Sandro Leuchter

Internet-Kommunikation

Steinbeis-Edition

Reihe „Verteilte Architekturen“ | Band 1

Sandro Leuchter hat seit 2016 die Professur für verteilte und mobile Anwendungen der Fakultät für Informatik der Hochschule Mannheim inne, ist Gründungsmitglied des Kompetenzzentrums Lehre & Lernen der Hochschule Mannheim und leitet das Steinbeis-Transferzentrum Verteilte und mobile Anwendungen. Vorher war er u. a. „Head of Software Engineering and Infrastructure Software“ bei Atlas Elektronik, einem gemeinsamen Unternehmen von ThyssenKrupp und EADS. 2009 promovierte er an der Fakultät für Verkehrs- und Maschinensysteme der Technischen Universität Berlin zum Dr.-Ing. Das Diplom in Informatik hat er 1999 ebenfalls an der Technischen Universität Berlin bekommen.

Steinbeis-Transferzentrum
Verteilte und mobile
Anwendungen

Sandro Leuchter

Internet-Kommunikation – Eine praktische Einführung mit Java

Impressum

2018 Steinbeis-Edition

Sandro Leuchter: Internet-Kommunikation. Eine praktische Einführung mit Java
Reihe „Verteilte Architekturen“ | Band 1

1. Auflage, 2018 | Steinbeis-Edition, Stuttgart
ISBN 978-3-95663-189-4

Diese Publikation ist auch als E-Book erhältlich.: ISBN 978-3-95663-190-0
Diese Publikation ist auch abrufbar unter: https://doi.org/10.5281/zenodo.1042168

Satz: Sandro Leuchter, Steinbeis-Transferzentrum Verteilte und mobile Anwendungen, Karlsruhe
Druck: e.kurz+co druck und medientechnik gmbh, Stuttgart

203683-2018-12 | www.steinbeis-edition.de

Vorwort

Die Mehrzahl der IT-Systeme basiert auf einer verteilten Architektur: Sei es, dass Menschen in sozialen Netzwerken miteinander interagieren wollen, Ressourcen in der Unternehmens-IT zwischen mehreren Nutzern geteilt oder redundant vorgehalten werden um Lastspitzen bzw. Ausfälle abzufedern oder dass Sensoren in weltumspannende Messnetze eingebunden werden. Für IT-Entwicklerinnen und Entwickler aus nahezu allen Anwendungsbereichen ist das Aufteilen der Funktionalität auf voneinander getrennte Einheiten, die durch Netzwerkkommunikation zusammenwirken, heute allgegenwärtig. Immer mehr wird bei der Umsetzung auf Internet-Technologien gesetzt. Das ermöglicht die Implementierung durchgehender Funktionsketten ohne Protokollbrüche und erleichtert die Integration von bisher isolierten Systemen im Zuge der Digitalisierung aller Lebensbereiche.

Die Entwurfsparadigmen verteilter Architekturen haben dabei vielfältige Auswirkungen auf die nichtfunktionalen Anforderungen an IT-Systeme. Sie ermöglichen z. B. die Berücksichtigung von Ausfallsicherheit und Skalierbarkeit, erleichtern aber auch gleichzeitig das unerlaubte Ausspähen oder Stören von Komponenten verteilter Anwendungen. Deshalb ist es für alle Rollen, die zum Entwicklungsprozess moderner IT-Systeme beitragen, immens wichtig, die Funktionsweise der Internet-Kommunikation zu kennen und ein breites Repertoire für Gestaltungsalternativen für verteilte Architekturen zur Hand zu haben.

Diese Buchreihe ist an alle gerichtet, die aufbauend auf grundlegenden Programmierkenntnissen die Kompetenz erlangen wollen, verteilte Architekturen zu planen, umzusetzen und zu betreiben. Die Reihe ist als Lehrbuch für das Selbststudium konzipiert. Am Ende sind die Leserinnen und Leser in der Lage selbstständig alle architekturellen Anforderungen an verteilte Anwendungen zu analysieren, geeignete Architekturen für verteilte Anwendungen auszuwählen und zu bewerten, eine geeignete *Middleware* und Ablaufumgebung auszuwählen sowie verteilte Anwendungen kompetent zu entwerfen, zu implementieren und in Betrieb zu nehmen.

Die Algorithmik dieses Lehrprogramms ist zunächst leicht. Die Hürden für den Einsatz der vorgestellten Technologiekonzepte für die Umsetzung verteilter Architekturen liegen nicht in der Komplexität der Programme, sondern in der Ablaufumgebung: Aufgrund der Abhängigkeiten zwischen Elementen der Ablaufumgebung kann es ziemlich aufwändig und fehleranfällig sein eine Entwicklungsumgebung für Programmierung und Test verteilter Architekturen aufzubauen.

Deshalb ist diese Lehrbuchreihe als Praktikum aufgebaut: Zu jedem Technologiebereich gibt es eine sehr kurze Einführung in die Konzepte. Darauf folgt immer ein umfangreiches Praktikum, in dem sich Leserinnen und Leser die jeweilige Technologie mit einer Schritt-für-Schritt-Anleitung selbstständig erarbeiten.

Aktivitätsschritt 0.1:
Ziel: Aufbau des Buches verstehen
Jeder Aktivitätsschritt beschreibt eine abgrenzbare Tätigkeit, die jeweils ein konkretes Ziel hat. Zu Beginn jedes Praktikums sind die Aktivitätsschritte noch ziemlich ausführlich erläutert. In späteren Schritten werden die Funktionalität und Leistungsfähigkeit der erarbeiteten Technologie durch praktische Experimente genauer untersucht und die Anleitungen dazu werden weniger detailliert, sind aber immer durch Transfer aus den früheren Aktivitätsschritten zu meistern.

Die Aktivitätsschritte bauen immer aufeinander auf und sollten in der vorgegebenen Reihenfolge bearbeitet werden. Am Ende jedes Praktikums sollten die Leserinnen und Leser sich ein gutes überblicksartiges Verständnis der architekturellen Möglichkeiten der jeweiligen Technologie erarbeitet haben und in der Lage sein sie selbstständig einzusetzen. Zu allen Programmierpraktika gibt es Musterlösungen.
Die Auswahl der Themen geht über das Minimum hinaus, das für eine pragmatische Beherrschung aktueller Technologien für die Umsetzung verteilter Architekturen erforderlich ist. Der thematischen Erweiterung in dieser Buchreihe liegt die Überzeugung zugrunde, dass für eine kompetente Anwendung der benutzten Technologien das Verständnis für mindestens eine darunterliegende Ebene vorhanden sein sollte.

Die allerwichtigsten Konzepte und Zusammenhänge sind mit einem Ausrufezeichen am Rand markiert.

Jeder Band stellt den Stoff eines Semesters (von vier bis fünf ECTS *Credit Points*, also ca. 120-150 Stunden Aufwand) dar. Aufgrund der Vielzahl der architekturellen Ansätze ist das Konzept dieser Lehrbuchreihe: «Breite vor Tiefe». An vielen Stellen gibt es interessante weiterführende Informationen, die nicht zu den Basisinformationen des jeweiligen Themas gehören und letztlich auch nicht zu den beschriebenen Lernzielen beitragen. Leser, die einem straffen Programm folgen wollen, können diese als «Exkurs» gekennzeichneten Abschnitte überspringen.

Exkurs: Abschweifungen für mehr Kontext

Trotzdem sind diese jeweils kurz dargestellten Themen in die Lehrbuchreihe aufgenommen worden, um Leserinnen und Lesern mehr Kontext und Anknüpfungspunkte zu bieten und zu eigener Recherche zu motivieren.

Letztlich ist die Stoffauswahl und der Umfang der Praktika ziemlich knapp. Es hat sich gezeigt, dass einige Leserinnen und Leser den Stoff gerne über den Standardumfang hinaus vertiefen wollen – zum einen um die neu angeeigneten Fertigkeiten zu festigen, zum anderen um die Technologien stärker in der Tiefe zu erforschen und weitergehende Konzepte zu entdecken. Deshalb gibt es am Ende jedes Praktikums Anregungen für eigene Projekte.

Aktivitätsschritt 0.2 (fakultativ):
Ziel: Fakultative Aktivitätsschritte als Anregung begreifen
Bei über den Standardumfang hinausgehenden Aufgaben sind die Aktivitätsschritte als «fakultativ» gekennzeichnet. Das bedeutet, dass es sich um Vertiefungen handelt. Für die meisten dieser Vertiefungsaufgaben gibt es Musterlösungen wie für die regulären.

Auf der *Companion Website* `http://verteiltearchitekturen.de/` stehen die Bände der Buchreihe als PDF zum Download bereit. Außerdem gibt es dort die Eclipse-Projekte für Praktika sowie Musterlösungen als Eclipse-Projekte. Bücher und Quelltexte werden unter einer «Creative Commons BY-NC-SA»-Lizenz weitergegeben. Jeder hat also das Recht die Inhalte und Dateien der Buchreihe für nichtkommerzielle Zwecke zu verwenden und unter Autorennennung und unter den gleichen Bedingungen weiterzugeben. Dozierende bekommen auf Anfrage sehr gerne Folien sowie Aufgaben und Musterlösungen für Testate zur Verfügung gestellt.

Karlsruhe, Oktober 2018 *Sandro Leuchter*

Inhaltsverzeichnis

Teil I

Internet-Standards

Internet-Adressierung

Prinzipiell kann man für die Kommunikation in verteilten Architekturen ganz unterschiedliche Netzwerktechnologien verwenden. Im Anwendungsbereich der Prozesssteuerung chemischer Anlagen kommen dafür z. B. besondere Feldbustechnologien zum Einsatz, die spezialisierte echtzeitfähige Kommunikationsverfahren benutzen. In dieser Buchreihe wird hingegen grundsätzlich Internet-Technologie für die Netzwerkkommunikation benutzt.

In ersten Teil dieses Bandes von *Verteilte Architekturen* werden die Grundlagen für die Programmierung von Netzwerkkommunikation gelegt. Im weiteren Verlauf werden dann unterschiedliche Techniken behandelt um Nachrichten in verteilten Systemen auszutauschen. Zur Repräsentation von Absender und Empfänger ist aber immer eine Benennung und Adressierung von Rechnern im Internet erforderlich. In diesem Kapitel lernen Sie deshalb mehrere Arten der Benennung von vernetzten Rechnern kennen: zwei unterschiedliche Arten von *IP-Adressen* und textuelle Rechnernamen, die vom *Domain Name System* verwaltet werden. Außerdem geht es um die Grundlagen der Kommunikation im Internet und die dynamische Verteilung von IP-Adressen über das *Dynamic Host Configuration Protocol*.

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

Exkurs: Internet-Technologien auch bei *Internet of Things* und *Industrie 4.0*

Bei aktuellen Technologietrends wie dem *Internet of Things* oder bei *Industrie 4.0* wird die Kommunikation von Anwendungen, die früher eher mit Feldbusnetzen umgesetzt wurde, inzwischen immer mehr mit Internet-Technologie abgewickelt.

1.1 IP-Adressen

Das Internet ist ein heterogener Zusammenschluss eigenständiger Netze. Jedes angeschlossene Netz kann seine eigene Technologie verwenden und wird unabhängig von den anderen Netzen organisiert. «Das Internet» ist aus dieser Sicht ein Bündel von aufeinander abgestimmten Protokollen, die ermöglichen, dass Rechner miteinander kommunizieren können, die sich in verschiedenen Teilnetzen befinden.
Computer und andere vernetzte Geräte im Internet werden allgemein Knoten (engl. «*node*») oder *Hosts* genannt. Jeder Knoten im Internet hat mindestens eine Adresse, über die er Nachrichten von anderen Internet-Hosts empfangen kann. Die Adresse gehört zur Netzwerkverbindung und normalerweise hat ein Netzwerkanschluss (z. B. Wifi oder Ethernet) genau eine Adresse.
Internet-Adressen (genauer: *Internet Protocol*, Version 4 (IPv4) Adressen) bestehen aus einer 32-Bit Zahl, die meistens in vier Gruppen von dezimal ausgedrückten 8-Bit Zahlen (also jeweils 0-255), getrennt durch einen «`.`» wiedergegeben wird:

```
172.217.17.132, 216.58.209.68
```

Unter anderem da die Anzahl der so adressierbaren Hosts nicht ausreichend ist, wird gegenwärtig an der Einführung des *Internet Protocol*, Version 6 (IPv6) gearbeitet. Eine von mehreren Weiterentwicklungen von IPv6 im Vergleich zu IPv4 ist, dass die Adressen 128 Bit groß sind. IPv6-Adressen werden in acht Blöcke von 16 Bit Zahlen, die hexadezimal notiert werden, also aus jeweils vier Hexadezimalziffern (`0`-`F`) bestehen, repräsentiert. Die Blöcke werden durch «`:`» voneinander getrennt:

```
2a00:1450:401b:0801:0000:0000:0000:2004
```

In dieser Buchreihe wird an den meisten Stellen die IPv4-Adressierung wegen der besseren Übersichtlichkeit benutzt.

Exkurs: ARPANET

Abb. 1.1 zeigt die Netze des Internet-Vorläufers ARPANET und deren Verbundstruktur. Die Boxen mit abgerundeten Kanten bezeichnen in dieser Darstellung Netzwerke («Subnetze», teilweise verbergen sich in dieser Darstellung hinter einem Netzwerk tatsächlich mehrere Subnetze) und die Boxen mit eckigen Kanten stellen Verbindungen dazwischen dar («*Router*»).

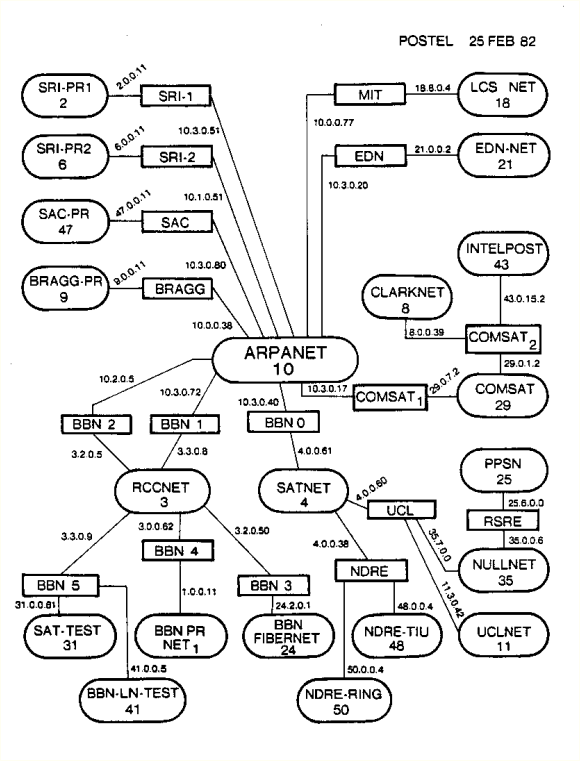


Abbildung 1.1: Netzwerkstruktur des Arpanet als Vorläufer des Internet (Stand 1982) von Jon Postel, Bildnachweis s. S. 341

Die damalige Struktur des ARPANET ermöglichte bereits, dass Daten unterschiedliche Wege vom Sender zum Empfänger nehmen: Ob eine Nachricht von einem Rechner aus `ARPANET 10` zu einem Rechner in `RCCNET 3` über `BBN 2` oder `BBN 1` geleitet wird, ist offen. Diese Redundanz verringert die Fehleranfälligkeit, da ein Ausfall bspw. von `BBN 2` durch `BBN 1` kompensiert werden kann.

Exkurs: IPv6: Verkürzte Schreibweise für Adressen

Es gibt einige Regeln zur Verkürzung von IPv6-Adressen: So dürfen bspw. führende Nullen innerhalb eines Blockes und komplette «0000»-Blöcke (auch mehrere aufeinander) ausgelassen werden. Um Doppeldeutigkeiten zu vermeiden dürfen maximal an einer Stelle einer IPv6-Adresse ein oder mehrere aufeinander folgende «0000»-Blöcke ausgelassen werden.
Aufgrund der hohen Relevanz für den Mischbetrieb von IPv4 und IPv6, können die letzten 32 Bit einer 128 Bit IPv6-Adresse auch in IPv4-Notation wiedergegeben werden. Die folgenden vier IPv6-Adressen sind deshalb alle gleichbedeutend und repräsentieren dieselbe 128-Bit Zahl:

```
2a00:1450:401b:0801::2004
2a00:1450:401b:801::2004
2a00:1450:401b:0801:0000:0000:0000:2004
2a00:1450:401b:0801:0000:0000:0.0.32.4
```

1.2 Subnetze und ihre Adressierung

Alle Knoten innerhalb desselben Subnetzes können sich gegenseitig Nachrichten schicken. Solche Nachrichten verlassen nie ihr Subnetz. In solch einem Subnetz kann es eine direkte Verbindung zwischen allen Knoten (z. B. im Fall von Wifi) geben oder ein vergleichsweise einfacher Subnetz-lokaler Vermittler in Form eines Hubs oder Switches sorgt für die Zustellung. Hubs kopieren eingehende Nachrichten an alle angeschlossenen Knoten, Switches sind selektiver und kopieren Nachrichten nur an den adressierten Empfänger. Soll eine Nachricht an einen Knoten außerhalb des Subnetzes des Senders verschickt werden, muss sie zuerst in das Subnetz des Zielempfängers gerouted werden.
Soll im Beispiel in Abb. 1.2 eine Nachricht von `10.136.117.77` an `192.168.99.67` geschickt werden, muss sie aus «Subnetz 1» über einen Router durch «Subnetz 2» und einen weiteren Router in das «Subnetz 3» bewegt werden. Längere Nachrichten werden in mehrere Datenpakete zerteilt. Jedes Datenpaket kann einen anderen Weg vom Absender zum Empfänger nehmen. Im Beispiel in Abb. 1.2 können Datenpakete sowohl über `91.89.194.5` als auch über `91.89.194.6` weitergeleitet werden.

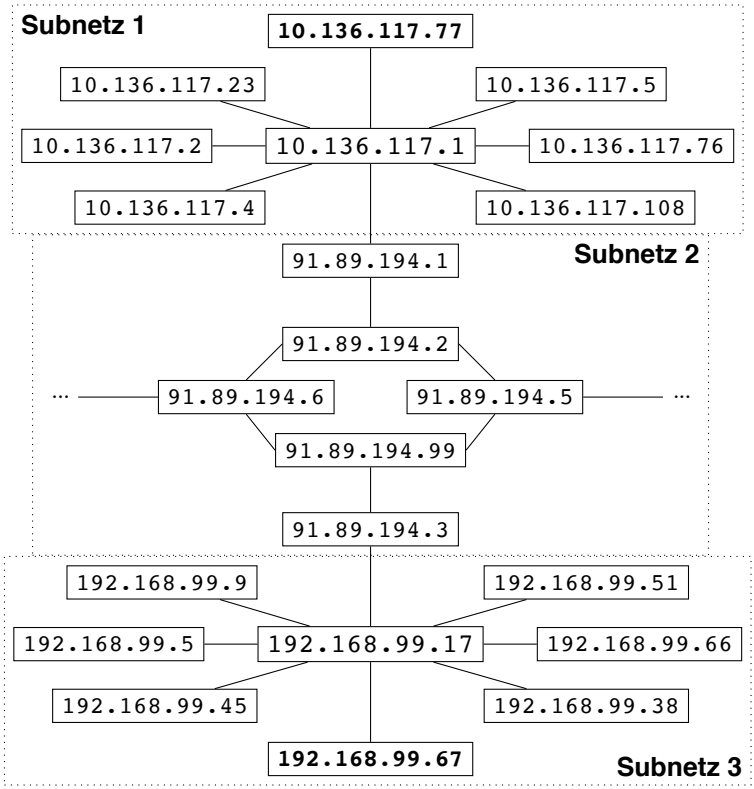


Abbildung 1.2: Drei Subnetze, die untereinander verbunden sind; Nachrichten von `10.136.117.77` an `192.168.99.67` werden aus «Subnetz 1» über `91.89.194.6` oder `91.89.194.6` in das «Subnetz 3» gerouted.

Datenpakete können *auf unterschiedlichen Wegen* vom Sender zum Empfänger geleitet werden. Das Internet ist u. a. deshalb *nicht echtzeitfähig*, man kann also keine maximale Übertragungszeit zwischen zwei Knoten im Internet spezifizieren. Außerdem muss die Reihenfolge, in der Nachrichten versandt wurden, nicht dieselbe sein, in der sie empfangen werden. Die Route zwischen zwei Hosts ergibt sich durch eine Reihe lokaler Entscheidungen an den einzelnen Gabelungen auf der Strecke. Sender und Empfänger können den Weg zwischen ihnen nicht beeinflussen.

IP-Adressen beinhalten als Bestandteil sowohl die Adresse eines Subnetzes, als auch die Adresse des *Hosts* innerhalb des Subnetzes. Die Netzmaske eines Subnetzes gibt

an, wie viele führende Bits für die Adressierung dieses Subnetzes erforderlich sind und wie viele verbleibende Bits für die Adressierung von Knoten innerhalb dieses Subnetzes benutzt werden. Zusammen stehen 32 Bit zur Verfügung.
Die Netzmaske beschreibt ein Bitmuster, das aus x gesetzten Bits und darauf folgend $32 - x$ nicht gesetzten Bits besteht. Eine solche Repräsentation hat den Vorteil, dass aus einer beliebigen IPv4-Adresse sehr effizient die Subnetz-Adresse durch bitweises «UND» mit der Netzmaske berechnet werden kann.
Da es sich bei der Netzmaske um eine 32 Bit Zahl genau wie bei einer IPv4-Adresse handelt, werden Netzmasken in derselben «`.`»-Notation wie IPv4-Adressen repräsentiert. Einige Beispiele für Netzmasken sind in der Tab. 1.1 dargestellt.

Tabelle 1.1: Netzwerkmasken in unterschiedlichen Darstellungen

Netzmaske	Netzlänge	Hostlänge	Bitmuster
`255.0.0.0`	8 Bits	24 Bits	`11111111000000000000000000000000`
`255.255.0.0`	16 Bits	16 Bits	`11111111111111110000000000000000`
`255.255.255.0`	24 Bits	8 Bits	`11111111111111111111111100000000`
`255.255.254.0`	23 Bits	9 Bits	`11111111111111111111111000000000`
`255.255.128.0`	17 Bits	15 Bits	`11111111111111111000000000000000`

Exkurs: IP-Netzklassen: Feste Adresslängen für Subnetze

Bis Anfang der Neunzigerjahre wurde eine feste Anzahl führender Bits einer IPv4-Adresse verwendet, um Subnetze zu adressieren. Da es erforderlich schien, Subnetze mit einer verschieden großen Maximalanzahl von Knoten vorzusehen, wurden drei unterschiedliche Klassen von Subnetz-Adressen definiert, die sich darin unterschieden, wie die Aufteilung der 32 Bits einer IPv4-Adresse auf Subnetz- und Knoten-Adresse erfolgt:

			Länge in Bits ($\sum 32$)				
Klasse	**Präfix**	**Netzmaske**	**Präfix**	**Netz**	**Host**	**adressierb. Netze**	**Knoten / Netz**
A	`0...`	`255.0.0.0`	1	7	24	$2^7 = 128$	16.777.214
B	`10...`	`255.255.0.0`	2	14	16	$2^{14} = 16.384$	65.534
C	`110...`	`255.255.255.0`	3	21	8	$2^{21} = 2.097.152$	254

Die maximal mögliche Anzahl der Knoten in einem Netz mit n Bits Platz für Host-Adressen ist $2^n - 2$: Von den zur Verfügung stehenden Adressen (2^n) müssen die Netzadresse (alle Bits der Host-Adresse `0`) und die Broadcast-Adresse (alle Bits der Host-Adresse `1`) abgezogen werden.

`0.......00000000000000000000000`	Netzadresse in «Class A»-Netzen
`0.......11111111111111111111111`	Broadcast-Adresse in «Class A»-Netzen
`10.............0000000000000000`	Netzadresse in «Class B»-Netzen
`10.............1111111111111111`	Broadcast-Adresse in «Class B»-Netzen
`110.....................00000000`	Netzadresse in «Class C»-Netzen
`110.....................11111111`	Broadcast-Adresse in «Class C»-Netzen

Bei allen IPv4-Adressen, die mit `0` im Bitmuster anfingen, war klar, dass es sich um eine Adresse für eines der 128 «Class A»-Netze handeln musste. Fing das Bitmuster einer Adresse mit `1` an, war das zweite Bit entscheidend: Bei `10` war es eine Adresse in einem der 16.384 «Class B»-Netze, bei `110` war es eines der 2^{21} «Class C»-Netze . Eine extra Netzmaske wurde also nicht benötigt.

Exkurs: IP-Subnetzadressierung: klassenlose Subnetze

Nachdem längere Zeit mit den Klassen A-C bei der Adressierung von IP-Subnetzen gearbeitet wurde, wurden diese Klassen später aufgegeben und die sog. *classless* Adressierung von Subnetzen eingeführt, bei der eine flexible Aufteilung der 32 Bit einer IPv4-Adresse in x führende Bits für die Adressierung des Subnetzes und $32 - x$ Bits für die Adressierung eines Knotens innerhalb des Subnetzes verwendet wird. Es ergeben sich dann ggf. anders lautende Netzmasken wie beispielsweise:

Netzmaske	Netzlänge	Hostlänge	Knoten / Netz
...			
255.255.254.0	23 Bits	9 Bits	510
255.255.252.0	22 Bits	10 Bits	1022
255.255.248.0	21 Bits	11 Bits	2046
...			

1.3 Forwarding und Routing

Das Internet ist ein paketvermittelndes Netzwerk. Nachrichten werden in Datenpakete unterteilt und jedes Datenpaket wird auf einem eigenen Weg durch das Netzwerk gerouted. Jedes Paket enthält einen Header, in dem u. a. der Zielrechner in Form seiner IP-Adresse vermerkt ist. Das Routing durch das Internet erfolgt dadurch, dass an jeder Zwischenstation (Router) auf dem Weg eines Pakets vom Sender zum Zielrechner eine lokale Entscheidung getroffen wird, zu welchem Anschluss ein empfangenes Paket weitergeleitet wird (*Forwarding*). Dabei untersucht der Router die Zieladresse des Pakets und vergleicht sie mit einer lokal gespeicherten Forwarding-Tabelle. Im Prinzip sind in dieser Tabelle Regeln hinterlegt, die für Adressen bestimmter Adressbereiche eine *forwarding*-Richtung festlegen. Aufgrund der Beschaffenheit der Netzadressen im Internet kann dazu einfach der *longest prefix match* der Zieladresse gegen die Subnetz-Adressen bekannter Zielnetze erfolgen. Dieses Verfahren lässt sich sehr effizient implementieren. Außerdem ist es dadurch leicht möglich, untergeordnete Subnetze in einem existierenden Netz einzurichten und den Netzwerkverkehr so zu strukturieren.

Für Subnetz-interne Adressen fungieren Router prinzipiell genauso wie ein Switch. Manche Adressen werden von Switches und Routern speziell behandelt. Das sind Broadcast oder Multicast-Adressen, die nicht nur genau einen Empfänger bezeichnen, sondern mehrere. Im Falle einer Broadcast-Adresse verbleibt das Datenpakt zwar im Subnetz, wird vom Switch jedoch an alle Knoten in diesem Subnetz zugestellt. Jeder

Exkurs: Netzneutralität

Für das Internet wird als Prinzip gefordert, dass Router ihre lokalen *forwarding*-Entscheidungen ausschließlich anhand der Zieladresse eines jeden Datenpakets treffen. Absender, Art oder der Inhalt der transportierten Daten sollen keinen Einfluss auf Priorisierung und Routing haben.
Dieser Grundsatz wird Netzneutralität genannt. Dadurch soll es innovativen Dienstanbietern ermöglicht werden, ohne Einstiegshürden am Internet teilzunehmen.

Exkurs: Adressbereiche für private IP-Adressen

Es gibt einige speziell reservierte Adressbereiche, deren IP-Adressen nicht aus ihrem Subnetz herausgeroutet werden und die dementsprechend auch nicht von außen adressierbar sind. Das sind Adressen, die niemals von der *Internet Assigned Numbers Authority* (IANA) vergeben werden. Deshalb kann solch eine IP-Adresse problemlos von mehreren Hosts gleichzeitig verwendet werden. Eine zentrale Anmeldung bei der IANA ist nicht erforderlich. Lediglich innerhalb eines Subnetzes darf eine private IP-Adresse nicht mehrfach verwendet werden.
Die folgenden Adressbereiche wurden in IETF-Standards (RFC 1918, 5735) als privat definiert:

`10.0.0.0`	bis	`10.255.255.255`	(Netzmaske: `255.0.0.0`)
`172.16.0.0`	bis	`172.31.255.255`	(Netzmaske: `255.240.0.0`)
`192.168.0.0`	bis	`192.168.255.255`	(Netzmaske: `255.255.0.0`)
`169.254.0.0`	bis	`169.254.255.255`	(Netzmaske: `255.255.0.0`)
`100.64.0.0`	bis	`100.127.255.255`	(Netzmaske: `255.192.0.0`)

IP-Adressen aus diesen Netzen können ohne Abstimmung (mehrfach in getrennten Subnetzen) verwendet werden. Private Heim-LANs verwenden im Regelfall *private IP-Adressen*.

Exkurs: Multicast

Multicast-Adressen bezeichnen mehrere Knoten gleichzeitig. Sie können sich auf mehrere Zielrechner im selben Subnetz oder auch in unterschiedlichen Subnetzen beziehen. Router müssen zur korrekten Behandlung von Multicast-Adressen Datenpakete möglicherweise duplizieren und auf getrennten Wegen weiterleiten. In Band 1 von *Verteilte Architekturen* wird Multicast nicht weiter behandelt.

Empfänger entscheidet dann, ob ein empfangenes Broadcast-Paket weiterverarbeitet wird oder nicht.

1.4 Dynamic Host Configuration Protocol (DHCP)

Prinzipiell ist es immer möglich, jeden Netzwerkanschluss eines Hosts manuell zu konfigurieren um ihm seine IP-Adresse und Netzmaske mitzuteilen. Das ist in der Praxis aber kaum managebar, weil mobile Endgeräte ständig das Netz wechseln und oft viele Hosts kurzfristig in ein Netzwerk aufgenommen werden sollen. Wird das manuell und möglicherweise auch noch von mehreren Stellen gemacht, kann es leicht passieren, dass IP-Adressen doppelt vergeben werden, was nicht passieren darf.
In den meisten Subnetzen wird das Dynamic Host Configuration Protocol (DHCP) unterstützt. Dafür muss ein Service eingerichtet werden, der alle Knoten im Subnetzwerk identifiziert und verwaltet. Kommt ein neuer Host hinzu, kann ihm darüber eine noch freie IP-Adresse des Subnetzwerkes und des Gateways (das ist der Router, über den Nachrichten an fremde Subnetze versendet werden können) zugewiesen und die anwendbare Netzmaske sowie optional auch noch andere Informationen wie der lokale DNS Server (s. u.) mitgeteilt werden.
Der Ablauf umfasst vier Schritte:

1. **«`Discover`» (Client → Server):** Anfangs hat der neu eintreffende Host noch keine eigene IP-Adresse und kennt auch nicht die Netzmaske des Subnetzes. Er schickt eine Broadcast-Nachricht an alle Knoten des Subnetzes, indem er die universelle Broadcast-Adresse `255.255.255.255` als Empfänger setzt. Die Nachricht sollte eigentlich die IP-Adresse des Absenders enthalten. Da der Absender noch keine eigene IP-Adresse hat, ist der Absender auf `0.0.0.0` gesetzt.

Client ⟶

DHCP-Pakettyp:	«Discover»
Server-Adr.:	255.255.255.255 (Adressat)
Client-Adr.:	0.0.0.0 (Absender)
zugeteilte Adr.:	0.0.0.0
Paket-ID:	z. B. 4711

⟶ Server

2. **«Offer» (Server → Client):** Der DHCP-Server (im Beispiel 10.136.117.1) im Subnetz empfängt wie alle anderen Knoten die Broadcast-Nachricht vom Typ «Discover», die an die Adresse 255.255.255.255 gegangen war. Aus der Liste der freien IP-Adressen des Subnetzes wird eine ausgewählt (hier 10.136.117.77) und für die Verwendung durch den neuen Host vorgesehen. Es ist dem Server noch nicht möglich, diese IP-Adresse (und ggf. weitere Informationen) direkt an den neuen Host zurückzuschicken, da er noch keine eigene Adresse hat. Deshalb muss die Antwort wieder in Form eines Broadcasts (Adresse 255.255.255.255) an alle angeschlossenen Knoten im Subnetz gehen.

Client ⟵

DHCP-Pakettyp:	«Offer»
Server-Adr.:	z. B. 10.136.117.1 (Absender)
Client-Adr.:	255.255.255.255 (Adressat)
zugeteilte Adr.:	z. B. 10.136.117.77
Paket-ID:	z. B. 4711

⟵ Server

3. **«Request» (Client → Server):** Der neue Client empfängt wie alle anderen Knoten im Subnetz die «Offer»-Nachricht des DHCP-Servers (z. B. 10.136.117.1). Im Prinzip ist es möglich, dass es mehrere DHCP-Server in demselben Subnetz gibt. Deshalb sammelt der Client nun alle «Offer»-Nachrichten der antwortenden DHCP-Server, wählt eine daraus aus (z. B. 10.136.117.77) und unterrichtet alle Server über die Auswahl.

   Der Client darf im Moment die ausgewählte IP-Adresse noch nicht verwenden. Deshalb geht die «Request»-Nachricht wieder mit der Absenderadresse 0.0.0.0 als Broadcast (255.255.255.255) an alle. In dieser Nachricht ist die ausgewählte IP-Adresse (z. B. 10.136.117.77) enthalten. Der Server, von dem der Vorschlag kam, kann den «Request» also als zu ihm gehörend zuordnen, während die anderen DHCP-Server, die möglicherweise eine DHCP-

Exkurs: Link Layer: MAC-Adresse

Zur Adressierung auf der Ebene des Links innerhalb eines Subnetzes wird eine andere Adressierung benutzt. In den dafür benutzten MAC (Medium Access Control) Adressen ist keine Netzwerkinformation enthalten. Jeder Netzwerkadapter hat eine einmalige 48 Bit-Adresse, über die er von Switches oder fremden Wifi-Adaptern identifiziert bzw. wiedererkannt werden kann. In Band 1 von *Verteilte Architekturen* wird nicht mehr weiter auf den Link Layer mit der MAC-Adressierung eingegangen.

«`Offer`» geschickt hatten, die nicht zum Zuge kamen, dies auch erkennen können.

Client ⟶

DHCP-Pakettyp:	«`Request`»
Server-Adr.:	`255.255.255.255` (Adressat)
Client-Adr.:	`0.0.0.0` (Absender)
zugeteilte Adr.:	z. B. `10.136.117.77`
Paket-ID:	z. B. 4712

⟶ Server

4. **«`Acknowledge`» (Server → Client):** Der DHCP-Server (im gezeigten Beispiel `10.136.117.1`) bestätigt nun dem Client, dass die gewählte IP-Adresse (hier `10.136.117.77`) tatsächlich verwendet werden darf. Der Client muss aber noch ein letztes Mal über eine Broadcast-Nachricht darüber informiert werden. Ab diesem Zeitpunkt darf der Client die übermittelte IP-Adresse (z. B. `10.136.117.77`) verwenden.

Client ⟵

DHCP-Pakettyp:	«`Acknowledge`»
Server-Adr.:	z. B. `10.136.117.1` (Absender)
Client-Adr.:	`255.255.255.255` (Adressat)
zugeteilte Adr.:	z. B. `10.136.117.77`
Paket-ID:	z. B. 4712

⟵ Server

1.5 Domain Name System (DNS)

Weil numerische IP-Adressen (IPv4: 32-Bit in «`.`»-Notation, IPv6: 128-Bit in «`:`»-Notation) schlecht zu merken sind und sich aus netzverwaltungstechnischen Gründen

ändern können (z. B. wegen Umadressierens eines Subnetzes mit allen darin enthaltenen Hosts), wurde das *Domain Name System* (DNS) als Vermittlungsmechanismus eingeführt. Das DNS ist eine verteilte Datenbank, die textuelle Namen von Hosts enthält. Zu jedem *Host*-Namen gibt es dann mindestens eine IP-Adresse. Durch das DNS wird dem IP-Adressierungsschema *Ortstransparenz* hinzugefügt. Durch eine zentrale Änderung des DNS-Eintrags kann der aus Netzwerktechniksicht physische Ort eines Hosts problemlos geändert werden: Wenn alle Kommunikationspartner des zu «bewegenden» Knoten ausschließlich die textuellen DNS-Rechnernamen zur Adressierung verwenden, brauchen sie nicht über eine IP-Adressänderung informiert werden, da sie die jeweils aktuelle Adresse aus dem DNS beziehen.

Der textuelle Name eines Zielrechners muss immer erst in dessen numerische IP-Adresse überführt werden, bevor Nachrichten über das Internet an ihn geschickt werden können, denn auf der Netzwerktechnik-Ebene von Routern und Subnetzen haben nur numerische IP-Adressen eine Funktion. Der Prozess, in dem ein textueller *Host*-Name unter Verwendung des DNS in eine numerische IP-Adresse übersetzt wird, muss also immer vor der eigentlichen Netzwerkkommunikation erfolgen. Ist die numerische IP-Adresse eines textuellen *Host*-Namens von einem potenziellen Nachrichtensender einmal aufgelöst worden, wird diese Zuordnung für die zukünftige Verwendung in einem lokalen *Cache* zwischengespeichert. Da das DNS dynamisch ist und sich die Zuordnung in Zukunft ändern könnte, haben die lokalen Zuordnungskopien nur eine begrenzte Lebensdauer. Ist sie abgelaufen, wird der nächste Zugriff auf den textuellen *Host*-Namen wieder erst durch eine DNS-Abfrage zu einer numerischen IP-Adresse aufgelöst.

Damit das DNS nicht als «*single point of failure*» und nicht skalierender Flaschenhals im Internet wirkt, ist dessen Datenbestand redundant verteilt. Ein Nachteil davon ist, dass Adressänderungen nicht augenblicklich überall im Internet wirksam werden und es Übergangszeiten geben kann, in denen es zu Inkonsistenzen kommen kann. Ein Vorteil der verteilten Datenhaltung ist hingegen, dass örtlich getrennte DNS-Teildatenbanken absichtlich unterschiedliche Hostadressen zu demselben textuellen Rechnernamen speichern können, damit die Last ortsbezogen auf redundante Rechenzentren verteilt werden kann.

Exkurs: «DNS-Sperre»: Internet-Sperren durch DNS-Manipulation

Bisweilen wird versucht, unerwünschte Internetdienste durch Manipulation ihrer DNS-Einträge unbrauchbar zu machen.
Beispielweise wurde 2010 im inzwischen wieder aufgehobenen Gesetz zur Erschwerung des Zugangs zu kinderpornographischen Inhalten in Kommunikationsnetzen (Zugangserschwerungsgesetz – ZugErschwG) eine solche «DNS-Sperre» vorgesehen, aber letztlich nicht umgesetzt.
Da die Funktionsfähigkeit eines solchen Internetdienstes technisch nicht von der korrekten Auflösung seines textuellen *Host*-Namens abhängt, kann über eine DNS-Sperre der Zugriff auf inkriminierte Inhalte nur erschwert, aber nicht verhindert werden, denn die Übertragung von Daten im Internet erfolgt ausschließlich anhand von numerischen IP-Adressen.
Die Twitter-Sperrungen in der Türkei in den Jahren 2014 und 2015 wurden zuerst auch über eine DNS-Sperre realisiert.
Der DNS-Eintrag für den *autoritative* DNS-Server der Domain `twitter.com` wurde regional manipuliert und auf eine IP-Adresse im Netzwerk von TTNET, ener Tochtergesellschaft von Türk Telekom, umgeleitet.

Abbildung 1.3: Foto von Negatif Pollyanna @FindikKahve von einem Graffito an einer Wand im Stadtteil Kadıköy von Istanbul im Jahr 2014 zum Umgehen der damaligen Twitter-Sperre in der Türkei mit dem Google-DNS `8.8.8.8`, Bildnachweis s. S. 341

Als Reaktion auf die DNS-Manipulation in der Türkei wurde dazu aufgerufen, den DNS-Server `8.8.8.8` (alternativ `8.8.4.4`) von Google als lokalen DNS-Server zu verwenden (s. z. B. Abb. 1.3) und nicht die per DHCP vom jeweiligen türkischen Internet Service Provider zugewiesenen.

1.5.1 DNS-Architektur

Eigentlich gibt es nicht «den» DNS-Dienst, sondern DNS ist prinzipiell nur ein Protokoll, das zum Einen vom potenziellen Sender einer Nachricht zur Abfrage der numerischen IP-Adresse zu einem textuellen *Host*-Namen beim DNS benutzt wird, zum Anderen beinhaltet der DNS-Standard Protokolle und Datenstrukturen, die zum Aufbau und zum Abgleich der verteilten Datenhaltung benutzt werden. Für die eigentlichen Daten des DNS gibt es aber «den» einen Anbieter, der als «offizielles» Internet-Namensverzeichnis angesehen wird (konkurrierende Anbieter sind denkbar, s. Praktikum, Aktivitätsschritt 1.5).

Root DNS Server

Dieses «offizielle» DNS wird auf der obersten Ebene («*root*») von der *Internet Corporation for Assigned Names and Numbers* (ICANN) verwaltet.

Die Root DNS Server «kennen» die DNS Server auf der nächsten Ebene. Das bedeutet, dass in der verteilten Datenbank der Root DNS Server ausschließlich textuelle Namen der nächsten Ebene zu IP-Adressen aufgelöst werden, die auf DNS Server der nächsten Ebene verweisen (s. Abb. 1.4). Damit die Wurzel nicht einen «*single point of failure*» und Flaschenhals darstellt, ist der Root DNS redundant ausgelegt: Es gibt hunderte Root DNS Server auf der Welt verteilt, die über gegenwärtig 13 IP-Adressen angefragt werden können. Jeder sollte im Prinzip denselben Inhalt aufweisen. Für jede der 13 IP-Adressen und die DNS Server dahinter ist ein anderer Betreiber verantwortlich. Das sind Telekommunikationsunternehmen (z. B. VeriSign), Hochschulen (z. B. die University of Maryland), Non-Profit Organisationen wie das Internet Software Consortium (ISC) oder das *Réseaux IP Européens Network Coordination Centre* (RIPE NCC), aber auch US-amerikanische staatliche Stellen wie NASA oder das Verteidigungsministerium. Für die Region Europa, Naher Osten und Zentralasien betreibt das RIPE NCC einen dieser 13 DNS Root Knoten.
Das Management unterhalb der obersten Ebene des DNS ist hierarchisch auf weitere Betreiber untergliedert. In Abb. 1.4 sind exemplarisch die beiden Betreiber DENIC und Nominet aufgeführt. DENIC ist mit dem eigenen DNS Server `s.de.net` für die Auskunft über alle Anfragen unterhalb von `.de` zuständig. Nominet für die meisten unter `.uk` (außer `.gov.uk`). Allerdings ist bei den DNS-Knoten von DENIC und Nominet auch nicht jeweils der gesamte Inhalt unterhalb von `.de` und `.uk` gespei-

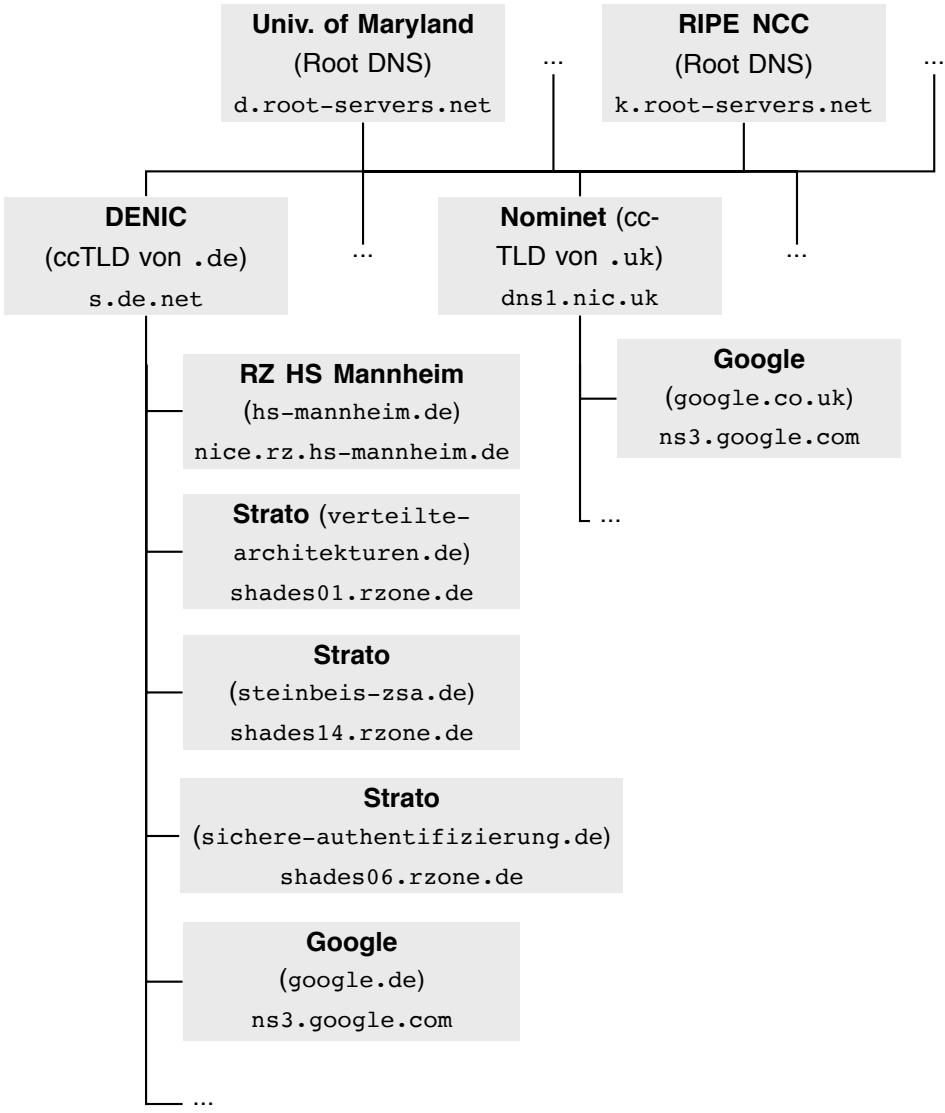


Abbildung 1.4: Hierarchische DNS-Architektur

Exkurs: ccTLD «.dd»

Die ccTLD der Deutschen Demokratischen Republik «`.dd`» wurde nicht von den internationalen Root DNS Servern zur IP-Adresse des TLD DNS Server in der DDR aufgelöst, sondern nur innerhalb der DDR benutzt.

chert, sondern diese Adressen sind in weitere Zonen untergliedert. Für jede Zone gibt es einen DNS-Betreiber, der alle Anfragen über DNS-Einträge zu dieser Zone beantworten kann. In Abb. 1.4 sind die DNS-Betreiber für die Zonen `hs-mannheim.de`, `verteiltearchitekturen.de`, `google.de` und `google.co.uk` aufgelistet. Man sieht an den Einträgen für `google.de` und `google.co.uk`, dass derselbe Server für unterschiedliche Zonen zuständig sein kann.

