, so sind alle HTTP-Methoden geschützt. `<auth-constraint>` legt fest, welche Rollen auf die geschützte Ressource zugreifen dürfen. `<transport-guarantee>` im Element `<user-data-constraint>` definiert den Grad des Datenschutzes. Der Wert `CONFIDENTIAL` verlangt Verschlüsselung wie sie z. B. mit SSL erreicht wird (siehe weiter unten).

```
@POST
@Consumes(MediaType.APPLICATION_XML)
public Response newMessage(@Context UriInfo uriInfo,
      @Context SecurityContext sec, Message message) {
   if (!sec.isUserInRole("admin")) {
      System.out.println(sec.getUserPrincipal().getName()
            + " hat eine neue Nachricht erstellt.");
   }

   int id = getId();
   message.setId(id);
   message.setTimestamp(getTimestamp());
   messages.put(id, message);
   URI location = uriInfo.getAbsolutePathBuilder()
         .path(String.valueOf(id)).build();
   return Response.created(location).build();
}
```

In der Methode `newMessage` wird eine Instanz vom Typ `SecurityContext` injiziert. Das Interface `javax.ws.rs.core.SecurityContext` hat u. a. die folgenden Methoden:

`boolean isUserInRole(String role)`
: prüft, ob der aktuelle User zu einer bestimmten Rolle gehört.

`Principal getUserPrincipal()`
: liefert eine Instanz vom Typ des Interface `java.security.Principal`, das den aktuellen User repräsentiert und dessen Methode `getName` den Usernamen liefert.

SSL und HTTPS

Secure Sockets Layer (*SSL*), die alte Bezeichnung für *Transport Layer Security* (*TLS*), ist ein Netzwerkprotokoll um Daten verschlüsselt über *HTTPS* (*HyperText Transfer Protocol Secure*) zu übertragen.[11] Der Server muss bei diesem Verfahren seine Identität gegenüber dem Client beweisen. Das geschieht mit einem *Zertifikat*, das von einer vertrauenswürdigen Zertifizierungsstelle stammen sollte. Wir nutzen hier allerdings zur Vereinfachung ein selbst signiertes Zertifikat. Wurde das Zertifikat vom Client akzeptiert und ist die Verbindung aufgebaut, werden die Daten zwischen Client und Server verschlüsselt übertragen.

Wir erweitern das Beispiel aus diesem Unterkapitel. Dazu sind die folgenden Schritte nötig.

---

[11] `http://de.wikipedia.org/wiki/Transport_Layer_Security`
`http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol_Secure`

1. Änderung in web.xml

In der Datei `web.xml` muss

```
<transport-guarantee>NONE</transport-guarantee>
```

durch

```
<transport-guarantee>CONFIDENTIAL</transport-guarantee>
```

ersetzt werden.

2. Tomcat für SSL einrichten

In `<Tomcat-Installationsort>/conf/server.xml` wird ein Konnektor für den Port 8443 eingerichtet:

```
<Connector port="8443"
   protocol="org.apache.coyote.http11.Http11NioProtocol"
   maxThreads="150" SSLEnabled="true" scheme="https" secure="true"
   clientAuth="false" sslProtocol="TLS" keystorePass="tomcat" />
```

3. Server-Zertifikat erzeugen

Für den Server muss ein Zertifikat erzeugt werden. Hierzu nutzen wir das Java-Tool `keytool`. Die Kommandos müssen jeweils in einer einzigen Zeile eingegeben werden.

a) Schlüsselpaar (public key/private key) erstellen:

```
keytool -genkeypair -dname "CN=demo" -ext san=ip:127.0.0.1 -alias rest
-keypass tomcat -storepass tomcat
```

Der Speicher zur Aufbewahrung der Schlüssel wird mit dem Namen `.keystore` im User-Verzeichnis (`C:\Users\`*`username`* bei Windows 7) eingerichtet. Antragsteller ist `demo`, der alternative Antragstellername (`san` = subject alternative name) ist die IP-Adresse `127.0.0.1`. Hier kann statt `127.0.0.1` auch die im Netz gültige IP-Adresse des Rechners, auf dem der Server läuft, angegeben werden.

b) Zertifikat exportieren:

```
keytool -exportcert -alias rest -file %USERPROFILE%\tomcat.cer -storepass
tomcat
```

Das Zertifikat wird unter `tomcat.cer` im User-Verzeichnis (`%USERPROFILE%` bei Windows 7) abgelegt.