Top-Level-Domain DNS Server

Die ICANN verwaltet für diese Root DNS Server Betreiber die Struktur auf der nächsten Ebene. Das sind die weithin bekannten *Top Level Domains* (TLD).

- **«klassische» TLDs:** «`.com`», «`.net`», «`.info`», «`.edu`», «`.org`», «`.gov`» und «`.mil`»
- mehr als 200 **länderspezifische Top-Level-Domains** (ccTLD $\widehat{=}$ *country code* TLD): z. B. «`.de`» (Bundesrepublik Deutschland), «`.tv`» (Tuvalu), «`.to`» (Tonga), «`.uk`» (Vereinigtes Königreich) oder «`.us`» (Vereinigte Staaten von Amerika)
- Inzwischen gibt es auch eine ganze Reihe **überstaatlicher TLDs**: z. B. «`.eu`» (Europäische Union), «`.int`» (u. a. «`.nato.int`»)
- **weitere TLDs** wie «`.name`», «`.museum`», «`.tel`» oder «`.travel`»

Jede TLD wird autonom verwaltet. Oft stehen kommerzielle Unternehmen hinter dem Betrieb von TLD DNS Servern. «`.com`» und «`.net`» wird beispielsweise von VeriSign betrieben. Die Organisation hinter «`.de`» ist hingegen die eingetragene Genossenschaft «*Deutsches Network Information Center*» (DENIC). In den skandinavischen Ländern sind wiederum Vereine oder andere non-profit Organisationen zuständig wie z. B. *The Internet Foundation In Sweden* für die schwedische «`.se`» TLD.

Bei manchen TLD-Betreibern gibt es besondere Vorgaben, wie die weitere hierarchische Organisation aussehen soll. In Großbritannien («.uk») gab es bis vor einigen Jahren noch folgende Strukturvorgabe: «.ac.uk», «.co.uk», «.gov.uk», «.ltd.uk», «.me.uk», «.net.uk», «.nhs.uk», «.nic.uk», «.org.uk», «.plc.uk», «.sch.uk», während es in Deutschland keine besondere zwischengelagerte Ebene unter «.de» gibt.

Autoritativer DNS Server

Der DNS Server auf TLD-Ebene kennt alle Domänen unterhalb der jeweiligen TLD. Im Fall von «.co.uk» ist das z. B. google.co.uk, im Fall von «.de» google.de oder hs-mannheim.de. Jede Domäne wird wiederum eigenständig verwaltet. Unterhalb der Domäne kann der jeweilige Betreiber eine eigene weitere Strukturierung vornehmen z. B. services.informatik.hs-mannheim.de oder www.cit.hs-mannheim.de. Wieviele Ebenen und welche Namen ein Domänenbetreiber nutzt, ist ihm selbst überlassen. Der gesamte Namensbereich, den ein Domäneninhaber verwendet, heißt «Zone». Es gibt mindestens einen DNS Server, der alle Anfragen zu der Zone beantworten kann. Seine Adresse muss dem DNS-Server der übergeordneten TLD bekannt sein.

Lokale und nicht autoritative DNS Server

Die bisher beschriebenen DNS-Ebenen verwalten alle Daten, die das DNS ausmachen – teils redundant – in einer hierarchisch gegliederten verteilten Organisationsstruktur. Möchte ein Host Daten aus dem DNS abrufen, erfolgt der Zugriff jedoch zumeist über einen lokalen DNS Server. Solche lokalen DNS Server werden bspw. von Internet Service Providern für ihre Kunden betrieben und können auch in Routern enthalten sein. In den meisten Subnetzen gibt es deshalb einen lokalen Standard DNS Server, der neu angeschlossenen Hosts per DHCP mitgeteilt wird. Es gibt aber auch öffentlich zugreifbare DNS Server, die von Firmen sowohl kostenfrei (z. B. «8.8.8.8» von Google) als auch als kommerziell nutzbarer Dienst (z. B. OpenDNS von CISCO) bereitgestellt werden. Solche DNS Server sind zwar nicht selber für Teile des Namensraums verantwortlich, können aber trotzdem alle DNS-Anfragen beantworten, indem die fehlenden Daten von autoritativen DNS Servern abgefragt (und zwischengespeichert) werden. Sie werden deshalb nicht autoritative DNS Server genannt.

1.5.2 Daten im DNS

Das DNS ist eine verteilte Datenbank. Informationen werden darin Domains zugeordnet. Domain-Namen sind hierarchisch gegliedert. Die Ebenen werden durch einen `"."` voneinander getrennt. Die Hierarchiebenenen werden von rechts nach links gelesen. Die oberste Hierarchieebene, i. Allg. eine TLD, steht also ganz rechts. Die maximale Länge eines Domain-Namens sind 255 Zeichen.
Zu jedem Domain-Namen können mehrere Datensätze (sog. *resource records*) in Form von Schlüssel/Wert Paaren gespeichert sein. Als Schlüssel sind dabei u. a. die folgenden Typen vorgesehen:

`A:` die IPv4-Adresse eines Hosts

`AAAA:` die IPv6-Adresse eines Hosts

`NS:` ein autoritativer Name Server

`CNAME:` der kanonische Name eines Alias

`MX:` Mail Server

`TXT:` freie Text Strings

1.5.3 Auflösung eines textuellen *Host*-Namens zu einer numerischen IP-Adresse

Der lokale DNS Server ist für Clients der Einstiegspunkt in das DNS. In den meisten Programmierumgebungen (so auch in Java) kann man an den Stellen, an denen eine numerische IP-Adresse benötigt wird, stattdessen auch einen textuellen *Host*-Namen angeben. Die entsprechende Betriebssystemfunktion löst dann in einem vorgelagerten Schritt erst diesen textuellen Namen durch eine Anfrage beim DNS zu einer numerischen IP-Adresse auf. Bereits bekannte *Host*-Name/IP-Adressenzuordnungen werden zwischengespeichert, damit zukünftige Anfragen schneller beantwortet werden können.
Zur Auflösung eines textuellen *Host*-Namens fragt ein Client beim DNS den *resource record* mit dem Schlüssel `A` (IPv4-Adresse) oder `AAAA` (IPv6-Adresse) für den *Host*-Namen ab. In der Antwort ist die IP-Adresse enthalten.

1.5.4 Abläufe

Die Anfrage eines Clients bei einem DNS Server kann entweder *iterativ* oder *rekursiv* beantwortet werden.
Ist die IP-Adresse des gesuchten *Host*-Namens weder lokal im Cache des Betriebssystems, noch beim kontaktierten lokalen DNS Server («Netgear R7000» in Abb. 1.5 und 1.6) bekannt, passiert folgendes (*iterativ*, s. Abb. 1.5):

1. Der Client möchte die IP-Adresse zum *Host*-Namen `www.informatik.hs-mannheim.de`. Er fragt im ersten Schritt bei seinem lokalen DNS Server nach. Der ist in diesem Beispiel auf dem Router, einem Netgear Nighthawk R7000, installiert. Dieses lokale DNS hat die IP-Adresse `10.136.117.1`.

2. Im Beispiel kennt dieser lokale DNS Server die IP-Adresse von `www.informatik.hs-mannheim.de` nicht. Er kann deshalb nicht mit dessen Adresse antworten, sondern kann nur die IP-Adresse zu seinem lokalen DNS Server, der beim ISP Liberty Global läuft, zurückliefern. Dies ist ein nicht autoritativer DNS. Er hat die IP-Adresse `80.69.96.12`.

3. Aufgrund der Antwort des lokalen DNS Servers kann der Client nun das nächste DNS kontaktieren. Das ist das nicht autoritative DNS bei Liberty Gobal DNS mit der IP-Adresse `80.69.96.12`.

4. Dieser DNS Server kennt auch nicht die Adresse von `www.informatik.hs-mannheim.de`. Daher liefert er im Beispiel in Abb. 1.6 die IP-Adresse eines der 13 Root DNS Server. Hier wird die IP-Adresse `193.0.14.129` des RIPE NCC Root DNS Servers zurückgegeben.

5. Der Client fragt den Root Server `193.0.14.129` nach der IP-Adresse von `www.informatik.hs-mannheim.de`.

6. Der Root DNS Server antwortet mit der IP-Adresse `195.243.137.26` des DNS Servers der ccTLD «`.de`», der von DENIC betrieben wird.

7. Der Client fragt bei `195.243.137.26` nach `www.informatik.hs-mannheim.de`.

8. `195.243.137.26` antwortet mit der IP-Adresse `141.19.1.75` des autoritativen DNS Servers der Zone `hs-mannheim.de`.

9. Der Client kann nun bei `141.19.1.75` nach der IP-Adresse von `www.informatik.hs-mannheim.de` fragen.
10. Dieser DNS Server kennt die IP-Adresse von `www.informatik.hs-mannheim.de` und gibt `141.19.1.171` an den Client zurück.
11. Nun kann der Client mit `www.informatik.hs-mannheim.de` unter seiner IP-Adresse `141.19.1.171` kommunizieren.

Bei der rekursiven Beantwortung übernimmt ein angefragter DNS Server seinerseits das Kontaktieren nachgeordneter DNS Server und liefert die letztliche Antwort an den Anfrager zurück (s. Abb. 1.6).

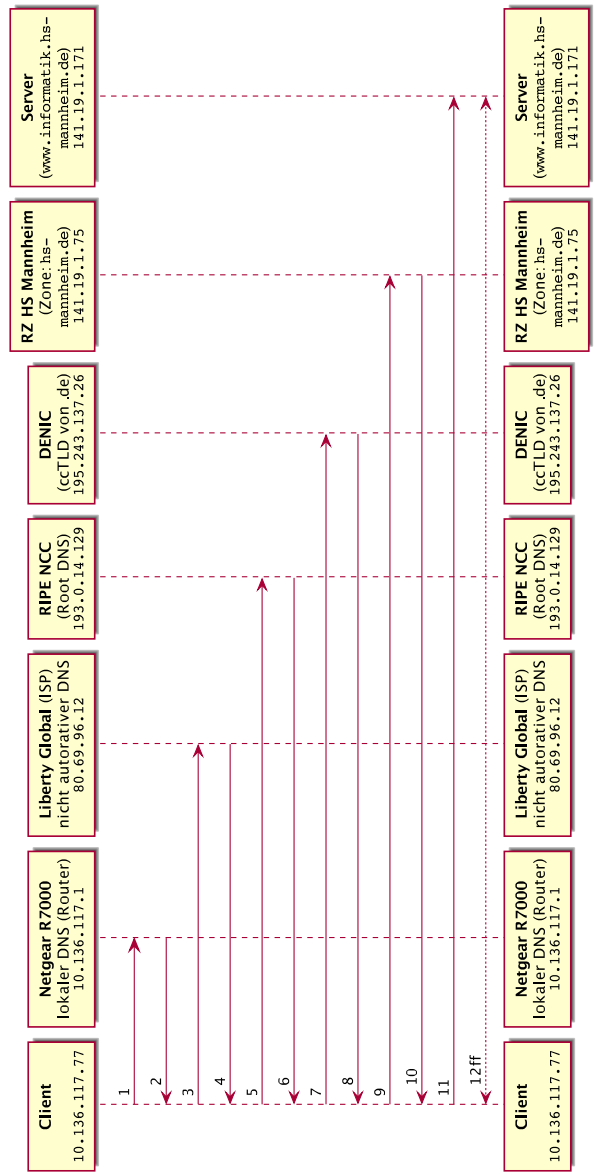


Abbildung 1.5: DNS-Anfrage nach `www.informatik.hs-mannheim.de` – iterative Auflösung des Namens (Schritte 1-10)

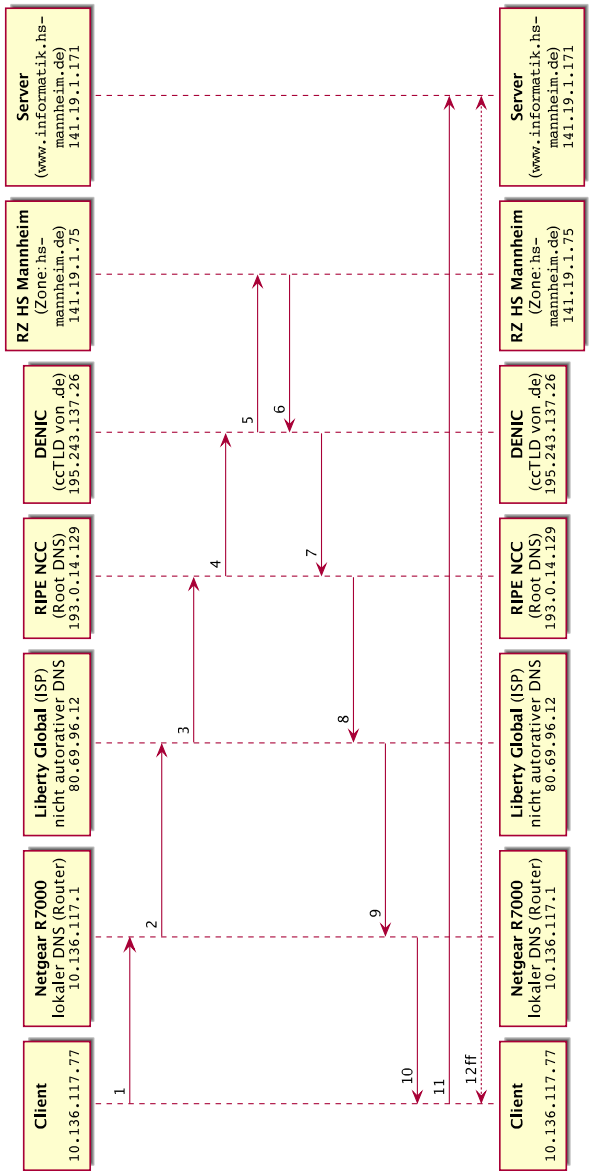


Abbildung 1.6: DNS-Anfrage nach `www.informatik.hs-mannheim.de` – rekursive Auflösung des Namens (Schritte 1-10); allerdings unterstützen Root DNS rekursive Anfragen aus Performance-Gründen nicht

Exkurs: Dynamisches DNS (DDNS)

Private Heim-LANs werden normalerweise über *Internet Service Provider* (ISP) mit dem Internet verbunden. Die Anbindung erfolgt dabei in der Regel über ein Modem, das die Verbindung zum ISP über einen Telefonanschluss (DSL oder ISDN), das Kabelnetz oder inzwischen auch per Mobilfunk (LTE) herstellt. Das Modem ist entweder Teil des Routers, der die Weiterleitung von Paketen aus dem LAN heraus und in das LAN hinein übernimmt, oder extern. Normalerweise verwendet man für die Hosts innerhalb des privaten Heim-LANs *private IP-Adressen* (s. Exkurs auf S. 11). Der Router, der über das Modem die Verbindung zum Internet herstellt, bekommt vom ISP hingegen eine normale IP-Adresse zugewiesen. Im Regelfall wird diese IP-Adresse bei jeder Verbindung (Einwahl bei Telefonanschluss und Mobilfunk, bzw. bei Neuverbindung nach automatisch erfolgter Trennung bei Kabelmodems) geändert und erneut zugewiesen.

Statt einer statischen IP-Adresse, die also bei jeder Einwahl und bzw. bei jeder Neuverbindung erhalten bleibt, kann man die jeweils neue IP-Adresse im DNS mit einem statischen Domain- bzw. *Host*-Namen verbinden. Die gespeicherte IP-Adresse wird sich bei einem solchen Eintrag im DNS also regelmäßig ändern. Man spricht in diesem Fall von *dynamischem DNS*. Über einen solchen DNS-Namen können dann Ressourcen innerhalb des Heimnetzes stabil adressiert werden, obwohl sich die zugewiesene IP-Adresse ständig ändert.

Für die Änderung des DNS-Eintrags existieren unterschiedliche Kommunikationsprotokolle, die von Routern mit Modem oft bereits softwaretechnisch unterstützt werden: Erhält der Router bei der Einwahl eine neue IP-Adresse, wird im DNS über das entsprechende Protokoll ein vorher konfigurierter dynamischer DNS-Eintrag mit der neuen IP-Adresse aktualisiert. Allerdings sind solche DNS-Änderungen nicht beliebig möglich, sondern man muss vorher einen entsprechenden Dienstleister mit dem Anlegen eines DNS-Eintrags beauftragen. Solch ein Dienstleister muss weiterhin eines der Protokolle für dynamisches DNS anbieten.

Der Router «Nighthawk R7000» von Netgear unterstützt beispielsweise die dynamischen DNS-Dienste von MyNETGEAR[a], no-ip[b] und Oracle Dyn[c] (ehem. DynDNS[d]).

[a] `https://my.netgear.com/`
[b] `https://www.noip.com/`
[c] `https://dyn.com/dns/`
[d] `http://dyndns.org/`

Praktikum

Aktivitätsschritte:

Im folgenden Praktikum werden Sie IP-Adressen analysieren und Ihre eigene Anbindung an das Internet untersuchen. Sie werden mit dem *Domain Name System* (DNS) unter Zuhilfenahme unterschiedlicher Kommandos interagieren. Anhand der Antworten des DNS können Sie die Struktur und Arbeitsweise des DNS und die Adressierung im Internet genauer kennenlernen.

1.6 Interaktion mit dem Domain Name System (DNS)

Es gibt Hilfsprogramme, mit denen man direkt mit dem *Domain Name System* (DNS) interagieren kann. Das Hilfsprogramm `nslookup` ist bspw. auf den meisten Unix-Plattformen (BSD, Linux, macOS) und unter Windows verfügbar. Moderner ist dagegen `dig`, das standardmäßig nicht auf Windows-Rechnern installiert ist. Auf neueren Unix-Systemen (BSD, Linux, macOS) ist die Wahrscheinlichkeit groß, dass `dig` installiert ist. Im Rahmen dieses Praktikums ist es egal, welches von beiden Programmen Sie verwenden. Beide Varianten sind mit eigenen Aktivitätsschritten beschrieben.

Aktivitätsschritt 1.1:
Ziel: `nslookup` oder `dig` starten
Um die Hilfsprogramme `nslookup` und `dig` auszuführen, geben Sie einfach den Befehl `nslookup` bzw. `dig` in einem *Shell*-Fenster ein (je nach Oberfläche Ihres Betriebssystems könnte es sein, dass der Zugang zur *Shell* auch «Terminal» oder «Konsole» heißt).
Um solche Hilfsprogramme unter Windows auszuführen, öffnen Sie die «Eingabeaufforderung» (z. B. *Win* + `r`, geben Sie dann `cmd` ein). Geben Sie in dem Fenster der «Eingabeaufforderung» `nslookup` ein.

In diesem Praktikum werden nur die grundlegenden Funktionen von `nslookup` und `dig` benutzt. Beide Tools ermöglichen es bei einem bestimmten DNS Server nach Einträgen in der DNS-Datenbank zu recherchieren. Der abgefragte DNS Server kann ein Root DNS Server, ein Top Level Domain DNS Server, ein autoritativer DNS Server oder ein lokaler DNS Server sein. Dazu senden sowohl `nslookup` als auch `dig` eine DNS-Abfrage an den angegebenen DNS-Server, empfangen eine DNS-Antwort vom Server und zeigen das Ergebnis an.

Aktivitätsschritt 1.2 (`nslookup`):
Ziel: Lokalen DNS Server feststellen, *Host*-Namen auflösen
Geben Sie in Shell oder Eingabeaufforderung folgendes ein und bestätigen Sie durch die *Return*-Taste:

```
nslookup www.verteiltearchitekturen.de
```

Das Ergebnis sollte von der Struktur her aussehen wie in Abb. 1.7. An der Antwort von `nslookup` kann man ableiten, dass der lokale DNS-Server, der entweder manuell konfiguriert oder per DHCP zugewiesen wurde, im abgebildeten Beispiel `10.136.117.1` ist. Abb. 1.8 zeigt einen Screenshot der entsprechenden Oberfläche der Systemeinstellungen unter macOS, der belegt, dass als lokaler DNS Server tatsächlich `10.136.117.1` konfiguriert ist.
Aus der Antwort von `nslookup` kann weiterhin abgelesen werden, dass der DNS-Service auf diesem Rechner auf Port `53`, dem Standard Port für DNS (s. Kapitel 2), angesprochen wird.
Die Antwort des für `www.verteiltearchitekturen.de` zuständigen DNS-Servers ist `216.239.32.21`. Aus der Information `canonical name = verteiltearchitekturen.de` kann man noch ablesen, dass die textuellen *Host*-Namen `verteiltearchitekturen.de` und `www.verteiltearchitekturen.de` auf dieselbe numerische IP-Adresse abgebildet werden.

Aktivitätsschritt 1.3 (`dig`):
Ziel: Lokalen DNS Server feststellen, *Host*-Namen auflösen
Geben Sie in Shell oder Eingabeaufforderung folgendes ein und bestätigen Sie durch die *Return*-Taste:

```
dig -t A www.verteiltearchitekturen.de
```

Die Option `-t A` bewirkt, dass nur IPv4-Adressen zurückgegeben werden.

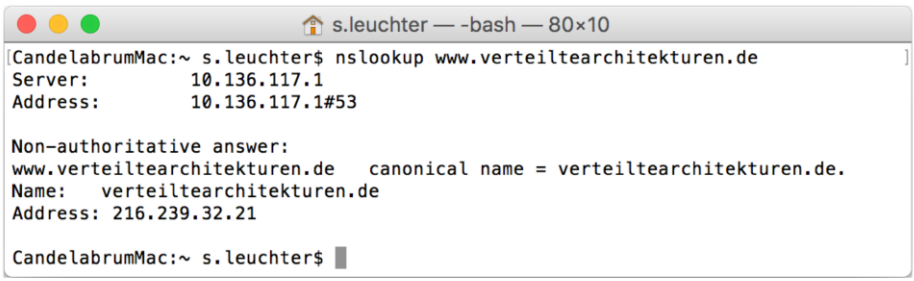


Abbildung 1.7: DNS-Abfrage zur Auflösung von `www.verteiltearchitekturen.de` beim lokalen DNS `10.136.117.1` mit `nslookup`

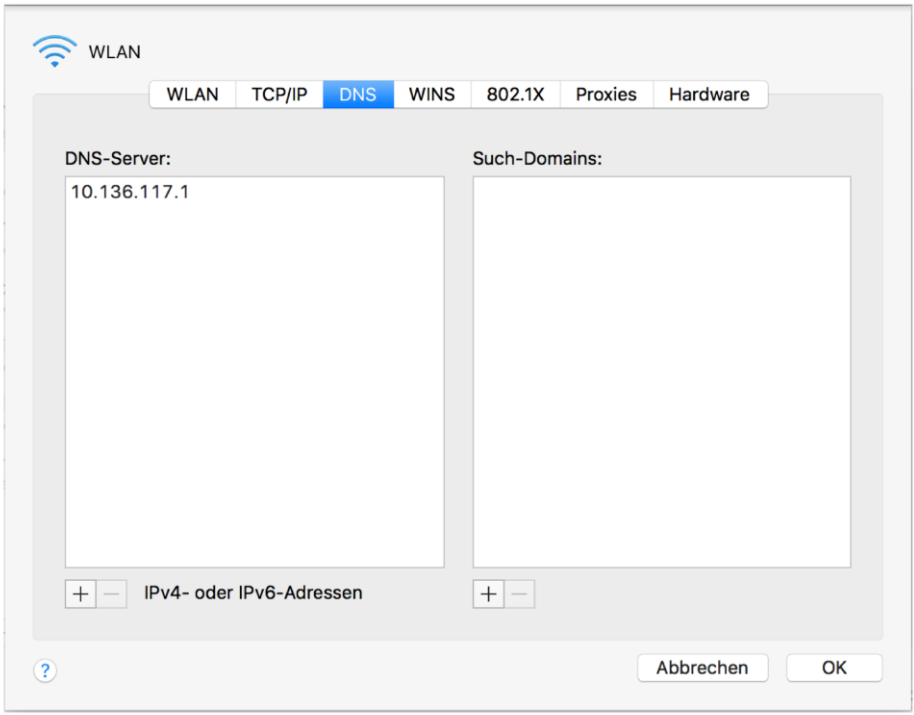


Abbildung 1.8: Screenshot der macOS Systemeinstellungen für den lokalen DNS Server (hier: `10.136.117.1`)

Das Ergebnis sollte von der Struktur her aussehen wie in Abb. 1.9. An der Antwort von `dig` kann man ableiten, dass der lokale DNS Server, der entweder manuell konfiguriert oder per DHCP zugewiesen wurde, im abgebildeten Beispiel `10.136.117.1` ist. Abb. 1.8 zeigt einen Screenshot der entsprechenden Oberfläche der Systemeinstellungen unter macOS, der belegt, dass als lokaler DNS Server tatsächlich `10.136.117.1` konfiguriert ist.
Aus der Antwort von `nslookup` kann weiterhin abgelesen werden, dass der DNS Service auf diesem Rechner auf Port `53`, dem Standard Port für DNS (s. Abschnitt 2), angesprochen wird.
Die Antwort des für `www.verteiltearchitekturen.de` zuständigen DNS Servers ist `216.239.32.21` gewesen. Aus der Information `CNAME verteiltearchitekturen.de` kann man noch ablesen, dass die textuellen *Host*-Namen `verteiltearchitekturen.de` und

`www.verteiltearchitekturen.de` auf dieselbe numerische IP-Adresse abgebildet werden.

Aktivitätsschritt 1.4 (`nslookup` oder `dig`):

Ziel: Analyse unterschiedlicher Domain-Namen

Analysieren Sie mit `nslookup` oder `dig`, ob jeweils die folgenden Paare von textuellen Domain-Namen auf dieselbe IP-Adresse verweisen und ziehen Sie Schlüsse auf die Intention der jeweiligen Eigentümer:

Nr.	*Host*-Name 1	*Host*-Name 2
1.)	www.google.de	www.google.com
2.)	www.dama.io	mail.dama.io
3.)	www.informatik.hs-mannheim.de	ftp.informatik.hs-mannheim.de
4.)	www.hsrw.eu	www.hochschule-rhein-waal.de

Probieren Sie auch aus, diese Adressen in einem Web-Browser aufzurufen.

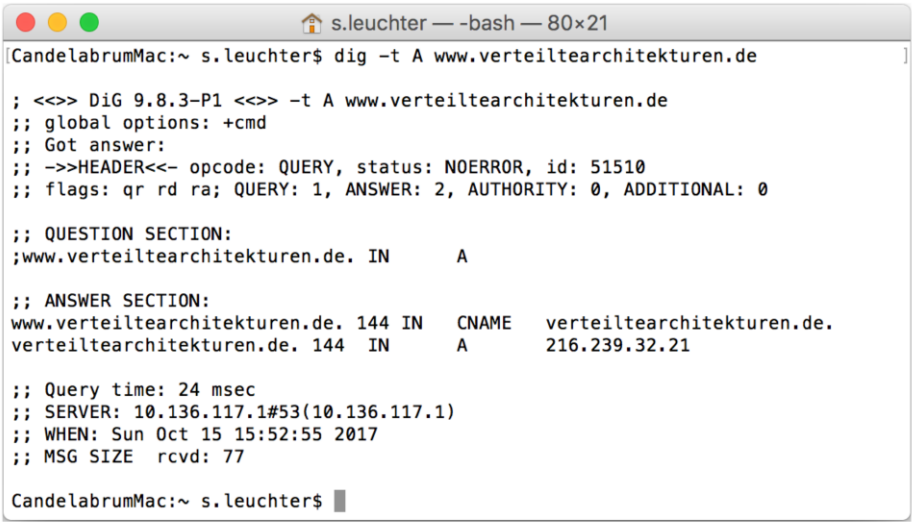


Abbildung 1.9: DNS-Abfrage zur Auflösung von `www.verteiltearchitekturen.de` beim lokalen DNS `10.136.117.1` mit `dig`

1.7 Alternative DNS Server

Bei den bisherigen Beispielen wurde immer eine vorgegebene Standardadresse zur Kontaktaufnahme zum lokalen DNS Server benutzt. Manchmal möchte man aber Anfragen an einen anderen DNS Server senden.
Abb. 1.10 zeigt den vergeblichen Versuch den textuellen *Host*-Namen `reg.oss` mit `nslookup` und `dig` aufzulösen. Das Problem ist, dass die TLD `oss` im DNS nicht bekannt ist. Das DNS wird genau wie die Zuteilung von IP-Adressen von der *Internet Assigned Numbers Authority* (IANA) verwaltet. Die IANA ist nur die Funktionsrolle, nicht die konkrete Organisation, die die Verwaltung ausführt. Ursprünglich von der staatlichen US-amerikanischen DARPA bereitgestellt, wurde die IANA-Funktion 1998 auf die *Internet Corporation for Assigned Names and Numbers* (ICANN) übertragen. Es gibt aber auch alternative «Neben-DNS». Eines davon ist das OpenNIC, das neben den TLD von ICANN auch noch weitere kennt, z. B. `oss` für Open Source Software Projekte.

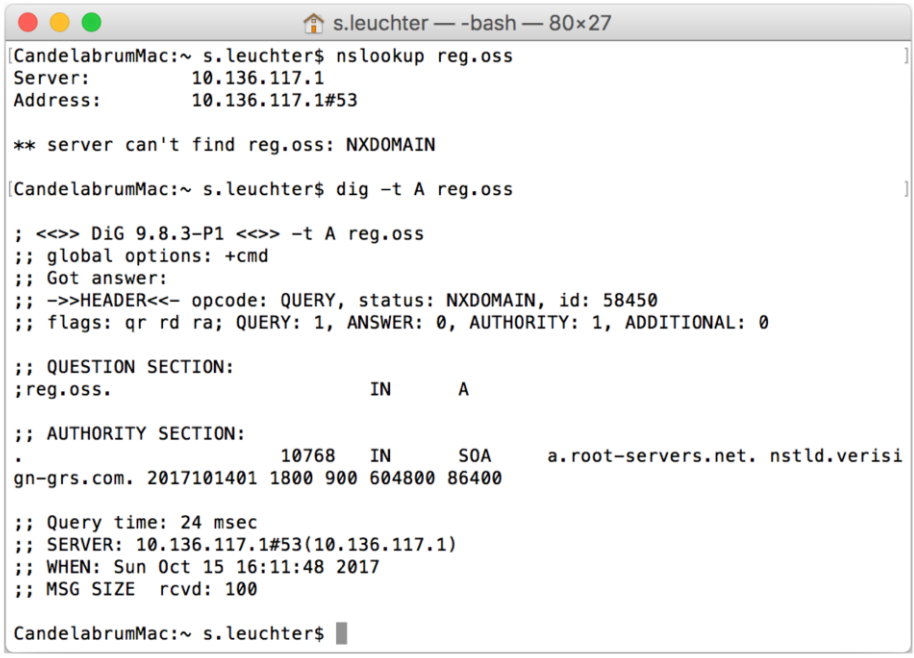


Abbildung 1.10: DNS-Abfrage zur Auflösung von `reg.oss` mit dem alternativen OpenNIC Root DNS

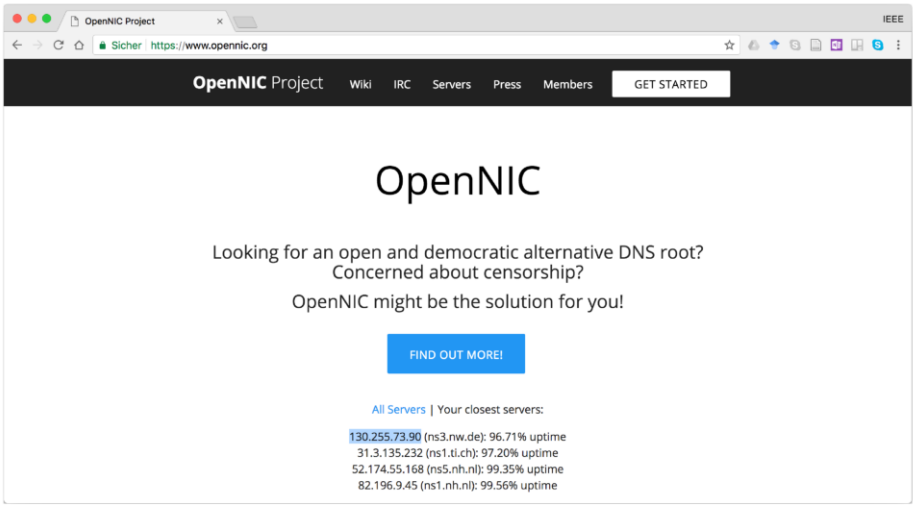


Abbildung 1.11: Website des OpenNIC Project (`http://www.opennic.org/`)

Aktivitätsschritt 1.5:

Ziel: IP-Adresse eines DNS Servers des OpenNIC ermitteln

Rufen Sie die Website `https://www.opennic.org/` des *OpenNIC Project* auf. Finden Sie die numerische IP-Adresse eines DNS Servers des OpenNIC heraus, die eine möglichst hohe Verfügbarkeit hat. Im Beispiel in Abb. 1.11 ist das die IP-Adresse `130.255.73.90`.

Aktivitätsschritt 1.6 (`nslookup`):

Ziel: *Host*-Namen in TLD `oss` auflösen mit OpenNIC DNS

Lösen Sie den textuellen *Host*-Namen `reg.oss` bei einem DNS Server von OpenNIC auf, indem Sie die IP-Adresse des OpenNIC DNS Servers am Ende anhängen, die Sie im Aktivitätsschritt 1.5 herausgefunden haben:

```
nslookup reg.oss 130.255.73.90
```

Das Resultat sollte analog zu Abb. 1.12 sein. Welche IP-Adresse hat `reg.oss`?

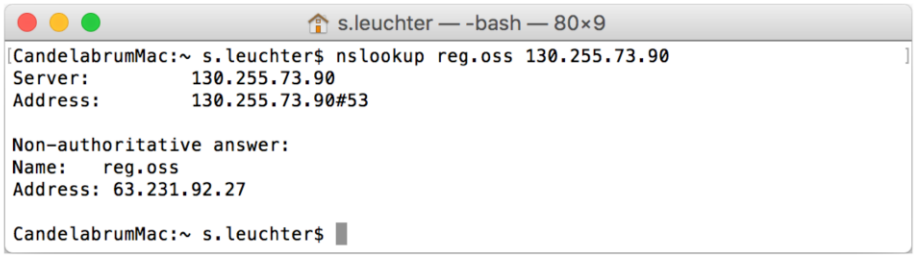


Abbildung 1.12: Abfrage von `reg.oss` bei OpenNIC mit `nslookup`

Aktivitätsschritt 1.7 (`dig`):
Ziel: *Host*-Namen in TLD `oss` auflösen mit OpenNIC DNS
Lösen Sie den textuellen *Host*-Namen `reg.oss` bei einem DNS Server von OpenNIC auf, indem Sie die IP-Adresse des OpenNIC DNS Servers, die Sie im Aktivitätsschritt 1.5 herausgefunden haben, mit einem `@`-Symbol am Anfang (keine Leerzeichen dazwischen) vor `reg.oss` setzen:

```
dig -t A @130.255.73.90 reg.oss
```

Das Resultat sollte analog zu Abb. 1.13 sein. Welche IP-Adresse hat `reg.oss`?

Aktivitätsschritt 1.8 (`nslookup` oder `dig`):
Ziel: Vergleich ICANN DNS mit OpenNIC DNS
Überprüfen Sie, ob ICANN DNS und OpenNIC DNS identische Ergebnisse liefern, indem Sie unterschiedliche Domain-Namen auflösen:
`verteiltearchitekturen.de`
`www.hs-mannheim.de`
`ftp.informatik.hs-mannheim.de`

Google unterhält öffentlich zugängliche nicht-autoritative DNS Server (für die reguläre ICANN DNS Hierarchie). Sie haben die öffentlich bekannten (und einprägsamen) numerischen IP-Adressen `8.8.8.8` und `8.8.4.4`.

Aktivitätsschritt 1.9 (`nslookup` oder `dig`):
Ziel: Vergleich Ihres lokalen DNS Servers mit `8.8.8.8`
Vergleichen Sie die Antworten Ihres aktuellen lokalen DNS Servers mit denen des

Google DNS unter `8.8.8.8`, indem Sie unterschiedliche Domain-Namen auflösen:

`verteiltearchitekturen.de`

`www.hs-mannheim.de`

`ftp.informatik.hs-mannheim.de`

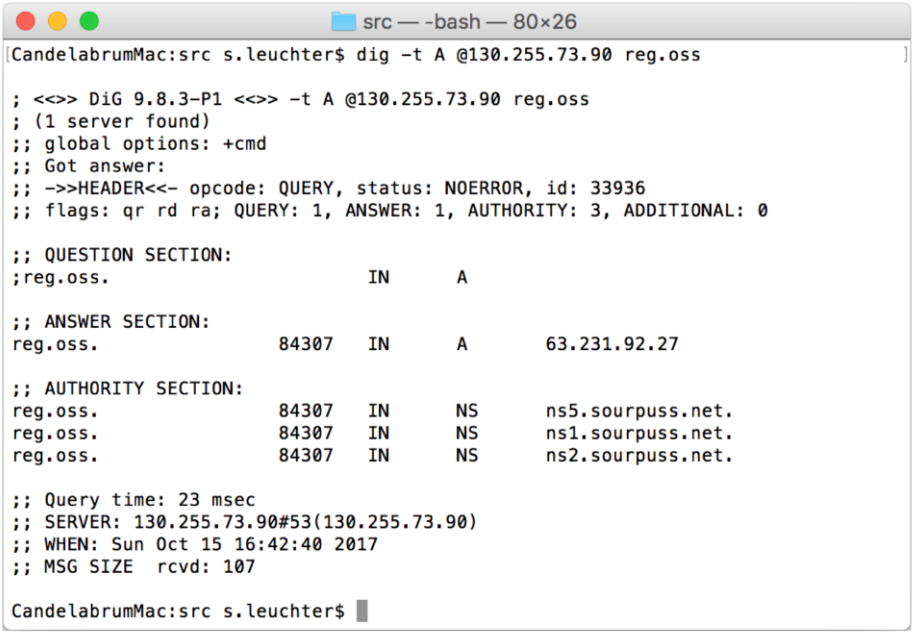


Abbildung 1.13: Abfrage von `reg.oss` bei OpenNIC mit `dig`

1.8 Domaininhaber feststellen

In der DNS-Datenbank sind viele Informationen über Domain-Namen zu finden. Prinzipiell auch, wer der Besitzer oder technisch Verantwortliche eines Eintrags ist. Allerdings gibt es keine `nslookup`- oder `dig`-Abfrage um diese Informationen zu bekommen. Das DNS-Protokoll «*whois*» ist eigentlich für solche Abfragen gedacht. Aus Datenschutzgründen kann man aber inzwischen solche Informationen nur noch über eine interaktive Web-Schnittstelle beim TLD-Registrar erfragen. Das ist bspw. *DENIC* für die `de`-Domain.

Aktivitätsschritt 1.10:
Ziel: Inhaber der Domain `hs-mannheim.de` ermitteln
Öffnen Sie im Web-Browser die Seite `https://www.denic.de/` des Registrars *DENIC*. Über die Funktion «Domainabfrage» (s. Abb. 1.14) können Sie «*whois*»-Informationen über die Domain `hs-mannheim.de` abrufen. Analysieren Sie, welche Informationen über die Web-Schnittstelle zur Verfügung gestellt werden.

Abbildung 1.14: Web-Schnittstelle zu «*whois*»-Informationen von `de`-Domains beim DENIC (`https://www.denic.de/`)

Aktivitätsschritt 1.11:
Ziel: Inhaber der Domain `nice.org.uk` ermitteln
Ermitteln Sie den Inhaber der Domain `nice.org.uk`. Für die TLD `uk` ist der Registrar *Nominet* (`https://www.nominet.uk/`) zuständig.

1.9 Paketvermittlung im Internet: Routing

Grundlage des Internets ist die Paketvermittlung. Nachrichten werden in kleine Pakete zerlegt, die einzeln auf einen Weg vom Sender zum Empfänger geschickt werden. Ein einzelnes Datenpaket wird dafür von Vermittlungsknoten zu Vermittlungsknoten gereicht. Jeder Knoten auf dem Weg inspiziert die Empfängerinformation im Header des Pakets und entscheidet daraufhin zu welchem Folgeknoten das Paket weitergereicht wird («*forwarding*»). Um Netzwerkprobleme zu analysieren, kann es wichtig sein, den Weg solcher Pakete im Internet zu verfolgen. Ein nützliches Hilfsmittel ist dafür das Programm `traceroute` (`tracert` auf Windows-Rechnern).

Aktivitätsschritt 1.12 (Unix (BSD, Linux, macOS)):
Ziel: `traceroute` starten
Um das Hilfsprogramm `traceroute` auszuführen, geben Sie einfach den Befehl `traceroute` in einem *Shell*-Fenster ein (je nach Oberfläche Ihres Betriebssystems könnte es sein, dass der Zugang zur *Shell* auch «Terminal» oder «Konsole» heißt.

Aktivitätsschritt 1.13 (Windows):
Ziel: `tracert` starten
Um das Hilfsprogramm `tracert` auszuführen, öffnen Sie die «Eingabeaufforderung» (z. B. *Win* + `r`, geben Sie dann `cmd` ein). Geben Sie in dem Fenster der «Eingabeaufforderung» `tracert` ein.

Mit dem Hilfsprogramm `traceroute`/`tracert` werden leere (bis auf Header-Informationen) Datenpakete erzeugt, die eine begrenzte Lebensdauer haben. Die Lebensdauer wird im Header als Anzahl von maximal erlaubten *forwards* von einem Vermittlungsknoten zum nächsten vermerkt. Diese sog. «*time to live*» (`TTL`) wird im Hea-

der jedes von `traceroute`/`tracert` erzeugten Datenpakets vermerkt. Aus dem Header geht neben der begrenzten Lebensdauer hervor, dass sie am Ende ihrer Lebenszeit an den Absender zurückgeschickt werden sollen. `traceroute`/`tracert` erzeugt im ersten Schritt ein Datenpaket mit der `TTL=1` und schickt es auf den Weg zum Ziel. Der erste Vermittlungsknoten erkennt, dass die TTL erreicht ist und schickt ein Paket zurück. `traceroute`/`tracert` empfängt das Paket und stellt fest, dass der Absender noch nicht der Zielknoten war. Im nächsten Schritt erzeugt `traceroute`/`tracert` ein neues Paket mit einer um `1` längeren `TTL`. Dadurch erhält der Absender, also das Programm `traceroute`/`tracert` Informationen über den Weg eines Pakets bis zum Ziel.
`traceroute`/`tracert` ermittelt dabei auch das Timing der Übertragungen. Das Internet ist jedoch nicht echtzeitfähig. Die Übermittlung kann an einem Vermittlungsknoten unterschiedlich lange dauern. Deshalb werden jeweils drei sog. «*probes*» geschickt, aus denen ein Mittelwert generiert werden kann, um die generelle Latenz an diesem Abschnitt zu schätzen.

Aktivitätsschritt 1.14:
Ziel: Ermitteln der Netzwerkanbindung
Starten Sie die Verfolgung von Datenpaketen durch

```
traceroute ziel
```

bzw.

```
tracert ziel
```

Setzen Sie als `ziel` eine numerische IP-Adresse oder einen textuellen *Host*-Namen, der zuerst über das DNS zu einer numerischen IP-Adresse aufgelöst wird. Prüfen Sie die Adressen aus der folgenden Liste als `ziel`:
`www.google.de`
`www.google.co.jp`
`www.twitter.com`
`verteiltearchitekturen.de`

Im Beispiel in Abb. 1.15 und 1.17 war der Rechner über das Kabelnetz von Unitymedia mit dem Internet verbunden. Im Beispiel in Abb. 1.16 und 1.18 war der Rechner über das Mobilfunknetz von Telefonica mit dem Internet verbunden.

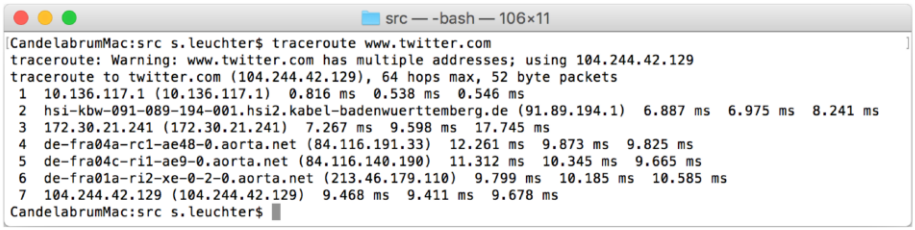


Abbildung 1.15: `traceroute` nach `www.twitter.com` aus dem Netz von Unitymedia

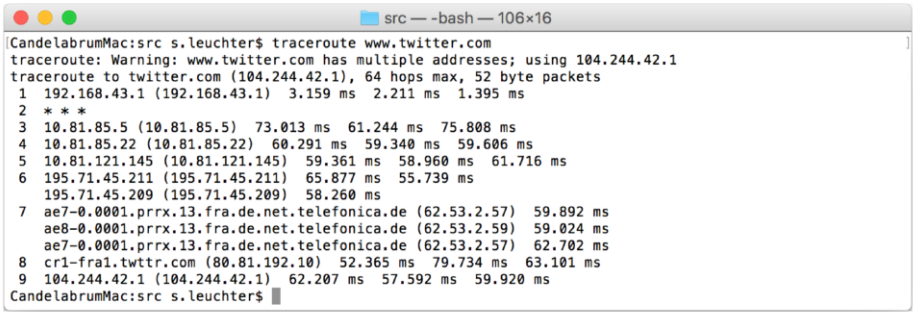


Abbildung 1.16: `traceroute` nach `www.twitter.com` aus dem Netz von Telefonica

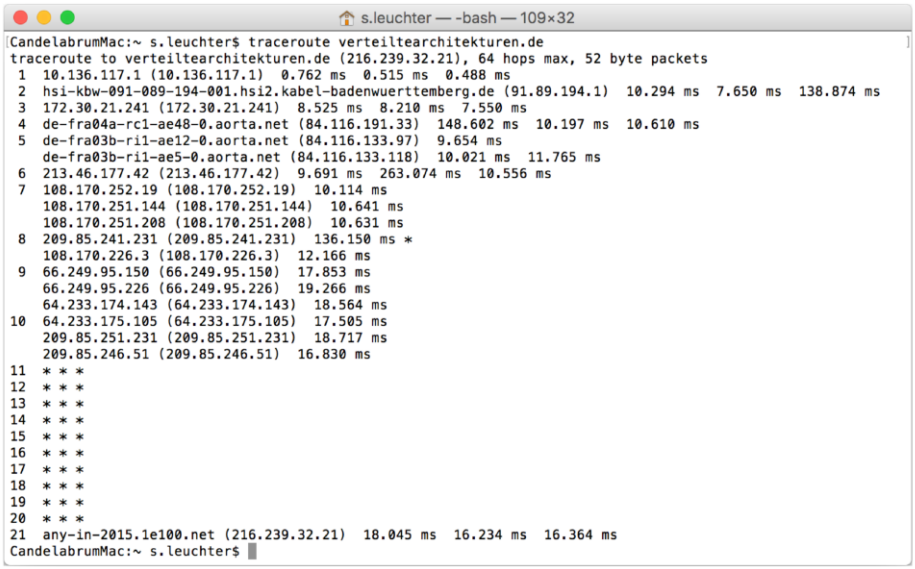


Abbildung 1.17: `traceroute` nach `verteiltearchitekturen.de` aus dem Netz von Unitymedia

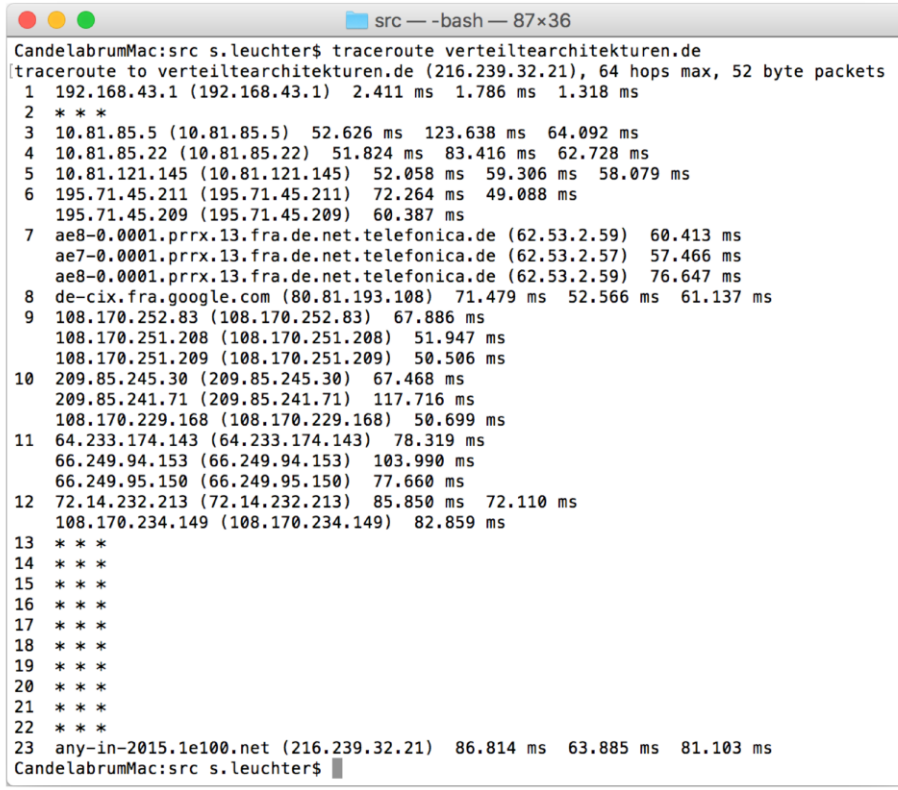


Abbildung 1.18: `traceroute` nach `verteiltearchitekturen.de` aus dem Netz von Telefonica

Aktivitätsschritt 1.15:
Ziel: Änderung der Netzanbindung
Ändern Sie Ihre Anbindung an das Internet, z. B. indem Sie von WLAN auf Mobilverbindung umschalten oder indem Sie eine VPN-Verbindung aufbauen. Alternativ können Sie auch an einen anderen Rechner gehen, von dem Sie annehmen, dass er anders als Ihr eigener an das Internet angebunden ist.

Aktivitätsschritt 1.16:
Ziel: Erneutes Ermitteln der Netzwerkanbindung
Wiederholen Sie Aktivitätsschritt 1.14 und vergleichen Sie die Ausgaben von `traceroute`/`tracert` in beiden Fällen. Hat sich der Weg zu den Zielen verkürzt oder verlängert?

1.10 Netzwerkkonfiguration ermitteln und ändern

Die Dienstprogramme `ipconfig` (für Windows) und `ifconfig` (Unix) sind sehr nützlich um die Netzwerkkonfiguration eines Rechners zu analysieren und zu ändern.

1.10.1 Windows: `ipconfig`

Aktivitätsschritt 1.17 (Windows/DOS):
Ziel: Anzeigen der Netzwerkkonfiguration
Das Hilfsprogramm `ipconfig` ist im Prinzip dem Unix- (BSD, Linux, macOS) Programm `ifconfig` ähnlich. Man muss es jedoch anders starten. Um die aktuelle IP-Konfiguration des Rechners anzuzeigen wird `ipconfig` mit dem Parameter `\all` gestartet:

```
ipconfig \all
```

Das Ergebnis könnte – abhängig von der Anzahl und Art der Netzwerkanschlüsse – wie in Abb. 1.19 aussehen. Vergleichen Sie Ihre Konfiguration mit der in Abb. 1.19 dargestellten. Welche Unterschiede fallen Ihnen auf?

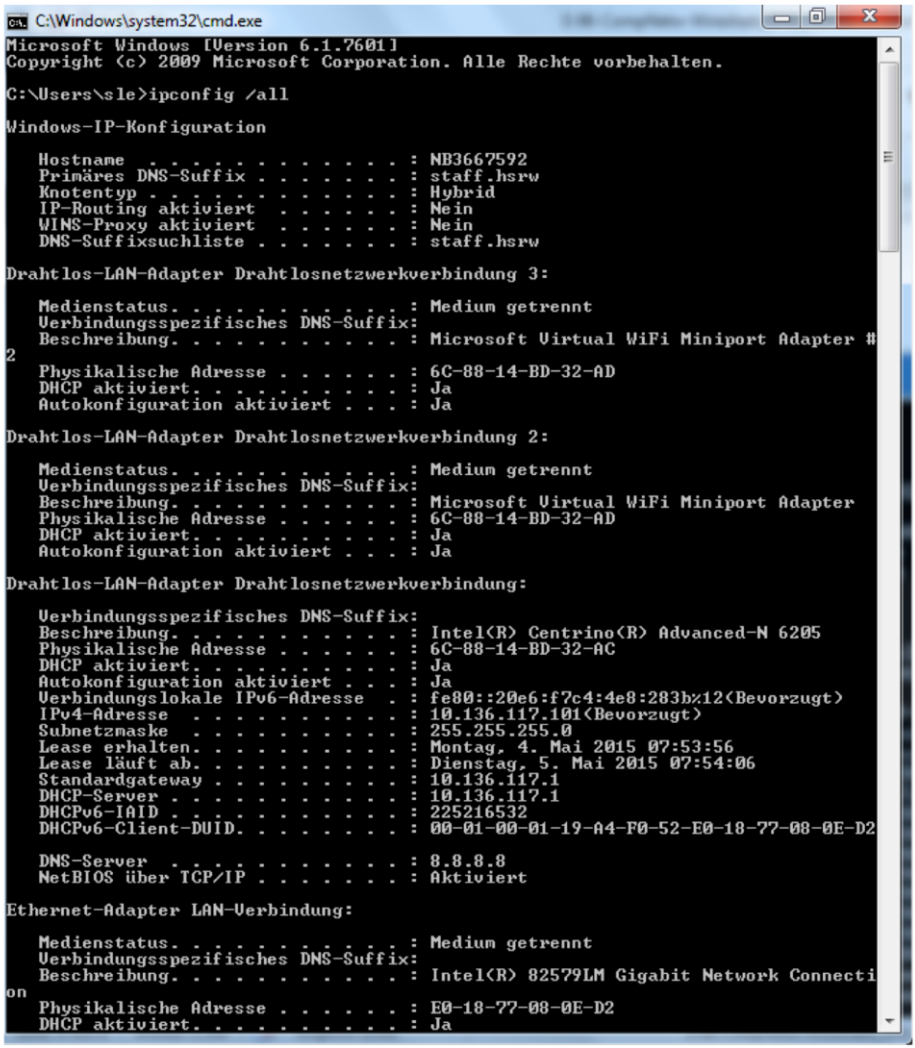


Abbildung 1.19: `ipconfig \all` auf einem Windows-Rechner mit mehreren Netzwerkanschlüssen

Mit `ipconfig` kann auch die Interaktion mit dem DNS beeinflusst werden. Der Client zum lokalen DNS Server speichert empfangene Antworten, damit zukünftige Anfragen nach demselben textuellen Domain-Namen schneller, nämlich ohne Rückfrage beim lokalen DNS Server, beantwortet werden können.

Aktivitätsschritt 1.18 (Windows/DOS):
Ziel: Anzeigen der DNS-Inhalte im lokalen Cache
Zeigen Sie mit dem folgenden Kommando die DNS-Inhalte im lokalen Cache an:

```
ipconfig /displaydns
```

Jeder Eintrag in der Ergebnisliste zeigt die verbleibende *time to live* (`TTL`) an, die er noch im Cache gespeichert bleibt. Der Cache kann geleert werden, in dem `ipconfig` mit der Option «`/flushdns`» aufgerufen wird.

Aktivitätsschritt 1.19 (Windows/DOS):
Ziel: Leeren des lokalen Caches
Leeren Sie mit dem folgenden Kommando die DNS-Inhalte aus dem lokalen Cache:

```
ipconfig /flushdns
```

Aktivitätsschritt 1.20 (Windows/DOS):
Ziel: Erneutes Anzeigen der DNS-Inhalte im lokalen Cache
Zeigen Sie die DNS-Inhalte im lokalen Cache an. Öffnen Sie die Website `verteiltearchitekturen.de` in einem Web-Browser und zeigen Sie die DNS-Inhalte im lokalen Cache erneut an.

1.10.2 Unix (BSD, Linux, macOS): `ifconfig`

Auf den Unix-artigen Betriebssystemen (BSD, Linux, macOS) steht meistens das Kommando `ifconfig` zur Verfügung («`if`» steht dabei für *Interface*). Mit `ifconfig` können alle relevanten Informationen über Netzwerkadapter abgefragt und (mit Adminstratorrechten) gesetzt werden.

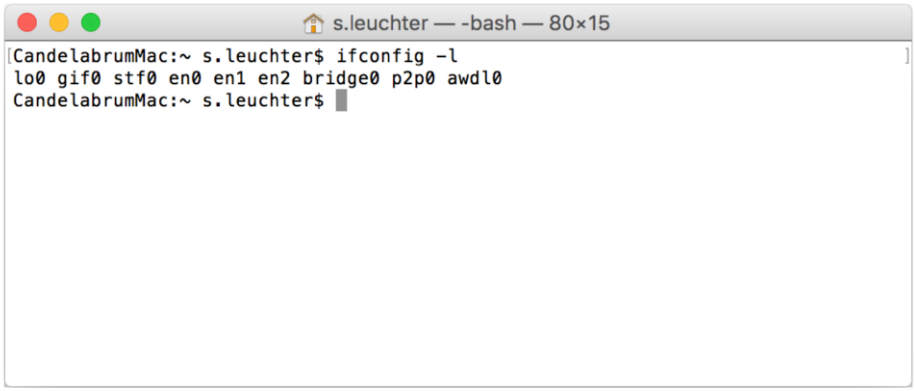


Abbildung 1.20: Ausgabe der Liste aller Netzwerkanschlüsse: `ifconfig -l`

Aktivitätsschritt 1.21 (Unix (BSD, Linux, macOS)):
Ziel: Alle Netzwerkanschlüsse ausgeben

Um eine Liste aller an einem Unix- (BSD, Linux, macOS) Rechner vorhandenen Netzwerkanschlüsse zu bekommen kann `ifconfig` mit der Option «`-l`»[a] aufgerufen werden:

```
ifconfig -l
```

Das Ergebnis ist eine Liste aller Namen von Netzwerkanschlüssen (z. B. wie in Abb. 1.20). Diese Namen erlauben einen Rückschluss auf die Art des Anschlusses (s. u.). Über den Namen können Eigenschaften des jeweiligen Anschlusses eingesehen und geändert werden.

Geben Sie die Liste Ihrer Netzwerkanschlüsse aus.

[a] *kleiner Buchstabe L*

Aktivitätsschritt 1.22 (Unix (BSD, Linux, macOS)):
Ziel: Netzwerkanschluss genauer analysieren

Analysieren Sie einzelne der Netzwerkanschlüsse, die Sie im vorigen Aktivitätsschritt 1.21 ermittelt haben, beispielsweise:

```
ifconfig lo0
```

```
ifconfig en0
```

```
ifconfig en1
```

Die konkreten Ausgaben können sich je nach Gerät und Fähigkeiten unterscheiden. Abb. 1.21 zeigt die Ausgabe für das *loop back device*. Mit dieser virtuellen Netzwerkkarte kann nur der Rechner selber erreicht werden. Dieses Gerät sollte auf jedem Rechner vorhanden sein und hat i. Allg. die IP-Adresse `127.0.0.1`. Der Wifi-Anschluss hat im Beispiel in Abb. 1.21 den Namen `en0`. Er ist IPv4 und IPv6 fähig, während der Ethernet-Anschluss `en1` in Abb. 1.21 ausschließlich IPv6 benutzt.
Versuchen Sie interessante Eigenschaften Ihrer Netzwerkanschlüsse genauer zu untersuchen.

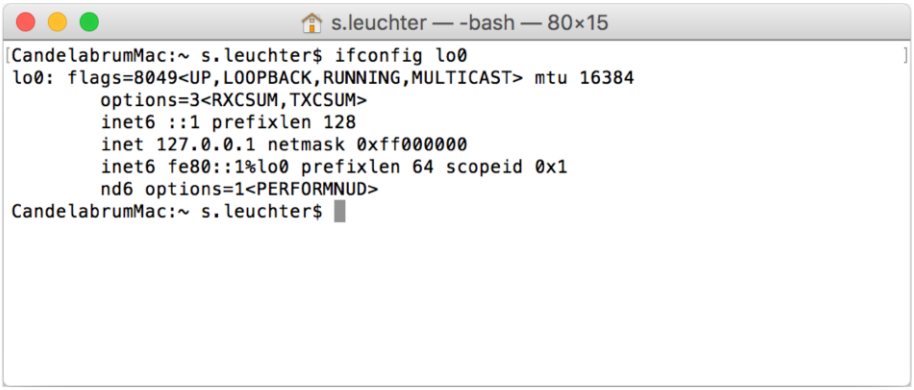


Abbildung 1.21: Netzwerkkonfiguration des *loop back device*: `ifconfig lo0`

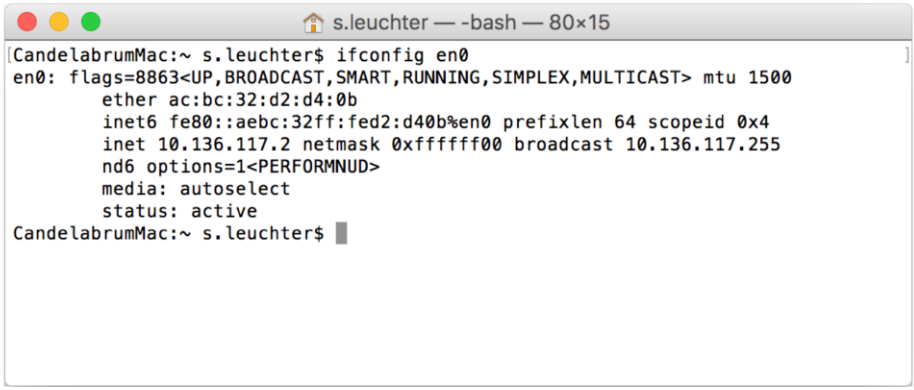


Abbildung 1.22: Netzwerkkonfiguration der WiFi-Karte: `ifconfig en0`

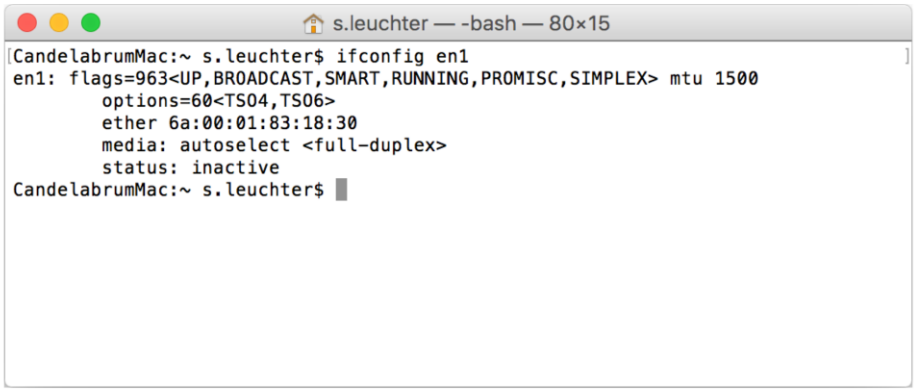


Abbildung 1.23: Netzwerkkonfiguration des Ethernet-Anschlusses: `ifconfig en1`

Socket-Kommunikation mit dem User Datagram Protocol (UDP)

Sockets erlauben Prozessen miteinander zu kommunizieren, auch wenn sie auf unterschiedlichen Hosts im Internet laufen. Die Sockets werden über Internet-Adresse (bei IPv4: 32 Bit Zahl, bei IPv6: 128 Bit Zahl) und Port (16 Bit Zahl von 0 bis 65535) adressiert. Wird alternativ ein textueller *Host*-Name statt einer numerischen Internet-Adresse als Socket-Adressierungsbestandteil benutzt, wird der Name über das DNS (s. Abschnitt 1.5) zu einer numerischen IP-Adresse aufgelöst. Ein Prozess kann mehrere Sockets verwenden. Es handelt sich bei der Socket-Adresse also nicht um eine Prozessadresse. Sockets funktionieren immer bidirektional, ðüber einen Socket können Daten sowohl versendet als auch empfangen werden.

In Java wird `java.net.InetSocketAddress` verwendet um Socket-Adressen zu repräsentieren. Es gibt mehrere Konstruktoren: Ein Ziel-*Host* kann dabei mit einem `InetAddress`-Objekt oder einen `String` für den `host`-Anteil und einem `int` für die `port`-Nummer erzeugt werden. Der `String` kann hierbei die Form

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

einer numerischen IP-Adresse haben (`216.239.32.21`) oder ein DNS-auflösbarer *Host*-Name sein (`verteiltearchitekturen.de`). Statt `host` und `port` zusammen anzugeben, gibt es auch einen Konstruktor, der nur die Port-Nummer als einzigen Parameter hat, wobei dann ein Socket auf dem lokalen Rechner (`localhost` bzw. `127.0.0.1`) gemeint ist.

UDP ist im Gegensatz zu TCP **verbindungslos**. Ein UDP Socket kann daher von mehreren Kommunikationspartnern gleichzeitig verwendet werden. Beispielsweise kann ein Server einen UDP Socket öffnen, der von allen Clients des Services gleichzeitig verwendet wird.

Bei UDP werden **Datagramme** zwischen Sender und Empfänger ausgetauscht. Der englische Begriff «*Datagram*» steht für «*Data Telegram*». Ein *Datagramm* ist eine in sich abgeschlossene Nachricht, die je eine `java.net.InetSocketAddress` für den Sender und den Empfänger beinhaltet.

Sendet ein Host mehrere *Datagramme* an denselben Empfänger, kann man sich allerdings nicht über die Reihenfolge des Empfangs sicher sein. Außerdem kann man sich nicht darauf verlassen, dass das *Datagramm* überhaupt ankommt. Die *Quality of Service* von UDP ist «*best effort*». Dafür hat das UDP aber wenig Overhead und die Latenz ist daher vergleichsweise gering.

UDP wird in Java über ein `java.net.DatagramSocket`-Objekt benutzt. Ein `DatagramSocket` kann zum Senden und Empfangen verwendet werden.

2.1 Lebenszyklus von java.net.DatagramSocket-Objekten

2.1.1 UDP-Socket-Konstruktoren

Es gibt eine Reihe von Konstruktoren für typejava.net.DatagramSocket-Objekte, die nützlich sind, je nachdem, ob der `DatagramSocket` von einem Client oder einem Server erzeugt wird. Die gebräuchlichsten sind:

- `DatagramSocket() throws SocketException`

UDP Socket auf irgendeinem verfügbaren Port auf `localhost` (für UDP Clients geeignet)

- `DatagramSocket(int port) throws SocketException`

  UDP Socket auf einem bestimmten Port auf `localhost` (für UDP Server geeignet; darf nicht schon von einem anderen Socket (TCP oder UDP) belegt sein, sonst wird eine `SocketException` geworfen)

- `DatagramSocket(int port, InetAddress iaddr) throws SocketException`

  Für UDP Server geeignet; falls ein Server mehrere IP-Adressen hat (z. B. wenn mehrere Netzwerkkarten – bspw. eine Wifi und eine Ethernet-Karte – vorhanden sind), kann mit diesem Konstruktor über den Parameter `iaddr` bestimmt werden, welche IP-Adresse der Socket verwendet.

2.1.2 Vor der Verwendung: *Port binding*

Bevor ein Socket verwendet werden kann, muss er an einen Port gebunden werden. Der folgende Konstruktor erzeugt einen ungebundenen *Socket*, da keine Implementierung übergeben wird. Direkt anschließend muss er aber mit `bind(SocketAddress addr)` an einen Port gebunden werden.

```
DatagramSocket s = new DatagramSocket(null);
s.bind(new InetSocketAddress(8888));
```

... hat deshalb denselben Effekt wie ...

```
DatagramSocket s = new DatagramSocket(8888);
```

2.1.3 Am Ende: Socket schließen

Nach Verwendung sollte ein Socket mit `close()` geschlossen werden, um seine Ressourcen wieder freizugeben. Andernfalls kann der Port nicht für weitere Sockets (TCP oder UDP) benutzt werden.

In mehreren der folgenden Beispiele wird statt des expliziten `close()`-Aufrufs das Java SE 7 *try-with-resources* Statement[1] verwendet. Das ist möglich, da die Klasse `DatagramSocket` das Interface `java.lang.AutoCloseable` implementiert. Dadurch wird der Code etwas kompakter. Beachten Sie aber, dass durch die Verwendung des *try-with-resources* Statements der Aufruf von `close()` am Ende des Lebenszyklus eines `DatagramSocket`-Objekts implizit erfolgt.

2.2 Lesen und Schreiben über einen UDP Socket

Die *Datagramme*, die über `DatagramSocket`-Objekte gesendet und empfangen werden, werden als `java.net.DatagramPacket` repräsentiert. Die wichtigsten Konstruktoren für diese Klasse sind:

- `DatagramPacket(byte[] buf, int length)`
  Zum Empfangen eines Datagrams mit `length` Bytes, die in `buf` gespeichert werden sollen (`length` $\leq$ `buf.length`).
- `DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
  Erzeugt ein *Datagramm* zum Senden von `length` Bytes, die in `buf` gespeichert sind, an einen /typeDatagramSocket an `address` und `port` (`length` $\leq$ `buf.length`).

`DatagramPacket`-Objekte haben die folgenden gebräuchlichen *Properties*, die mit den entsprechenden *Getter*- und *Setter*-Methoden benutzt werden können:

- `Address` (Typ: `InetAddress`): Die IP-Adresse des Servers, zu dem das *Datagramm* gesendet werden soll oder von dem es empfangen wurde.
- `Data` (Typ: `byte[]`): Die *payload* des *Datagramms*.
- `Length` (Typ: `int`): Die Länge des Byte Arrays, das die *payload* des *Datagramms* ist.
- `Port` (Typ: `int`): Die Port-Nummer des Services, zu dem das *Datagramm* gesendet werden soll oder von dem es empfangen wurde.

[1] `https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`

- `SocketAddress` (Typ: `SocketAddress`): Die Adresse des Sockets auf dem Server, zu dem das *Datagramm* gesendet werden soll oder von dem es empfangen wurde (enthält IP-Adresse + Port-Nummer).

Über ein existierendes und gebundenes `DatagramSocket`-Objekt können einzelne `DatagramPacket`-Objekte gesendet und empfangen werden:

- `void send(DatagramPacket p) throws IOException`
- `void receive(DatagramPacket p) throws IOException`
  Der Aufruf `receive(p)` blockiert, bis ein *Datagramm* empfangen wurde. Nachdem `receive(p)` fertig abgearbeitet ist und der Programmfluss beim nächsten Statement weitergeht, ist die empfangene Nachricht im `DatagramPacket`-Objekt `p` repräsentiert. Der Inhalt von `p` wird dabei überschrieben.

2.2.1 Timeout beim Lesen von einem UDP Socket

Der Aufruf von `receive(p)` blockiert standardmäßig solange, bis neue Daten am Socket anliegen. Da man sich in verteilten Systemen nicht darauf verlassen kann, dass alle Kommunikationspartner immer verfügbar sind (z. B. wegen schlechter Funknetzwerkverbindungen), ist es erforderlich bei der Programmierung einen «Plan B» vorsehen zu können: Man kann das Verhalten von `receive(p)` durch Aufruf der Methode `setSoTimeout(int timeout)` beeinflussen. Damit kann die maximale Zeitdauer (in ms) gesetzt werden, die `receive(p)`-Aufrufe blockieren. Wird zwischen dem (blockierenden) Aufruf von `receive(p)` und der Zeitdauer des *Timeouts* kein *Datagramm* an diesem Socket empfangen, endet die Blockade des `receive(p)`-Aufrufs. Um zu signalisieren, dass mit dem *Timeout* ein asynchrones Ereignis eingetreten ist, wird eine `java.net.SocketTimeoutException` geworfen, die entsprechend zu behandeln ist.
Die Dauer bis zum *Timeout* wird in Java immer in Millisekunden angegeben. Falls dieser Wert `0` ist, werden *Timeouts* deaktiviert.

Praktikum

Aktivitätsschritte:

Unter `http://verteiltearchitekturen.de/vol01/VAR-UDP-solution.zip` können Sie eine Musterlösung zu diesem Praktikum herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-UDP_solution`.

Im folgenden Praktikum werden Sie einfache Client-/Server-Anwendungen programmieren, bzw. insbesondere die dafür erforderliche Netzwerkkommunikation in Java. Die Kommunikation findet direkt (also noch ohne die Verwendung von Middleware-Produkten oder von Frameworks wie in den nächsten Kapiteln) auf der Ebene des Transportprotokolls UDP statt.

Aktivitätsschritt 2.1:
Ziel: Projekt downloaden und in Eclipse importieren
Laden Sie das ZIP-Archiv mit dem Projekt `VAR-UDP` unter `http://verteiltearchitekturen.de/vol01/VAR-UDP.zip` herunter. Importieren

Sie das ZIP-File in Eclipse mit der Funktion *File → Import... → General / Existing Projects Into Workspace*. Wählen Sie dort die Option *Select Archive File*. Sie sollten auch darauf achten, dass die Option *Copy Projects Into Workspace* aktiviert ist, sonst arbeiten Sie möglicherweise nur auf den Dateien in Ihrem Download-Ordner und nicht in Ihrem Eclipse Workspace.

Als Ergebnis sollte das Projekt `VAR-UDP` in Ihrem Package oder Project Explorer in Eclipse auftauchen. Im `src`-Ordner dieses Projektes sollten Sie drei Packages sehen:

- `var.sockets.udp.time`
- `var.sockets.udp.echo`
- `var.sockets.udp.messwerte`

In jedem der drei Pakete dieses Praktikums gibt es je eine Klasse für einen Client und einen Server zu einem Service.

Ein *Service* ist eine Funktionalität, die ein Server mehreren Clients bereitstellt. Die Clients nutzen die Funktionalität des Services, die auf dem Server bereitgestellt wird. Der Begriff *Server* ist mehrdeutig: Es kann sich um den Rechner handeln, der einen Service bereitstellt, oder es kann sich um den Prozess auf einem Server-Rechner handeln, der den Service implementiert.

In Ihrem Eclipse Workspace Directory sollte ein neues Verzeichnis für das Projekt entstanden sein. In dem Projektverzeichnis gibt es mindestens einen Ordner namens

```
└── var
    └── sockets
        └── udp
            ├── echo
            │   ├── EchoClient.java
            │   └── EchoServer.java
            ├── messwerte
            │   ├── MesswertClient.java
            │   └── MesswertServer.java
            └── time
                ├── TimeClient.java
                └── TimeServer.java
```

Abbildung 2.1: Verzeichnis- und Dateibaum für das Projekt `VAR-UDP`

`src/`, in dem die Packages als (Unter-)Verzeichnisse auftauchen. Abb. 2.1 zeigt die Struktur unterhalb des Folders `src/`.

Im folgenden Praktikum erwarten die Server jeweils in einer Endlosschleife *Datagramme* von Clients (in diesen Beispielen immer `packetIn`). Die Aufgaben unterscheiden sich darin, ob das empfangene Paket Nutzdaten (engl. *payload*) enthält, die weiterverarbeitet werden. Die Services der Pakete `var.sockets.udp.echo` und `var.sockets.udp.messwerte` enthalten Nutzdaten, die vom Server weiterverarbeitet werden sollen.

Auf den Inhalt eines *Datagramms* kann mit `packetIn.getData()` zugegriffen werden. Das Ergebnis ist ein Byte-Array der Länge `packetIn.getLength()`.

Weiter unterscheiden sich die Services der Praktikumsaufgaben darin, ob der Server eine Antwort in Form eines neuen *Datagramms* (in diesen Beispielen immer `packetOut`) erzeugen und an den Client zurückschicken soll. Bei den ersten zwei Aufgaben (Services der Pakete `var.sockets.udp.echo` und `var.sockets.udp.time`) wird nach Empfang eines *Datagramms* eine Antwort in Form eines neuen *Datagramms* (`packetOut`) an den Sender zurückgeschickt.

Die Zieladresse des zurückzuschickenden Pakets wird immer aus dem eingegangenen *Datagramm* (in diesen Beispielen immer `packetIn`) ermittelt, indem die Methoden `packetIn.getSocketAddress()` (liefert mit einem `SocketAddress`-Objekt Port und Internet-Adresse zusammen) oder `packetIn.getAddress()` und `packetIn.getPort()` einzeln ausgeführt werden. Das Ergebnis wird dann mit `packetOut.setSocketAddress(...)` bzw. mit `packetOut.setAddress(...)` und `packetOut.setPort(...)` im zu versendenden *Datagramm* eingefügt.

Es ist wichtig zu beachten, dass UDP-Kommunikation immer verbindungslos ist. Das bedeutet u. a., dass ein `DatagrammSocket` von mehreren Clients quasi gleichzeitig benutzt werden kann.

Das ist der Grund dafür, dass die Zieladresse nicht zum `DatagramSocket`-Objekt gehört und deshalb nicht im Konstruktor der Klasse `DatagramSocket` übergeben wird. Stattdessen muss die Zieladresse am `DatagrammPacket` spezifiziert werden. Das kann über den Konstruktor oder später über `setSocketAddress(...)` bzw. mit `setAddress(...)` und `setPort(...)` geschehen.

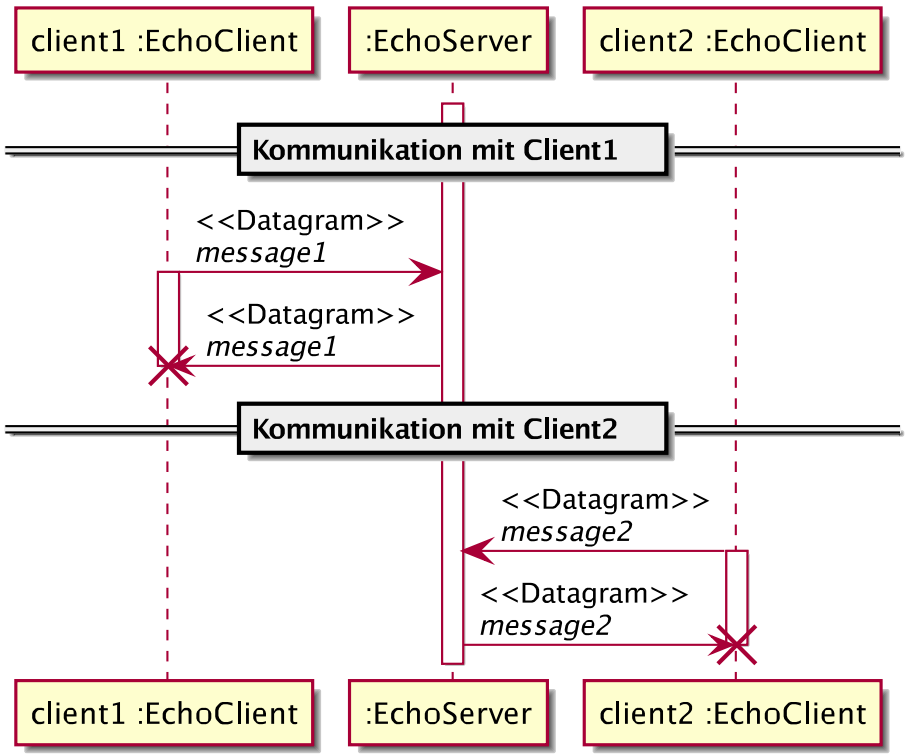


Abbildung 2.2: Prinzipielle Funktion des Echo Service

2.3 Aufgabe: Echo Service

Ein erster Service soll als einfache Funktionalität alle empfangenen Daten an den Sender zurückmelden («**echo**»). Dazu soll der Server die *payload* jedes empfangenen *Datagramms* in den Nutzdaten eines neuen *Datagramms* an den Client zurücksenden (s. Abb. 2.2). Der Client soll ein *Datagramm* mit dem ersten Kommandozeilenparameter an den Server senden und bis zu einem *Timeout* auf das Eintreffen eines Antwort-*Datagramms* warten. Der Inhalt der Antwort soll ausgegeben werden.
Im Package `var.sockets.udp.echo` sind Client und Server bereits vorgegeben. Der Code dort ist allerdings noch nicht ganz lauffähig. Finden Sie die Fehler und testen Sie einen Server mit mehreren Clients gleichzeitig. Variieren Sie, auf welchen Rechnern die Clients sind.

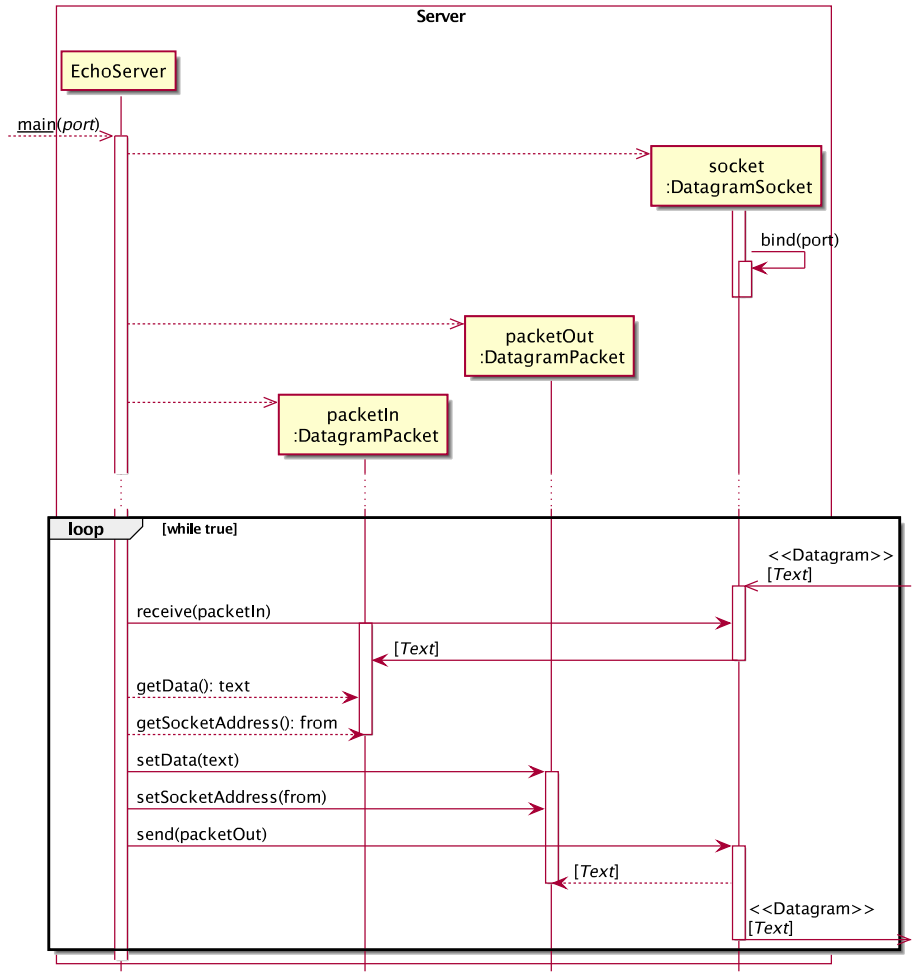


Abbildung 2.3: Sequenzdiagramm des Servers für den *Echo Service* (vereinfacht)

2.3.1 Server für den Echo Service

Aktivitätsschritt 2.2:
Ziel: Analyse des Ablaufs in `EchoServer`
Analysieren Sie den Quelltext des Servers (`EchoServer`): Was sind die wesentlichen Schritte, die innerhalb der Endlosschleife ablaufen? Können Sie den Quellcode von `EchoServer` mit dem (vereinfachten) Sequenzdiagramm in Abb. 2.3 in Einklang bringen?

Eine Endlosschleife ist ein starker Hinweis darauf, dass es sich um einen Server handelt. (Achtung: Das ist nur ein Indiz und kein Beweis. Im Client zum `var.sockets.udp.messwerte`-Service gibt es z. B. auch eine Endlosschleife.)

Aktivitätsschritt 2.3:
Ziel: Zieladresse des *Ausgangsdatagramms* aus Eingang ermitteln und setzen
Vor dem Senden muss `packetOut` noch eine Zieladresse bekommen. Rufen Sie die Dokumentation von Java SE 7 zur Methode `setSocketAddress()` auf: `http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramPacket.html#setSocketAddress(java.net.SocketAddress)` und leiten Sie ab, welche Methode Sie an `packetOut` aufrufen müssen und wie Sie die Zieladresse dieser Rückantwort an den Client auf `packetIn` auslesen können.

2.3.2 Client für den Echo Service

Aktivitätsschritt 2.4:
Ziel: Analyse des Ablaufs in `EchoClient`
Analysieren Sie den Quelltext des Servers (`EchoClient`): Aus welcher Abfolge von Schritten besteht das Hauptprogramm? Können Sie den Quellcode von

`EchoClient` mit dem (vereinfachten) Sequenzdiagramm in Abb. 2.4 in Einklang bringen?

Aktivitätsschritt 2.5:
Ziel: Fehler in `EchoClient` erkennen und verbessern
Achten Sie darauf, welches Paket der Client vom Socket liest. Soll das wirklich `packetOut` sein? Korrigieren Sie den Parameter.

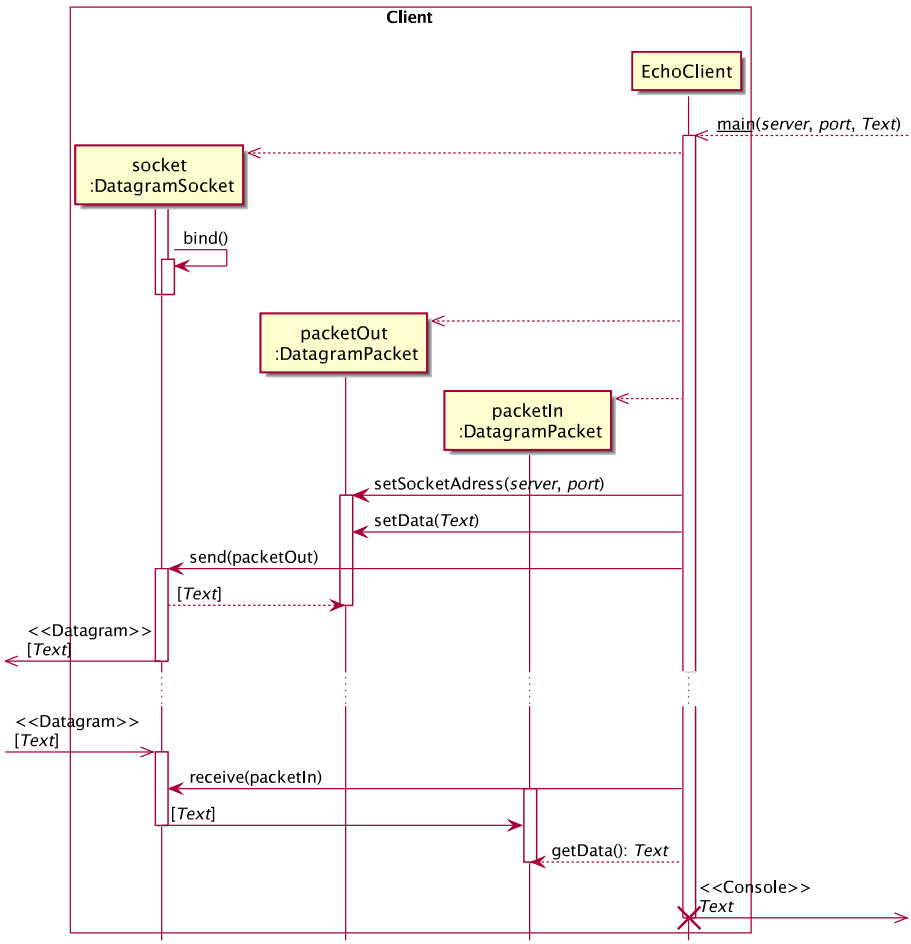


Abbildung 2.4: Sequenzdiagramm des Clients für den *Echo Service* (vereinfacht)

Aktivitätsschritt 2.6:

Ziel: Kommandozeilenparameter über Eclipse an Hauptprogramm übergeben

Um den Client zu starten benötigen Sie Kommandozeilenparameter. In Eclipse können Sie Kommandozeilenparameter beim Start einer `main()`-Klasse über eine *Run Configuration* übergeben. Sie können die Run Configurations über das Menü *Run → Run Configurations...* anlegen. Im Dialog zum Anlegen einer *Run Configuration* kann über das *New*-Icon eine neue *Run Configuration* für *Java Application* angelegt werden. Im nächsten Schritt können im Reiter *Main* das entsprechende Projekt und die Klasse mit der zu startenden `main(...)`-Methode spezifiziert werden (s. Abb. 2.5). Im nächsten Schritt können dann unter dem Reiter *Arguments* wie in Abb. 2.6 gezeigt die Kommandozeilenparameter im Bereich *Program arguments* angegeben werden.

Da im Beispiel-Client nur genau der erste Kommandozeilenparameter (`args[0]`) an den Server gesendet wird, und mehrere Kommandozeilenparameter mit Leerzeichen voneinander getrennt werden, müssen Nachrichten mit Leerzeichen und Zeilenumbrüchen mit «`"`» und «`\`» quotiert werden wie im Beispiel:

```
"Die Hochschule Mannheim ist bekannt für eine \
praxisnahe und theoretisch fundierte Ausbildung."
```

Verteilte Anwendungen sind unpraktisch zu testen: Es werden i. Allg. mehrere Prozesse benötigt, die manchmal in einer speziellen Reihenfolge gestartet werden müssen. Wenn das nicht automatisiert wird (im professionellen Software Engineering oft mit Hilfe von Scripting-Sprachen) und stattdessen interaktiv in Eclipse auf einem Entwicklungsrechner getestet wird, muss zwischen den Ein-/Ausgabe-Konsolen der Prozesse gewechselt werden. Das kann durch Klick auf das Icon *Display Selected Console*, wie in Abb. 2.7 dargestellt, bewerkstelligt werden.

Aktivitätsschritt 2.7:

Ziel: Clients und Server auf mehrere Rechner verteilen und testen

In der vorgegebenen Konfiguration des Echo Service Beispiels laufen Server und Clients auf demselben Rechner (`localhost`). Tauschen Sie mit anderen Studierenden die IP-Adressen und rufen Sie entfernte (also auf anderen Rechnern be-

findliche) Server mit Ihrem Client auf oder umgekehrt. Dazu müssen Sie sich auf dieselbe Portnummer (Konstante `PORT`) einigen. Der Client muss die IP-Adresse des Servers kennen (Konstante `HOST`).