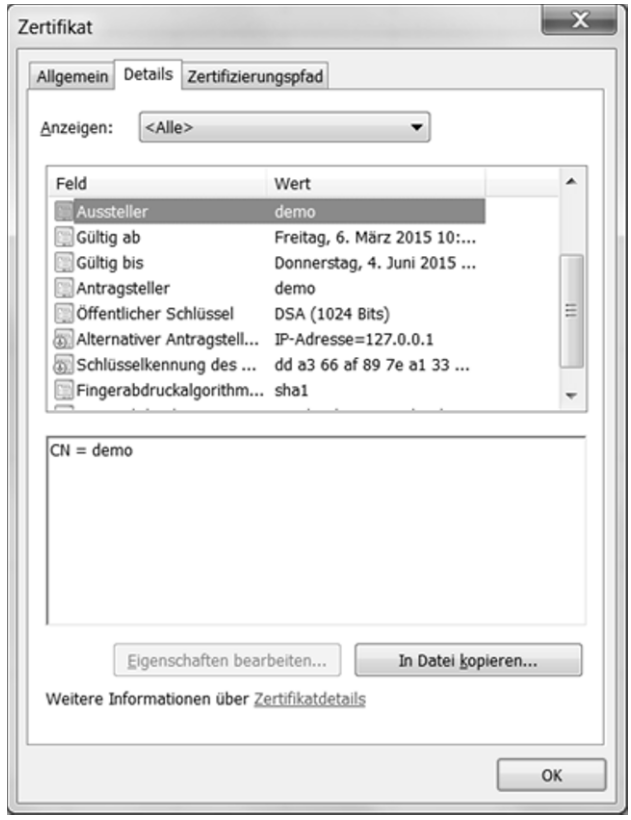

Abbildung 10-3: Zertifikat

4. Zertifikat in den Truststore des Client aufnehmen

Das Zertifikat wird dem Client ausgehändigt. In unserem Beispiel befinden sich Client und Server auf demselben Rechner, deshalb kann `tomcat.cer` an derselben Stelle bleiben.

Der Client importiert nun das Zertifikat in seinen Truststore:

```
keytool -importcert -noprompt -alias rest -file %USERPROFILE%\tomcat.cer
-keystore %USERPROFILE%\.truststore -storepass secret
```

Der Client muss nun für SSL konfiguriert werden.

```
public class JerseyClientSSL {
   private static final String BASE_URL = "https://127.0.0.1:8443/rest";
   private static Client client;
```

```
    public static void main(String[] args) {
       String home = System.getProperty("user.home");
       System.setProperty("javax.net.ssl.trustStore", home + "/.truststore");
       System.setProperty("javax.net.ssl.trustStorePassword", "secret");

       ...
    }

    ...
}
```

Die IP-Adresse im `BASE_URL` wird mit der IP-Adresse im Zertifikat verglichen. Der Pfadname des Truststore und das Passwort werden über Systemeigenschaften gesetzt.

10.11 Aufgaben

1. Durch eine Hash-Funktion wird eine Nachricht auf einen Hashwert mit einer festen Länge abgebildet. In Java steht hierfür die Klasse `java.security.MessageDigest` zur Verfügung.

   Entwickeln Sie einen REST-Service, der für einen vorgegebenen Algorithmus (MD5, SHA-1, SHA-256 oder SHA-512) und einen Text den Hashwert berechnet und als Antwort liefert. Der Client sendet die Daten im URL-codierten Format über POST.