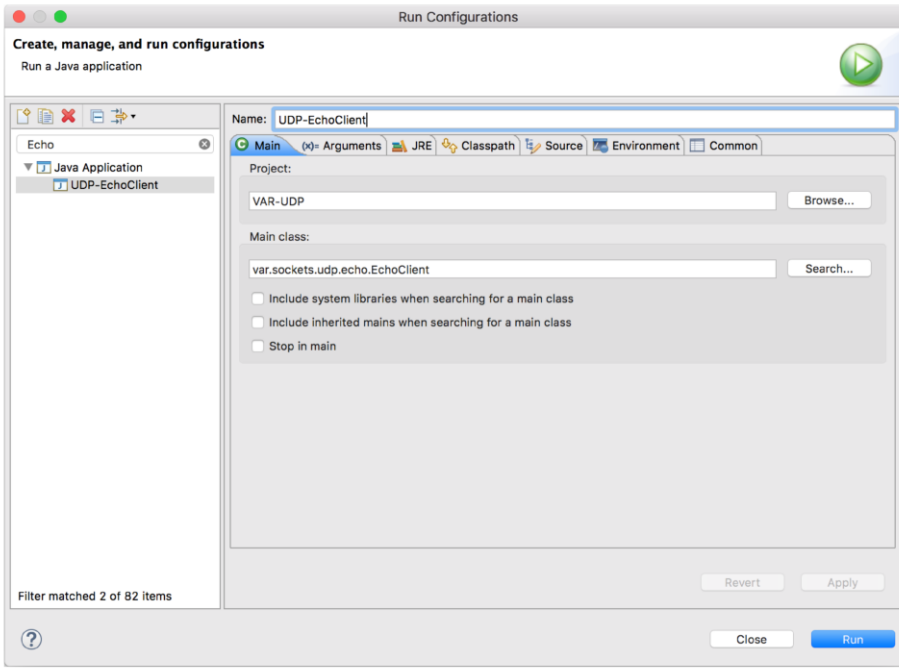


Abbildung 2.5: Screenshot aus Eclipse: Anlegen einer *Run Configuration*, Schritt 1

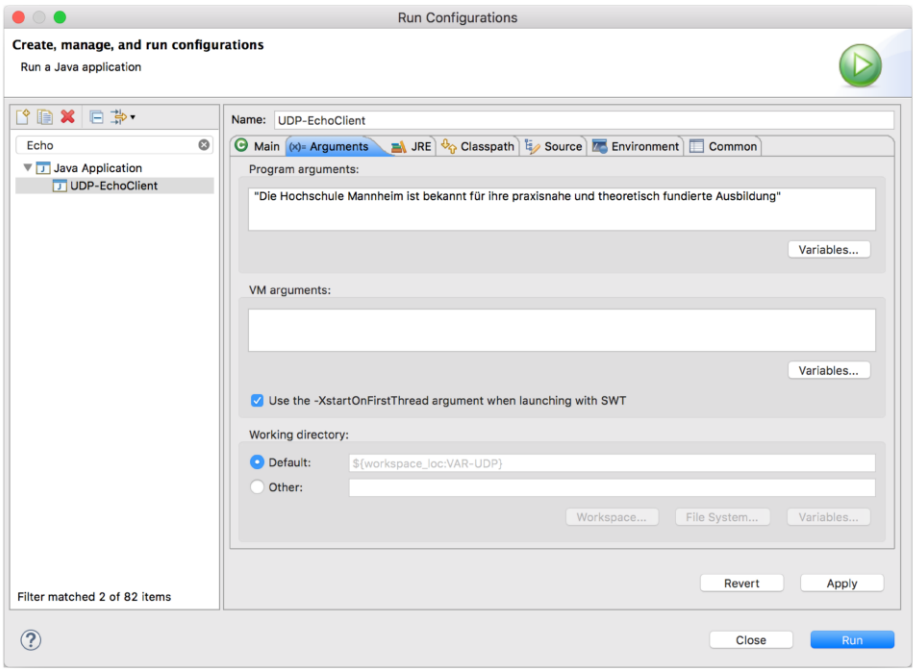


Abbildung 2.6: Screenshot aus Eclipse: Anlegen einer *Run Configuration*, Schritt 2

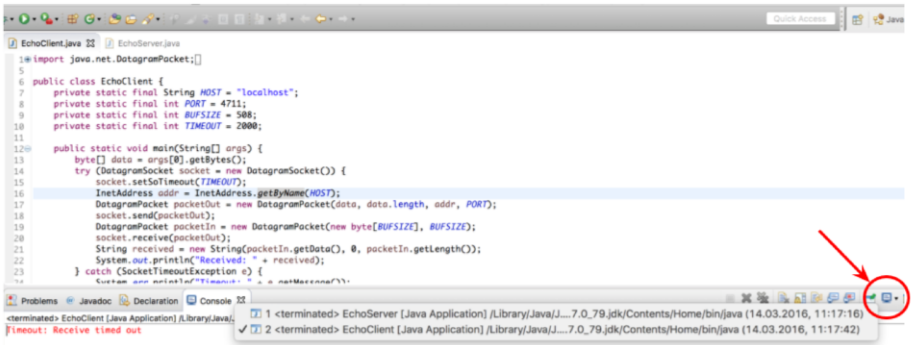


Abbildung 2.7: Screenshot aus Eclipse: Wechsel zwischen mehreren Prozessen

2.4 Aufgabe: Time Service

Der `TimeServer` erzeugt als Reaktion auf den Empfang eines «leeren» *Datagramms* einen Zeitstempel der aktuellen Systemzeit des Servers in Form einer Zeichenkette, die in einem Antwort-*Datagramm* an den Client zurück geschickt wird (s. Abb. 2.8).
Der dazu passende `TimeClient` (s. Abb. 2.10) erzeugt ein leeres *Datagramm*, schickt es über einen UDP Socket an den Server, wartet auf die Antwort des Servers (s. Abb. 2.9), auf deren Empfang hin die Systemzeit des Servers ausgegeben wird. Der Client wartet maximal zwei Sekunden auf die Antwort des Servers. Falls bis dahin keine Antwort eingeht, wird der Client mit einem Timeout-Fehler beendet.

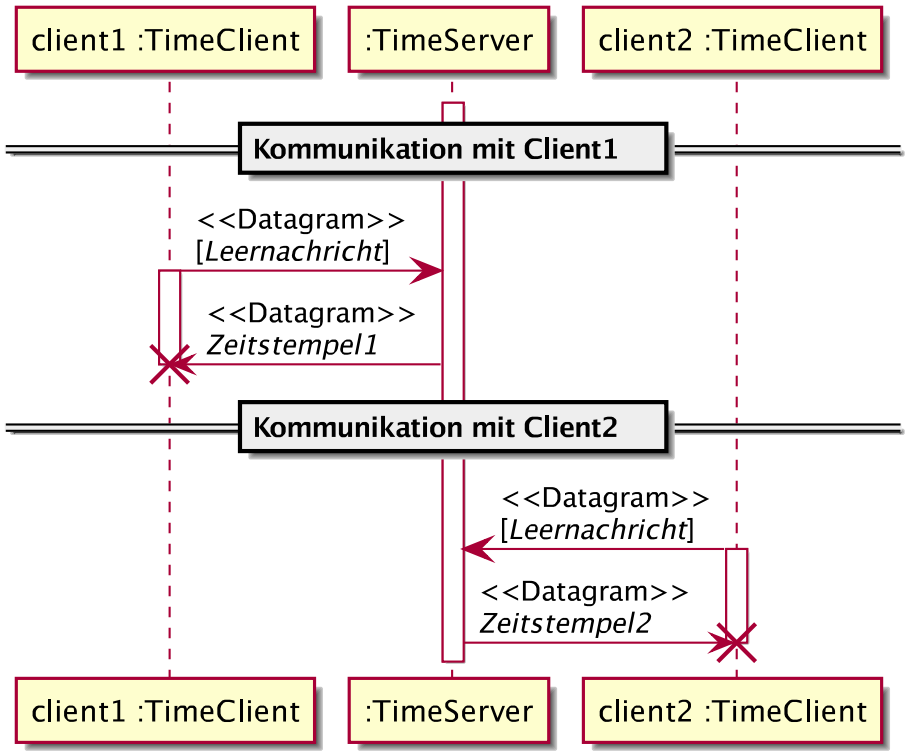


Abbildung 2.8: Prinzipielle Funktion des Time Service

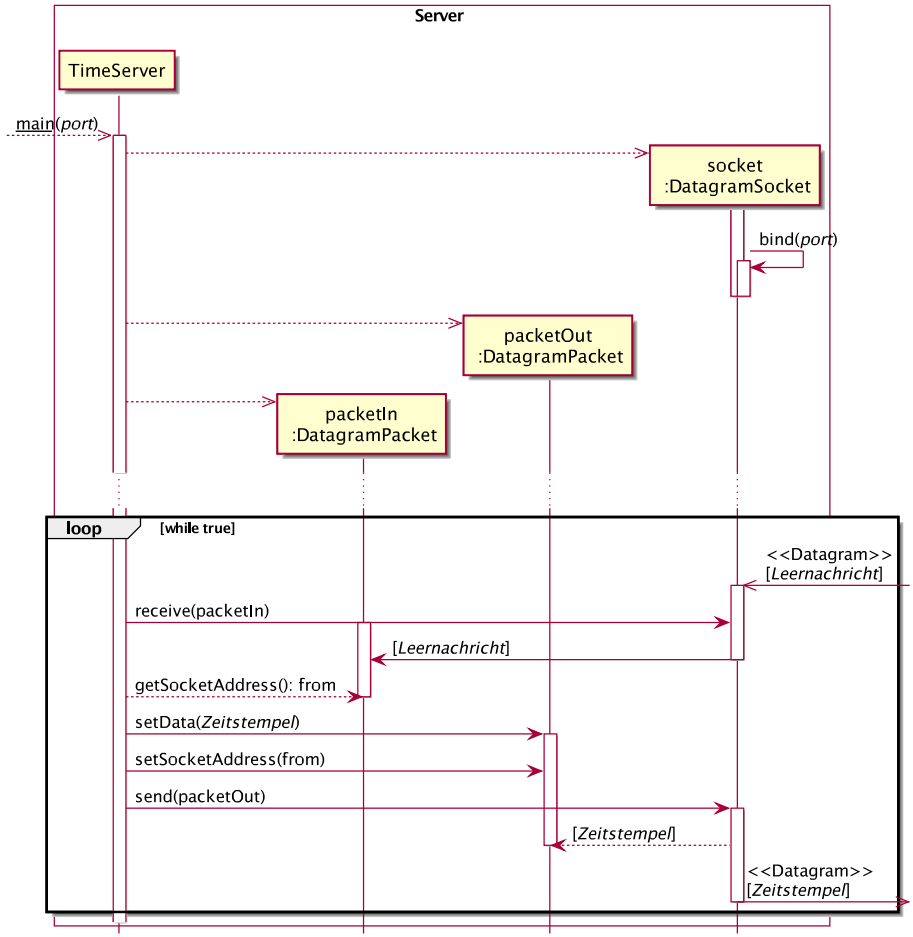


Abbildung 2.9: Sequenzdiagramm des Servers für den *Time Service* (vereinfacht)

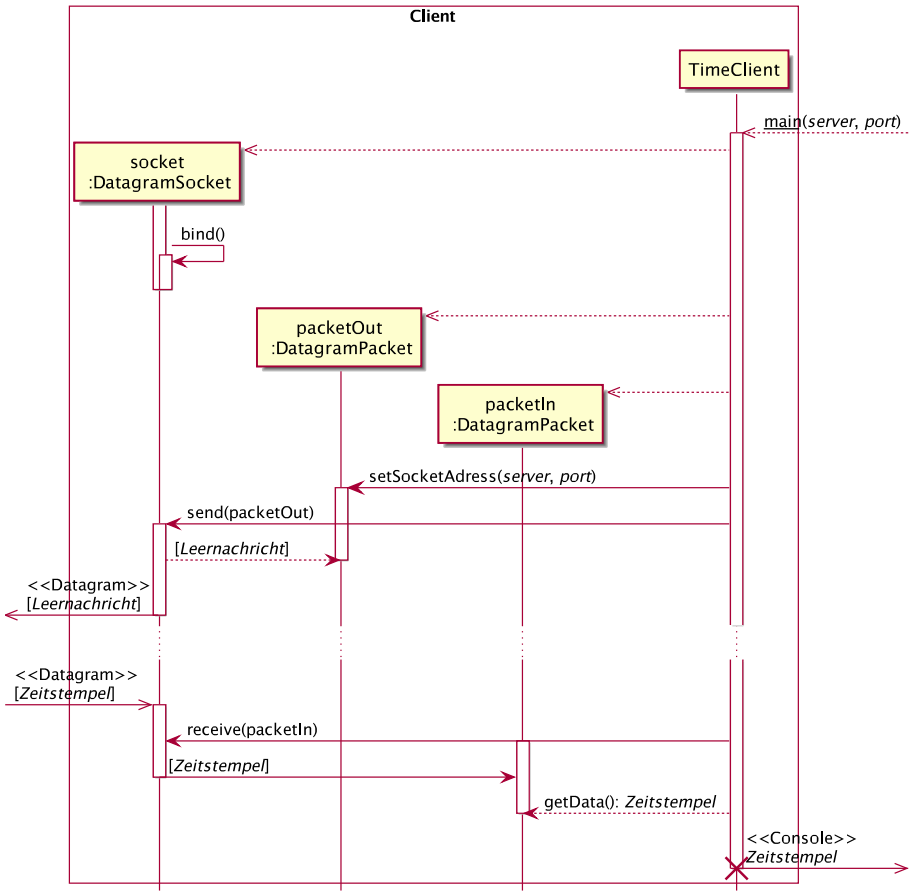


Abbildung 2.10: Sequenzdiagramm des Clients für den *Time Service* (vereinfacht)

Listing 2.1: Gerüst für `var.sockets.udp.time.TimeClient`

```
1 package var.sockets.udp.time;
2 public class TimeClient {
3
4   public static void main(String[] args) {
5     // TODO Auto-generated method stub
6   }
7 }
```

Listing 2.2: Gerüst für `var.sockets.udp.time.TimeServer`

```
1 package var.sockets.udp.time;
2 public class TimeServer {
3
4   public static void main(String[] args) {
5     // TODO Auto-generated method stub
6   }
7 }
```

Aktivitätsschritt 2.8:

Ziel: Fundament für `TimeClient` und `TimeServer` aus Echo Service kopieren

Im Projekt `VAR-UDP` gibt es im Ordner `src` das Paket `var.sockets.udp.time`. Dort sind Client (s. Listing 2.1) und Server (s. Listing 2.2) als Gerüst vorbereitet.

Nutzen Sie den Code von `EchoClient` und `EchoServer` aus `var.sockets.udp.echo`. Kopieren Sie alles aus diesen beiden Klassen an die entsprechenden Stellen der vorgegebenen Gerüste.

2.4.1 Server für den Time Service

Aktivitätsschritt 2.9:

Ziel: Time Service spezifischen Code für `TimeServer` entwickeln

Der `TimeServer` empfängt wie der `EchoServer` ein Paket, sendet aber einen anderen Inhalt als `EchoServer` zurück. Der Server muss dafür den Inhalt von `packetIn` nicht auswerten, sondern erzeugt selber die *payload* von `packetOut` (s. Listing 2.3). Ändern Sie den enstprechenden Teil des `TimeServer` Codes.

Listing 2.3: Code-Schnipsel zum Setzen des Zeitstempels als Nutzdateninhalt des *Datagramms* `packetOut`

```
1 String jetzt = (new Date()).toString();
2 packetOut.setData(jetzt.getBytes());
3 packetOut.setLength(jetzt.length());
```

2.4.2 Client für den Time Service

Aktivitätsschritt 2.10:
Ziel: `TimeClient` für Time Service umbauen
Der `TimeClient` unterscheidet sich vom `EchoClient` nur darin, dass er ein leeres Paket an den Server sendet. Sie brauchen also `args[0]` nicht der Variable `data` zuweisen. Statt `data` und `data.length` können Sie `packetOut` mit einem leeren Byte-Array (`new byte[0]`) und der Länge 0 erzeugen.

2.5 Aufgabe: Messwert Service

Der Client für den Messwert Service (`MesswertClient`) sendet in Abständen von fünf Sekunden Zufallszahlen an den Server. Mit den Zufallszahlen wird der Einfachheit halber das Auslesen eines Sensors simuliert. Der Server (`MesswertServer`) gibt diese Zahlen zusammen mit der IP-Adresse des Senders, dem Port des Sockets des Senders und dem Zeitpunkt der Messung am Bildschirm aus.

Beispiel

```
127.0.0.1:62253 Mon Mar 27 07:31:24 CEST 2017 36.514389799756114
127.0.0.1:62252 Mon Mar 27 07:31:25 CEST 2017 44.470316990128784
```

Im Gegensatz zu *Echo Service* und *Time Service* hat der *Messwert Service* einen anderen Ablauf. Das (vereinfachte) Sequenzdiagramm in Abb. 2.11 soll verdeutlichen, dass nun auch jeder Client in einer Endlosschleife Nachrichten mit Messwerten produziert und an den Server sendet.
Der Server sendet nun keine Nachrichten mehr als Antwort an den Client zurück, sondern speichert die Nachricht der Clients nur bei sich in einer Datenbank. Demzufolge ist kein Objekt `packetOut` mehr vorhanden.

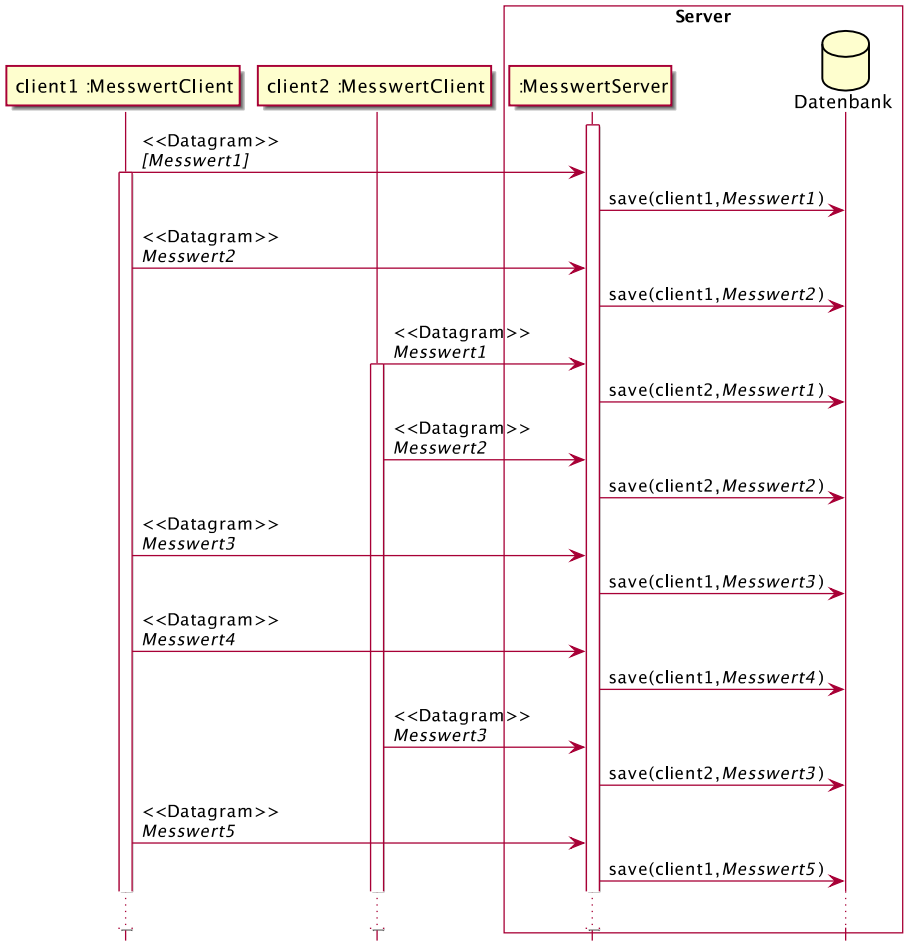


Abbildung 2.11: Prinzipielle Funktion des Messwert Service

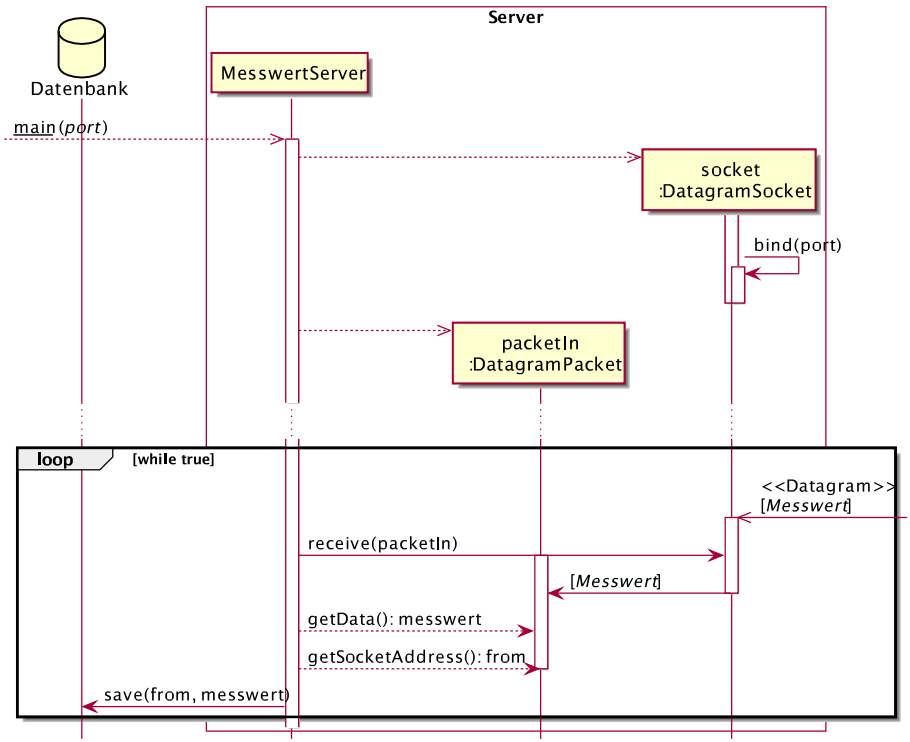


Abbildung 2.12: Sequenzdiagramm des Servers für den *Messwert Service* (vereinfacht)

Der Server arbeitet wie bei den bisherigen Services in einer Endlosschleife (s. Abb. 2.12).
Der Client (s. Abb. 2.13) braucht auch kein Objekt `packetIn` mehr und sendet seinerseits in einer Endlosschleife Messwerte an den Server.

2.5.1 Server für den Messwert Service

Aktivitätsschritt 2.11:
Ziel: Fundament für `MesswertServer` aus `EchoServer` kopieren
Kopieren Sie wieder die Funktionalität aus der Klasse `EchoServer` in das vorgefertigte Gerüst von `MesswertServer` (s. Listing 2.4) .

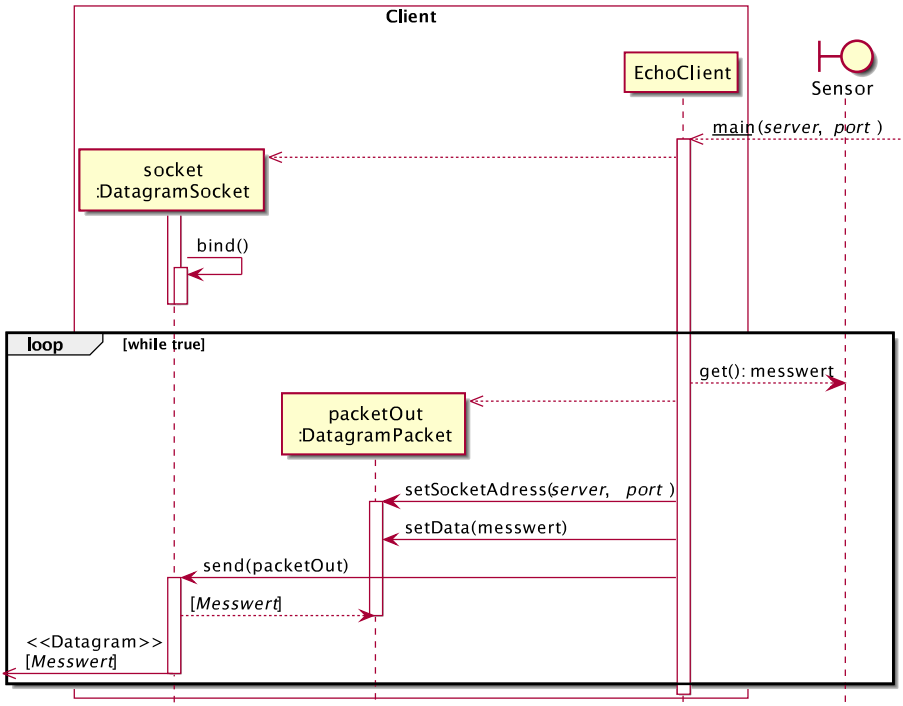


Abbildung 2.13: Sequenzdiagramm des Clients für den *Messwert Service* (vereinfacht)

Listing 2.4: Gerüst für `var.sockets.udp.messwerte.MesswertServer`

```
1 package var.sockets.udp.messwerte;
2 public class MesswertServer {
3
4   public static void main(String[] args) {
5     // TODO Auto-generated method stub
6   }
7 }
```

Aktivitätsschritt 2.12:
Ziel: `MesswertServer` für Messwert Service umbauen
Diesmal braucht der Server nichts an die Clients zurückzusenden. Das `packetOut`-*Datagramm* wird also hier nicht benötigt.
Allerdings sollen empfangene Pakete etwas anders ausgegeben werden als beim `EchoServer`. Geben Sie die folgenden Informationen entsprechend formatiert in einer Zeile aus:

- `packetIn.getAddress().getHostAddress()`
- `packetIn.getPort()`
- `packetIn.getData()`

2.5.2 Client für den Messwert Service

Aktivitätsschritt 2.13:
Ziel: Fundament für `MesswertClient` aus `EchoClient` kopieren
Kopieren Sie wieder die benötigte Funktionalität aus `EchoClient` in das vorgegebene Gerüst von `MesswertClient` (s. Listing 2.5).

Beachten Sie, dass dieser Server dauerhaft mit mehreren Clients zusammenarbeiten kann. Alle Clients schicken ihre *Datagramme* an denselben Socket des Servers, der sie alle gleichzeitig bearbeitet. UDP ist nicht verbindungsorientiert. Daher können alle Clients gleichzeitig mit demselben Socket des Servers kommunizieren.

Listing 2.5: Gerüst für `var.sockets.udp.messwerte.MesswertClient`

```
1  package var.sockets.udp.messwerte;
2  public class MesswertClient {
3
4      public static void main(String[] args) {
5          Random rg = new Random();
6          try {
7              // ...
8              while (true) {
9                  String jetzt = (new Date()).toString();
10                 String messung = Double.toString(rg.nextDouble() *
                       100.0);
11                 // ...
12                 Thread.sleep(5000);
13             }
14         } catch (Exception e) {
15             System.err.println(e);
16         }
17     }
18 }
```

2.6 Anregung ReactionGame: Ein einfaches Reaktionsspiel mit Timeouts

Die folgende Aufgabe ist fakultativ: Neue Kernkonzepte verteilter Architekturen kommen hier nicht mehr vor. Diese Aufgabe ist mehr als Ideengeber für eigene Projekte gedacht und soll Sie motivieren, Experimente mit UDP zu machen. Da UDP so einfach ist, kann man leicht eigene vorhandene Programme um einen Socket erweitern, von dem man z. B. Kommandos zur Fernsteuerung entgegennehmen kann. Oft ist eine asynchrone Verwendung des Sockets erforderlich, weil das Programm bereits seinem eigenen Kontrollfluss folgt. In dem Fall kann das Verhalten der `receive(p)`-Methode durch `setSoTimeout(t)` asynchron gemacht werden: Die `receive(p)`-Methode blockiert nur maximal für `t` Millisekunden. Sollte bis dahin kein *Datagramm* am Socket ankommen, wird eine `java.net.SocketTimeoutException` geworfen.

Abb. 2.14 zeigt den prinzipiellen Ablauf des Reaktionsspiels: Der Server fungiert als «Torwart», der Client ist der «Stürmer». Ist das Tor ungeschützt, während ein Leernachricht-Datagramm des Clients beim Server eintrifft, hat der Stürmer Erfolg. Trifft die UDP-Nachricht des Client-Stürmers hingegen ein, während das Tor «nicht

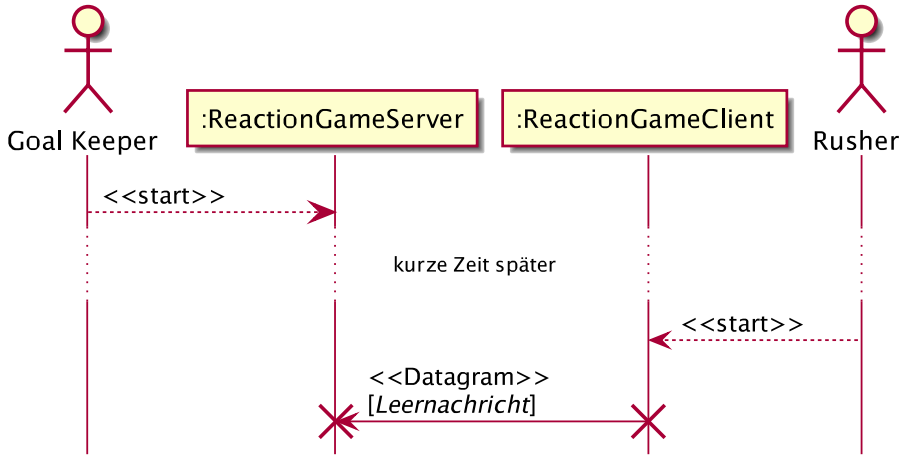


Abbildung 2.14: Prinzipieller Ablauf des Reaktionsspiels

auf Empfang» ist, handelt es sich um einen Fehlschuss des Stürmers und der Torwart hat gewonnen.

Aktivitätsschritt 2.14 (fakultativ):

Ziel: Server und Client für das *ReactionGame* entwickeln

Das *ReactionGame* ist ein einfaches Reaktionsspiel. Man kann es sehr einfach mit UDP Sockets und einem *Timeout* umsetzen. In Abb. 2.15 ist der Ablauf der Spielidee dargestellt. `GoalKeeper` und `Rusher` spielen gegeneinander.

Der `GoalKeeper` startet sein Programm `ReactionGameServer` durch Aufruf der `main()`-Methode. Als erstes wird eine zufällig lange Zeit $\leq$ `MAX_WAITING_TIME_MS` ms gewartet, indem `Thread.sleep(...)` aufgerufen wird. Dann wird ein neuer `DatagramSocket` auf einem bekannten Port geöffnet. Der Socket wird so konfiguriert, dass `receive(p)`-Aufrufe einen *Timeout* nach `GATE_OPEN_DURATION_MS` ms erzeugen. Dann wird `receive(p)` aufgerufen.

Schafft es der `Rusher` seinen Client genau so zu starten, dass er im Zeitfenster zwischen dem `receive(p)`-Aufruf des Servers und dem *Timeout* nach `GATE_OPEN_DURATION_MS` ms ein leeres *Datagramm* an den Server schicken kann, hat er gewonnen. Falls sich das Tor aber aufgrund des *Timeouts* schon ge-

schlossen hat, wenn das Paket beim Server eintrifft, hat der `GoalKeeper` gewonnen.

Aktivitätsschritt 2.15 (fakultativ):
Ziel: Server für das *ReactionGame* multiplayerfähig machen
Man kann das Spiel leicht erweitern, sodass mehrere `Rusher` gleichzeitig gegen einen `GoalKeeper` spielen können. Die Identität eines erfolgreichen `Rusher`'s könnte entweder in der *Payload* seines *Datagramms* übermittelt oder durch die Absender-IP-Adresse im (leeren) *Datagramm* ermittelt werden.

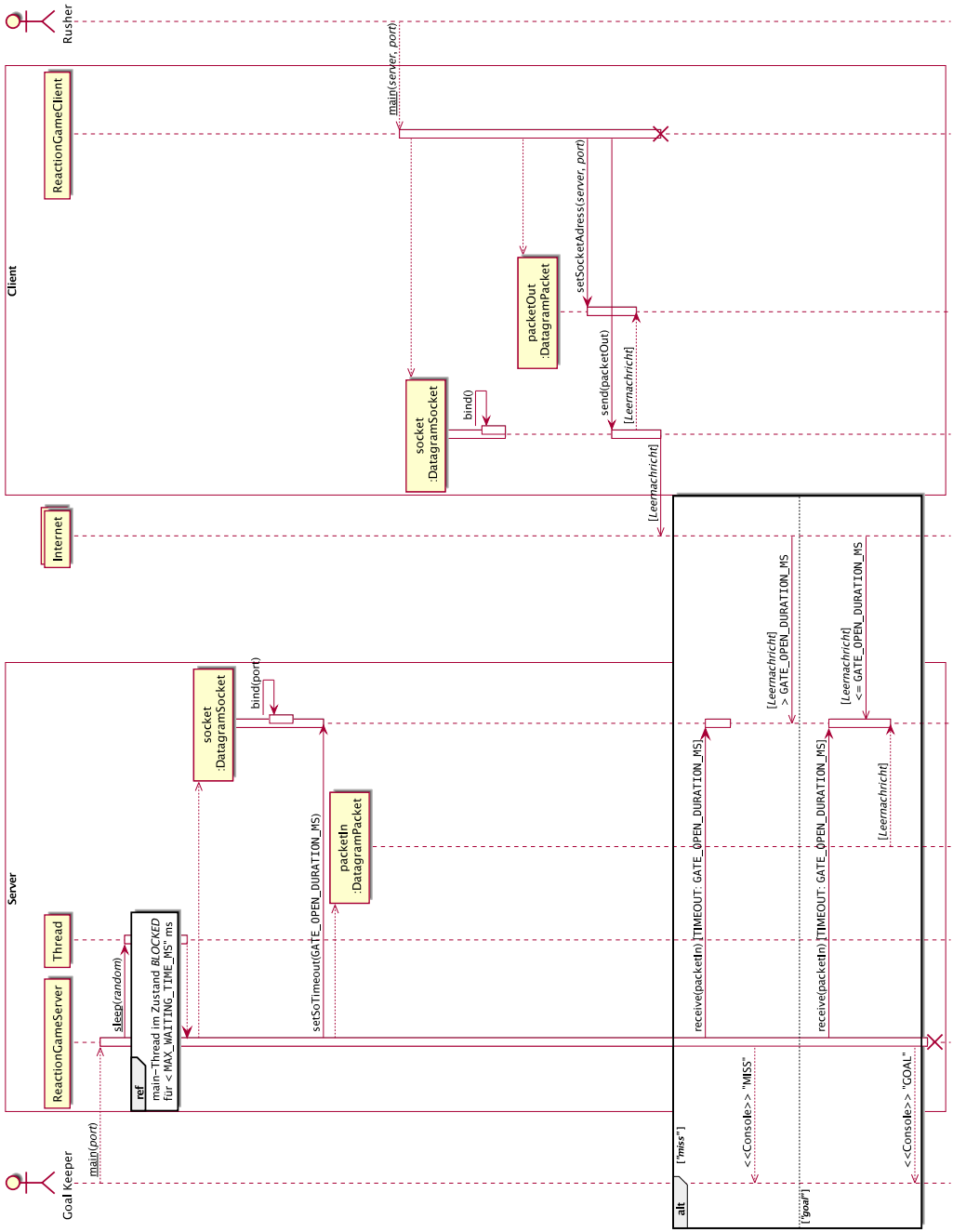


Abbildung 2.15: Sequenzdiagramm des *ReactionGame* (vereinfacht)

Socket-Kommunikation mit dem Transmission Control Protocol (TCP)

Sockets verbinden Prozesse miteinander, die auf unterschiedlichen *Hosts* im Internet laufen können. Die Sockets werden über Internet-Adresse (bei IPv4: 32-Bit Zahl und bei IPv6: 128-Bit Zahl oder DNS-*Host*-Name) und Port (16-Bit Zahl von 0 bis 65535) adressiert.

In Java wird `java.net.InetSocketAddress` verwendet, um Socket-Adressen zu repräsentieren. Es gibt mehrere Konstruktoren: Ein Ziel-*Host* kann dabei mit einem `InetAddress`-Objekt oder einen `String` für den `host`-Anteil und einem `int` für die `port`-Nummer erzeugt werden. Der `String` kann hierbei die Form einer numerischen IP-Adresse haben («`141.19.1.171`») oder ein DNS-auflösbarer *Host*-Name sein («`verteiltearchitekturen.de`»). Statt `host` und `port` zusammen anzugeben, gibt es auch einen Konstruktor, der nur die Port-Nummer als einzigen Parameter hat, wobei dann ein Socket auf dem lokalen Rechner («`localhost`» bzw. «`127.0.0.1`») gemeint ist.

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

3.1 Das Transportprotokoll TCP

Das Transportprotokoll TCP arbeitet wie UDP immer unter Zuhilfenahme des Internet-Protokolls (IP). Im Gegensatz zu UDP wird mit TCP ein exklusiv zu nutzender bidirektionaler Kommunikationskanal zwischen Sender und Empfänger etabliert.

TCP ist im Gegensatz zu UDP **verbindungsorientiert**. Ein TCP Socket an einem Server kann daher immer nur von genau einem Client verwendet werden.

3.1.1 Dienstgüteparameter von TCP

Auch bei TCP werden auf Netzwerkebene die Nachrichten, die vom Sender an den Empfänger geschickt werden, in Datenpakete («Segmente») zergliedert, die beim Empfänger wieder zusammengesetzt werden.
TCP garantiert im Gegensatz zu UDP, dass die Reihenfolge des Empfangs der Segmente der Reihenfolge des Versands entspricht, auch wenn einzelne Pakete unterschiedliche Wege durch das Internet nehmen. Außerdem kann man sich darauf verlassen, dass keine Pakete unterwegs verloren gehen – es gibt eine Zustellungsgarantie für versandte Nachrichten. Sollte eine aufgebaute Verbindung trotzdem nicht benutzbar sein (z. B. weil zwischenzeitlich ein Netzwerkproblem vorliegt), wird nach einer gewissen Zeit ein Timeout-Ereignis generiert. Auch Übertragungsfehler werden erkannt und korrigiert. Weiterhin bietet das TCP eine Flusskontrolle. Wenn ein Sender schneller Nachrichten sendet als der Empfänger aus seinem Socket-Ende lesen und verarbeiten kann, wird der Sender gebremst.
Für den Benutzer eines TCP Sockets sind die Maßnahmen zur Erbringung der Dienstgüte *transparent*: Das TCP-Protokoll sorgt intern beispielsweise dafür, dass verlorengegangene oder fehlerhafte Pakete erkannt und erneut angefordert werden.
Um diese im Vergleich zu UDP höhere Dienstgüte zu erreichen, ist das Protokoll von TCP umfangreicher und komplexer abzuarbeiten. Außerdem werden zusätzliche Daten im Header benötigt, die mitübertragen werden müssen und die zur Verfügung stehende Bandbreite schmälern.

Durch diesen *Overhead* ist die Übertragung von Daten mit TCP langsamer als mit UDP. TCP-Übertragung führt außerdem zu einer erhöhten Latenz, weswegen das einfachere UDP für besonders zeitkritische Anwendungen wie Sprachkommunikation oder bei der Übertragung von Positionsdaten zwischen mehreren interagierenden Partnern verwendet werden sollte. Sollen hingegen Dateien fehlerfrei in einem Netzwerk ausgetauscht werden, sollte TCP benutzt werden.

Obwohl TCP eine ganze Reihe von Transportdiensten anbietet, gibt es durchaus auch weitergehende vorstellbare Funktionen, die in TCP nicht umgesetzt sind, wie beispielsweise die Priorisierung von Verbindungen oder die Reservierung einer bestimmten Bandbreite für eine Verbindung oder einen Dienst.

3.1.2 TCP Sockets

Die speziellen Eigenschaften von TCP Sockets und die Art, wie man sie normalerweise verwendet, sind keine Features einer Bibliothek oder von Programmiersprachen, sondern im Wesentlichen Vorgaben des Netzwerk-Stacks, der auf den meisten Plattformen als Teil des Betriebssystems realisiert ist. Die folgenden Ausführungen sind somit nicht spezifisch für die Programmiersprache Java, denn alle Konzepte auf Betriebssystemebene werden von Java nur durchgereicht.

Lebenszyklus von TCP Sockets

Bei TCP sind im Gegensatz zu UDP Server Socket und Client Socket unterschiedlich. Am Server Socket kann die Funktion `accept()` aufgerufen werden, die solange blockiert, bis sich ein Client mit dem Server Socket verbindet. Clientseitig wird dafür ein `Socket`-Objekt erzeugt, an dem die `connect(...)`-Methode aufgerufen wird. In dem Moment wird ein neuer Socket auf Server-Seite erzeugt.
Dieser neue Socket hat eine neue Port-Nummer und steht exklusiv dem gerade anfragenden Client zur Verfügung. Die Verbindung wird also zwischen dem `Socket`-Objekt des Clients und dem neu erzeugten Socket aufgebaut. Der Client bekommt die Port-Nummer dieses neuen Sockets mitgeteilt und kommuniziert im Folgenden mit diesem Socket. Der Server Socket, an dem `accept()` aufgerufen worden war, ist hingegen nun wieder frei um den nächsten Client-Verbindungswunsch mit einem neu zu erzeugenden Socket zu beantworten.

Der Server Socket, an dem `accept()` aufgerufen wird, wird auch «*Welcome Socket*» genannt. Das im Vergleich zu UDP kompliziert wirkende Verfahren ist erforderlich, weil TCP verbindungsorientiert arbeitet und so der «*Welcome Socket*» immer frei für weitere Verbindungsanfragen von Clients bleibt, auch wenn der Server schon Verbindungen zu anderen Clients unterhält. Diese Verbindungen haben serverseitig unterschiedliche Port-Nummern, die auch nicht vorhersehbar sind, sondern nach Bedarf aus dem Pool freier 16 Bit Port-Nummern über 1024 (die Portnummern bis 1024 sind reserviert für wohlbekannte Dienste) vergeben werden. Der «*Welcome Socket*», dessen Port-Nummer die Clients kennen müssen um sich mit dem Server zu verbinden, kann so eine feste Port-Nummer verwenden, während die konkreten Verbindungen andere Ports benutzen.

Timeouts bei TCP Sockets

Bei TCP Sockets gibt es zwei unterschiedliche Arten von Timeouts, deren Eintritt durch eine `java.net.SocketTimeoutException` angezeigt wird.
Zum Einen kann ein Timeout wie bei UDP *beim Lesen vom Socket* auftreten: Der Aufruf von `read()` und ähnlichen Methoden, mit denen vom `InputStream` des `Socket`-Objekts gelesen wird, blockiert solange, bis neue Daten am Socket anliegen. Dieses Verhalten kann man durch Aufruf der Methode `setSoTimeout(int timeout)` beeinflussen. Damit kann die maximale Dauer gesetzt werden, die `read()`-Aufrufe blockieren.
Zum Anderen kann ein Timeout *beim Verbindungsaufbau* eintreten: Versucht man das `Socket`-Objekt des Clients durch den Aufruf von `connect(...)` mit dem Server zu verbinden, kann es clientseitig zu einem Timeout kommen. Die maximale Zeit, die der Aufbau der Verbindung dauern darf, kann als Parameter bei `connect(...)` angegeben werden.
Auch serverseitig kann ein Timeout *beim Verbindungsaufbau* eintreten: Dauert es nach dem Aufruf von `accept()` an einem `ServerSocket`-Objekt zu lange, bis sich ein Client verbindet, tritt ein Verbindungsaufbau-Timeout beim Server ein. Die Dauer bis zum Timeout wird jedoch im Gegensatz zum clientseitigen `connect(...)` nicht als Parameter der `accept()`-Methode übergeben, sondern muss am `ServerSocket`-Objekt mit der Methode `setSoTimeout(int timeout)` gesetzt werden. Inhaltlich wird man aber selten eine Anwendung finden, bei der solch ein serverseitiger Verbindungsaufbau-Timeout sinnvoll wäre.

Die Dauer bis zum Timeout wird in Java immer in Millisekunden angegeben. Falls dieser Wert `0` ist, werden Timeouts deaktiviert.

Backlog

Noch nicht verarbeitete Verbindungswünsche von Clients werden im *Backlog*-Speicher des `ServerSocket`-Objekts gehalten. Es geht dabei um Clients, die per `connect()` den «*Welcome Socket*» des Servers angesprochen haben, deren Verbindungswünsche aber serverseitig noch nicht von `accept()` bearbeitet wurden. Die Größe des Backlogs kann im Konstruktor von `ServerSocket` angegeben werden. Eine `java.net.ConnectException` wird erzeugt, falls das Backlog wegen zu vieler unbearbeiteter Client Verbindungswünsche überläuft.

Exkurs: TCP Handshake («Drei-Wege Handschlag»)

Der erfolgreiche Aufbau der TCP-Verbindung geschieht über den Austausch einer Sequenz von drei leeren (ohne Nutzlastdaten) Datenpaketen:

1. Der Client sendet als Verbindungsanfrage ein leeres TCP-Paket mit dem `SYN`-Bit im Header gesetzt. Das `seq`-Feld im Header hat einen beliebigen Startwert seq_1.

   Client → Server («Client-Verbindungsanfrage»)
   `SYN`-Bit `1`
   `ACK`-Bit `0`
   `seq`-Feld seq_1
   `ack`-Feld —

2. Der Server sendet als Antwort ein leeres TCP-Paket zurück. Darin sind `SYN`- und `ACK`-Bit gesetzt und das `ack`-Feld (ack_2) ist auf den Nachfolgewert des `seq`-Feldes des empfangenen Pakets ($ack_2 = seq_1 + 1$) gesetzt. Das `seq`-Feld des zurückgesendeten Pakets hat einen eigenen Startwert (seq_2).

   Server → Client («Server-Antwort»)
   `SYN`-Bit `1`
   `ACK`-Bit `1`
   `seq`-Feld seq_2
   `ack`-Feld $ack_2 = seq_1 + 1$

3. Der Client sendet zur Bestätigung ein leeres TCP-Paket mit gesetztem `ACK`-Bit, zum ersten Paket passendem `seq`-Feld Nachfolger ($seq_3 = ack_2 = seq_1 + 1$) und hat das `ack`-Feld mit dem Wert des `seq`-Feldes der Server-Antwort ($ack_3 = seq_2 + 1$) gesetzt.

   Client → Server («Client-Bestätigung»)
   `SYN`-Bit `0`
   `ACK`-Bit `1`
   `seq`-Feld $seq_3 = ack_2 = seq_1 + 1$
   `ack`-Feld $ack_3 = seq_2 + 1$

Praktikum

Aktivitätsschritte:

Unter `http://verteiltearchitekturen.de/vol01/VAR-TCP-solution.zip` können Sie eine Musterlösung zu diesem Praktikum herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-TCP_solution`.

Im folgenden Praktikum programmieren Sie einfache Client-/Server-Anwendungen, bzw. insbesondere die dafür erforderliche Netzwerkkommunikation in Java. Die Kom-

munikation findet direkt (also noch ohne die Verwendung von Middleware-Produkten oder von Frameworks wie in den nächsten Kapiteln) auf der Ebene des Transportprotokolls TCP statt.

Aktivitätsschritt 3.1:
Ziel: Projekt downloaden und in Eclipse importieren
Laden Sie das ZIP-Archiv mit dem Projekt `VAR-TCP` unter `http://verteiltearchitekturen.de/vol01/VAR-TCP.zip` herunter. Importieren Sie das ZIP-File in Eclipse mit der Funktion *File → Import... → General / Existing Projects Into Workspace*. Wählen Sie dort die Option *Select Archive File*. Sie sollten auch darauf achten, dass die Option *Copy Projects Into Workspace* aktiviert ist, sonst arbeiten Sie möglicherweise nur auf den Dateien in Ihrem Download-Ordner und nicht in Ihrem Eclipse-Workspace.

Als Ergebnis sollte das Projekt `VAR-TCP` in Ihrem Package Explorer in Eclipse auftauchen. Im `src`-Ordner dieses Projektes sollten Sie drei Packages sehen:

- `var.sockets.tcp.echo`
- `var.sockets.tcp.filer`
- `var.sockets.tcp.time`

In jedem der drei Pakete dieses Praktikums gibt es Klassen für einen Client und jeweils zwei unterschiedliche Server-Implementierungen zum jeweiligen Service.
In Ihrem Eclipse-Workspace-Directory sollte ein neues Verzeichnis für das Projekt entstanden sein. In dem Projektverzeichnis gibt es mindestens einen Folder namens `src`, in dem die Packages als (Unter-) Verzeichnisse auftauchen. Abb. 3.1 zeigt die Struktur unterhalb des Folders `src/`.
Die beiden Server-Implementierungen unterscheiden sich in den Packages `var.sockets.tcp.echo` und `var.sockets.tcp.filer` in der Art wie Clients bedient werden (iterativ oder parallel) bzw. im Package `var.sockets.tcp.time` verwenden die beiden Server unterschiedliche Repräsentationsformate, um denselben Inhalt an Clients zu schicken.

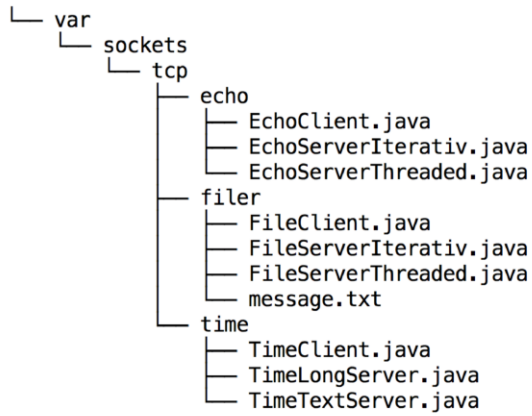


Abbildung 3.1: Verzeichnis- und Dateibaum für das Projekt `VAR-TCP`

3.2 Aufgabe: Echo Service

Der Echo Service liest zeilenweise Daten vom Client und sendet sie zeilenweise an den jeweiligen Client zurück (s. Abb. 3.2). Jeder Zeile wird dabei der `String` «`echo:␣`» vorangestellt. Der Server macht das solange, bis der Client die Verbindung beendet.

3.2.1 Client für den Echo Service

Die Implementierung eines funktionierenden Echo Service Clients ist bereits komplett vorgegeben (s. Listing 3.1).

Aktivitätsschritt 3.2:

Ziel: Analyse des Programmablaufs von `EchoClient`

Analysieren Sie die einzelnen Schritte des Programms in Listing 3.1. Was ist der Unterschied zwischen dem `readLine()`-Aufruf in Zeile 10 und dem in Zeile 14? Wann beendet der Client die Verbindung zum Server? Nehmen Sie zur Analyse des Ablaufs des Clients für den TCP *Echo Service* auch das Sequenzdiagramm in Abb. 3.3 zur Hilfe.

`in.readLine()` ist hier *synchron*. In diesem Zusammenhang bedeutet dies, dass der Aufruf solange blockiert, bis eine komplette Zeile am `InputStream` des `Socket`-Objekts vorliegt. Wie schon im UDP-Kapitel gibt es die Möglichkeit den

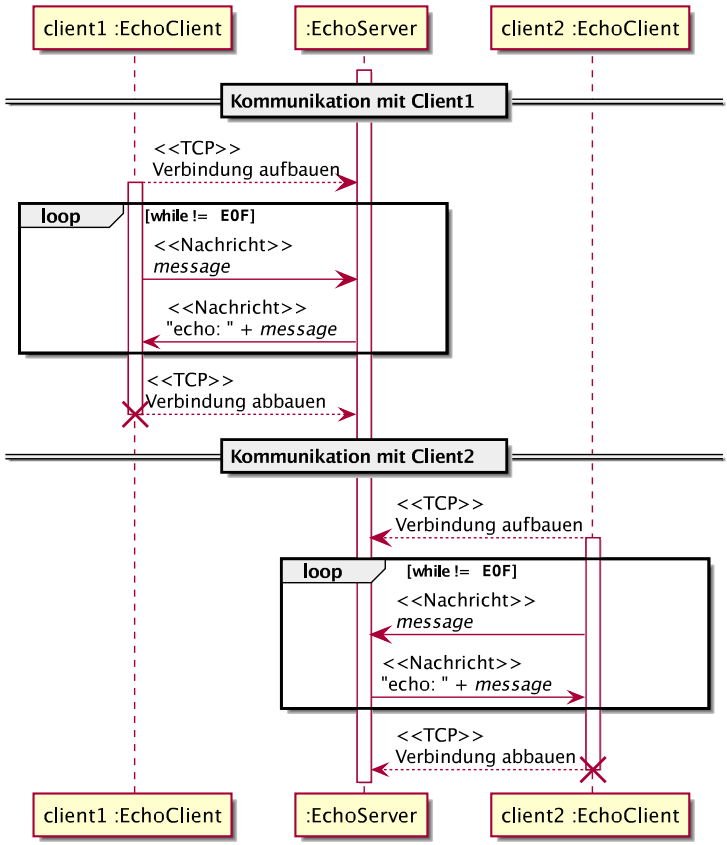


Abbildung 3.2: Prinzipielle Funktion des Echo Services (TCP-Version, iterativ)

Listing 3.1: Der vorgegebene `var.sockets.tcp.echo.EchoClient`

```
1  public class EchoClient {
2      public static void main(String[] args) {
3          String host = args[0];
4          int port = Integer.parseInt(args[1]);
5
6          try (Socket socket = new Socket(host, port);
7                  BufferedReader in = new BufferedReader(new
                       InputStreamReader(socket.getInputStream()));
8                  PrintWriter out = new PrintWriter(socket.
                       getOutputStream(), true);
9                  BufferedReader stdin = new BufferedReader(new
                       InputStreamReader(System.in))) {
10             String msg = in.readLine();
11             System.out.println(msg);
12             while (true) {
13                 System.out.print(">> ");
14                 String line = stdin.readLine();
15                 if ("q".equals(line)) {
16                     break;
17                 }
18                 out.println(line);
19                 System.out.println(in.readLine());
20             }
21         } catch (Exception e) {
22             System.err.println(e);
23         }
24     }
25 }
```

Socket mit einem Timeout zu konfigurieren, sodass das Blockieren beim Eintreten einer Zeitbedingung mit einer `SocketTimeoutException` beendet wird.

Aktivitätsschritt 3.3:

Ziel: Vergleich mit UDP-Version des `EchoClient`

Vergleichen Sie diesen ersten TCP Client mit einem UDP Client. Beim UDP Client wird die Zieladresse (*Host* + Port) im Datagramm, also der Nachricht, repräsentiert. Wie wird im Unterschied dazu der Empfänger einer Nachricht bei TCP angegeben? Erklären Sie sich, warum sich in diesem Programmierunterschied die Natur von verbindungsloser und verbindungsorientierter Kommunikation widerspiegelt.

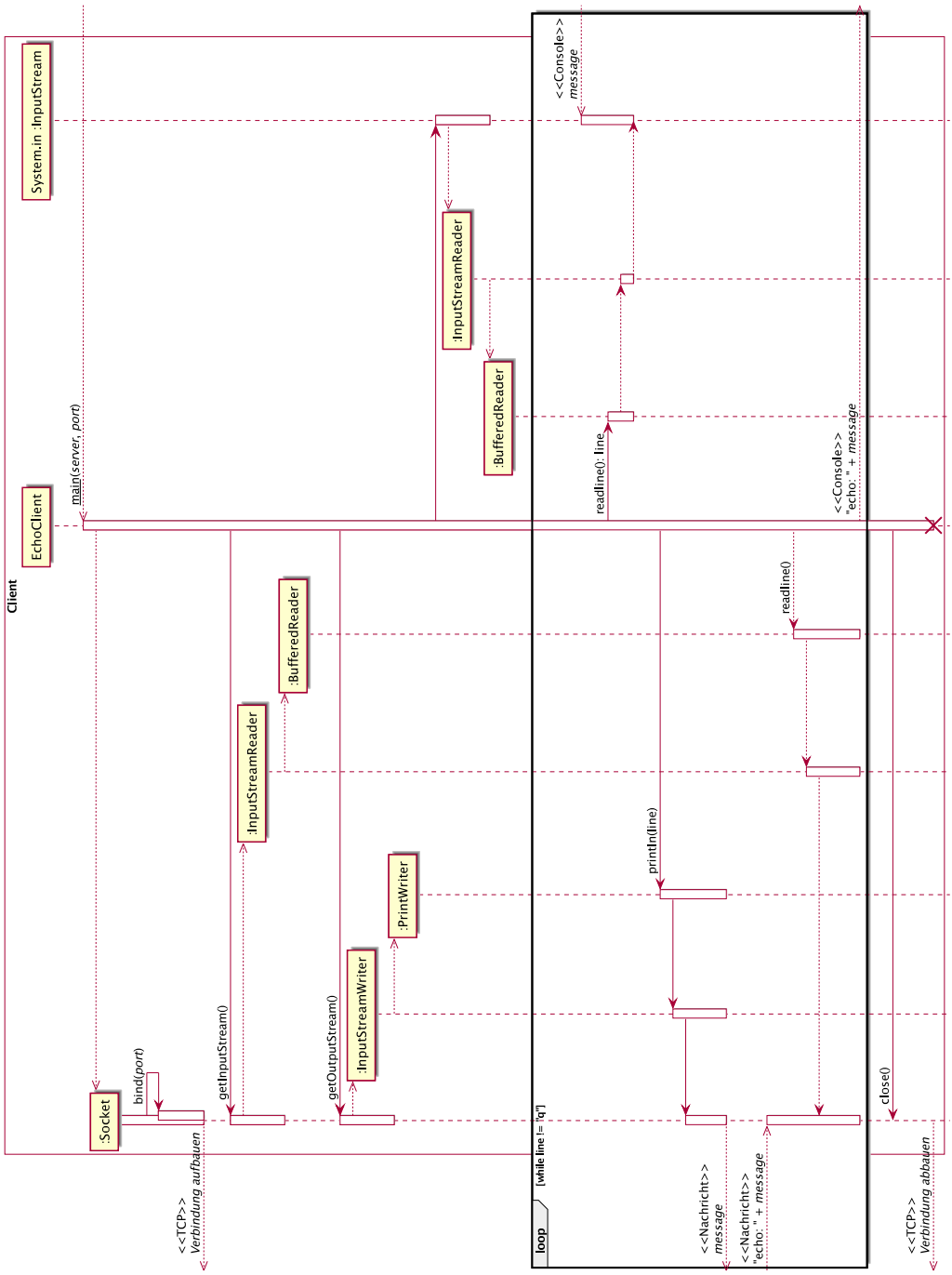


Abbildung 3.3: Sequenzdiagramm des Clients beim TCP *Echo Service* (vereinfacht)

3.2.2 Iterativer Server für den Echo Service

Die iterative Variante des Echo Services Servers `var.sockets.tcp.echo.EchoServerIterativ` ist ebenfalls bereits voll funktionsfähig im Eclipse-Projekt vorhanden (s. Listings 3.2 und 3.3). Der prinzipielle Ablauf ist in Abb. 3.4 dargestellt.

Aktivitätsschritt 3.4:
Ziel: Analyse des Programmablaufs von `EchoServerIterativ`
Untersuchen Sie zuerst wieder anhand des Quelltextes den Aufbau dieses Programms. Fangen Sie bei der `main(...)`-Methode an: Dort wird eine neue Instanz der Klasse erzeugt (der Konstruktor `EchoServerIterativ(...)` speichert nur die übergebenen Parameter in den Instanzvariablen). An diesem Objekt wird dann `start()` aufgerufen. Dort wird der «*Welcome Socket*» `serverSocket` erzeugt. Dieser Socket wird `handleClient(...)` übergeben, wo anfangs `accept()` aufgerufen wird.

`accept()` wird immer am «*Welcome Socket*», einer Instanz der Klasse `ServerSocket`, aufgerufen. Der Aufruf blockiert solange, bis ein Client sich mit diesem Socket verbindet. Daraufhin erzeugt `accept()` ein neues Socket-Objekt, das nun nicht mehr vom Typ `ServerSocket`, sondern vom Typ `Socket` ist. Dieses Socket hat eine andere (und nicht vorhersehbare) Port-Nummer als das «*Welcome Socket*», dessen Port-Nummer dem Client bekannt sein muss. Nachdem die `Socket`-Instanz erzeugt wurde, ist das «*Welcome Socket*» für den nächsten Client Verbindungswunsch frei und der Aufruf der Methode `accept()` endet.

Aktivitätsschritt 3.5:
Ziel: `EchoServerIterativ` laufen lassen und Verhalten analysieren
Untersuchen Sie das Laufzeitverhalten von `var.sockets.tcp.echo.EchoServerIterativ`. Lassen Sie dazu den Server laufen und starten Sie mehrere Clients, möglichst so, dass die Aufrufe sich überlappen und Sie die sequentielle Arbeitsweise des iterativen Servers analysieren können. Sie können wie in Abb. 3.5 zwischen der Konsole des Server-Prozesses und den Konsolen mehrerer Client-Prozesse hin- und herschalten.

Listing 3.2: Der vorgegebene `var.sockets.tcp.echo.EchoServerIterativ` ohne `main`-Methode (s. Listing 3.3)

```
1  public class EchoServerIterativ {
2      private int port;
3      private int backlog;
4
5      public EchoServerIterativ(int port, int backlog) {
6          this.port = port;
7          this.backlog = backlog;
8      }
9
10     public void start() {
11         try (ServerSocket serverSocket = new ServerSocket(port, backlog
               )) {
12             System.out.println("EchoServer (iterativ) auf " +
                   serverSocket.getLocalSocketAddress() + " gestartet ...
                   ");
13             while (true) {
14                 handleClient(serverSocket);
15             }
16         } catch (IOException e) {
17             System.err.println(e);
18         }
19     }
20
21     private void handleClient(ServerSocket server) {
22         SocketAddress socketAddress = null;
23         try (Socket socket = server.accept();
24                 BufferedReader in = new BufferedReader(new
                       InputStreamReader(socket.getInputStream()));
25                 PrintWriter out = new PrintWriter(socket.
                       getOutputStream(), true)) {
26             socketAddress = socket.getRemoteSocketAddress();
27             System.out.println("Verbindung zu " + socketAddress + "
                   aufgebaut");
28             out.println("Server ist bereit ...");
29             String input;
30             while ((input = in.readLine()) != null) {
31                 System.out.println(socketAddress + ">> [" + input + "]"
                       );
32                 out.println("echo: " + input);
33             }
34         } catch (IOException e) {
35             System.err.println(e);
36         } finally {
37             System.out.println("Verbindung zu " + socketAddress + "
                   abgebaut");
38         }
39     }
40 }
```

Listing 3.3: `main`-Methode der vorgegebenen Klasse `var.sockets.tcp.echo.EchoServerIterativ` (s. Listing 3.2)

```
1 public static void main(String[] args) {
2     int port = Integer.parseInt(args[0]);
3     int backlog = 50;
4     if (args.length == 2) {
5         backlog = Integer.parseInt(args[1]);
6     }
7     new EchoServerIterativ(port, backlog).start();
8 }
```

Bei iterativen Servern blockiert die Abarbeitung der Kommunikation mit einem Client den «*Welcome Socket*» für andere Clients, deren Verbindungswünsche im *backlog* gespeichert werden. Die Größe des *backlog*-Speichers kann beim Anlegen des `ServerSocket`-Objekts angegeben werden. Für die meisten Anwendungen ist dieses sequentielle Verhalten nicht akzeptabel, weil die Anzahl der Clients dann nur sehr klein sein kann und die Latenzzeit zu groß wird. Deshalb wird bei den meisten praktischen Netzanwendungen stattdessen mit *Threads* gearbeitet, die parallel mehrere Verbindungen abarbeiten können.

3.2.3 *Threaded* Server für den Echo Service

Im Gegensatz zum Ablauf, der in Abb. 3.2 dargestellt ist, sollte es in «real world»-Anwendungen möglich sein, dass ein Server mit mehreren Clients gleichzeitig Verbindungen unterhält und das jeweilige Anwendungsprotokoll zwischen Server und Client abarbeitet. Abb. 3.6 soll dagegen veranschaulichen, dass beide Clients gleichzeitig vom Server bedient werden können. Um diese Art der Nebenläufigkeit in Java zu realisieren, werden *Threads* benötigt.

Für die Variante des Echo Service Servers mit Threads `var.sockets.tcp.echo.EchoServerThreaded`, die die Anforderung nach der gleichzeitigen Verbindung zu mehren Clients erfüllt, ist noch keine funktionsfähige Version im Eclipse-Projekt vorhanden. Listing 3.4 zeigt den Rumpf eines TCP Servers mit Threads. Im Prinzip ist dieses Gerüst unabhängig von der eigentlichen Funktion des Servers und kann auch für andere Services weiterverwendet werden.

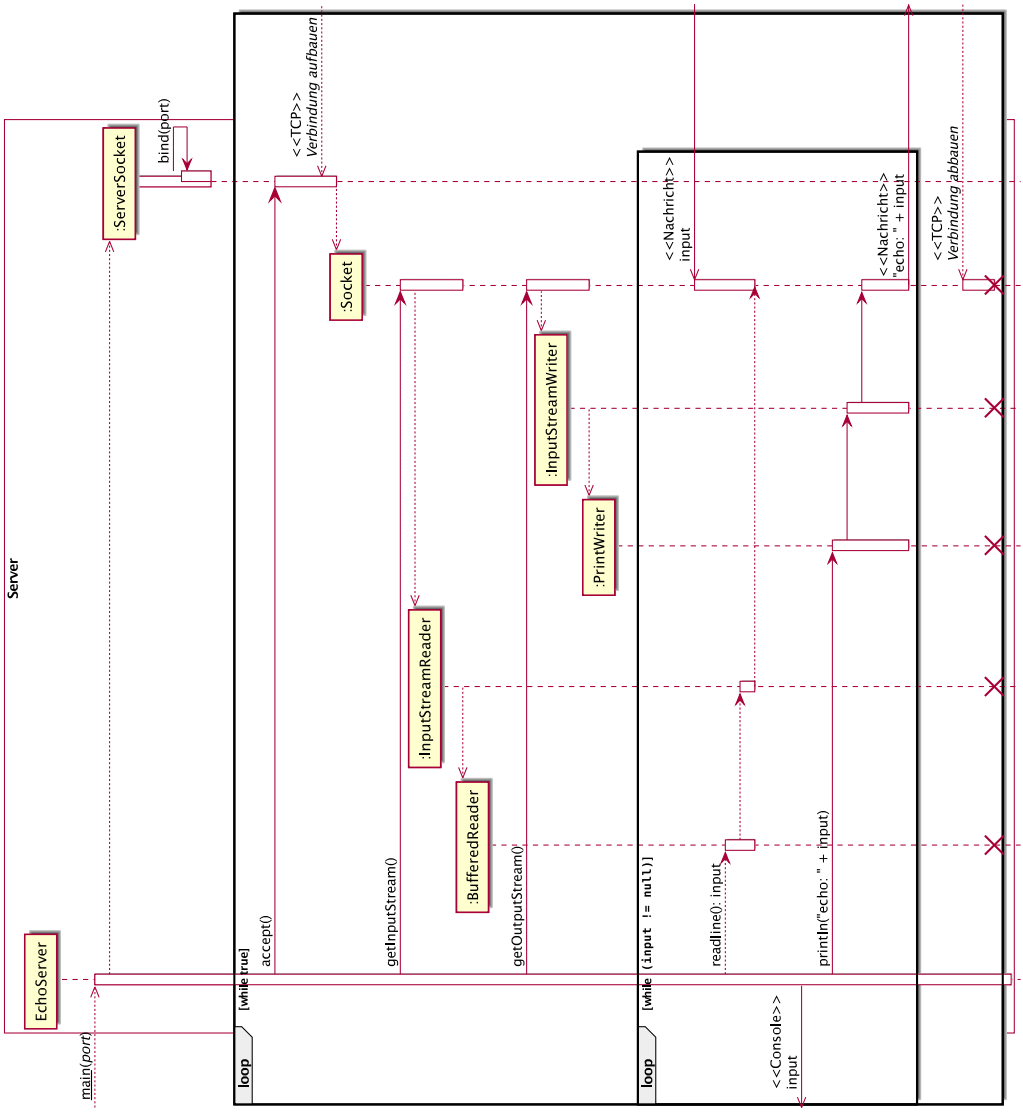


Abbildung 3.4: Sequenzdiagramm des iterativen TCP Servers des Echo Services (vereinfacht)

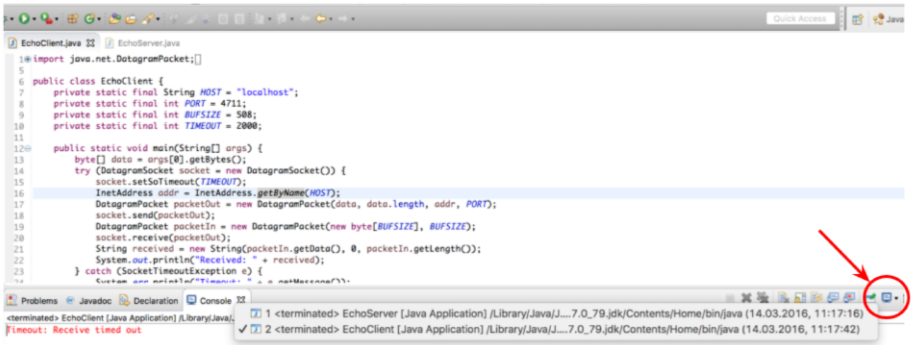


Abbildung 3.5: Screenshot aus Eclipse: Wechsel zwischen mehreren Prozessen

Aktivitätsschritt 3.6:

Ziel: Programmablauf von `EchoServerThreaded` entwickeln

Implementieren Sie das Folgende in `start(...)` an der Stelle des Zeilenkommentars:

- Es wird an dieser Stelle eine Endlosschleife benötigt, die fortwährend Verbindungen entgegennimmt und `EchoServer`-Objekte für die Verbindungen erzeugt und zur nebenläufigen Ausführung startet.
- Innerhalb der Schleife muss am `ServerSocket`-Objekt, das in der lokalen Variable `serverSocket` verfügbar ist, `accept()` aufgerufen werden, was solange blockiert, bis ein neuer Verbindungswunsch eines Clients vorliegt. Das Ergebnis von `accept()` ist ein neues `Socket`-Objekt.
- Wenn eine neue Verbindung vorliegt (`accept()` ist fertig abgearbeitet), muss eine neue Instanz von `EchoThread` mit dessen Konstruktor erzeugt werden, dem das Ergebnis von `accept()` als Parameter übergeben wird.
- Da diese `EchoThread`-Instanz ein `Thread`-Objekt ist, muss man es noch nebenläufig laufen lassen, indem die `start()`-Methode aufgerufen wird.

Die innere Klasse (in diesem Fall `EchoThread`) realisiert in ihrer `run()`-Methode die spezifische Funktion des Services. Dabei ist in diesem Gerüst vorgesehen, dass in `run()` über die Instanzvariable `socket` auf die aktuelle Verbindung zugegriffen werden kann.

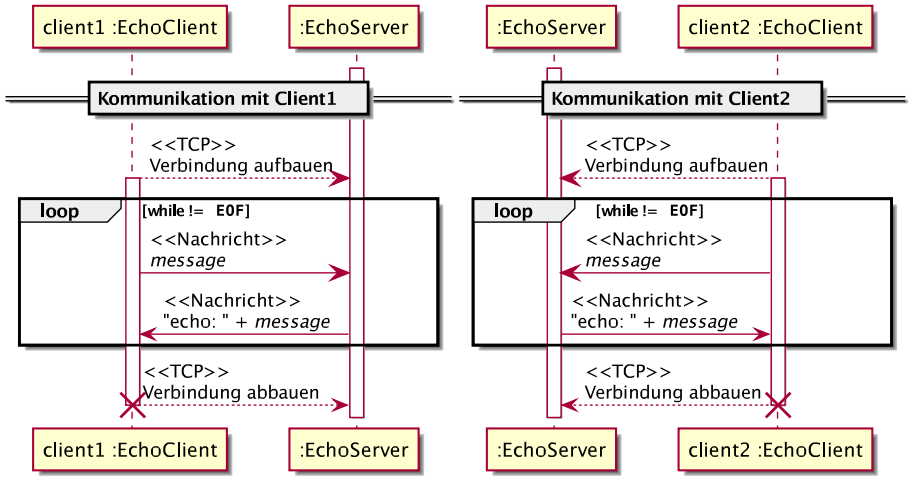


Abbildung 3.6: Prinzipielle Funktion des Echo Services (TCP-Version, threaded)

Aktivitätsschritt 3.7:

Ziel: Funktion für `EchoServerThreaded` aus iterativem Server kopieren

Implementieren Sie die Funktionalität des Echo Services in der `run()`-Methode der inneren Klasse an der Stelle des Zeilenkommentars. Kopieren Sie dazu alles, was dafür erforderlich ist, aus der Implementierung des Verhaltens des iterativen Servers. Alles, was dazu erforderlich ist, steckt in der Methode `handleClient(ServerSocket server)`.

Achtung: In `run()` steht das `Socket`-Objekt schon in der Instanzvariable `socket` der inneren Klasse zur Verfügung. Der Aufruf von `server.accept()`, der in `handleClient(ServerSocket server)` enthalten ist, ist deshalb in `run()` nicht mehr erforderlich.

Listing 3.4: `var.sockets.tcp.echo.EchoServerThreaded`: der vorgegebene Code zur Implementierung des Servers

```
1  public class EchoServerThreaded {
2      private int port;
3
4      public EchoServerThreaded(int port) {
5          this.port = port;
6      }
7
8      public void start() {
9          try (ServerSocket serverSocket = new ServerSocket(port)) {
10             System.out.println("EchoServer (threaded) auf " +
                   serverSocket.getLocalSocketAddress() + " gestartet ...
                   ");
11             // hier müssen Verbindungswünsche von Clients in
12             // einem neuen Thread angenommen werden
13         } catch (IOException e) {
14             System.err.println(e);
15         }
16     }
17
18     private class EchoThread extends Thread {
19         private Socket socket;
20
21         public EchoThread(Socket socket) {
22             this.socket = socket;
23         }
24
25         public void run() {
26             // hier muss die Verbindung mit dem Client über this.socket
                   abgearbeitet werden
27         }
28     }
29
30     public static void main(String[] args) {
31         int port = Integer.parseInt(args[0]);
32         new EchoServerThreaded(port).start();
33     }
34 }
```

Exkurs: Threads

Objekte, die von `java.lang.Thread` erben, können parallel zum Hauptprogramm abgearbeitet werden. Die Einstiegsmethode (vergleichbar mit `public static void main(String[] args)`) heißt `public void run()`. Sie muss in Unterklassen von `Thread` überschrieben werden.
Die zum bisherigen Programmablauf parallele Abarbeitung der `run()`-Methode einer `Thread`-Instanz beginnt, wenn an der Instanz die Methode `start()` aufgerufen wird.
Wird der `Thread` unterbrochen, wird eine `InterruptedException` geworfen, die behandelt werden muss. Mit der Klassenmethode `sleep` wird die Ausführung für eine Zeitdauer unterbrochen. Die Methode erwartet die Dauer als Parameter in Millisekunden.

Aktivitätsschritt 3.8:

Ziel: `EchoServerThreaded` laufen lassen und Verhalten analysieren

Untersuchen Sie wieder das Laufzeitverhalten des Servers. Lassen Sie diesmal die Klasse `var.sockets.tcp.echo.EchoServerThreaded` laufen und starten Sie mehre Clients möglichst so, dass die Aufrufe sich überlappen und Sie die nebenläufige Arbeitsweise des *threaded* Servers analysieren können. Sie können wieder wie in Abb. 3.5 zwischen der Konsole des Server-Prozesses und den Konsolen mehrerer Client-Prozesse hin- und herschalten.

Machen Sie sich die unterschiedliche Arbeitsweise von iterativer und Thread-basierter Server-Variante klar. UDP Server brauchen dieses Vorgehen nicht, weil UDP nicht verbindungsorientiert ist. Stattdessen reicht dort ein einziges Socket beim Server, über das Datagramme von allen Clients empfangen werden können. Sollte die Verarbeitung eines einzelnen Datagramms jedoch zeitintensiv sein, wäre es auch bei UDP Servern denkbar, diese Datagramm-Behandlung nebenläufig in Threads auszuführen, um währenddessen das nächste Datagramm vom UDP Server Socket entgegenzunehmen.

3.3 Aufgabe: File Service

In diesem Abschnitt sollen die Konzepte, die im Echo Server angewandt wurden, extrahiert und auf einen etwas anderen Dienst angepasst übertragen werden. Dieser Service soll jedem Client eine fest vorgegebene Datei zurücksenden. Der Client verbindet sich dazu mit dem Server, dann beginnt der Server sofort diese Datei Zeile für Zeile an den Client zu senden. Ist die Datei komplett übertragen, schließt der Server die Verbindung zum Client. Im Vergleich zum Echo Service Server ist der Dialogablauf und die Initiative, die Verbindung zu beenden, also anders geregelt.

3.3.1 Client für den File Service

Im Projekt ist in der Klasse `var.sockets.tcp.filer.FileClient` (s. Listing 3.5) ein Client zu dem beschriebenen File Service vorhanden. Er soll als Richtlinie für die Programmierung der Server-Funktionalität dienen: Im Bereich verteilter Systeme wird oft eine Implementierung von Client oder Server als implizite Spezifikation eines Kommunikationsprotokolls verwendet. Damit lässt sich auch die eigene Implementierung während der Entwicklung testen.

Listing 3.5: Der vorgegebene `var.sockets.tcp.filer.FileClient`

```
1  public class FileClient {
2      public static void main(String[] args) {
3          String host = args[0];
4          int port = Integer.parseInt(args[1]);
5
6          try (Socket socket = new Socket(host, port);
7                  BufferedReader in = new BufferedReader(new
                       InputStreamReader(socket.getInputStream()))) {
8              String line;
9              while ((line = in.readLine()) != null) {
10                 System.out.println(line);
11             }
12         } catch (Exception e) {
13             System.err.println(e);
14         }
15     }
16 }
```

Aktivitätsschritt 3.9:

Ziel: Vergleich der Clients für File Service und Echo Service mit Eclipse

Strukturell gleicht der vorgegebene File Service Client dem `EchoClient` (s. Listing 3.1) für den Echo Service. Vergleichen Sie die beiden Client-Klassen und suchen Sie Unterschiede und Gemeinsamkeiten heraus. Sie können dazu in Eclipse die Funktion *Compare With → Each Other* aus dem Kontextmenü («rechte Maustaste») zweier markierter Dateien verwenden (s. Abb. 3.7). Das Ergebnis könnte wie in Abb. 3.8 aussehen.

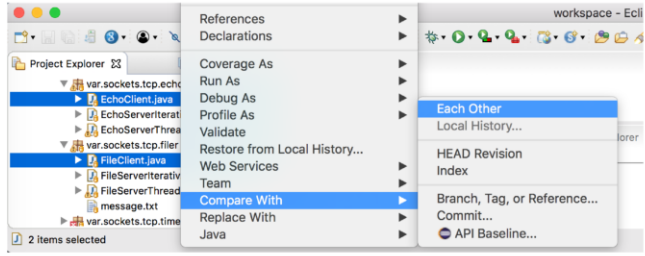


Abbildung 3.7: Screenshot aus Eclipse: Vergleich zweier Klassen

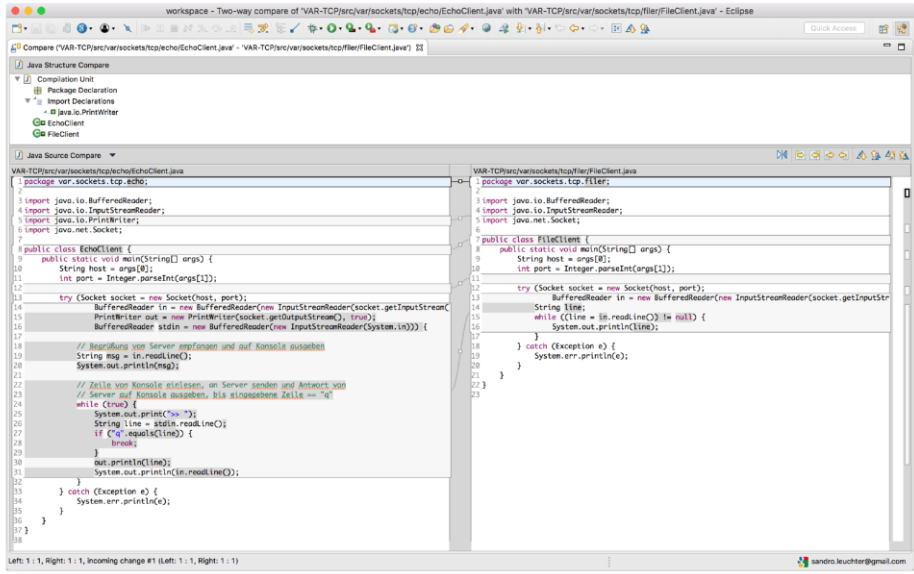


Abbildung 3.8: Screenshot aus Eclipse: Unterschiede zwischen zwei Klassen

3.3.2 Iterativer Server für den File Service

Der iterative File Service Server ist im Eclipse-Projekt als Gerüst vorhanden (s. Listings 3.6 und 3.7). Die eigentliche Funktionalität des Dienstes fehlt noch, strukturell arbeitet dieser Server aber korrekt.

Listing 3.6: Die vorgegebene Klasse für den iterativen Server zum File Service: `var.sockets.tcp.filer.FileServerIterativ` ohne `main`-Methode (s. Listing 3.6)

```
1  public class FileServerIterativ {
2      private static final String FILE = "src/var/sockets/tcp/filer/
           message.txt";
3      private int port;
4      private int backlog;
5
6      public FileServerIterativ(int port, int backlog) {
7          this.port = port;
8          this.backlog = backlog;
9      }
10
11     public void start() {
12         try (ServerSocket serverSocket = new ServerSocket(port, backlog
               )) {
13             System.out.println("FileServer (iterativ) auf " +
                   serverSocket.getLocalSocketAddress() + " gestartet ...
                   ");
14             File file = new File(FILE);
15             if (file.exists()) {
16                 System.out.println("\"" + file.getAbsolutePath() + "\"
                       soll gesendet werden.");
17                 while (true) {
18                     handleClient(serverSocket);
19                 }
20             }
21         } catch (IOException e) {
22             System.err.println(e);
23         }
24     }
25
26     private void handleClient(ServerSocket server) {
27         SocketAddress socketAddress = null;
28         try (Socket socket = server.accept();
29                 BufferedReader in = new BufferedReader(new FileReader(
                       FILE));
30                 PrintWriter out = new PrintWriter(socket.
                       getOutputStream(), true)) {
31             socketAddress = socket.getRemoteSocketAddress();
32             System.out.println("Verbindung zu  " + socketAddress + "
                   aufgebaut");
33             // Inhalt von in zeilenweise an out senden
34         } catch (IOException e) {
35             System.err.println(e);
36         } finally {
37             System.out.println("Verbindung zu  " + socketAddress + "
                   abgebaut");
38         }
39     }
40 }
```

Listing 3.7: `main`-Methode der vorgegebenen Klasse für den iterativen Server zum File Service: `var.sockets.tcp.filer.FileServerIterativ` (s. Listing 3.6)

```
1 public static void main(String[] args) {
2     int port = Integer.parseInt(args[0]);
3     int backlog = 50;
4     if (args.length == 2) {
5         backlog = Integer.parseInt(args[1]);
6     }
7
8     new FileServerIterativ(port, backlog).start();
9 }
```

Aktivitätsschritt 3.10:

Ziel: Vergleich der iterativen Server für File und Echo Service mit Eclipse

Vergleichen Sie in Eclipse wieder mit der eingebauten Funktion *Compare With → Each Other* den Quelltext der iterativen Server für den Echo-Dienst `var.sockets.tcp.echo.EchoServerIterativ` und `var.sockets.tcp.filer.FileServerIterativ` miteinander. Es gibt hier einige Unterschiede, die sich auf die vorgegebene Datei beziehen, die an die Clients ausgegeben werden soll. Diese Datei ist bereits im Projekt vorhanden.

Aktivitätsschritt 3.11:

Ziel: Funktionalität für `FileServerIterativ` ausarbeiten

Implementieren Sie an der Stelle des Zeilenkommentars in `handleClient(ServerSocket server)` die Funktionalität, mit der der Server zeilenweise den Inhalt aus der vorgegebenen Datei (`FILE`) liest und in den Socket schreibt, damit der Dateiinhalt an den Client übertragen wird. Die Datei (`FILE`) ist bereits geöffnet: Aus dem `BufferedReader in` kann wieder wie beim `EchoServerIterativ` mit `readline()` gelesen werden.

Aktivitätsschritt 3.12:

Ziel: Verhalten von `FileServerIterativ` ggü. Clients beobachten

Untersuchen Sie wieder das Laufzeitverhalten von `var.sockets.tcp.filer.FileServerIterativ` mit dem vorgegebenen Client. Lassen Sie dazu den Server laufen und starten Sie mehrere Clients, möglichst so, dass die Aufrufe

sich überlappen und Sie die iterative Arbeitsweise des Servers analysieren können. Im Gegensatz zum Test des Echo-Dienstes ist das nun schwieriger, da die Verbindung nur so lange besteht, wie der Server braucht um die vorgegebene Datei an den Client zu senden.

Tipp: Verlangsamen Sie die Kommunikation des Servers, indem Sie nach der Ausgabe jeder Zeile der Datei eine Sekunde warten:

```
Thread.sleep(1000);
```

Allerdings kann hierbei eine `InterruptedException` auftreten, die noch im umschließenden `try`/`catch` behandelt werden muss.

3.3.3 *Threaded* Server für den File Service

Zum File Service gibt es im Eclipse-Projekt wieder eine Variante, die Threads benutzt, um Verbindungen mit mehreren Clients gleichzeitig verwenden zu können. In Listing 3.8 ist wieder ein Gerüst vorgegeben, das im Vergleich zum *threaded* Server für den Echo-Dienst um einige Zeilen zum Handling der auszuliefernden Datei erweitert wurde.

Aktivitätsschritt 3.13:
Ziel: Threaded Server für File und Echo Service mit Eclipse vergleichen
Vergleichen Sie den `Thread`-basierten File-Dienst Server `FileServerThreaded` mit dem für den Echo Service `EchoServerThreaded` mit der Funktion *Compare With* → *Each Other* von Eclipse.

Neben den unterschiedlichen Namen findet sich zusätzlich wieder die Konstante `FILE` wie schon in der iterativen Variante des Servers. Zusätzlich ist alles dafür vorbereitet, dass Verbindungen von Clients nur akzeptiert werden, wenn die Datei, die ausgeliefert werden soll, auch existiert.

Listing 3.8: Die vorgegebene Implementierung des Gerüsts für einen *threaded* Server des File Service: `var.sockets.tcp.filer.FileServerThreaded`

```
1  public class FileServerThreaded {
2      private static final String FILE = "src/var/sockets/tcp/filer/
           message.txt";
3      private int port;
4
5      public FileServerThreaded(int port) {
6          this.port = port;
7      }
8
9      public void start() {
10         try (ServerSocket serverSocket = new ServerSocket(port)) {
11             System.out.println("FileServer (threaded) auf " +
                   serverSocket.getLocalSocketAddress() + " gestartet ...
                   ");
12             File file = new File(FILE);
13             if (file.exists()) {
14                 System.out.println("\"" + file.getAbsolutePath() + "\"
                       soll gesendet werden.");
15                 while (true) {
16                     // hier müssen Verbindungswünsche von Clients in
17                     // einem neuen Thread angenommen werden
18                 }
19             }
20         } catch (IOException e) {
21             System.err.println(e);
22         }
23     }
24
25     private class FileThread extends Thread {
26         private Socket socket;
27
28         public FileThread(Socket socket) {
29             this.socket = socket;
30         }
31
32         public void run() {
33             // hier muss die Verbindung mit dem Client über this.socket
34             // abgearbeitet werden
35         }
36     }
37
38     public static void main(String[] args) {
39         int port = Integer.parseInt(args[0]);
40         new FileServerThreaded(port).start();
41     }
42 }
```

Aktivitätsschritt 3.14:
Ziel: `FileServerThreaded`: Clients nebenläufig bedienen können
Vervollständigen Sie das *threaded* Verhalten des Servers `FileServerThreaded`, sodass er Verbindungswünsche von Clients annimmt und nebenläufige Threads für die Verbindungen startet.

Aktivitätsschritt 3.15:
Ziel: Code zur Behandlung von Clients aus `FileServerIterativ` kopieren
Implementieren Sie die Funktion des Servers, indem Sie aus der iterativen Server-Variante den Inhalt aus `handleClient(ServerSocket server)` an die entsprechende Stelle des `FileServerThreaded` übernehmen.

Aktivitätsschritt 3.16:
Ziel: `FileServerThreaded` mit Client testen
Testen Sie `FileServerThreaded` mit dem Client. Lassen Sie dazu den Server laufen und starten Sie mehrere Clients, möglichst so, dass die Aufrufe sich überlappen und Sie die nebenläufige Arbeitsweise des *threaded* Servers analysieren können. Im Gegensatz zum Test des Echo-Dienstes ist das nun schwieriger, da die Verbindung nur so lange besteht, wie der Server braucht um die vorgegebene Datei an den Client zu senden.

Tipp: Verlangsamen Sie die Kommunikation des Servers, indem Sie nach der Ausgabe jeder Zeile der Datei eine Sekunde warten:

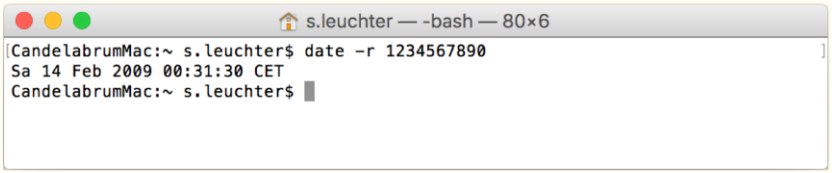
```
Thread.sleep(1000);
```

Allerdings kann hierbei eine `InterruptedException` auftreten, die noch im umschließenden `try/catch` behandelt werden muss.

Mit der `Thread`-basierten Variante des Servers können mehrere Clients gleichzeitig bedient werden. Die Auslieferung der Datei an mehrere Clients ist untereinander unabhängig.

Exkurs: UNIX-Zeit: ms seit Beginn der «UNIX-Epoche»

Zeitstempel werden von Betriebssystemen an vielen Stellen benötigt. Beispielsweise wird der Zeitpunkt der letzten Änderung einer Datei in den meisten Filesystemen gespeichert. Im Betriebssystem UNIX werden seit der Version 6 Zeitstempel als Zahl von Millisekunden seit dem 1. Januar 1970 00:00:00 GMT, dem sog. Beginn der «UNIX-Epoche», repräsentiert. Diese Konvention wird als Teil des internationalen POSIX-Standards IEEE 1003.1:2016 für Betriebssystemschnittstellen in vielen aktuellen IT-Systemen verwendet.



Das Kommando `date` unter dem Betriebssystem macOS 11.11.06 auf einem MacBook Pro zeigt, dass am 14.02.2009 00:31:30 CET genau 1234567890 Sekunden seit dem Beginn der UNIX-Epoche vergangen waren.
Auch Java verwendet diese Repräsentation von Zeitstempeln bei der Klasse `java.util.Date`. Das ist der Grund, warum `Date` den Konstruktor `Date(long date)` und die Methode `getTime()` hat. Auch die Methode `System.currentTimeMillis()` liefert dasselbe Format.

3.4 Aufgabe: Time Service

Die Funktion des Time Service ist einem Client, der sich mit dem Server verbindet, sofort Uhrzeit und Datum (zusammen der «Zeitstempel»), die aus der Sicht des Servers gerade aktuell sind, zurückzusenden. Danach schließt der Server die Verbindung. Das Package des Time Service im Eclipse-Projekt hat wieder zwei Server-Implementierungen. Allerdings unterscheiden sie sich diesmal nicht in der Art, wie sie Clients iterativ oder nebenläufig bedienen, sondern beide Server sind iterativ umgesetzt. Aber der eine Server sendet den Zeitstempel in Form eines menschenlesbaren Textes zurück, der andere Server sendet die Uhrzeit in einer numerischen Repräsentation in Millisekunden seit dem 01.01.1970 00:00:00 GMT.