   Entwickeln Sie auch einen Java-Client mit dem JAX-RS Client API sowie ein HTML-Formular.

2. In Tomcat können auch Datenbanken als *JNDI Datasource* konfiguriert werden. Tomcat arbeitet dann mit einem *Database Connection Pool*, um die Client-bezogenen Verbindungen zur Datenbank optimal zu verwalten. Dazu muss der Context wie folgt ergänzt werden:

   ```
   <Context docBase="<Pfad-zum-Projekt>/aufgabe2_tomcat/webapp"
         reloadable="true">
      <Resource name="jdbc/myDB" auth="Container"
         type="javax.sql.DataSource"
         maxTotal="100" maxIdle="30" maxWaitMillis="10000"
         driverClassName="org.h2.Driver" url="jdbc:h2:~/contactsdb" />
   </Context>
   ```

   In der Java-Anwendung erfolgt der Zugriff auf eine Verbindung wie folgt:

   ```
   Context ctx = new InitialContext();
   DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/myDB");
   ```

   Ändern Sie das Beispiel aus Kapitel 10.9 so, dass die Datenbankverbindungen nun von Tomcat selbst verwaltet werden.

3. Realisieren Sie einen REST-Service zur Verwaltung von Aufgaben (Todos). Neben einer eindeutigen Id soll eine Aufgabe den Usernamen, die Beschreibung der eigentlichen Aufgabe und den Status für die Erledigung (`true`/`false`) enthalten. Die Daten sollen in einer von Tomcat verwalteten Datenbank gespeichert werden (siehe Aufgabe 2). Jeder User soll seine Aufgaben verwalten können und keinen Zugriff auf Aufgaben anderer User haben. Nutzen Sie die HTTP-Authentifizierung *Basic Authentication* aus Kapitel 10.10. Innerhalb der Java-Anwendung kann der aktuelle User über den `SecurityContext` abgefragt werden. Die Datenbankzugriffsmethoden müssen den User berücksichtigen (WHERE-Klausel in SQL).

   Entwickeln Sie auch einen Client mit dem JAX-RS Client API.

Quellen im Internet

Beispielprogramme und Lösungen zu den Aufgaben

Den Zugang zum Begleitmaterial finden Sie auf der Website des Verlags

```
www.springer-vieweg.de
```

bei den bibliographischen Angaben zu diesem Buch.

Java Standard Edition

```
http://www.oracle.com/technetwork/java/javase/
```

Hier finden Sie die neueste Version zur *Java Standard Edition* (Java SE) für diverse Plattformen sowie die zugehörige Dokumentation. Beachten Sie die für Ihre Plattform zutreffende Installationsanweisung.

Java-Entwicklungsumgebungen

Eclipse	`http://www.eclipse.org`
NetBeans	`http://www.netbeans.org`
IntelliJ IDEA	`http://www.jetbrains.com/idea/`
JDeveloper	`http://www.oracle.com/technetwork/developer-tools/jdev/`
JCreator	`http://www.jcreator.com`

Protokollspezifikationen

UDP	`http://www.ietf.org/rfc/rfc768.txt`
TCP	`http://www.ietf.org/rfc/rfc793.txt`
Ports	`http://www.iana.org/assignments/port-numbers`
HTTP 1.0	`http://www.ietf.org/rfc/rfc1945.txt`
HTTP 1.1	`http://www.ietf.org/rfc/rfc2616.txt`
HTTP Status-Code	`http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`
WebSocket	`http://www.ietf.org/rfc/rfc6455.txt`
MIME:	`http://www.ietf.org/rfc/rfc1521.txt`
Base64	`http://www.ietf.org/rfc/rfc2045.txt`
Basic Authentication	`http://www.ietf.org/rfc/rfc2617.txt`
XML-RPC	`http://www.xmlrpc.com`