Listing 3.9: Die vorgegebene Implementierung des Time Service für Zeitstempel in ms: `var.sockets.tcp.time.TimeLongServer`

```
1  public class TimeLongServer {
2      private int port;
3
4      public TimeLongServer(int port) {
5          this.port = port;
6      }
7
8      public void startServer() {
9          try (ServerSocket serverSocket = new ServerSocket(port)) {
10             while (true) {
11                 try (Socket socket = serverSocket.accept();
12                         PrintWriter out = new PrintWriter(socket.
                               getOutputStream())) {
13                     Date now = new Date();
14                     long currentTime = // Zeit von now in ms seit
                           01.01.1970 00:00:00 GMT abrufen
15                     out.print(currentTime);
16                     out.flush();
17                 } catch (IOException e) {
18                     System.err.println(e);
19                 }
20             }
21         } catch (IOException e) {
22             System.err.println(e);
23         }
24     }
25
26     public static void main(String[] args) {
27         new TimeLongServer(Integer.parseInt(args[0])).startServer();
28     }
29 }
```

3.4.1 Time Service: iterativer Server für Zeitstempel in ms

Aktivitätsschritt 3.17:

Ziel: `java.util.Date` recherchieren

Recherchieren Sie, wie aus einem `java.util.Date`-Objekt in Java der repräsentierte Zeitstempel als `long` abgerufen werden kann, und korrigieren Sie den Quellcode des `TimeLongServer` aus dem Eclipse-Projekt (s. Listing 3.9) an der entsprechenden Stelle unter Verwendung des aktuellen Zeitstempels in der `now`.

Aktivitätsschritt 3.18 (Unix (BSD, Linux, macOS)):
Ziel: `telnet` zum Test von TCP Servern

Auf den Unix-artigen Betriebssystemen (BSD, Linux, macOS) können Sie Ihren Server mit dem Kommando `telnet` prüfen: Starten Sie Ihren Server z. B. auf Port `4723` und starten Sie in einer Shell («Konsole») das Kommando:

```
telnet localhost 4723
```

Abb. 3.9 zeigt das Resultat: Der Server liefert die aktuelle Zeit in dem oben beschriebenen Format `1491057754193` zurück und schließt augenblicklich die Verbindung. Hier kann man auch beobachten, dass diesmal die Antwort des Servers nicht als komplette Zeile wie in den Beispielen bisher gesendet wird, da am Ende kein Zeilenumbruch mitgeschickt wird.

Das Programm `netcat`, das in den meisten Umgebungen nicht standardmäßig installiert ist, kann in ähnlicher Weise verwendet werden wie `telnet`

```
nc localhost 4723
```

Unter Windows kann auch das Programm *putty* installiert werden, das von einer grafischen Benutzungsschnittstelle aus wie das `telnet`-Kommando benutzt werden kann.

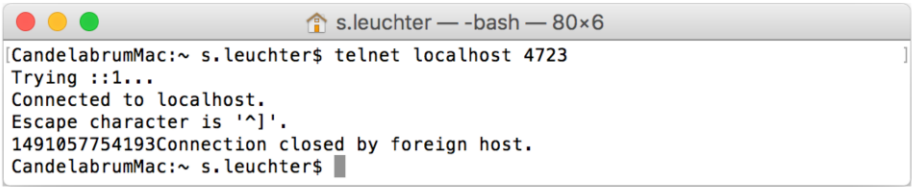


Abbildung 3.9: `telnet`-Verbindung zum laufenden `TimeLongServer`

3.4.2 Time Service: iterativer Server für Zeitstempel in lokalisiertem Textformat

Aktivitätsschritt 3.19:
Ziel: `DateFormat` recherchieren und `TimeTextServer` reparieren
Recherchieren Sie, wie aus einem `java.util.Date`-Objekt in Java der repräsentierte Zeitstempel mit `DateFormat`-Instanz gemäß den lokalen Gepflogenheiten formatiert werden kann, und korrigieren Sie den Quellcode des `TimeTextServer` aus dem Eclipse-Projekt (s. Listing 3.10) an der entsprechenden Stelle unter Verwendung des aktuellen Zeitstempels in der `now`.

Aktivitätsschritt 3.20 (Unix (BSD, Linux, macOS)):
Ziel: `TimeTextServer` mit `telnet` testen
Auf den Unix-artigen Betriebssystemen (BSD, Linux, macOS) können Sie Ihren Server wieder mit dem Kommando `telnet` prüfen: Starten Sie Ihren Server z. B. auf Port `4723` und starten Sie in einer Shell das Kommando:

```
telnet localhost 4723
```

Abb. 3.10 zeigt das Resultat: Der Server liefert die aktuelle Zeit in dem oben beschriebenen Format `01.04.17 17:05` zurück und schließt augenblicklich die Verbindung. Wie schon beim `TimeLongServer` kann man an der Ausgabe von `telnet` sehen, dass die Antwort des Servers nicht als komplette Zeile, sondern ohne Vorschubzeichen am Ende gesendet wird.

Listing 3.10: Die vorgegebene Implementierung des Time Service für Zeitstempel in lokalisiertem Textformat: `var.sockets.tcp.time.TimeTextServer`

```
1  public class TimeTextServer {
2      private int port;
3
4      public TimeTextServer(int port) {
5          this.port = port;
6      }
7
8      public void startServer() {
9          try (ServerSocket serverSocket = new ServerSocket(port)) {
10             while (true) {
11                 try (Socket socket = serverSocket.accept();
12                         PrintWriter out = new PrintWriter(socket.
                               getOutputStream())) {
13                     Date now = new Date();
14                     String currentTime = // DateFormat Instanz holen
                           und mit dessen format Methode now zum String
                           machen
15                     out.print(currentTime);
16                     out.flush();
17                 } catch (IOException e) {
18                     System.err.println(e);
19                 }
20             }
21         } catch (IOException e) {
22             System.err.println(e);
23         }
24     }
25
26     public static void main(String[] args) {
27         new TimeTextServer(Integer.parseInt(args[0])).startServer();
28     }
29 }
```

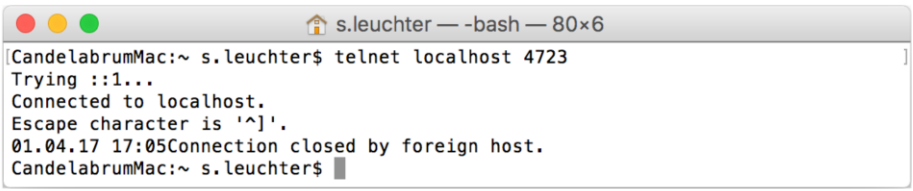


Abbildung 3.10: `telnet`-Verbindung zum laufenden `TimeTextServer`

Listing 3.11: Das vorgegebene Gerüst zur Implementierung des Clients für den Time Service: `var.sockets.tcp.time.TimeClient`

```
1  public class TimeClient {
2      public static void main(String[] args) {
3          try (Socket socket = new Socket(args[0], Integer.parseInt(args
               [1]));
4                  InputStream in = socket.getInputStream()) {
5              StringBuilder stringBuilder = new StringBuilder();
6              int c;
7              while ((c = in.read()) != -1) {
8                  stringBuilder.append((char) c);
9              }
10             // stringBuilder-Inhalt in ein Date-Objekt konvertieren und
                   ausgeben
11         } catch (Exception e) {
12             System.err.println(e);
13         }
14     }
15 }
```

3.4.3 Client für den Time Service

Im Eclipse-Projekt ist das Gerüst für einen Client zum Time Service enthalten (s. Listing 3.11).

Aktivitätsschritt 3.21:
Ziel: `TimeClient` Quelltext analysieren
Analysieren Sie den Ablauf im Client-Gerüst. In der Variable `stringBuilder` vom Typ `StringBuilder` wird die Antwort des Servers Zeichen für Zeichen hinzugefügt. Wenn `in.read()` `-1` liefert, heißt das, dass der Socket vom Server geschlossen wurde. `stringBuilder` enthält nun die komplette Antwort des Servers.

Aktivitätsschritt 3.22:
Ziel: `TimeClient` erweitern um Antwort zu analysieren
Zu diesem Zeitpunkt ist allerdings nicht bekannt, ob die Antwort von einem `TimeLongServer` oder von einem `TimeTextServer` empfangen wurde. Das Format der Antwort ist also unklar. Den Inhalt des `stringBuilder`-Objekts kann man mit der `toString()` abrufen. Ergänzen Sie den Code von

`TimeClient` an der Stelle des Zeilenkommentars: Inspizieren Sie die Antwort des Servers, finden Sie das Format heraus und erzeugen Sie daraus ein `Date`-Objekt, das Sie auf der Konsole ausgeben.

Tipp: Viele Methoden, die als Eingabe ein bestimmtes Format erwarten, werfen eine `Exception`, wenn die Eingabe nicht dem erwarteten Format entspricht.

3.5 Ausblick und Anregungen für eigene Projekte

3.5.1 *Thread Pool*: Skalierung der Anzahl der Clients

Auch wenn die *threaded* Server mit mehreren Clients durch interne Kontextwechsel zwischen den aktiven Threads quasi gleichzeitig umgehen können, profitieren solche Systeme von den Parallelverabeitungsmöglichkeiten moderner CPUs. Jeder Core einer CPU kann unabhängig von den anderen rechnen und im Prinzip auch darauf einen Java-Thread ausführen. Sie teilen aber den Zugriff auf den Speicher und andere Ressourcen.
Die Anzahl der sinnvollerweise gleichzeitig laufenden Threads (mit `start()` nebenläufig gestartet) ist von den Parallelverabeitungsressourcen (z. B. Anzahl Cores, *Hyperthreading*) des Servers, seiner Auslastung durch andere neben der Java Virtual Machine (JVM) laufende Prozesse und dem konkreten Ressourcennutzungsprofil der Threads (z. B. wegen gegenseitig blockierendem Zugriff auf geteilte Mittel wie I/O) abhängig. Bei zu vielen Threads (im Verhältnis zur Anzahl der Cores, zu den sonst abzuarbeitenden Prozessen und zum Ressourcennutzungsprofil der Threads) wird jedoch durch die dann erforderlichen Kontextwechsel zwischen den Threads die Performance leiden und relativ mehr Rechenzeit für die Verwaltung als für die Abarbeitung von Aufgaben anfallen.
Zudem erfordert die Instanziierung eines neuen `Thread`-Objekts einen gewissen Aufwand: Speicher muss für den Stack und den lokalen Heap Cache des neuen Threads alloziert und initialisiert werden.
Deshalb ist es sinnvoll nicht für jeden Client ein neues `Thread`-Objekt zu erzeugen, sondern `Thread`-Objekte weiterzuverwenden und nicht zu viele `Thread`-Objekte gleichzeitig zu verwenden. Um dies zu erreichen, kann das *Thread Pool* Design Pattern angewendet werden.

In Java gibt es dafür das Interface `java.util.concurrent.ExecutorService`. An einem `ExecutorService` kann man die Methode `execute(Runnable command)` aufrufen, der man ein `Runnable`, also ein Objekt einer Klasse, die die `run()`-Methode implementiert, als Parameter übergibt. Das `Runnable` wird vom `ExecutorService` in einer ihm eigenen Strategie unter Wiederverwendung der verfügbaren `Thread`-Objekte seines *Thread Pools* ausgeführt.
Mit den folgenden statischen Methoden der *Factory* `Executors` kann man unterschiedliche *Thread Pools* in Java erzeugen, die als `ExecutorService` in der Lage sind, `Thread`-Instanzen für `Runnable`-Objekte in unterschiedlicher Weise wiederzuverwenden:

`Executors.newSingleThreadExecutor()`: Führt genau einen `Thread` im *Thread Pool* aus. Die mit `execute(...)` übergebenen `Runnable`-Objekte werden in der Reihenfolge des Aufrufs von `execute(...)` nacheinander in der `Thread`-Instanz ausgeführt.

`Executors.newFixedThreadPool(int n)`: Lässt genau n `Thread`-Instanzen nebenläufig ablaufen, in denen die übergebenen `Runnable`-Objekte ausgeführt werden. Werden mehr `Runnable` mit `execute(...)` übergeben als `Thread`-Instanzen zur Verfügung stehen, wird eine Warteschlange gebildet. Endet ein `Runnable`, wird der `Thread`, in dem es ausgeführt wurde, frei und kann für das am längsten in der Warteschlange wartende `Runnable` wiederverwendet werden. Zu jedem Zeitpunkt gibt es genau n `Thread`-Instanzen in dem *Thread Pool*.

`Executors.newCachedThreadPool()`: Erzeugt einen *Thread Pool* mit dynamischer Größe: Sind im *Thread Pool* keine `Thread`-Objekte zur Wiederverwendung vorhanden, wenn ein `Runnable` mit `execute(...)` übergeben wird, wird ein neuer `Thread` instanziiert und dieses `Runnable` darin ausgeführt. Wird eine frei gewordene `Thread`-Instanz 60 Sekunden lang nicht mehr benutzt, wird sie aus dem *Thread Pool* entfernt.

Mit diesen *Thread Pool* Implementierungen können *threaded* TCP Server effizienter laufen. Die innere Klasse, die die anwendungsspezifische Abarbeitung der Verbindungen in der `run()`-Methode übernimmt, muss dann nicht mehr `Thread` erweitern, sondern das Interface `Runnable` implementieren. In der Endlosschleife in `start()`

Listing 3.12: Die Implementierung des Echo Servers mit Thread Pool `EchoServerThreadPool`

```
1  public class EchoServerThreadPool {
2      private int port;
3      private ExecutorService threadPool;
4
5      public EchoServerThreadPool(int port) {
6          this.port = port;
7          this.threadPool = Executors.newSingleThreadExecutor();
8      }
9
10     public void start() {
11         try (ServerSocket serverSocket = new ServerSocket(port)) {
12             while (true) {
13                 Socket socket = serverSocket.accept();
14                 this.threadPool.execute(new EchoThread(socket));
15             }
16         } catch (IOException e) {
17             System.err.println(e);
18         }
19     }
20
21     private class EchoThread implements Runnable {
22         // ...
23      }
24
25     public static void main(String[] args) {
26         // ...
27     }
28 }
```

wird dann die Instanz der inneren Klasse nicht mit `start()` als eigener `Thread` gestartet, sondern per `execute(...)` einem `ExecutorService` übergeben, der das `Runnable` nach seiner eigenen Strategie mit den im *Thread Pool* zur Verfügung stehenden `Thread`-Objekten ausführen wird (s. Listing 3.12).

Aktivitätsschritt 3.23 (fakultativ):

Ziel: Framework zur Implementierung von TCP Servern entwerfen

Die drei *threaded* Server aus diesem Kapitel teilten alle dasselbe Grundgerüst. Die konkrete Implementierung des Services liegt dabei jeweils isoliert in der inneren Klasse. Extrahieren Sie den generellen Aufbau der Server und entwerfen Sie ein eigenes Framework für *threaded* TCP Server zu beliebigen Services in Java. Es spricht nichts dagegen, die bisher innere Klasse außerhalb der `Server`-Klasse zu implementieren.

Exkurs: *Thread Pools* als Mittel gegen «Distributed Denial of Service»-Angriffe

Speziell für *threaded* Server bieten *Thread Pools* eine einfache Möglichkeit sich vor den Auswirkungen von *Distributed Denial of Service* (DDoS) zu schützen. Bei einem DDoS-Angriff blockieren verteilte Angreifer einen Service durch eine hohe Anzahl sinnloser *Requests*. Bedient ein Server alle Client-Anfragen, indem jeweils ein `Thread` für die Verbindung erzeugt wird, kann es bei einem solchen Angriff dazu kommen, dass so viele `Thread`-Instanzen aktiv sind, dass der Server abstürzt oder das Betriebssystem keinem einzelnen `Thread` mehr genug Ressourcen zur Verfügung stellen kann. Die Verwendung eines *Fixed* oder *Single Thread Pools* deckelt die Anzahl der aktiven `Thread`-Instanzen. Client *Requests* würden ggf. in der Warteschlange oder im TCP Backlog liegen bleiben. Legitime Client *Requests*, also solche, die nicht zum DDoS-Angriff gehören, würden dann zwar z. T. auch nicht bedient werden, aber die grundsätzliche Funktionsfähigkeit des Servers bliebe bestehen.

Tipp: Es wäre vorteilhaft in solch einem Framework *Thread Pools* zu verwenden.

3.5.2 Funktionale Erweiterung von Kommunikationsprotokollen

Die Protokolle der bisher implementierten Services sind sehr einfach. Sie folgen einem einzigen Ablaufschema. Reale Kommunikationsprotokolle sind komplexer, weil sie einen größeren funktionalen Umfang haben.

Anfragen vom Client: Abrufe und Aufrufe

Oft möchte man Clients ermöglichen Dateien oder andere Daten vom Server **abzurufen**. Ein Client könnte dazu Anfragen an den Server richten, die dieser je nach Inhalt der Anfrage anders beantworten soll. Der Server müsste dabei die Nachrichten, die vom Client kommen, z. B. als Dateinamen interpretieren, die zurückgesendet werden sollen.

Serverseitig könnten auch aufwändige Rechendienste angeboten werden, die rechenschwache Clients **aufrufen** können. Dann müsste der Client in seiner Anfrage die Rechenfunktion auswählen und dazugehörende Parameter übermitteln. Wenn Client und Server in derselben Programmiersprache entwickelt wurden, gibt es meistens einfache Möglichkeiten ein Format zu definieren, an das Client und Server sich halten können um interoperabel zusammenarbeiten zu können. In Java stehen dafür

bspw. `ObjectOutputStream` und `ObjectInputStream` zur Verfügung. Will man aber zulassen, dass Client und Server auf unterschiedlichen technologischen Plattformen basieren können, wird es aufwändiger ein neutrales Austauschformat zu entwickeln.

Die absichtliche Verletzung solch eines vorgegebenen Protokolls oder Austauschformats eines Service kann ein Angriffsvektor zum Lahmlegen oder zum Einbruch in einen Server sein. Man sollte also auch immer unvorhergesehene Anfragen des Clients behandeln können, ohne dass der Server mit einer `ParseException` oder einer anderen Ausnahme abstürzt!

Komplexere Protokolle mit mehreren Phasen

Die Protokolle bisher ließen sich im Wesentlichen als Anfrage-/Antwortdialog umsetzen. Auch wenn viele verbreitete Kommunikationsprotokolle so arbeiten, ergeben sich interessante Möglichkeiten, wenn man einen komplexen Ablauf mit mehreren Schritten vorsieht. Formal lassen sich solche mehrstufigen Protokolle leicht durch endliche Automaten ausdrücken. Ausgehend von einem Startzustand können Client oder Server beim Empfang eines bestimmten Ausdrucks in einen Folgezustand übergehen, in dem der Empfang desselben Ausdrucks möglicherweise eine andere Reaktion bewirkt.
Ein Beispiel dafür sind *sessions*: Am Anfang solch einer Sitzung ist der Client noch anonym und nicht authentifiziert. Versucht er durch das Senden eines Befehls zum Abruf einer geschützten Datei die Anmeldung zu umgehen, würde der Server eine Fehlermeldung zurückmelden und sich weiter im nicht authentifizierten Zustand befinden. Sendet der Client hingegen seine *Credentials*, die der Server mit einem autorisierten Benutzer abgleichen kann, würde der Server in den Zustand «Benutzer angemeldet» gehen. In diesem Zustand könnte der Abruf der Datei, deren Zugriff vorher gescheitert war, erlaubt sein. Hingegen könnte das Senden von weiteren *Credentials* in diesem Zustand eine Protokollverletzung darstellen.
Im Fall eines Thread-basierten Servers müsste man jeweils einen Zustandsautomaten für jeden Client vorhalten. In dem Gerüst der Java-Implementierung aus den vorherigen Beispielen würde eine Instanzvariable in der inneren Klasse geeignet sein, den aktuellen Zustand der Verbindung zu speichern. In Java würde man als Typ der Zustandsvariablen zweckmäßigerweise eine `Enumeration` benutzen, die für jeden erlaubten Zustand einen Wert hat.

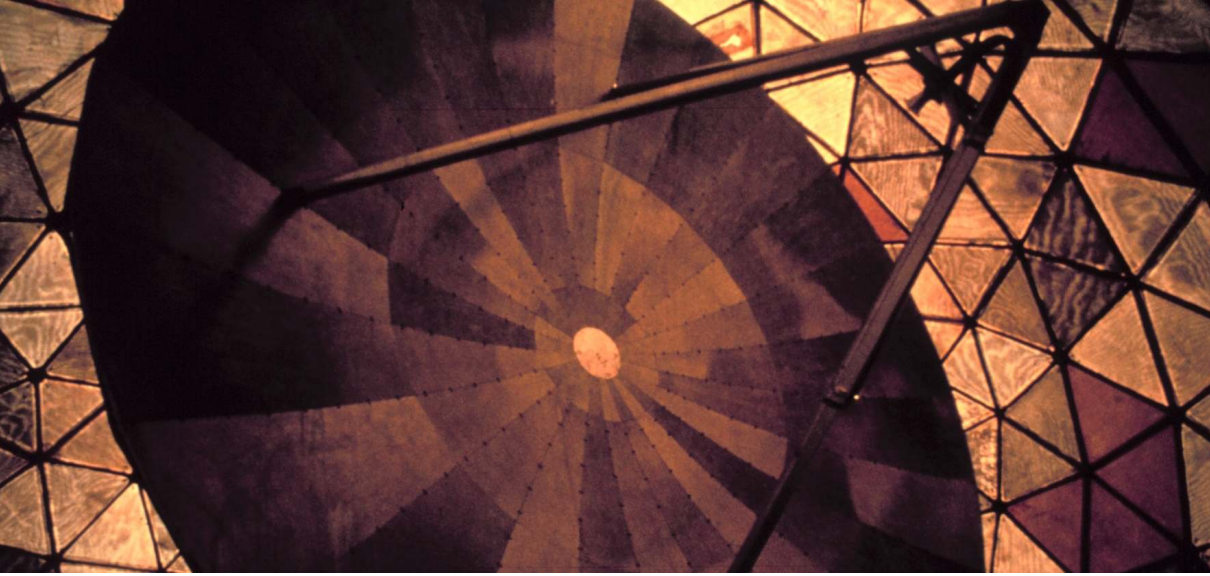
Aktivitätsschritt 3.24 (fakultativ):

Ziel: Service für Berechnungen entwerfen

Entwerfen Sie einen Service für Berechnungen. Der Service sollte mehrere unterschiedliche Funktionen wie «Fakultät» oder «Test auf Primzahleneigenschaft» anbieten. Damit die Ressourcen des Servers nicht unbefugt verwendet werden können, soll ein Login-Prozess vorgeschaltet sein.

Teil II

Indirekte, gepufferte Kommunikation



Indirekte, gepufferte Kommunikation

Die Verwendung von *Message Queues* erlaubt die Entkopplung von Sender und Empfänger in verteilten Systemarchitekturen. Ein möglicher Realisierungsansatz für *Message Queue* Systeme ist, einen zentralen Vermittlerprozess (auch «Message Broker» genannt) im Netzwerk bereitzustellen, der Warteschlangen («Queues») verwaltet. Der Broker agiert als Server. Systeme, die untereinander kommunizieren wollen, verbinden sich als Client mit dem Broker. Solche Clients können Nachrichten in eine *Message Queue* des Brokers schreiben (dann fungieren sie als *Message Producer*) oder sie können Nachrichten aus einer bestimmten *Queue* abrufen (dann fungieren sie als *Message Consumer*).
Die indirekte Vermittlung von Nachrichten über einen Broker als Zwischenstation hat viele Vorteile:

- Der Broker wirkt als Puffer. Es ist für den Empfänger nicht erforderlich in derselben Geschwindigkeit Nachrichten zu verarbeiten, wie der Sender sie schickt.
- Es ist für einen Empfänger möglich Nachrichten von mehreren Sendern über nur eine Verbindung zum Broker zu erhalten.

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

- Umgekehrt kann der Broker eingehende Nachrichten an mehrere Empfänger weiterleiten. Es kann also mehrere Empfänger geben, ohne dass der Sender das wissen oder programmtechnisch berücksichtigen muss.
- Die Fehlertoleranz wird erhöht, da der Empfänger nicht immer empfangsbereit sein muss.
- Nachrichten können zur effizienteren Ausnutzung der Bandbreite eines Netzwerkes gebündelt werden.
- Nachrichten können am Vermittlungsknoten gefiltert werden, sodass nur wirklich wichtige oder relevante Nachrichten an einzelne Empfänger weitergereicht werden, ohne dass Sender oder Empfänger dafür eigene Funktionalitäten implementieren müssten.

4.1 Enterprise Application Integration

Ein wichtiger Einsatzbereich für *Message Queue Systeme* ist in der *Enterprise Application Integration* (EAI), also der Integration bereits existierender Anwendungssysteme. Soll beispielsweise die Telefonanlage A die Nummer eines eingehenden Anrufs an das *Customer Relationship Management System* (CRM) B schicken, um am Arbeitsplatzrechner eines *Call Center Agents* die richtige Fallakte anzuzeigen, müsste man ohne Broker ein eigenes Protokoll ersinnen, das zwischen einem an das CRM angebundenen Socket Server und einem in die Telefonanlage integrierten Client umgesetzt werden müsste. Das ist sehr aufwändig, weil das weitreichende Eingriffe in die Programmlogik bedingen würde.
Es ist hingegen sehr viel einfacher die Integration über eine *Message Queue* zu realisieren: Man braucht einen *Broker* als Server. Der *Broker* braucht keine anwendungsspezifische Programmierung. Deshalb gibt es vorgefertigte Produkte, die man einfach installieren und starten kann.
A schickt die Telefonnummer an eine vorher definierte *Queue* auf dem *Broker*. A fungiert als *Message Producer*. Dazu muss in A ein Client realisiert werden. Das Protokoll zwischen *Message Producer* und *Broker* ist wiederum nicht von der Anwendung abhängig. Es gibt deshalb auch dafür vorgefertigte Routinen, die in einer *Library* als Teil des *Broker*-Produkts bereitgestellt werden. Die *Broker*-Produkte unterscheiden sich

allerdings durchaus bei den unterstützen Programmiersprachen der mitgelieferten *Libraries*.
Für B muss dann wiederum ein Client geschrieben werden, der die Nachricht vom *Broker* abholt und die entsprechende Fallakte bereitstellt. Dadurch, dass B nun nicht mehr als Server arbeitet, sondern seinerseits einen Client für den *Broker* darstellt, kann man die entsprechende Funktion einfach in B integrieren. Auch wenn B in einer anderen Programmiersprache programmiert wurde als A, kann man i. Allg. ein *Broker*-Produkt finden, das die erforderlichen Plattformen unterstützt.

Message Broker und ihre *Client Libraries* beinhalten Kommunikationsmuster, mit denen sowohl die synchrone Kommunikation (B fragt aktiv beim *Broker* nach, ob eine neue Nachricht vorliegt) als auch asynchrone Kommunikation (B wird vom Broker über neu vorliegende Nachrichten informiert) unterstützt werden.

In den meisten Fällen wird eine Rückantwort von B zurück zu A erforderlich sein. Daher kehren sich die Rollen um und B schickt eine Nachricht als Antwort (z. B. den Namen eines bereits bekannten Kunden) zurück. B wird *Message Producer* und A holt die Rückantwort von einer *Queue* als *Message Consumer* wieder ab. Ein einzelner Kommunikationspartner (Knoten) kann also sowohl als *Message Producer* als auch als *Message Consumer* fungieren.
Weitere Funktionen, die *Message Broker* bereitstellen können und die in der EAI besonders nützlich sein können, sind die *Formatwandlung*, *Duplizierung* oder *Filterung* der Nachrichten abhängig von Sender, Empfänger oder Inhalt der Nachrichten.

4.2 Message Oriented Middleware

Natürlich kann die asynchrone Art der Kommunikation auch mit direkter Socket-Programmierung erreicht werden. Dafür müsste aber ein umfangreiches Framework programmiert werden, mit dem Nachrichten beispielsweise zwischengespeichert werden, wenn der Empfänger gerade nicht für die Kommunikation zur Verfügung steht. Und tatsächlich verwenden *Message Queue* Systeme intern Sockets (TCP oder UDP) und stellen genau solch ein Framework bereit, das sehr viel einfacher zu nutzen ist.
Durch die Verwendung eines *Message Brokers* ist die Kopplung zwischen Sender und Empfänger weniger eng. Man spricht deshalb von *loser Kopplung*. Verglichen mit einem eng gekoppelten verteilten System, in dem alle Knoten direkt miteinander kom-

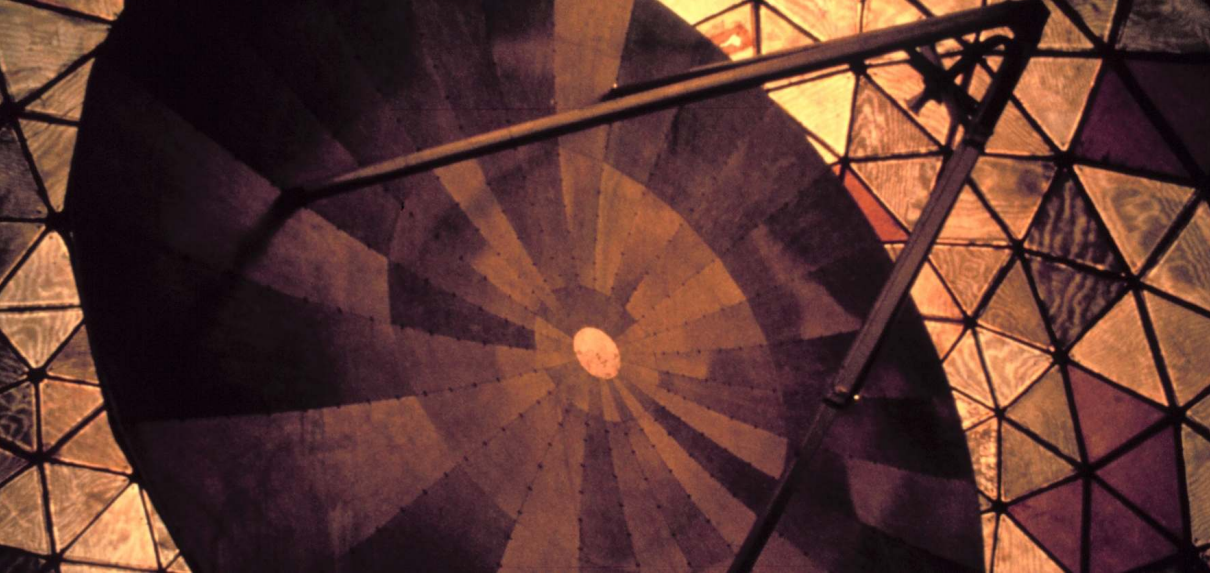
Exkurs: Verteilter Broker (Cluster)

Ein Nachteil eines zentralen *Message Broker Servers* ist, dass er einen *single point of failure* darstellt: Wenn dieser eine Knoten ausfällt, können alle anderen nicht mehr miteinander kommunizieren. Außerdem würde die Verarbeitungskapazität eines zentralen *Message Brokers* wie ein Flaschenhals wirken und die Anzahl der zwischen Knoten ausgetauschten Nachrichten pro Zeiteinheit oder auch die maximale Anzahl der kommunizierenden Knoten begrenzen. Um dem zu begegnen wird in großen Systemen oder in Systemen, die besonders hohe Anforderungen an die Verarbeitungsgeschwindigkeit stellen, der Message Broker nicht als ein singuläres System, sondern als ein Netzwerk untereinander kommunizierender Message Broker angelegt.

munizieren, ist es dadurch einfacher Sender oder Empfänger auszutauschen, weil sie weniger Abhängigkeiten voneinander haben. Stattdessen werden Bezüge untereinander durch die jeweilige Kopplung zu einer standardisierten *Middleware* ersetzt, deren konkrete Realisierung durch die Normierung der Protokolle und Schnittstellen leicht gegen vergleichbare Implementierungen austauschbar ist. Im besten Fall brauchen die *Message Producer* und *Message Consumer* nicht geändert zu werden, wenn der *Message Broker* ausgetauscht wird.
Es gibt allerdings mehrere teils konkurrierende Standards für nachrichtenorientierte Middleware (*message-oriented Middleware*, abgekürzt MOM):

- **Data Distribution Service** (DDS): hauptsächlich für komplexe technische Systeme, vor allem in der Luft- und Raumfahrttechnik; standardisiert von der *Object Management Group* (OMG)
- **Advanced Message Queuing Protocol** (AMQP): hauptsächlich für Geschäftsanwendungen; von einem Industriekonsortium von Finanzunternehmen und Softwarefirmen standardisiert
- **Message Queue Telemetry Transport** (MQTT): hauptsächlich für «Internet of Things»-Anwendungen, standardisiert von der *Organization for the Advancement of Structured Information Standards* (OASIS)
- **Java Message Service** (JMS): hauptsächlich für Geschäftsanwendungen und *Enterprise Application Integration*; standardisiert als Teil der *Java Enterprise Edition Plattform* im Rahmen des *Java Community Process*.

Manche MOM-Produkte bieten Schnittstellen für mehrere Protokolle. In Band 1 von *Verteilte Architekturen* wird bspw. ActiveMQ von Apache als MOM benutzt. ActiveMQ unterstützt u.a. die Protokolle AMQP, MQTT (s. Kapitel 6) und JMS (s. Kapitel 5).



Java Message Service (JMS)

In diesem Kapitel wird JMS für die asynchrone Kommunikation zwischen Java-Prozessen benutzt. Dabei werden unterschiedliche Kommunikationsmuster behandelt. Außerdem werden die folgenden Features von JMS in Programmierbeispielen angewendet:

- *Message Properties*
- *Message Listener* für asynchrone Kommunikation
- Filter

Von Java aus existiert mit den Java Message Service (JMS) eine einheitliche Schnittstelle, die viele MOM-Produkte anbieten. In diesem Zusammenhang fungiert ein «JMS Provider» als MOM Server. «JMS Clients» sind Programme, die über den JMS Provider miteinander kommunizieren.
JMS Clients beinhalten *Message Producer* (zum Senden von Nachrichten) oder *Message Consumer* (zum Empfangen bzw. Abrufen von Nachrichten). JMS Clients können dabei sowohl *Message Producer* als auch *Message Consumer* gleichzeitig enthalten.

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

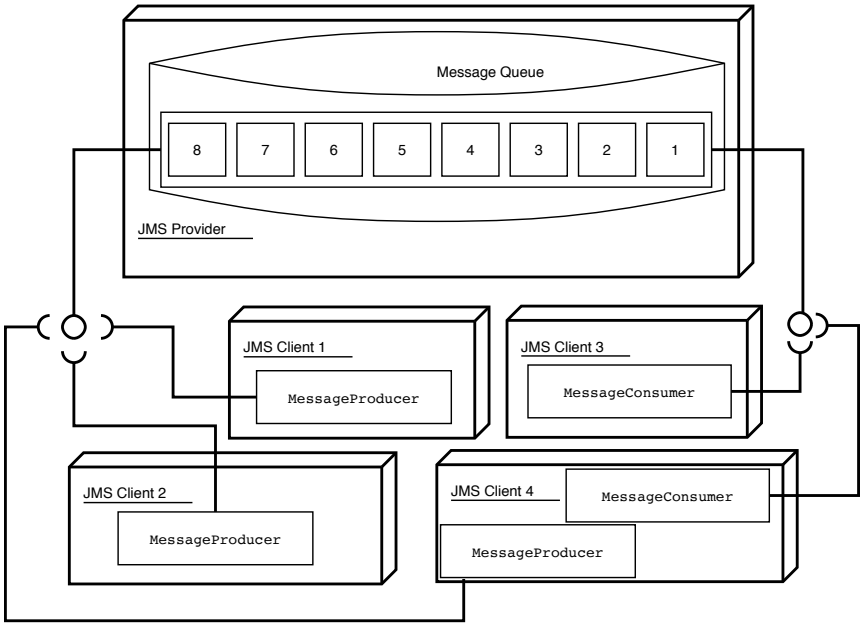


Abbildung 5.1: Beispiel für ein verteiltes System mit einem JMS Provider und vier JMS Clients, die jeweils `MessageProducer` und `MessageConsumer` beinhalten

Abb. 5.1 zeigt beispielhaft eine Situation, in der die JMS Clients 1-4 mit einem JMS Provider kommunizieren. JMS Client 1 und 2 enthalten jeweils genau einen `MessageProducer`, JMS Client 3 genau einen `MessageConsumer`. Der JMS Client 4 enthält sowohl einen `MessageProducer`, als auch einen `MessageConsumer`. Der JMS Provider bietet auf der einen Seite eine Schnittstelle zum Schreiben in seine Message Queue an, die von den `MessageProducern` der JMS Clients benutzt wird. Auf der anderen Seite stellt er eine Schnittstelle zum Lesen aus seiner Message Queue bereit, die von den `MessageConsumern` in den JMS Clients benutzt wird.

Der Middleware-Standard JMS ist spezifisch für Java. Ein festes «Wire-Protokoll», also eine Spezifikation der Datenpakete, die über die Sockets zwischen MOM Servern und JMS Clients ausgetauscht werden, ist nicht vorgegeben. Deshalb ist es nicht möglich, aus einer anderen Programmiersprache heraus solche Datenpakete zu erzeugen und zu verarbeiten.

Man kann stattdessen nur durch die Verwendung der vorgegebenen Java-Klassen an einem JMS-Kommunikationsverbund teilnehmen. Die dafür erforderlichen Packages in `javax.jms.*` sind Teil der Java EE-Spezifikation. Sie beinhalten aber nur Java Interfaces. Damit ist es möglich JMS Clients alleine gegen diese Spezifikation zu programmieren und zu `*.class`-Files zu kompilieren.
Zur Laufzeit sind dann allerdings Implementierungen der Interfaces erforderlich. Ein JMS Provider muss zu allen standardisierten JMS Interfaces Implementierungsklassen bereitstellen. Sie sind spezifisch für die konkrete technische Umsetzung der Middleware. In ihnen ist das jeweilige proprietäre Wire-Protokoll zwischen JMS Provider und JMS Clients implementiert.
Die Implementierungsklassen der JMS Interfaces («Client Library») müssen, zumeist in Form eines JAR, im *Classpath* der JMS Clients bereitgestellt werden.

5.1 Java Naming and Directory Interface (JNDI)

JMS abstrahiert von der konkreten Implementierung. Deshalb muss ein Vermittler eingeschaltet werden, um die Realisierungsklassen (z. B. der Apache ActiveMQ-Implementierung) bereitzustellen bzw. anzusprechen. Auf diesen Vermittler wird über eine abstrakte Schnittstelle aus dem Java-eigenen Framework *Java Naming and Directory Interface* (JNDI) zugegriffen, das in späteren Kapiteln auch noch bei anderen Middleware-Systemen verwendet wird.
Das JNDI wird über die *Properties*-Datei `jndi.properties` im Wurzelverzeichnis des Projekts (also z. B. `.../workspace/VAR-JMS/src/jndi.properties`) konfiguriert:

```
1 java.naming.factory.initial = org.apache.activemq.jndi.
      ActiveMQInitialContextFactory
2 java.naming.provider.url = tcp://localhost:61616
3
4 ## var.mom.jms.log
5 queue.var.jms.log.queue = queue4719
```

Beim Eintrag `java.naming.provider.url` muss statt `localhost` der Rechnername bzw. die IP-Adresse des Rechners angegeben werden, auf dem der JMS Provider läuft. Die Portnummer `61616` ist spezifisch für Apache ActiveMQ und kann in der Konfiguration von Apache ActiveMQ auch geändert werden.

Über den Namensdienst des JNDI bekommen die JMS Clients alle Informationen, die zur Nutzung des JMS Providers erforderlich sind. Die Klassen und Interfaces, mit denen das JNDI von Java-Programmen aus angesprochen wird, liegen im Package `javax.naming.*`. JNDI kann zwischen mehreren verschiedenen Kontexten (`Context`) für unterschiedliche Anwendungen unterscheiden. Allgemein kann dann mit der Methode `lookup()` eine spezifische Konfigurationsinformation aus einem `Context` abgerufen werden. Der Typ der Information, die mit `lookup()` aus dem JNDI bereitgestellt wird, ist abhängig von Schlüssel und Kontext. Deshalb ist die Rückgabe der Methode `lookup()` vom Typ `Object`.
Im folgenden Beispiel-Listing soll diejenige Klasse der JMS-Provider-Implementierung geholt werden, über die später mit dem JMS Provider interagiert wird. Das wird mit JNDI dadurch realisiert, dass ein *Factory*-Objekt zurückgegeben wird (`factory` vom Typ `ConnectionFactory`), von dem später durch «*Factory-Methoden*» (`createConnection()`) neue Instanzen der eigentlich interessierenden Klasse geholt werden können. Der konkrete Typ der so abgerufenen Objekte ist allerdings spezifisch für die Implementierung des JMS Providers und zur Entwicklungszeit somit noch nicht bekannt, denn es soll möglich sein, die JMS-Provider-Implementierung jederzeit auswechseln zu können, ohne dass das Programm neu übersetzt werden muss. Der «Trick» ist nun, dass die JMS-Provider-spezifischen Klassen immer ein JMS-standardisiertes Interface implementieren, über das die Typisierung erfolgen kann. Bspw. implementieren alle Klassen, die von der `ConnectionFactory` erzeugt werden, das Interface `Connection` aus dem Package `javax.jms` (s. Zeilen 2 und 11 in Listing 5.1).

```
1 Context ctx = new InitialContext();
2 ConnectionFactory factory = (ConnectionFactory) ctx.lookup("
      ConnectionFactory");
3 Destination queue = (Destination) ctx.lookup("queue.var.mom.jms.QUEUE34
      ");
```

Außerdem wird eine Referenz auf die konkrete *Message Queue* (`queue` vom Typ `Destination`) über einen symbolischen Namen («`var.mom.jms.QUEUE34`») geholt. `factory` und `queue` sind spezifisch für die Konfiguration der konkreten JMS-Anwendung, aber unabhängig vom verwendeten JMS Provider. Sie würden in einem operativen System angepasst und in `jndi.properties`, also nicht im Quelltext des JMS-Client-Programms, bereitgestellt werden.

Ein Vorteil bei der Verwendung des JNDI (im Vergleich zur direkten Codierung der Spezifika des verwendeten JMS Providers im Programmtext) ist, dass der JMS Provider ohne Änderung der JMS Clients ausgetauscht werden kann. Lediglich die Konfiguration in `jndi.properties` muss dafür angepasst werden.

5.2 Programmierung von JMS Clients

JMS beinhaltet eine Reihe von Klassen, die auf eine spezielle Weise zusammenarbeiten. Für die Nutzung von JMS in einem Client ist es erforderlich, deren Abhängigkeiten und Zusammenwirken zu kennen.

5.2.1 Behandlung von Ausnahmen

Bei der Verwendung von JMS-Funktionen können Fehlersituationen durch `JMSException` signalisiert werden. Bei der Verwendung von JNDI können Fehlersituationen durch `NamingException` signalisiert werden.

5.2.2 Verbindungsaufbau zum JMS Provider

Das folgende Listing 5.1 enthält ein minimales Grundgerüst zum Erzeugen und Starten eines JMS Clients (zur JMS 1.1 Spezifikation).
Das Zusammenwirken der Objekte im Konstruktor in Listing 5.1 ist folgendermaßen:

- **`ctx`** (Typ:**`Context`**): JNDI-Kontext (s. Abschnitt 5.1)
- **`factory`** (Typ:**`ConnectionFactory`**): Über die `factory` kann später eine `Connection` zum JMS Provider geöffnet werden. `factory` ist das Ergebnis eines JNDI-`lookup()` Aufrufs (s. Abschnitt 5.1).
- **`connection`** (Typ: **`Connection`**): Die `Connection` wird vom `ConnectionFactory`-Objekt durch die Methode `createConnection()` abgerufen bzw. erzeugt.
- **`session`** (Typ: **`Session`**): Von der `connection` kann mit der Methode `createSession(...)` ein `Session`-Objekt erzeugt werden. Die `Session`

hat eine bestimmte Dienstgütecharakteristik, die festlegt, wie Nachrichten versendet werden und ob der JMS Provider vom JMS Client Empfangsbestätigungen einfordert. In diesem Kapitel wird immer dieselbe Konfiguration benutzt (s. Exkurs «Dienstgüte bei JMS Sessions» auf S. 130).

- **`destOut`** und **`destIn`** (Typ: **`Destination`**): Bei einer `Destination` kann es sich um eine `Queue` (für *Point-To-Point*-Kommunikation) oder ein `Topic` (für *Publish/Subscribe*-Kommunikation) handeln (s. Abschnitt 5.3). Die `Destination`-Objekte werden über das JNDI bereitgestellt, indem wie auch beim `ConnectionFactory`-Objekt ein `lookup()` am `Context`-Objekt ausgeführt wird. Ob von dort ein `Topic` oder `Queue` als Typ zurückgeliefert wird, hängt von der Konfiguration des JNDI ab. In Listing 5.1 wird das durch die Verwendung des Supertyps `Destination` offen gehalten.
  - **`destOut`** ist im Beispielquelltext die Warteschlange, über die der JMS Client Nachrichten versendet.
  - **`destIn`**: Zum Empfangen von *Message Producer* und *Message Consumer* wird später eine `Destination` benötigt.

Listing 5.1: Gerüst für einen JMS Client mit `MessageProducer` und `MessageConsumer` und automatischem Verbindungsaufbau im Konstruktor

```
1  public class JMSClient {
2      private Connection connection;
3      private Session session;
4      private MessageProducer producer;
5      private MessageConsumer consumer;
6
7      public JMSClient(String sendDest, String receiveDest)
8          throws NamingException, JMSException {
9          Context ctx = new InitialContext();
10         ConnectionFactory factory = (ConnectionFactory) ctx.lookup("
               ConnectionFactory");
11         connection = factory.createConnection();
12         session = connection.createSession(false, Session.
               AUTO_ACKNOWLEDGE);
13         Destination destOut = (Destination) ctx.lookup(sendDest);
14         Destination destIn = (Destination) ctx.lookup(receiveDest);
15         producer = session.createProducer(destOut);
16         consumer = session.createConsumer(destIn);
17         connection.start();
18     }
19     // Hauptprogramm und Verbindung abbauen...
20 }
```

- **`producer`** (Typ: **`MessageProducer`**): Von der `Session` kann nun ein `MessageProducer` erzeugt werden. Beim Erzeugen muss das `Destination`-Objekt übergeben werden, das dieser `MessageProducer` als Message Queue benutzen soll.
- **`consumer`** (Typ: **`MessageConsumer`**): Analog zum `producer` wird an der `Session` auch ein `MessageConsumer` erzeugt. Wieder muss das `Destination`-Objekt übergeben, von wo dieser `MessageConsumer` von nun an seine Nachrichten abholen soll.
- Nachdem der `consumer` erzeugt wurde, kann mit dem Aufruf der Methode `start()` am `Connection`-Objekt die Zustellung von Nachrichten an alle `MessageConsumer` dieses JMS Clients, die an dieser Verbindung hängen, gestartet werden.

5.2.3 Ressourcen im JMS Client freigeben

Am Ende sollten die Ressourcen der JMS-Nutzung vom JMS Client wieder frei gegeben werden. Das ist im Listing 5.2 im `finally`-Block der Methode `main(...)` demonstriert.

5.2.4 Nachrichten mit JMS synchron empfangen

Der Empfang einer Nachricht kann entweder *synchron* erfolgen, indem an einem `MessageConsumer`-Objekt die Methode `receive()` benutzt wird, oder *asynchron* mit einem `MessageListener` (s. Abschnitt 5.2.5).
Im synchronen Fall kann die Methode `receive` entweder ohne Parameter aufgerufen werden, dann blockiert der Aufruf so lange, bis eine Nachricht an der `Destination` des `MessageConsumer`-Objekts am JMS Provider verfügbar ist und abgerufen wurde bzw. bis der `MessageConsumer` geschlossen wurde.
Alternativ kann `receive` auch mit einer Zeitdauer in Millisekunden als Parameter aufgerufen werden, der angibt, wie lange der `receive`-Aufruf maximal blockieren darf. Ist die Maximaldauer erreicht, ohne dass eine `Message` empfangen wurde, ist das ein *Timeout*-Ereignis und das Ergebnis von `receive(...)` ist `null`. Wird `0` als Parameter für die maximale Dauer übergeben, blockiert der Aufruf wie bei `receive()` für immer.

Exkurs: Dienstgüte bei JMS Sessions

Ein JMS Client kann die erwartete Dienstgüte der Kommunikation mit dem JMS Provider spezifizieren, wenn die `Session` erzeugt wird.
Eine `Session` kann entweder *transaktionsbasiert* oder *ohne Transaktionskontrolle* ablaufen. Falls Transaktionsunterstützung aktiviert wird, können einzelne, aber inhaltlich zusammengehörende `Message`-Objekte zu einem Kommunikationsschritt, einer Transaktion, zusammengefasst werden. Sollte während der Abarbeitung der Transaktion ein Fehler oder auch anwendungsabhängig eine neue Situation eintreten, die die Transaktion ungültig macht, kann die gesamte Transaktion «zurückgerollt» werden, solange sie noch nicht durch einen Aufruf der Methode `commit()` am `Session`-Objekt abgeschlossen ist. Wenn keine Transaktionsunterstützung aktiviert wird, kann für die Kommunikation in der `Session` einer von drei unterschiedlichen *Acknowledge Modes* spezifiziert werden.

`AUTO_ACKNOWLEDGE`: Jedes Mal, wenn ein JMS Client erfolgreich eine Nachricht abholt bzw. bekommt, wird dies automatisch bestätigt.

`CLIENT_ACKNOWLEDGE`: Der JMS Client muss explizit die Methode `acknowledge()` an der Nachricht aufrufen, um den Empfang zu bestätigen.

`DUPS_OK_ACKNOWLEDGE`: Die Zustellung von Nachrichten wird weniger stringent als bei `AUTO_ACKNOWLEDGE` bestätigt, was zu Mehrfachzustellungen von Nachrichten an JMS Clients führen kann, aber Kommunikation zwischen JMS Client und JMS Provider spart.

Diese Konfiguration der Session hat Einfluss auf die Dienstgüte der Kommunikationsverbindung und damit auch auf deren Effizienz. Die Mechanismen, die ablaufen müssen, um die Dienstgüte umzusetzen, sind aber voll transparent aus Sicht der JMS Clients: Die Funktionalität, die dafür erforderlich ist, steckt in der Implementierung des JMS Providers und in der Client Library, die zur Kommunikation mit dem JMS Provider von den JMS Clients verwendet wird.

Listing 5.2: Gerüst für einen JMS Client mit Verbindungsabbau am Ende des Hauptprogramms; der automatische Verbindungsaufbau im Konstruktor wie in Listing 5.1 wurde hier weggelassen

```
1  public class JMSClient {
2      private Connection connection;
3      private Session session;
4      private MessageProducer producer;
5      private MessageConsumer consumer;
6
7      // wie beim Konstruktor im vorigen Listing zum JMSClient
8      public static void main(String[] args) {
9          JMSClient node = null;
10         try {
11             node = new JMSClient();
12             // node nutzen (senden, empfangen)
13         } catch (NamingException | JMSException e) {
14             System.err.println(e);
15         } finally {
16             try {
17                 if (node != null && node.producer != null) {
18                     node.producer.close();
19                 }
20                 if (node != null && node.consumer != null) {
21                     node.consumer.close();
22                 }
23                 if (node != null && node.session != null) {
24                     node.session.close();
25                 }
26                 if (node != null && node.connection != null) {
27                     node.connection.close();
28                 }
29             } catch (JMSException e) {
30                 System.err.println(e);
31             }
32         }
33     }
34 }
```

Bevor Nachrichten synchron mit `receive()` empfangen werden können, muss die Verbindung mit `start()` aufgebaut werden. Der Empfang von Nachrichten kann zeitweilig durch den Aufruf von `stop()` ausgesetzt und danach mit `start()` wieder fortgesetzt werden.

5.2.5 Nachrichten mit JMS asynchron empfangen

Alternativ zum synchronen Abruf von Nachrichten von einer `Destination` mit `receive()` kann dem `MessageConsumer`-Objekt mit der Methode `setMessageListener(...)` ein Objekt übergeben werden, das das `MessageListener` Interface implementiert. Das `MessageListener` Interface stellt nur eine Methode bereit:

```
public void onMessage(Message message);
```

Diese Methode wird asynchron vom JMS Framework aufgerufen, sobald eine Nachricht in der `Destination` des `MessageConsumer`-Objekts vorliegt und vom `MessageConsumer` empfangen wurde.
Dazu erzeugt die JMS Client Library die erforderliche Ablauflogik, die intern abläuft, um asynchron Benachrichtigungen vom *Message Broker* empfangen zu können. Beispielsweise könnte ein Socket Server in einem Thread des JMS Clients gestartet werden, der auf Nachrichten vom JMS Provider wartet, `Message`-Objekte entgegennimmt und dann `onMessage(...)` aufruft.
Ein oft verwendetes Muster ist, wie in Listing 5.3 an der Klasse, die das `MessageConsumer`-Objekt enthält, `onMessage(Message message)` zu implementieren und dann direkt nach dem Erzeugen eines `MessageConsumer` Objektes `this` als `MessageListener`-Objekt zu übergeben.

5.2.6 Nachrichtentypen in JMS

JMS Clients können über den JMS Provider die folgenden Nachrichtentypen austauschen:

`Message`: Interface für generelle Nachrichten, bei denen die Payload nicht betrachtet wird. Die folgenden Nachrichtentypen erben alle von diesem Interface.

Listing 5.3: Gerüst für einen JMS Client mit `MessageProducer` und `MessageConsumer`; der automatische Verbindungsaufbau im Konstruktor wie in Listing 5.1 ist hier weggelassen

```
1  public class JMSClient implements MessageListener {
2      private Connection connection;
3      private Session session;
4      private MessageProducer producer;
5      private MessageConsumer consumer;
6
7      public JMSClient(String sendDest, String receiveDest)
8          throws NamingException, JMSException {
9          // Beginn ... wie im Konstruktor beim vorigen Listing zum
               JMSClient
10         Destination destIn = (Destination) ctx.lookup(receiveDest);
11         consumer = session.createConsumer(destIn);
12         consumer.setMessageListener(this);
13         connection.start();
14     }
15     public void onMessage(Message message) {
16         // Verarbeitung von message
17     }
18     // Hauptprogramm und Verbindung abbauen...
19 }
```

`BytesMessage`: Beinhaltet als Payload einen *Stream* von strukturell nicht weiter interpretierten Bytes.

`MapMessage`: Beinhaltet als Payload eine unsortierte Liste von Schlüssel-/Wert-Paaren. Für die Schlüssel wird jeweils ein `String` benutzt. Als Werte sind alle primitiven Datentypen von Java sowie `Object` zugelassen.

`ObjectMessage`: Beinhaltet als Payload ein serialisiertes Java-Objekt.

`StreamMessage`: Beinhaltet als Payload ein Stream von primitiven Werten.

`TextMessage`: Beinhaltet als Payload eine textuelle Nachricht, die als `String` zur Verfügung steht.

`Message` ist ein Super-Interface, das von `BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage` und `TextMessage` spezialisiert wird. Alle Nachrichtentypen sind also auch vom Interface `Message`.
Der Typ wird entsprechend gewählt, je nachdem welcher Art die Nutzlast der Nachricht sein soll. In den Praktika in diesem Kapitel werden die Typen `TextMessage` und `BytesMessage` benutzt.

Beim Empfang eines `Message`-Objekts ist für den JMS Client zuerst unbestimmt, welche Art von `Message` vom `MessageConsumer`-Objekt empfangen wurde. Bevor spezifische Methoden am `Message`-Objekt aufgerufen werden, muss der dynamische Typ des zurückgelieferten Objekts z. B. mit `instanceof` geprüft werden:

```
1 Message message = consumer.receive();
2 if (message instanceof BytesMessage) {
3     BytesMessage bytesMessage = (BytesMessage) message;
4     // mit bytesMessage weiterarbeiten und deren spezifische Methoden
          aufrufen
5 }
```

Wenn das anwendungsspezifische Protokoll zum Zeitpunkt des Empfangs einer Nachricht nur genau einen bestimmten Nachrichtentyp erlaubt und man sich darauf verlässt, dass der Kommunikationspartner der Absprache folgt, kann man über einen Type Cast gleich die richtige Art von `Message` weiterverarbeiten. Es kann dann aber zur Laufzeit einer `ClassCastException` kommen, sollte doch eine andere Art von `Message` empfangen worden sein.

```
1 TextMessage message = null;
2 try {
3     message = (TextMessage) consumer.receive();
4 } (catch ClassCastException e) {
5     // Fehlerbehandlung
6 }
```

5.2.7 Nachrichten mit JMS versenden

Zum Senden muss zuerst am `Session`-Objekt eine neue Nachricht erzeugt werden, die dann über ein `MessageProducer`-Objekt an die `Destination` des `MessageProducers` auf dem JMS Provider geschickt wird.

5.2.8 Message Properties

`Message`-Objekte haben einen anwendungsspezifischen Nutzinhalt («*payload*») und beliebig viele zusätzliche *Properties*. Ein *Property* hat einen Typ, einen Schlüssel vom Typ `String` und einen Wert, der dem Typ des *Properties* entspricht.

Listing 5.4: Gerüst für einen JMS Client mit `MessageProducer`, über den eine leere `Message` versendet wird

```
1  public class JMSClient {
2      private Session session;
3      private MessageProducer producer;
4
5      // Konstruktor (initialisert u. a. session und producer) ...
6
7      public static void main(String[] args) {
8          ProducerNode node = null;
9          try {
10             node = new JMSClient();
11             node.producer.send(node.session.createMessage());
12         } // Ausnahmebehandlung und Verbindung abbauen...
13     }
14 }
```

Properties werden vom `MessageProducer` mit `setProperty(String schluessel, Typ wert)` gesetzt und mit `getTypProperty(String schluessel)` vom `MessageConsumer` ausgelesen. Als Typen sind erlaubt: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `String` und `Object`. Mit `propertyExists(String schluessel)` kann geprüft werden, ob ein bestimmtes *Property* in einer `Message` gesetzt ist.

5.2.9 Filter beim Nachrichtenaustausch über JMS

`MessageConsumer` können beim JMS Provider kundtun, dass sie nur an bestimmten Nachrichten interessiert sind. Dafür kann eine Bedingung, der sogenannte *Message Selector*, als Filter formuliert werden. Der *Message Selector* ist ein `String`, der bei einem `Session`-Objekt an einer Variante der `createMessageConsumer(...)` Methode als Argument übergeben wird wie im folgenden Beispiel:

```
1 MessageConsumer cUnfiltered = session.createConsumer(queue1);
2 MessageConsumer cFiltered = session.createConsumer(queue2, selector);
```

Der *Broker* des JMS Providers wird dann nur *passende* Nachrichten, also solche, bei denen die Bedingung des `messageSelector` zutrifft, an `cFiltered` weitergeben, aber ungefiltert alle an `cUnfiltered`.

Exkurs: Persistenz von Nachrichten

Die Nachrichten von `MessageProducer` JMS Clients werden beim JMS Provider in `Queue`-Datenstrukturen gespeichert. Ob diese Warteschlangen im RAM oder persistent gespeichert werden, ist eine Performance bzw. Robustheitsfrage. Bei einer Zwischenspeicherung z. B. in einer Datenbank ergibt sich die Möglichkeit, dass der Inhalt solcher *Queues* einen Absturz bzw. Neustart des JMS Providers übersteht.
Dazu gibt es den Typ `javax.jms.DeliveryMode`, von dem es genau zwei Ausprägungen als statische Konstanten gibt:

`PERSISTENT` Eine Nachricht wird persistiert, also auf einem Datenträger gespeichert, falls sie nicht sofort an den *Consumer* zugestellt werden kann. Sollte der JMS Provider vor der Wiedererreichbarkeit des *Consumers* abstürzen oder auch kontrolliert heruntergefahren werden (z. B. weil er auf einen anderen Server verschoben werden soll), geht die Nachricht nicht verloren.

`NON_PERSISTENT` Eine Nachricht wird nur im RAM gehalten, falls sie nicht sofort an den *Consumer* zugestellt werden kann. Sollte der JMS Provider vor der Wiedererreichbarkeit des *Consumers* neu gestartet werden, geht die Nachricht verloren und kann später nicht mehr zugestellt werden.

Wird von dem Default-Modus `PERSISTENT` abgewichen, kann es dazu kommen, dass Nachrichten vor dem Versand verlorengehen. Dafür ist keine Speicherung am JMS Provider erforderlich.
Der `DeliveryMode` kann generell für einen `MessageProducer` gesetzt werden:

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

Alternativ können auch nur einzelne Nachrichten als zu persistieren markiert werden:

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 9, 1000);
```

Exkurs: Priorität und Lebensdauer von Nachrichten

Wenn der `DeliveryMode` bei der Methode `send` benutzt wird, müssen noch zwei andere Parameter gesetzt werden:

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 9, 1000);
```

Die anderen beiden Parameter beziehen sich auch auf die Art des Versands von Nachrichten: Der dritte Parameter bezeichnet die Priorität der Nachricht mit Werten von `0`–`9`, wobei der Default-Wert `4` ist und `0`–`4` eine eher normale Priorität kennzeichnet, während Werte zwischen `5` und `9` die außergewöhnliche Dringlichkeit einer Nachricht anzeigen.
Der letzte Parameter ist die *time to live* in Millisekunden, mit der die maximale Lebensdauer einer Nachricht vorgegeben wird, die verstreichen darf, bevor die Nachricht am `MessageConsumer` angekommen sein darf (`0` bedeutet eine unbegrenzte Lebensdauer).
time to live und Priorität können wie der Auslieferungsmodus auch für alle zu sendenden Nachrichten an einem `MessageProducer`-Objekt vorgegeben werden:

```
producer.setTimeToLive(1000);
producer.setPriority(9);
```

Exkurs: Header von JMS-Message-Objekten

JMS`Message`-Objekte enthalten auch immer *Header*-Felder, die z. B. während des Transports zur Priorisierung von Nachrichten benutzt werden oder deren Verfallsdatum kennzeichnen. Beim *Request/Reply* Muster (s. Abschnitt 5.3.2) wird bspw. das Header-Feld `JMSReplyTo` verwendet.
In diesem Kapitel wird nicht direkt auf solche Felder zugegriffen, sondern nur über Methoden des Interface `Message` wie bspw. `setJMSReplyTo(...)` und `getJMSReplyTo()`, die in JMS als einheitlicher Weg vorgesehen sind, um das Header-Feld `JMSReplyTo` zu benutzen.

Die Filterung findet auf der Seite des JMS Providers statt.

Der *Message Selector* Filter `String` wird in einer Syntax wie bei den `WHERE`-Klauseln von SQL92 *Queries* formuliert. Die Felder, deren Wert geprüft oder verglichen werden kann, sind die *Properties* der Nachricht. Die *Message Selector*-Bedingung kann nicht auf Eigenschaften der Nutzlast einer Nachricht bezogen werden.
Wenn die Nachrichten ein *Property* `NewsType` vom Typ `String` haben, könnte folgender Filter definiert werden:

```
1 String selector = "NewsType = \'Sports\' OR NewsType = \'Culture\'";
2 MessageConsumer cFiltered = session.createConsumer(queue2, selector);
```

Der `MessageConsumer cFiltered` würde dann nur noch Nachrichten bekommen, bei denen die folgende Bedingung zutrifft:

```
1 getStringProperty("NewsType").equals("Sports")
2 || getStringProperty("NewsType").equals("Culture")
```

5.3 Kommunikationsmuster bei JMS

Mit Message Brokern kann man unterschiedliche Kommunikationsmuster umsetzen. Sie unterscheiden sich danach, wie die *Message Queues* sich verhalten, wenn eine Nachricht von einem `MessageConsumer` abgerufen wurde und welche `MessageProducer` und `MessageConsumer` welche Warteschlangen nutzen. In den folgenden Abschnitten werden die folgenden Kommunikationsmuster vorgestellt:

- *Point To Point* (PTP)
- *Request/Reply* (R/R)
- *Publish/Subscribe* (P/S)

5.3.1 Point To Point

Bei *Point To Point* (PTP) heißt die `Destination` beim JMS Provider, über die `Message`-Objekte ausgetauscht werden, `Queue`.

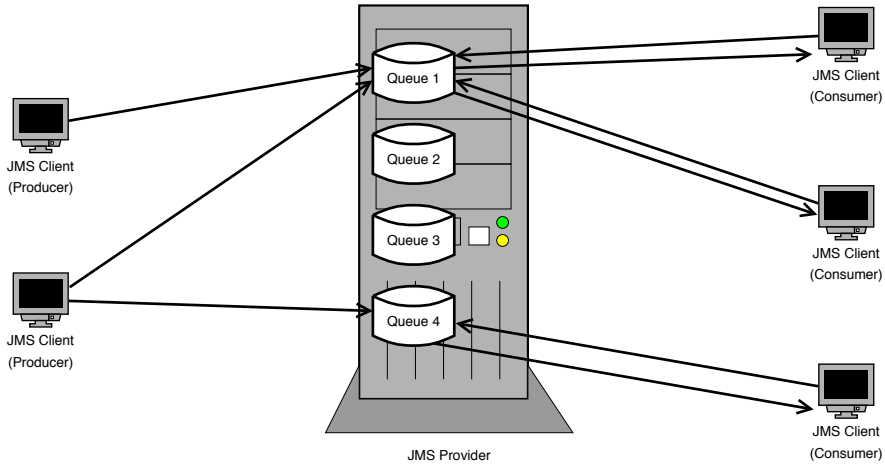


Abbildung 5.2: «1:n»-Verbindung beim Kommunikationsmuster *Point To Point*

Bei PTP hat jede `Message` genau einen `MessageConsumer`, von dem sie aus der Queue abgerufen wird. Der *Producer* sendet also quasi über den Umweg des Brokers eine Nachricht an genau einen Empfänger, den `MessageConsumer`. Es handelt sich deshalb um eine «1:1»-Verbindung. Aus der Sicht des JMS Providers verbraucht das Konsumieren einer Nachricht aus einer Queue diese Nachricht. Andere `MessageConsumer` können eine so verbrauchte Nachricht nicht mehr selber abrufen. Es ist aber möglich, dass ein `MessageProducer` mit mehreren `MessageConsumer`-Instanzen in Verbindung steht wie in Abb. 5.2. Dann handelt es sich *de facto* um eine «1:n»-Verbindung.
Bei der beispielhaften Konstellation in Abb. 5.2 kommunizieren fünf JMS Clients über einen JMS Provider. Sie verwenden dazu zwei der vier vorhandenen *Message Queues*. Die beiden JMS Clients auf der linken Seite der Abb. 5.2 fungieren als `MessageProducer`. Der obere Producer sendet eine Nachricht über «Queue 1». Der andere Producer sendet zwei Nachrichten: eine über «Queue 1», die andere über «Queue 4».
Die beiden oberen *Consumer* auf der rechten Seite fordern jeweils eine Nachricht von «Queue 1» an (z. B. synchron) und löschen sie dadurch aus dieser Warteschlange. Sie lesen die Nachrichten dabei in der Reihenfolge ihres Empfangs beim JMS Provider aus («*first in – first out*»). Der JMS Client unten rechts in Abb. 5.2 holt hingegen eine Nachricht auf «Queue 4» beim JMS Provider ab.

Wird Apache ActiveMQ als *Message Broker* benutzt, muss in der JNDI-Konfiguration `jndi.properties` der Eintrag für die *Destination* mit «`queue.*`» beginnen, bspw. `queue.var.jms.test = VAR4711`.
Im Quelltext eines JMS Clients könnte dann beispielsweise so zugegriffen werden:

```
Destination queue = (Destination) ctx.lookup("var.jms.test");
```

Der Bezeichner `VAR4711` ist dann der Name, der intern vom JMS Provider Apache ActiveMQ zur Identifikation der *Queue* verwendet wird. Der Name der *Queue* ist in der Administrationsoberfläche sichtbar. Bei Apache ActiveMQ kann man darüber die Konfiguration und Anzahl der JMS Clients sowie die darüber versendeten Nachrichten inspizieren.

5.3.2 Request/Reply

Eine Spezialisierung des PTP-Kommunikationsmusters ist *Request/Reply* (R/R). Hier soll der Empfänger einer Nachricht Antworten direkt an den Sender zurückschicken.
Strukturell sind Anwendungen zu diesem Kommunikationsmuster nach dem Client/Server-Prinzip aufgebaut. Da bei JMS technisch der JMS Provider als Server fungiert und JMS Clients sowohl als Dienstnachfrager als auch als Dienstbereitsteller arbeiten, werden im Folgenden die Namen bzw. Funktionsrollenbezeichnungen *Requester* (statt «Client») für den Dienstnachfrager und *Replier* (statt «Server») für den Dienstbereitsteller benutzt.
Da die Antwort in Form eines `Message`-Objekts ausschließlich für den ursprünglichen Sender der Anfrage (*Requester*) gedacht ist, kann der Rückversand nicht über eine allgemeine *Queue* erfolgen, auf der mehrere `MessageConsumer` die Antwort empfangen und dem intendierten Empfänger «wegnehmen» könnten.
Als Lösung für dieses Problem werden temporäre `Queue`-Objekte für die 1:1 Rückkommunikation eingerichtet. Der *Requester* sendet die Anfrage als `Message` an die allgemeine `Queue`, von der der Dienst-Provider Anfragen aller *Requester* entgegennimmt. Gleichzeitig richtet jeder *Requester* beim JMS Provider eine quasi private `Queue` ein, die jeweils ausschließlich für die Rückantworten zu diesem *Requester* verwendet wird. Solche `Queue`-Objekte sind *de facto* privat, weil die Kennung dieser `Queue`-Objekte nur dem Dienst-*Replier* bekannt gemacht wird. Andere JMS Clients kennen diese `Queue`-Objekte nicht.

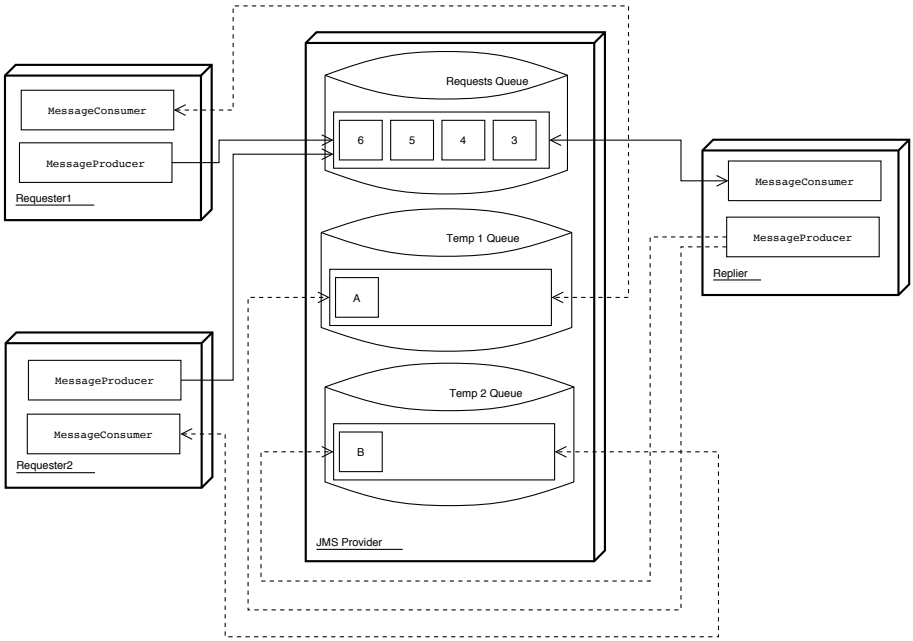


Abbildung 5.3: Zwei *Requester*, ein *Replier* Kommunikationsmuster *Request/Reply*

Abb. 5.3 zeigt als Beispiel für das Request/Reply Kommunikationsmuster eine Situation mit zwei *Requestern* und einem *Replier*. Sowohl *Requester*, als auch *Replier* sind JMS Clients mit jeweils einem `MessageProducer` und einem `MessageConsumer`. Die beiden *Requester* senden über ihren jeweiligen `MessageProducer` Anfragen an den *Replier* über die allgemein bekannte «Requests Queue». Der Replier holt sie bei der «Requests Queue» mit seinem `MessageConsumer` ab (das Konsumieren ist mit einem bidirektionalen Pfeil gekennzeichnet).

Der *Replier* verarbeitet die Anfragen und sendet individuelle Antworten an die *Requester* zurück. Die Behandlung der Antworten des *Repliers* sind in der Abb. durch gepunktete Pfeile abgehoben von den Anfragen der *Requester*, deren Transport mit durchgezogenen Pfeilen gekennzeichnet ist. Die Antworten erfolgen in getrennte temporäre Queues, aus denen die beiden Requester ihre jeweiligen Antworten getrennt abholen (das Konsumieren ist wieder mit einem bidirektionalen Pfeil gekennzeichnet).

Listing 5.5: Gerüst für einen JMS Client als *Requester* nach dem Request/Reply Muster

```
1  public class EchoRequesterNode {
2      private Connection connection;
3      private Session session;
4      private MessageProducer producer;
5      private MessageConsumer consumer;
6      private Queue replyQueue;
7
8      public EchoRequesterNode() throws NamingException, JMSException {
9          Context ctx = new InitialContext();
10         ConnectionFactory factory = (ConnectionFactory) ctx.lookup("
               ConnectionFactory");
11         connection = factory.createConnection();
12         session = connection.createSession(false, Session.
               AUTO_ACKNOWLEDGE);
13         Queue queue = (Queue) ctx.lookup(Conf.QUEUE);
14         producer = session.createProducer(queue);
15         replyQueue = session.createTemporaryQueue();
16         consumer = session.createConsumer(replyQueue);
17         connection.start();
18     }
19
20     public void sendMessage(String text) throws JMSException {
21         TextMessage message = session.createTextMessage();
22         message.setText(text);
23         message.setJMSReplyTo(replyQueue);
24         producer.send(message);
25     }
26     // ...
27 }
```

Mit der Methode `createTemporaryQueue()`, kann ein *Requester* an der `Session` solch eine «private» `Queue` anlegen. Zunächst ist sie nur dem *Requester* bekannt (s. Zeile 15 in Listing 5.5).

Damit der JMS Client, der eine Nachricht vom *Requester* empfängt, seine private Antwort-`Queue` für zurückgesendete Nachrichten benutzt, muss sie ihm bekanntgegeben werden. Der *Requester* verwendet dazu ein Header-Feld der Nachricht, die die Anfrage an den *Replier* enthält. Das Feld wird mit der `Message`-Methode `setJMSReplyTo(...)` gesetzt (vom *Requester*, s. Zeile 23 in Listing 5.5) und mit `getJMSReplyTo()` ausgelesen (vom *Replier*, s. Zeile 12 in Listing 5.6).

5.3.3 *Publish/Subscribe*

Beim Kommunikationsmuster *Publish/Subscribe* (P/S) wird eine Nachricht, die von einem JMS Client gesendet wird (im Folgenden *Publisher*), nicht nur von genau ei-

Listing 5.6: Gerüst für einen JMS Client als *Replier* nach dem Request/Reply Muster

```
1  public class EchoReplierNode implements MessageListener {
2      private Connection connection;
3      private Session session;
4      private MessageConsumer consumer;
5
6      // ...
7      @Override
8      public void onMessage(Message request) {
9          try {
10             if (request instanceof TextMessage) {
11                 TextMessage requestText = (TextMessage) request;
12                 Queue replyQueue = request.getJMSReplyTo();
13                 MessageProducer replyProducer = session.createProducer(
                       replyQueue);
14                 TextMessage reply = session.createTextMessage();
15                 reply.setText("echo: " + requestText.getText());
16                 replyProducer.send(reply);
17                 replyProducer.close();
18             }
19         } catch (JMSException e) {
20             System.err.println(e);
21         }
22     }
23     // ...
24 }
```

nem JMS Client empfangen wie bei PTP-Kommunikation, sondern die Nachricht des *Publishers* wird an alle «interessierten» Empfänger weitergeleitet (im Folgenden *Subscriber*). Es ergibt sich eine «1:n»-Verbindung bzw. eine «m:n»-Verbindung, wenn mehrere Publisher beteiligt sind wie in Abb. 5.4.

In Abb. 5.4 kommunizieren die JMS Clients nach dem P/S-Kommunikationsmuster über den JMS Provider. Die JMS Clients auf der linke Seite haben `MessageProducer`. Sie fungieren also als *Publisher*. Der obere Publisher sendet eine Nachrichten über die *Queue* «Topic 1», der untere über die Queue «Topic 4».

Auf der rechten Seite arbeiten drei JMS Clients als *Subscriber*. Sie holen mit ihrem jeweiligen `MessageConsumer` Nachrichten aus den *Queues* des JMS Providers. Der obere *Subscriber* ist an Nachrichten von «Topic 1» und «Topic 2» interessiert. In «Topic 1» ist eine Nachricht vom *Publisher* oben links vorhanden. Sie wird an den *Subscriber* zurückgesandt. In «Topic 2» ist in diesem Beispiel noch keine Nachricht eingetroffen. Deshalb gibt es keinen Rückfluss zum anfragenden *Subscriber*.

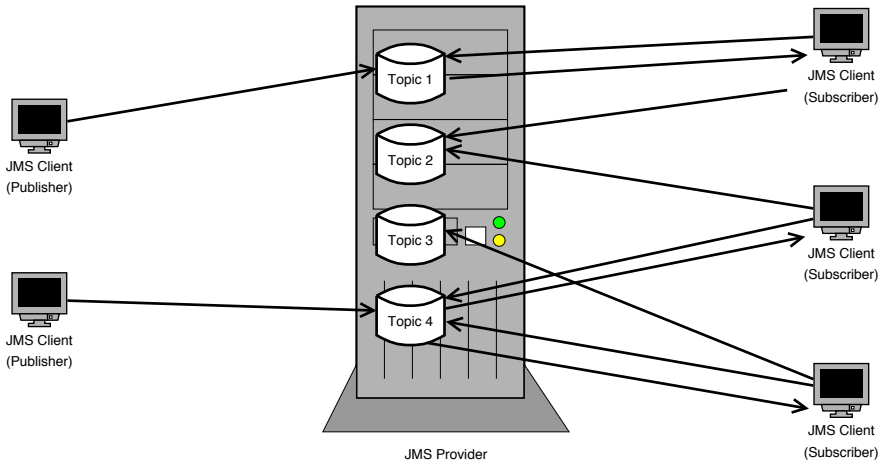


Abbildung 5.4: «m:n»-Verbindung beim Kommunikationsmuster *Publish/Subscribe*

Tabelle 5.1: Benennung der `Destination` in JNDI

Kommunikationsmuster	Benennung (JNDI-Ressourcenname)
Point To Point	`queue.*`
Request/Reply	
Publish/Subscribe	`topic.*`

Der mittlere *Subscriber* versucht Nachrichten aus «Topic 2» und «Topic 4» zu empfangen. In «Topic 4» ist eine Nachricht vom unteren *Publisher*, die an diesen *Subscriber* ausgeliefert wird.
Der untere *Subscriber* möchte Nachrichten aus «Topic 3» und «Topic 4» beziehen. In «Topic 4» ist immer noch die Nachricht vom unteren *Publisher*, die auch an diesen *Subscriber* ausgeliefert wird, da diese Nachricht durch das vorige Auslesen des mittleren *Subscribers* auf der rechten Seite nicht verbraucht wurde. Es ergibt sich daher eine «m:n»-Verbindung zwischen den JMS Clients.
Im Vergleich zu PTP verwaltet der JMS Provider die `Destination` bei P/S anders. Der Quelltext der *Publisher*- und *Subscriber*-JMS Clients bleibt beim PTP- und P/S-Kommunikationsmuster strukturell unverändert. Der Unterschied liegt darin, wie der JMS Provider konfiguriert wird. Dies geschieht über die Benennung der `Destination` in JNDI wie in Tab. 5.1 aufgelistet.

Listing 5.7: `jndi.properties` für gleichzeitige Verwendung der Kommunikationsmuster *Point To Point*, *Request/Reply* und *Publish/Subscribe*

```
1 java.naming.factory.initial = org.apache.activemq.jndi.
      ActiveMQInitialContextFactory
2 java.naming.provider.url = tcp://localhost:61616
3 queue.var.jms.ptp = VAR4733
4 queue.var.jms.rr  = VAR4734
5 topic.var.jms.ps  = VAR4735
```

Exkurs: Hierarchische Strukturierung von *Topics* bei MOM

JMS bietet zwar keine hierarchische Strukturierung von *Topics*. In vielen Anwendungen ist aber eine hierarchische Strukturierung wie bei Vererbung in der Objektorientierung wünschenswert: Je nachdem kann ein *Subscriber* sich dann dazu entscheiden, ein allgemeineres oder spezielleres *Topic* zu abonnieren. Ist das allgemeinere *Topic* abonniert, werden auch Nachrichten an die spezielleren *Topics* empfangen.
Dazu ist nicht unbedingt Objektorientierung erforderlich. In MQTT gibt es z. B. hierarchische *Topics*, deren Gliederung mit Einfachvererbung anhand der Benennung von Topics unterstützt wird.

Ein Beispiel für eine `jndi.properties`-Datei mit einer gemischten Verwendung der Kommunikationsmuster *Point To Point*, *Request/Reply* und *Publish/Subscribe* könnte beispielsweise aussehen wie in Listing 5.7.
Bei dieser Konfiguration würde Apache ActiveMQ als *JMS Provider* die `Destination queue.var.jms.ptp` und die `Destination queue.var.jms.rr` als `Queue` (also für 1:1-Kommunikation) verwenden. Wird eine Nachricht aus einer dieser `Destination` von einem JMS Client konsumiert, ist sie verbraucht und wird nicht an weitere JMS Clients ausgeliefert. `topic.var.jms.ps` würde hingegen als `Topic` behandelt werden: Eine Nachricht an diese `Destination` wird an alle `MessageConsumer` verteilt, die mit dieser `Destination` verbunden sind.
Die `Topic`-Warteschlangen erfüllen daher auch eine inhaltliche Funktion. Ein `Topic` kann eine thematische Eingrenzung bzw. Klassifizierung von Nachrichten sein.
Beispiele für *Publish/Subscribe*-Anwendungen sind Sensordaten-, Nachrichten- oder Börsenkursverteilung. Die Quelle (Sensor, Redakteur, Börsen-Broker) fungiert als *Publisher*, die Empfänger sind *Subscriber* (z. B. Displays, Entscheidungsalgorithmen).

Exkurs: *Durable Subscriber* bei JMS

Ein besonderer Vorteil von MOM ist, dass `MessageProducer` und `MessageConsumer` nicht gleichzeitig online und mit dem JMS Provider verbunden sein müssen. Das macht den Betrieb einfacher.
Um die Robustheit solch eines Systemdesigns noch weiter zu erhöhen können die `MessageConsumer` dauerhaft («*durable*») arbeiten. Das bedeutet, dass JMS Clients Nachrichten vom JMS Provider abrufen können, auch wenn der `MessageConsumer` zwischenzeitlich geschlossen wird bzw. abstürzt. Nachrichten, die über den JMS Provider verschickt werden, werden dann zwischengespeichert und später beim erneuten Verbinden an den `MessageConsumer` übermittelt.
Dazu müssen JMS Clients eine wiedererkennbare Identität bekommen. Für diesen Fall gibt es eine spezielle Methode, mit der ein solcher *Durable Subscriber* erzeugt werden kann:

```
Session session;
MessageConsumer consumer;
Topic topic;
//...initialisieren...
consumer = session.createDurableSubscriber(topic, name);
```

Der Parameter `name` ist ein String, mit dem die Identität des `MessageConsumer`-Objekts markiert wird. Es dürfen keine zwei Konsumenten denselben Namen verwenden. Wird der `consumer` geschlossen und später wieder unter demselben Namen am Topic angemeldet, werden alle Nachrichten, die in der Zwischenzeit eingegangen waren, zugestellt.

Praktikum

Aktivitätsschritte:

Unter `http://verteiltearchitekturen.de/vol01/VAR-JMS-solution.zip` können Sie eine Musterlösung zu diesem Praktikum herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-JMS_solution`.

Im folgenden Praktikum entwickeln Sie wieder einfache Client-/Server-Anwendungen. Die Kommunikation läuft aber nicht direkt, sondern über den JMS Provider Apache ActiveMQ nach dem JMS Standard. Sie werden also JMS Clients programmieren.
Während die bisherigen Praktika noch keine weitergehenden Anforderungen an die Entwicklungsumgebung gestellt hatten (eine Java Development Kit Installation ab 1.7 reichte aus, auf Eclipse hätte man im Zweifelsfall verzichten können), wird die Entwicklungsumgebung nun komplexer. Es wird ein externer JMS Provider benötigt, dessen Client Library in das Eclipse-Projekt eingebunden werden muss, und der als Server parallel zur Eclipse IDE laufen muss.

Die Erfahrung zeigt, dass es je nach eingesetztem Betriebssystem und Konfiguration ziemlich aufwändig sein kann alles zum Laufen zu bringen. Ein Großteil des Aufwands geht in die Konfiguration und das Zusammenspiel unterschiedlicher Frameworks. Die algorithmische Seite dieses Praktikums ist hingegen sehr einfach. Lassen Sie sich nicht frustrieren, wenn es nicht auf Anhieb läuft und bleiben Sie dran, bis Sie eine einsatzfähige Umgebung haben!

Wie schon in den vorigen Praktika könnten alle Prozesse auf unterschiedliche Rechner verteilt werden. So könnte z. B. der JMS Provider auf einem anderen Computer gestartet werden als die JMS Clients, die Sie in diesem Praktikum programmieren werden.

Aktivitätsschritt 5.1:
Ziel: Projekt in Eclipse importieren
Laden Sie das ZIP-Archiv mit dem Projekt `VAR-JMS` unter `http://verteiltearchitekturen.de/vol01/VAR-JMS.zip` herunter. Importieren Sie das ZIP-File in Eclipse mit der Funktion *File* → *Import...* → *General* / *Existing Projects Into Workspace*. Wählen Sie dort die Option *Select Archive File*. Sie sollten auch darauf achten, dass die Option *Copy Projects Into Workspace* aktiviert ist,

sonst arbeiten Sie möglicherweise nur auf den Dateien in Ihrem Download-Ordner und nicht in Ihrem Eclipse Workspace.

Für dieses Praktikum wird in Eclipse ein Projekt mit fünf Java Packages angelegt. Das Package `var.mom.jms.revers` ist allerdings anfangs noch leer. Sie werden es im Verlauf der Aktivitätsschritte in Abschnitt 5.6 füllen.
In Ihrem Eclipse Workspace Directory sollte ein neues Verzeichnis für das Projekt entstanden sein. In dem Projektverzeichnis gibt es mindestens einen Folder namens `src`, in dem die Packages als (Unter-)Verzeichnisse auftauchen. Abb. 5.5 zeigt die Struktur unterhalb des Folders `src/`. Neben den Verzeichnissen für die Java Packages gibt es in `src/` auch noch die Datei `jndi.properties`, die der Ausgangspunkt für Ihre JNDI-Konfiguration sein wird.

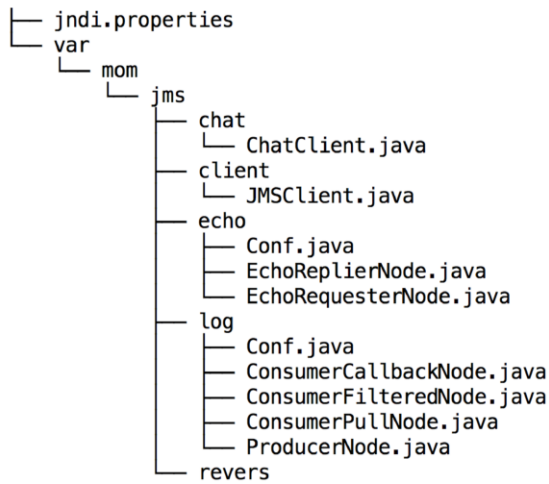


Abbildung 5.5: Verzeichnis- und Dateibaum für das Projekt `VAR-JMS`

5.4 Aufgabe: Installation von Apache ActiveMQ als JMS Provider

In diesem Praktikum wird Apache ActiveMQ als JMS Provider benutzt.
Falls Sie Apache ActiveMQ noch nicht haben, müssen Sie das Softwarepaket erst herunterladen und installieren.

Aktivitätsschritt 5.2:

Ziel: Feststellen, ob Java richtig für Apache ActiveMQ installiert ist

Da Apache ActiveMQ Java 8 verwendet, brauchen Sie auch ein *Java Runtime Environment* (ist Teil des *Java Development Kits*), das direkt in der Shell ausführbar ist. Es hängt vom Betriebssystem und der Installationsart ab, ob sich Java bei Ihnen «im Pfad befindet». Mit «Pfad» ist hier die Environment-Variable `$PATH` gemeint. Diese Umgebungsvariable wird vom Betriebssystem genutzt, um den Speicherort eines ausführbaren Programms zu finden, wenn es ohne Pfadnamen gestartet wird. Um zu prüfen, ob Java bei Ihnen läuft, öffnen Sie eine Shell (unter Windows können «Win» + «R» und im Eingabefeld `cmd` eingeben). In der dann erscheinenden Windows Eingabeaufforderung bzw. Shell können Sie versuchen das Java Runtime Environment zu starten, indem Sie Folgendes eingeben:

```
java -version
```

Sie sollten als Ergebnis des Aufrufs eine Meldung sehen, in der die Version Ihres Java Runtime Environments genauer beschrieben wird. Das kann beispielsweise so aussehen:

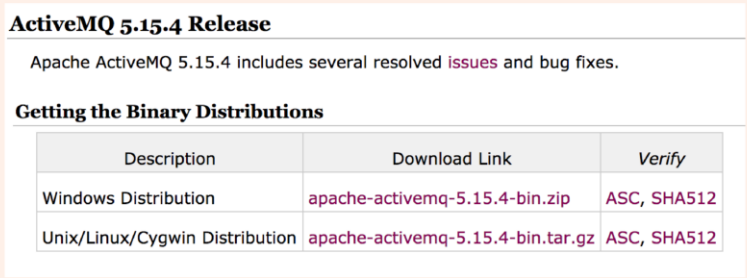
Die Versionsangabe «`1.8.0_112`» bedeutet, dass Java 8 installiert ist (JDK 1.8 entspricht Java 8).
Falls der Aufruf nicht funktioniert, ist Java wahrscheinlich nicht im Pfad. Oracle stellt eine Anleitung zum Setzen der $PATH-Umgebungsvariable für die Betriebssysteme Solaris SPARC, Red Hat Linux, SUSE Linux, Oracle Linux, Windows

10, Windows 8, Windows 7, Windows Vista, Windows XP und macOS bereit: `https://www.java.com/de/download/help/path.xml`.

Aktivitätsschritt 5.3 (falls nicht vorhanden):

Ziel: Apache ActiveMQ installieren

Laden Sie die aktuelle Version von Apache ActiveMQ von der Seite `http://activemq.apache.org/download.html` herunter («*latest stable release*»). Folgen Sie dort dem Link zur binär-Distribution. Abhängig von Ihrem Betriebssystem wählen Sie dann die passende Variante und laden Sie sie über den Download Link herunter (für macOS wählen Sie die Unix/Linux/Cygwin-Variante):



ActiveMQ 5.15.4 Release

Apache ActiveMQ 5.15.4 includes several resolved issues and bug fixes.

Getting the Binary Distributions

Description	Download Link	*Verify*
Windows Distribution	apache-activemq-5.15.4-bin.zip	ASC, SHA512
Unix/Linux/Cygwin Distribution	apache-activemq-5.15.4-bin.tar.gz	ASC, SHA512

Entpacken Sie das Archiv. Verschieben Sie das entpackte Verzeichnis aus Ihrem Download-Ordner, falls es dort entpackt wurde, an einen Platz, an dem das Verzeichnis längerfristig bleiben kann. Speichern Sie den Inhalt des Archivs aber nicht direkt in Ihrem Eclipse Workspace. Bei manchen Versionen von Apache ActiveMQ und auf einigen Betriebssystemen darf **kein Leerzeichen im Pfad zum entpackten Archiv** vorkommen.