Software und Tools

Apache ActiveMQ	`http://activemq.apache.org`
Apache Tomcat	`http://tomcat.apache.org`
Apache XML-RPC *	`http://archive.apache.org/dist/ws/xmlrpc/binaries/`
cURL	`http://curl.haxx.se`
H2	`http://www.h2database.com`

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3, © Springer Fachmedien Wiesbaden 2015

Jackson JSON Processor *	http://wiki.fasterxml.com/JacksonHome
Jersey	https://jersey.java.net
Jersey-Container *	http://mvnrepository.com/artifact/org.glassfish.jersey.containers/jersey-container-jdk-http/2.16
JMS 1.1	http://www.oracle.com/technetwork/java/javaee/docs-136352.html
Metro Web Service Stack	http://metro.java.net
Online-Tool Message Digester	http://www.freeformatter.com/message-digest.html
Tyrus Standalone Client *	https://tyrus.java.net

Die mit * markierten Produkte sind im Begleitmaterial vorhanden.

Literaturhinweise

Zur Vertiefung einiger Themen, die hier nicht im Vordergrund stehen bzw. im Rahmen dieses Buches nicht ausführlich behandelt werden können, sind die folgenden Quellen gut geeignet.

Anwendungsentwicklung mit Java

- Abts, D.: Grundkurs Java. Von den Grundlagen bis zu Datenbank- und Netzanwendungen. Springer Vieweg, 8. Auflage 2015
- Build-Management-Tool Apache Ant:
  `http://ant.apache.org/manual/index.html`
- Goll, J.: Architektur- und Entwurfsmuster der Softwaretechnik. Springer Vieweg, 2. Auflage 2014
- Künneth, T.: Einstieg in Eclipse: Die Werkzeuge für Java-Entwickler. Galileo Computing, 5. Auflage 2014
- Vogt, C.: Nebenläufige Programmierung: Ein Arbeitsbuch mit UNIX/Linux und Java. Hanser 2012

Verteilte Systeme

- Bengel, G.: Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing. Springer Vieweg, 4. Auflage 2014
- Comer, D. E.: TCP/IP - Studienausgabe: Konzepte, Protokolle, Architekturen. mitp 2011
- Harold, E. R.: Java Network Programming. O'Reilly Media, 4. Auflage 2013
- Mandl, P.: Masterkurs Verteilte betriebliche Informationssysteme. Springer Vieweg 2009
- Oechsle, R.: Parallele und verteilte Anwendungen in Java. Hanser, 4. Auflage 2014

Web-Technologien

- Gorski, P. L.; Lo Iacono, L.; Nguyen, H. V.: WebSockets. Moderne HTML5-Echtzeitanwendungen entwickeln. Hanser 2015
- Meinel, C.; Sack, H.: Web-Technologien. Grundlagen, Web-Programmierung, Suchmaschinen, Semantic Web. Springer 2015

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3, © Springer Fachmedien Wiesbaden 2015

- Münz, S.; Gull, C.: HTML5 Handbuch. Franzis Verlag, 10. Auflage 2014
- Tilkov, S.: REST und HTTP. dpunkt.verlag, 2. Auflage 2011
- Vonhoegen, H.: Einstieg in XML: Grundlagen, Praxis, Referenz. Galileo Computing, 7. Auflage 2013

Datenbanken, SQL und JDBC

- JDBC Database Access Tutorial:
  `http://docs.oracle.com/javase/tutorial/jdbc/`
- Steiner, R.: Grundkurs Relationale Datenbanken. Springer Vieweg, 8. Auflage 2014
- Unterstein. M., Matthiessen, G.: Anwendungsentwicklung mit Datenbanken. Springer Vieweg, 5. Auflage 2013
- Unterstein. M., Matthiessen, G.: Relationale Datenbanken und SQL in Theorie und Praxis. Springer Vieweg, 5. Auflage 2012

Sachwortverzeichnis

D. Abts, *Masterkurs Client/Server-Programmierung mit Java,*
DOI 10.1007/978-3-658-09921-3, © Springer Fachmedien Wiesbaden 2015