Im hier gezeigten Beispiel liegt der Inhalt des Archivs im Verzeichnis `/opt/apache-activemq-5.15.4/`. In den folgenden Ausführungen wird dieser Pfad benutzt. Sollten Sie eine andere Version von ActiveMQ installiert oder das Verzeichnis an einen anderen Ort entpackt haben, müssen Sie stattdessen Ihren Pfad einsetzen.

Aktivitätsschritt 5.4:

Ziel: Apache ActiveMQ starten

Starten Sie den ActiveMQ *Message Broker*. Wechseln Sie dazu in das entsprechende Verzeichnis in `/opt/apache-activemq-5.15.4/bin`. Im Fall von Linux ist das `linux-x86-32` (für 32-Bit) oder `linux-x86-64` (für 64-Bit). Unter macOS wechseln Sie in das Verzeichnis `macosx`. Wenn Sie die Windows-Version heruntergeladen haben, bleiben Sie im Verzeichnis `/opt/apache-activemq-5.15.4/bin/` (**wechseln Sie nicht** in `win32` oder `win64`).

```
cd /opt/apache-activemq-5.15.4/bin/linux-x86-32
```

Starten Sie dort

```
./activemq start
```

Das «./» ist erforderlich, falls Sie nicht das «current working directory» («./») in der Environment-Variable `$PATH` haben (was auch nicht empfehlenswert wäre), um dem Betriebssystem mitzuteilen, dass das `activemq`-Programm aus dem aktuellen Verzeichnis gestartet werden soll.
Später kann der Server mit dem folgenden Kommando angehalten werden.

```
./activemq stop
```

Ob der ActiveMQ Server läuft, kann durch Zugriff auf die Management-Oberfläche nachgeprüft werden. ActiveMQ bietet (wie die meisten Middleware Server) einen Zugang über eine Web-Schnittstelle (per HTTP) an. Über `http://localhost:8161/admin` sollte auf dem Rechner, auf dem der Server läuft, auf die Konfiguration von Apache ActiveMQ zugegriffen werden können.
Damit Unbefugte nicht in die Konfiguration dieses zentralen Kommunikationsvermittlers eingreifen können, wird die Web-Schnittstelle durch eine Autorisierungsabfrage geschützt. Zur Autorisierung muss man sich als Administrator authentifizieren. Die ActiveMQ-Benutzer sind unabhängig von der Benutzerverwaltung des Rechners, auf dem der Server läuft. Stattdessen könnten die Benutzerinformationen für ActiveMQ Accounts in der ActiveMQ-Konfiguration geändert werden. Standardmäßig kann über den Benutzernamen `admin` mit dem Passwort `admin` zugegriffen werden.

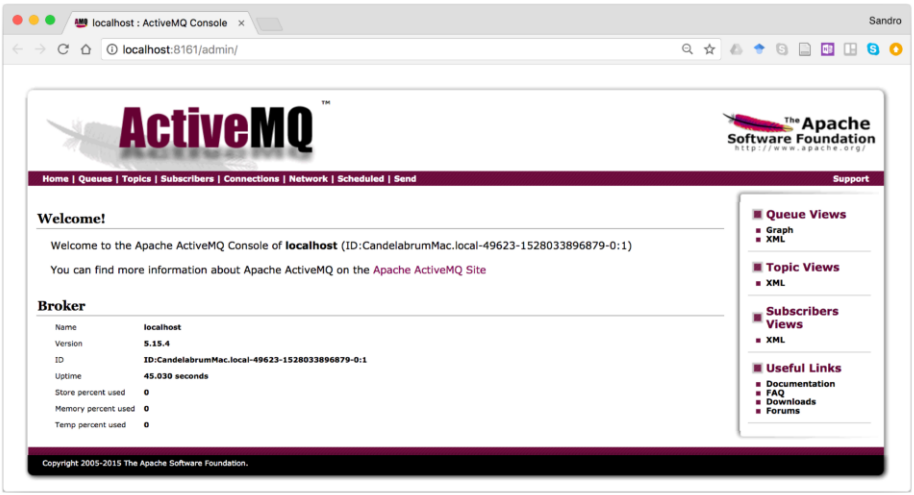


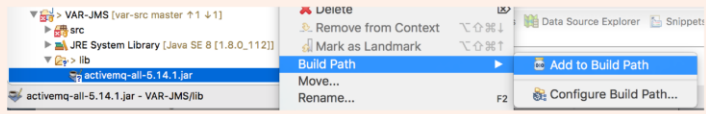
Abbildung 5.6: Management-Oberfläche von Apache ActiveMQ unter der Adresse `http://localhost:8161/admin` (Benutzer-Kennung: `admin`/`admin`)

Aktivitätsschritt 5.5:
Ziel: Testen, ob Apache ActiveMQ läuft
Öffnen Sie mit einem Webbrowser die Adresse `http://localhost:8161/admin` und melden Sie sich mit der Benutzer-Kennung `admin`/`admin` an. Sie sollten daraufhin die Management-Oberfläche des *Brokers* sehen wie sie in Abb. 5.6 dargestellt ist.

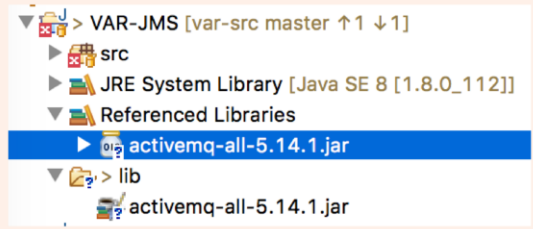
Außerdem muss eine Library von den JMS Clients eingebunden werden, die die spezifischen Kommunikationsmechanismen der JMS Provider Implementierung umsetzt. Im Fall von ActiveMQ ist das `activemq-all-5.15.4.jar`. Dieses JAR enthält neben der eigentlichen Client Library auch alle anderen Libraries, die für die Verwendung von ActiveMQ erforderlich sind. Dazu gehören auch die JMS Interfaces, sodass mit dieser Library auch der Zugriff von der *Java Platform, Standard Edition* (Java SE) aus möglich ist, wo diese JMS Interfaces nicht mitgeliefert werden.

Aktivitätsschritt 5.6:
Ziel: Apache ActiveMQ Client Library in Eclipse integrieren
Erzeugen Sie einen neuen *Folder* `lib` im Wurzelverzeichnis Ihres Eclipse-

Projektes (*File* → *New* → *Folder* oder *File* → *New* → *General/Folder*). Kopieren Sie das «Über-JAR» `/opt/apache-activemq-5.15.4/activemq-all-5.15.4.jar` (bzw. mit angepasster Versionsnummer/Pfad; wichtig ist, dass es das `activemq-all-*.jar` ist.) in den neuen *Folder* `lib`. In dem JAR sind die JMS Interfaces und alle Apache ActiveMQ spezifischen Implementierungsklassen von `javax.jms.*` sowie deren externe Abhängigkeiten enthalten. Obwohl sich das JAR nun im Projekt befindet, ist es noch nicht nutzbringend integriert: Der Compiler kann noch nicht darauf zugreifen. Dazu ist noch erforderlich, dass dieses JAR dem `$CLASSPATH` des Projekts hinzugefügt wird. Es gibt unterschiedliche Wege dies zu erreichen. Das Ziel ist immer, dass das JAR Teil des «Build Path»'s wird. Eine Möglichkeit dafür ist im Kontextmenü des JARs (öffnen durch Rechts-Klick auf das JAR im *Package Explorer*) *Build Path* → *Add To Build Path* zu klicken.



Im Ergebnis wird das JAR als *Referenced Library* im Projekt integriert.



Die Fehlermeldungen im Quellcode der JMS Clients sollten nun verschwunden sein. Falls Sie weiterhin viele Fehlermeldungen sehen, die sich auf die ganzen JMS-spezifischen Klassen beziehen, versuchen Sie das Projekt neu zu «builden»: *Project* → *Build Project*.

Sie können die Bibliothek am Dreiecks-Icon in den *Referenced Libraries* (nicht am JAR in `lib/`) aufklappen und inspizieren, welche Klassen und Interfaces Sie in Ihr Projekt integriert haben. Alle diese Klassen können Sie nun in Ihren Java-Quelltexten nutzen, indem Sie sie bspw. importieren.

Exkurs: Über-JAR

Ein Java Archive, das alle externen Abhängigkeiten zu einer Library bzw. einem Framework enthält, wird «Über-JAR», bzw. mangels Umlaut-Taste auf internationalen Tastaturen «Uber JAR» genannt. Damit ist gemeint, es seit *über*geordnet, da alles enthalten ist.
Die Client Library von Apache ActiveMQ hat eine Reihe von Abhängigkeiten zu externen Paketen. Im Verzeichnis `/opt/apache-activemq-5.15.4/lib` finden sich diese Libraries. Damit sie nicht alle einzeln dem Classpath hinzugefügt werden müssen, wird die Vereinigungsmenge der externen und ActiveMQ-eigenen Bibliotheken als Über-JAR `/opt/apache-activemq-5.15.4/activemq-all-5.15.4.jar` bereitgestellt.

5.5 Aufgabe: Logger Service

Der Logger Service ist im Package `var.mom.jms.log`.

Aktivitätsschritt 5.7:
Ziel: Analyse des Programmablaufs
Analysieren Sie die einzelnen Schritte der Klassen in `var.mom.jms.log`. Beantworten Sie sich die folgenden Fragen:

- Welche Bedeutung haben die Kommandozeilenparameter beim `ProducerNode`?
- Wann beendet der `ProducerNode` die Verbindung zum Server?
- An welcher Stelle wird `jndi.properties` verwendet?
- Wie unterscheiden sich die *Consumer* voneinander?

Aktivitätsschritt 5.8 (falls noch nicht geschehen):
Ziel: Start der Middleware
Starten Sie Apache ActiveMQ Server oder konfigurieren Sie `jndi.properties`, sodass ein verfügbarer (und erreichbarer) ActiveMQ Server verwendet wird.

Aktivitätsschritt 5.9:
Ziel: JMS Clients miteinander kommunizieren lassen
Lassen Sie den Server laufen und lassen Sie *Producer* und *Consumer* in unterschiedlicher Reihenfolge und auch gleichzeitig laufen — nach Möglichkeit von unterschiedlichen Rechnern. Beantworten Sie sich die folgenden Fragen:

- Was sind die Unterschiede zwischen den unterschiedlichen *Consumern*?
- Muss ein `Consumer*` laufen, damit das Programm funktioniert?

5.6 Aufgabe: Textumdreher-Service

In dieser Aufgabe werden Sie die JMS Clients `Anfrager` und `Umdreher` für den *Textumdreher-Service* entwickeln. Sie werden dabei die Quelltexte aus der vorherigen Aufgabe als Ausgangspunkt nehmen und sie Schritt für Schritt anpassen.
Der `Anfrager` soll Textanfragen an den `Umdreher` schicken, die der `Umdreher` beantworten soll. Die Textanfragen sind beliebige `String`-Objekte. Der `Umdreher` dreht jede empfangene Zeichenkette Zeichen für Zeichen um und sendet sie zurück. Der `Umdreher` soll die empfangenen Nachrichten zusätzlich auf der Konsole ausgeben. Der `Anfrager` erhält die umgedrehte Zeichenkette zurück und gibt sie ebenfalls auf der Konsole aus.

Aktivitätsschritt 5.10:
Ziel: Gerüst für zwei JMS Clients erstellen
Erzeugen Sie auf der Basis der Quelltexte des Pakets `var.mom.jms.log` zwei neue Klassen `Anfrager` und `Umdreher` im Paket `var.mom.jms.revers`, die Nachrichten untereinander austauschen können. Im Moment können Sie den `Anfrager` aus dem `EchoRequesterNode` übernehmen und den `Umdreher` aus `EchoReplierNode`.

Aktivitätsschritt 5.11:
Ziel: Neue Queue zum Nachrichtenaustausch (Request) einrichten
Richten Sie zum Nachrichtenaustausch eine neue *Queue* ein und passen Sie

Listing 5.8: zur `Conf`-Konstante unten passende `jndi.properties` für Requests an `Umdreher`

```
1 java.naming.factory.initial = org.apache.activemq.jndi.
      ActiveMQInitialContextFactory
2 java.naming.provider.url = tcp://localhost:61616
3 queue.var.mom.jms.rev = requestsForReversService
```

Listing 5.9: zur JNDI-Konfiguration passende `Conf`-Konstante `QUEUE` für Bezeichnung der Destination für Requests an `Umdreher`

```
1 class Conf {
2     public static final String QUEUE = "var.mom.jms.rev";
3 }
4 // ...
5 Queue queue = (Queue) ctx.lookup(Conf.QUEUE);
```

Ihre Klassen `Anfrager` und `Umdreher` an. Editieren Sie dazu die Datei `jndi.properties` im `src/`-Wurzelverzeichnis des Eclipse-Projekts.
Es wäre sinnvoll, wenn die `String`-Konstanten, mit denen die *Queues* bezeichnet werden, wie bei `var.mom.jms.log.Conf` in eine ausgelagerte Klasse verschoben werden.

Die Benennung der *Queue* (links von =) in `jndi.properties` muss wie in Listing 5.8 mit «`queue.`» beginnen, eindeutig sein und sollte als Wert (rechts neben dem =) einen neuen Namen verwenden.
Der Rest der Benennung (das, was nach `queue.` links vom = kommt) muss im JMS Client-Programm für die Verwendung der *Queue* angegeben werden (wie in Listing 5.9).

Aktivitätsschritt 5.12:

Ziel: `Anfrager` aus `EchoRequesterNode` umformen

Im `Anfrager` sollen in einer Endlosschleife Texte von der Konsole eingelesen werden und an den `Umdreher` gesendet werden. Das zeilenweise Einlesen von Eingaben und Versenden soll im Hauptprogramm nach dem Erzeugen des `node`-Objekts erfolgen.

```
BufferedReader input = new BufferedReader(new InputStreamReader(
    System.in));
String line;
while (true) {
    line = input.readLine();
    node.sendMessage(line);
}
```

Da `readLine()` eine `IOException` liefern kann, muss diese Situation behandelt werden. Bspw. kann dieser Fall im Multi Catch hinzugefügt werden.

Aktivitätsschritt 5.13:

Ziel: «Umdrehen»-Funktionalität im `Umdreher` bereitstellen

In `Umdreher` wird eine Methode benötigt, die einen String umdreht. Eine einfache Möglichkeit wäre:

```
public String revers(String in) {
    String out = "";
    for (char c : in.toCharArray()) {
        out = c + out;
    }
    return out;
}
```

Implementieren Sie eine `String`-Umdreher Methode in `Umdreher`.
Verwenden Sie die Funktion beim Zurücksenden der Antwort.

Aktivitätsschritt 5.14:

Ziel: `Anfrager` Antwort vom `Umdreher` empfangen lassen

Der `Anfrager` soll die Antworten asynchron empfangen. Die Methode `receiveAndPrintMessages()` und ihr Aufruf im Hauptprogramm können entfallen – sie waren für den synchronen Empfang gedacht. Implementieren Sie stattdessen das `MessageListener` Interface beim `Anfrager`. Sie können die Methode `onMessage` aus `ConsumerCallbackNode` übernehmen. Vergessen Sie nicht die `@Override`-Annotation und damit einhergehend, dass die Klasse `Anfrager` mit `implements` spezifiziert, dass sie das `MessageListener`

Interface implementiert. Im Konstruktor von `Anfrager` sollte dann vor dem `start()` der *Listener* bekanntgemacht werden:

```
consumer.setMessageListener(this);
```

Die empfangenen Nachrichten werden diesmal kein *Property* für die Priorität haben. Der entsprechende Abfrage und Ausgabe in `onMessage` muss entfernt werden.

Aktivitätsschritt 5.15 (falls noch nicht geschehen):
Ziel: Start der Middleware
Starten Sie Apache ActiveMQ Server oder konfigurieren Sie `jndi.properties`, sodass ein verfügbarer (und erreichbarer) ActiveMQ Server verwendet wird.

Aktivitätsschritt 5.16:
Ziel: `Umdreher` und `Anfrager` laufen lassen
Starten Sie mehrere `Anfrager` und einen `Umdreher`. Achten Sie darauf, dass die Anfrage eines *Requesters* auch wirklich nur bei ihm ankommt und nicht bei anderen *Requestern*. Falls das doch geschieht, sollten Sie prüfen, ob der `Anfrager` vor dem Versenden seiner `TextMessage`-Objekte eine temporäre *Reply Queue* (`session.createTemporaryQueue()`) erzeugt und mit der Methode `setJMSReplyTo(...)` am `Message`-Objekt setzt. Entsprechend muss der `Umdreher` für jeden Requester einen neuen `MessageProducer` für die individuelle *Reply Queue* erzeugen: Dazu muss er mit der Methode `getJMSReplyTo()` die *Reply Queue* abfragen.
Modifizieren Sie die *Temporary Queue* und prüfen Sie wieder mit mehreren *Requestern* das Verhalten:

- Alle *Requester* verwenden dieselbe *Queue* statt jeweils eigene mit `session.createTemporaryQueue()` erzeugte. Die Anfrage-*Queue* bleibt davon getrennt.
- Alle *Requester* und der *Replier* verwenden genau eine *Queue*.

Aktivitätsschritt 5.17 (fakultativ):
Ziel: `Umdreher` durch Liste der Requester verbessern
Im Moment erzeugt der `Umdreher` für jeden `Anfrager` so viele `MessageProducer`-Objekte wie dieser *Requests* sendet, obwohl alle *Requests* dieselbe *Temporary Queue* benutzen. Speichern Sie die bereits erzeugten `MessageProducer`-Objekte in einer `HashMap` mit der *Temporary Queue* als Schlüssel und prüfen Sie vor dem Erzeugen eines neuen `MessageProducer`-Objekts, ob bereits eines zur Wiederverwendung vorhanden ist. Wie könnte man erreichen, dass die `HashMap` nicht streng monoton wächst, sondern dass inaktive *Requester* gelöscht werden?

5.7 Aufgabe: Chat Service (*Publish/Subscribe*)

Mit einem JMS Provider lassen sich unterschiedliche Kommunikationsmuster sehr einfach umsetzen. Im folgenden Beispiel wird das *Publish/Subscribe*-Muster für eine einfache Chat-Anwendung benutzt. Sie werden feststellen, dass im Vergleich zu den bisherigen JMS Clients nur wenige Änderungen erforderlich sind um eine grundlegend andere Art von Kommunikationsverhalten zu erzeugen.
Für die Chat-Anwendung wird ein JMS Client benötigt, der sowohl einen `MessageProducer` als auch einen `MessageConsumer` zum Empfang benötigt. Das abonnierte *Topic* ist der gemeinsame Kommunikationskanal: Alle Nachrichten werden über das *Topic* empfangen und gesendet. Der JMS Provider stellt keine weitere Funktionalität bereit. Sollten mehr Kanäle bzw. Chat-Räume benutzt werden, müssten weitere *Topics* definiert werden. Im folgenden Beispiel wird aber nur genau ein *Topic* verwendet um unübersichtlichen Code für die Kanalauswahl und -verwaltung zu vermeiden.

Aktivitätsschritt 5.18:
Ziel: `ChatClient` aus `JMSClient` erstellen
Da ein JMS Client mit `MessageProducer` und `MessageConsumer` benötigt wird, der asynchron Nachrichten vom JMS Provider empfangen kann, scheint `JMSClient` ein guter Startpunkt zu sein. Erzeugen Sie im Paket `var.mom.jms.chat` die Klasse `ChatClient` auf der Basis des Codes von `JMSClient`

(kopieren Sie vorerst der Einfachheit halber Code der Klasse `JMSClient` statt Vererbung zu benutzen).

Alle ChatClients verwenden zum Senden und Empfangen dieselbe `Destination`, die aufgrund des *Publish/Subscribe*-Kommunikationsmusters als `Topic` spezifiziert werden muss. In `jndi.properties` ist das *Topic* bereits vorbereitet:

```
topic.var.mom.jms.channel1 = topic4735
```

Aktivitätsschritt 5.19:
Ziel: Spezifikation des Topics als `Destination` für den `ChatClient`
Damit dieses *Topic* verwendet wird, muss der String «`var.mom.jms.channel1`» beim Auffinden der `Destination` über das JNDI angegeben werden. Aufgrund des Aufbaus von `JMSClient` reicht es, diesen String über den Konstruktor sowohl als `sendDest` als auch als `receiveDest` Parameter zu übergeben:

```
String topic = "var.mom.jms.channel1";
node = new ChatClient(topic, topic);
```

Aktivitätsschritt 5.20 (falls noch nicht geschehen):
Ziel: Start der Middleware
Starten Sie Apache ActiveMQ Server oder konfigurieren Sie `jndi.properties`, sodass ein verfügbarer (und erreichbarer) ActiveMQ Server verwendet wird.

Aktivitätsschritt 5.21:
Ziel: Chat Clients miteinander kommunizieren lassen
Starten Sie mehrere Chat Clients mit unterschiedlichem Nutzernamen. Nach Möglichkeit auf unterschiedlichen Rechnern. Dabei müssen die jeweiligen `jndi.properties` so konfiguriert sein, dass alle dasselbe *Topic* auf demselben JMS Provider verwenden.

Aktivitätsschritt 5.22:
Ziel: `ChatClient` personalisieren
Beim Testlauf wird aufgefallen sein, dass die einzelnen Teilnehmer des Chats in den Messages nicht voneinander zu unterscheiden sind. Übergeben Sie einen Namen als Kommandozeilenparameter und fügen Sie ihn den `TextMessage`-Objekten vor dem Versand über das *Topic* hinzu. Sie können entweder den Namens-`String` mit der von der Konsole gelesenen Zeile konkatenieren oder der `TextMessage` ein *String Property* für den Namen hinzufügen. Das zweite Vorgehen ist zwar etwas aufwändiger, bietet dafür aber mehr Möglichkeiten z. B. für unterschiedliche Formatierung von Benutzernamen und Nachricht.

5.8 Ausblick und Anregungen für eigene Projekte

JMS als Beispiel für ein nachrichtenbasiertes Kommunikationssystem ist eine effektive Grundlage für leistungsfähige verteilte Systemarchitekturen. Im Folgenden werden Ideen für weitergehende Projekte vorgestellt, die JMS benutzen.

5.8.1 Skalierbare und robuste verteilte Architekturen mit JMS

Mit JMS kann man sehr einfach eine gute Skalierbarkeit und Robustheit erreichen. Im folgenden Projektvorschlag wird das *Point-To-Point*-Muster für die Verteilung von Arbeitsaufträgen in einem Netzwerk benutzt. Dabei werden Arbeitsaufträge in eine *Queue* auf einem JMS Provider geschrieben. JMS Clients, die auf unterschiedlichen Rechnern verteilt sind, konsumieren Arbeitsaufträge aus der *Queue* und nehmen diese Nachrichten als Input für ihre Berechnung. So kann eine größere Aufgabe effektiv auf einzelne Arbeitsknoten verteilt werden. Diese Architektur kann in sehr großem Maß um Rechenressourcen erweitert werden, indem einfach mehr Rechner mit jeweils einem (oder anhand der Anzahl der Cores auch mehreren) JMS Clients hinzukommen, die auch auf Arbeitsaufträge aus der *Queue* zugreifen.
Entscheiden Sie sich für eine rechenintensive Aufgabe, die gut beschreibbare Eingangsgrößen hat. Es wäre nützlich, wenn sich der Problemraum gut in einzelne Teilaufgaben zerlegen ließe. Ein Beispiel wären *Brute Force*-Angriffe beim Passwort-Knacken. Das Problem lässt sich unter bestimmten Annahmen auf die Berechnung einer Hash-Funktion für einen Passwortkandidaten mit anschließendem String-Vergleich

Listing 5.10: Bereitstellen von Arbeitsaufträgen für *Worker* in Form von `TextMessage`-Objekten

```
1  //...
2  final byte[] target = DatatypeConverter.parseHexBinary("865
       ae56c298c677e9d4987fe13268fa915bd041646526c595d293d5448481443");
3  final MessageDigest dig = MessageDigest.getInstance("SHA-256");
4  //...
5
6  @Override
7  public void onMessage(Message message) {
8      if (message instanceof TextMessage) {
9          TextMessage task = (TextMessage) message;
10         try {
11             byte[] hash = dig.digest(task.getText().getBytes());
12             if (Arrays.equals(hash, target)) {
13                 System.out.println("Lösung gefunden: " + textMessage.
                       getText());
14             }
15         } catch (JMSException e) {
16             System.err.println(e);
17         }
18     }
19 }
```

reduzieren. Die Berechnung eines einzelnen Hash-Wertes einer Passwort-Zeichenkette ist nicht besonders aufwändig, verbraucht aber doch einige Zeit (bei SHA-256 z. B. ca. 25 CPU-Zyklen pro Byte). Da man sehr viele Passwort-Kandidaten mit der Hash-Funktion verschlüsseln und das Ergebnis mit einem vorgegebenen String vergleichen muss, wäre eine Parallelisierung mit mehreren Rechnern wünschenswert.

Aktivitätsschritt 5.23 (fakultativ):
Ziel: Implementierung der Worker-Funktionalität
Implementieren Sie einen JMS Client als *Worker*, der Arbeitsaufträge aus einer *Queue* eines JMS Providers in Form von `ObjectMessage`-Objekten liest. Der Inhalt der `ObjectMessage` sollte der Eingangsparameter für die rechenintensive Funktion sein. Im Fall des SHA-256 Hashs würde auch eine `TextMessage` wie in Listing 5.10 passen.

Aktivitätsschritt 5.24 (fakultativ):
Ziel: Implementierung des Taskers

Die Systemarchitektur sieht vor, dass Arbeitsaufträge in einer zentralen *Queue* abgelegt werden, aus der sie von den *Workern* konsumiert werden. Es gibt unterschiedliche Möglichkeiten, woher die Aufträge kommen. Einfach und besonders übersichtlich wäre es, wenn die Aufträge von einem JMS Client generiert werden und von dessen `MessageProducer` in der Auftrags-*Queue* abgelegt werden, wie in Listing 5.11 skizziert.

Aktivitätsschritt 5.25 (fakultativ):
Ziel: Zusammenspiel von Tasker und Workern: Erfolg signalisieren
Wie kann man nun alle Teile so zusammenbringen, dass sie auch koordiniert zusammenspielen? Wieder bietet JMS bereits ein Mittel dafür. Es kann einfach ein zweiter Kommunikationskanal für die Signalisierung einer erfolgreichen Suche per *Publish/Subscribe* etabliert werden: Auf dem JMS Provider wird noch ein *Topic* zur Ergebnisübermittlung vorgesehen. Alle JMS Clients (*Tasker* und *Worker*) müssten dieses *Topic* abonnieren. Ein erfolgreicher *Worker* würde dann zusätzlich zur Konsolenausgabe eine Nachricht mit dem gefundenen Ergebnis über dieses *Topic* veröffentlichen, sodass alle JMS Clients davon informiert werden würden. Wenn das Signal empfangen wird, müsste eine globale Bool'sche Variable (z. B. `targetFoundFlag` im *Tasker*-Quellcode oben) gesetzt werden, die den Programmablauf steuert.
Bei der bisherigen Software-Architektur der *Worker* kommen sich möglicherweise die beiden Kommunikationskanäle in die Quere, wenn beide mit demselben `MessageListener`-Objekt asynchron ausgelesen werden. Eine Lösung wäre zwei innere Klassen mit dem `MessageListener` Interface bereitzustellen, die dann getrennt empfangen würden.

Listing 5.11: Aufträge für mit einem *Tasker* generieren und über eine Queue bereitstellen

```
1 try {
2     node = new ProducerNode("requests.queue");
3     while(!targetFoundFlag && taskIterator.hasNext()) {
4         node.send(taskIterator.next());
5     }
6 } // Ausnahmebehandlung und Aufräumen
```

Listing 5.12: Regelmäßige (alle 100 Nachrichten) Protokollierung des aktuellen Zeitstempels auf der Konsole (`System.out`)

```
1  //...
2  int messageCounter = 0;
3  //...
4
5  @Override
6  public void onMessage(Message message) {
7      if(messageCounter++ >= 99) {
8          System.out.println(System.currentTimeMillis());
9          messageCounter = 0;
10     }
11     // ...
12 }
```

Aktivitätsschritt 5.26 (fakultativ):
Ziel: Performance Tests
Die Leistungsfähigkeit und vor allem Skalierbarkeit solch einer Architektur sollte empirisch geprüft werden. Lassen Sie das System verteilt laufen. Passen Sie dazu den JNDI-Eintrag für die `java.naming.provider.url` an. Er darf nicht mehr `localhost` enthalten, sondern muss auf die Adresse des JMS Providers verweisen.
Protokollieren Sie die Performance in Arbeitseinheiten pro Zeiteinheit (z. B. SHA-256 Hashes pro Sekunde) in Abhängigkeit der JMS Clients pro Rechner und der Anzahl der Rechner. Bauen Sie dazu in die Worker eine Protokollierungsroutine ein. Protokollieren Sie beispielsweise alle 100 Messages die aktuelle Zeit win in Listing 5.12.

Abhängig von Problem und Konfiguration ist es interessant abzuschätzen, ab wann das Netzwerk und die Fähigkeiten des JMS Providers die Leistungsgrenze darstellen.

5.8.2 Ein Framework für JMS-basierte Anwendungen

In den vorigen Aufgaben wurde ein Grundprinzip der Softwareentwicklung verletzt: «Don't repeat yourself»! Entgegen dieser Regel wurden hier die immer gleichen Bausteine für die Initialisierung und Verwendung von `MessageConsumer` und `MessageProducer` in unterschiedliche, aber strukturell sehr ähnliche JMS Clients kopiert.

Aktivitätsschritt 5.27 (fakultativ):

Ziel: Entwurf eines Frameworks für JMS Clients

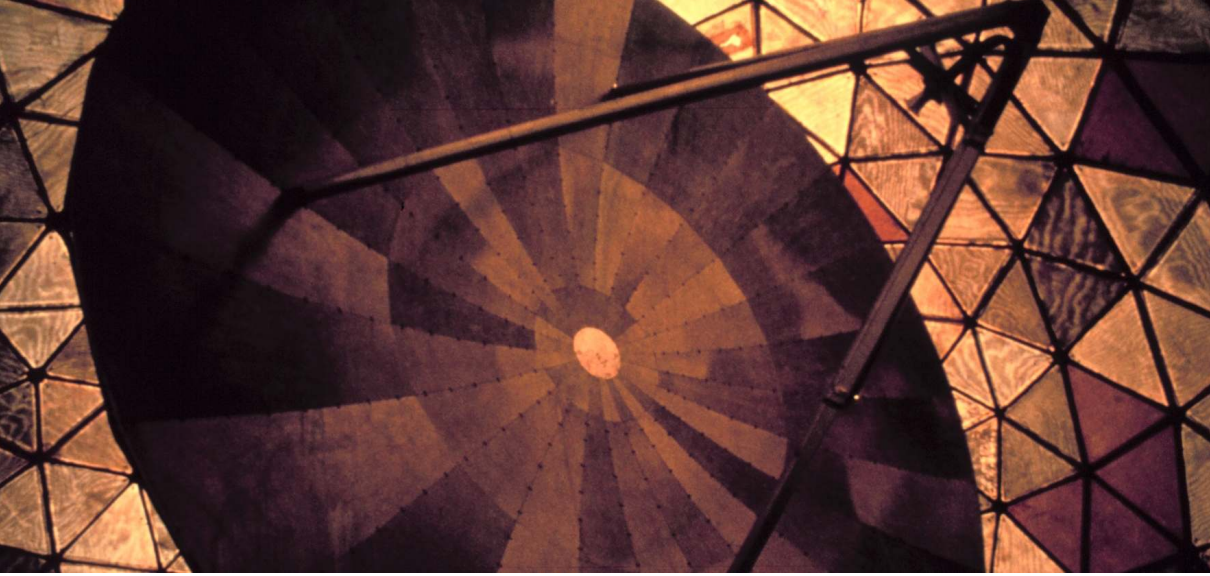
Sammeln Sie die Gemeinsamkeiten und vor allem die Unterschiede aus den konkreten Beispielen für *Logger*, *Echo*, *Umdreher* und *Chat* und entwerfen Sie daraus ein Framework, aus dem mindestens diese unterschiedlichen JMS Clients ohne Code-Kopieren abgeleitet werden können.

Tipp: Die Unterschiede zwischen den JMS Clients geben Ihnen einen Hinweis darauf, welche Erweiterungspunkte oder alternative Realisierungen solch ein Framework enthalten sollte. Scheuen Sie sich nicht das Framework vielen Refactoring-Schritten zu unterziehen, bis Sie wirklich zufrieden sind und Sie der Meinung sind, dass es gut einsetzbar wäre.

Aktivitätsschritt 5.28 (fakultativ):

Ziel: Qualitätskriterien für Architekturen und Frameworks

Besorgen Sie sich die internationale Norm ISO/IEC 25000 «*Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*» bzw. DIN ISO/IEC 25000 «Software-Engineering – Qualitätskriterien und Bewertung von Softwareprodukten (SQuaRE) – Leitfaden für SQuaRE» und definieren Sie anhand dessen die Qualitätsmerkmale, die aus Ihrer Sicht für Ihr Framework relevant sind. Sie können auch noch gute Hinweise aus der Vorgängernorm ISO/IEC 9126 bzw. DIN 66272 ziehen, falls sie Ihnen leichter zugänglich sein sollte.



Message Queue Telemetry Transport (MQTT)

MQTT ist ein Standard für Middleware, der auf Kommunikation in Systemarchitekturen mit beschränkten Ressourcen z. B. im *Internet of Things* abzielt. Er ist wegen seiner Einfachheit und einigen seiner Eigenschaften besonders geeignet für wenig verlässliche Netzwerke (hohe Latenz, geringe Bandbreite, eingeschränkte Verfügbarkeit) oder wenig leistungsfähige Knoten (insb. Embedded Devices/*Cyber Physical Systems*), die wenig RAM oder CPU-Performanz haben.

Wie bei JMS zielt MQTT als Protokoll für nachrichtenbasierte Kommunikation darauf ab, Knoten möglichst lose miteinander zu koppeln, asynchrone Benachrichtigungen zu ermöglichen, eine hohe Verlässlichkeit der Kommunikation zu garantieren und den Betrieb zu erleichtern, indem die Abhängigkeiten zwischen *Producern* und *Consumern* über einen *Broker* gemanagt werden. Im Gegensatz zu JMS ist *Publish/Subscribe* das einzige Kommunikationsmuster, das von MQTT unterstützt wird.

Der Standard MQTT beschreibt nur das Protokoll und die Formate der Nachrichten, die zwischen den Knoten zur Kommunikation verwendet werden. Im Gegensatz zum JMS ist die Programmierschnittstelle aber nicht standardisiert (bei JMS ist es das API

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

Exkurs: Ursprung und Standardisierung von MQTT

MQTT wurde ursprünglich von der Firma IBM für ein Projekt zur Pipelineüberwachung mit Sensornetzen entwickelt, inzwischen ist das Protokoll von OASIS standardisiert[a] (Port: `1883`, Transportprotokoll: TCP).

[a] `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`

im Paket `javax.jms`). Für MQTT gibt es hingegen unterschiedliche *Bindings* für viele Programmiersprachen wie Java, C und Python.

6.1 Topics

In MQTT haben *Topics* eine andere Bedeutung und werden anders verwaltet als in JMS. Jede Nachricht ist genau einem *Topic* zugeordnet. *Topics* können hierarchisch gegliedert sein. Die Gliederung spiegelt sich ausschließlich in der Benennung des *Topics*, einem `String` wieder: Im *Topic*-Namen können *Topic Level* getrennt durch den *Topic Level Separator* «/» verwendet werden:

```
daheim/erdgeschoss/wohnzimmer/temperatur
USA/California/SanFrancisco/SiliconValley
```

Die *Topic Level* dienen nicht nur der Organisation, sondern sie können durch die *Wildcard*-Symbole «+» (*single level*) und «#» (*multi level*) ersetzt werden.
Das «+»-*Wildcard* ersetzt genau ein *Topic Level*. Das *Topic*...

```
daheim/erdgeschoss/+/temperatur
```

kann also bspw. zu den folgenden konkreten *Topics* expandiert werden:

```
daheim/erdgeschoss/wohnzimmer/temperatur
daheim/erdgeschoss/kueche/temperatur
```

Das «#»-*Wildcard* kann auch mehrere *Topic Level* ersetzen. Das *Topic*...

```
daheim/#/temperatur
```

kann deshalb bspw. zu den folgenden konkreten *Topics* expandiert werden:

```
daheim/erdgeschoss/kueche/temperatur
daheim/obergeschoss/arbeitszimmer/temperatur
```

Neben den anwendungsspezifisch vergebenen *Topic*-Namen gibt es in vielen MQTT-*Middleware*-Implementierungen auch *interne Topics*. Sie können nicht von den Anwendungen angelegt werden, sondern sind fest vorgegeben. Darüber können Leistungsparameter der Middleware abgefragt werden. Interne Topics fangen grundsätzlich mit dem Symbol «$» an. Sie sind nicht standardisiert. Oft wird in MQTT-Middleware-Implementierungen der *Topic*-Name «$SYS/» für *interne Topics* verwendet:

```
$SYS/broker/clients/connected
$SYS/broker/uptime
```

Die internen mit «$» beginnenden *Topics* werden anders gehandhabt als die regulären. Ein Abonnement von allen Topics («#») schließt bspw. nicht die *internen Topics* mit ein.
Die dc-square GmbH, Hersteller des MQTT-fähigen MOM Servers HiveMQ, nennt in einem Blogbeitrag u. a. folgende *Best Practice*-Hinweise zum Umgang mit MQTT-*Topics*:

- *Topics* sollten keinen «/» am Beginn haben: Sie sähen dann zwar wie UNIX-Dateinamen aus, hätten aber am Beginn ein eigentlich überflüssiges (weil leeres) *Topic Level*.
- *Topic*-Namen sollten keine Leerzeichen beinhalten: Wie in Dateinamen erschwerten Leerzeichen die Notation von Namen und erforderten in vielen Kontexten eine Quotierung. Außerdem können unterschiedliche Arten von Whitespaces untereinander verwechselt werden, sodass die Fehlersuche ggf. erschwert werde.
- Aufgrund des Einsatzzwecks auf beschränkten *Embedded Devices* sollte die Benennung von Topics sparsam sein: Es sollten also möglichst kurze, jedoch prägnante Namen verwendet werden.
- Um Kompatibilität mit internationalen Umgebungen zu wahren, sollten nur druckbare 7-Bit ASCII Zeichen in *Topic*-Namen verwendet werden.
- Aufnehmen eines *Identifiers* oder einer *ClientId* in den *Topic*-Namen zur Trennung von Nachrichten unterschiedlicher Anwendungen.
- Man sollte in Clients nicht «#» abonnieren, denn der Durchsatz von Nachrichten könnte größer sein als die Verarbeitungskapazität solch eines Clients. Sollen tatsächlich alle Nachrichten verarbeitet werden, die über den *Broker* laufen (z. B. zur Archivierung), empfiehlt es sich stattdessen den *Broker* um eine entsprechende Funktionalität zu erweitern.
- Die Erweiterbarkeit von *Topics* vorsehen um eine Weiterentwicklung der Anwendung zu unterstützen: Mögliche zukünftige Erweiterungen sollten keine

nachträgliche Änderung von *Topic*-Namen nach sich ziehen, damit existierende Clients nicht geändert werden müssen. Wird bspw. später eine neue Sensorklasse in eine ursprüngliche monosensorische *Smart Home*-Anwendung eingeführt, müssen beide Sensorklassen unterschieden werden. Deshalb wäre es günstig von Anfang an die Sensorklasse in den *Topic*-Namen zu verwenden, auch wenn das zuerst nicht erforderlich ist.

- Man sollte inhaltlich trennbare Nachrichten auch in getrennten *Topics* veröffentlichen, also spezifische Topic-Namen statt zusammenfassender genereller Benennungen benutzen und durch die Einführung von *Topic Levels* und die spätere Zusammenfassung spezifischer *Topics* durch die Verwendung von *Wildcards* ermöglichen.

6.2 Das MQTT-Protokoll

Das MQTT-Protokoll ist speziell für «m:n»-Verbindungen nach dem *Publish/Subscribe*-Muster gemacht. Es erlaubt die Verwendung unterschiedlicher Ausprägungen für die Dienstgüte («Quality of Service») und hat eine Reihe von Features, die es besonders für IoT-Anwendungen interessant macht. In den folgenden Abschnitten werden zuerst die Eigenschaften zur Steuerung der Dienstgüte und diese Features vorgestellt.

6.2.1 Quality of Service

Die Dienstgüte wird von dem Aufwand bestimmt, mit dem versucht wird eine Nachricht zuzustellen. Je nach Absicherungsverfahren kann es dabei erforderlich sein, zusätzliche Daten in der Nachricht zu speichern und das Protokoll kann umfangreicher sein.
Im MQTT-Protokoll gibt es drei unterschiedliche Ausprägungen für die *Quality of Service* (QoS), die jeweils einen unterschiedlichen Protokollaufwand bedeuten und abhängig von den jeweiligen Systemanforderungen anwendbar sind.

QoS Level 0: «*at most once*»

Sollte eine Nachricht unterwegs verloren gehen, wird sie nicht ersetzt. Der *Subscriber* wartet stattdessen weiter bis zur nächsten Nachricht. In Fällen, in denen bspw. ein

Sensor regelmäßig neue Messdaten sendet, kann diese Dienstgüte anwendbar sein. Der Vorteil ist, dass der *Broker* durch dieses Sicherungsverfahren kaum belastet wird.

QoS Level 1: «*at least once*»

Die Zustellung der Nachricht wird sichergestellt. Diese Dienstgüte ist aufwändiger zu erreichen als QoS Level `0`. Es kann passieren, dass eine *Acknowledge*-Meldung verlorengeht. Dann würde dieselbe Nachricht doppelt zugestellt werden. Bei dieser Dienstgüteeinstellung müssen Clients in der Lage sein dieselbe Nachricht mehrfach zu empfangen.

QoS Level 2: «*exactly once*»

Die Zustellung wird wie bei QoS Level `1` garantiert. Durch zusätzlichen Protokoll-Overhead wird zusätzlich sichergestellt, dass die Nachricht nur genau einmal zugestellt wird.

6.2.2 Retained Messages

Jeder *Subscriber*, der ein *Topic* neu abonniert, erhält automatisch die letzte *Retained Nachricht*, die über das *Topic* gesendet wurde. Dadurch ist es z. B. möglich in Anwendungen mit seltenen Updates *Subscriber* von Beginn der Verbindung an mit einem aktuellen Stand zu versorgen.

6.2.3 Last Will and Testament (LWT)

Der *Publisher* hinterlegt eine spezielle Nachricht, die vom *Broker* genau dann gesendet wird, falls dieser publizierende Client nicht mehr aktiv ist oder wenn die Verbindung zu ihm abbricht. Diese LWT-Nachricht wird vom Client beim ersten Verbindungsaufbau hinterlegt. Ein Anwendungszweck für *Last Will and Testament* ist beim *Health-Monitoring*: Beim Empfang einer LWT *Message* könnte der *Publisher* erneut gestartet werden.

6.2.4 Persistent Session

Bei einer *persistenten Session* merkt sich der *Broker* alle *Subscriber*. Das jeweilige Abonnement eines *Topics* bleibt dann über mehrere Verbindungen des jeweiligen *Subscribers* erhalten. Bricht die Verbindung zu einem solchen *Subscriber* ab und verbindet

der sich später erneut, werden ihm dann alle in der Zwischenzeit verpassten Nachrichten gesendet, die sonst bis dahin zugestellt worden wären. *Persistent Sessions* sind nützlich in Anwendungen, in denen mit häufigen Verbindungsabbrüchen gerechnet werden muss, wenn die fortlaufende Historie der Nachrichten wichtig ist (z. B. wenn fortlaufend gemeldete Zustandsänderungen nachvollzogen werden müssen um den aktuellen Zustand zu kennen).

6.2.5 Nachrichten

Zunächst macht MQTT keine Vorgaben über Inhalt oder Struktur der *Payload* einer Nachricht. So gibt es im Gegensatz zum JMS auch keine Header oder einen generellen Mechanismus um *Properties* zu setzen. Wenn dies für eine MQTT-basierte Anwendung erforderlich wäre, müsste man das selber in der *Payload* der Nachrichten verwalten. Alle *Producer* und *Consumer* müssten dann dieses anwendungsspezifische Strukturformat berücksichtigen.
In den folgenden Abschnitten werden die wichtigsten Nachrichtentypen in MQTT kurz dargestellt.

«Publish»

Das «Publish»-Nachrichtenformat wird benutzt um Daten von einem Client in der Rolle *Publisher* an den *Broker* zu senden.

Publisher ⟶

MQTT-Pakettyp:	**«Publish»**
packetId:	4711
topicName:	/floor04/room06/temp
qos:	1
retainFlag:	false
payload:	22°C
dupFlag:	false

⟶ Broker

Dieses Nachrichtenformat enthält die folgenden Bestandteile:

packetId: Zur Prüfung des Erfolgs der Zustellung durch Rückantwort wird diese ID bei QoS-Level 1 und 2 verwendet.

topicName: Name *Topics*, über das die *Payload* dieser Nachricht veröffentlicht werden soll

qos: QoS-Level, unter dem die Nachricht veröffentlicht wird (0–2)

`retainFlag`: Gesetzt, wenn diese Nachricht als *Retained Message* behandelt werden soll.

`payload`: Der eigentliche Inhalt der Nachricht (Bytes).

`dupFlag`: Gesetzt, falls es sich um eine Wiederholungssendung bei nicht erfolgtem *Acknowledge* bei QoS 1 oder 2 handelt

Der *Broker* benutzt dasselbe Format um die so empfangenen Daten zu den *Subscriber*-Clients zu weiterzutransportieren.

Subscriber ⟵	**MQTT-Pakettyp:**	**«Publish»**	⟵ Broker
`packetId`:	`4711`
`topicName`:	`/floor04/room06/temp`
`qos`:	`1`
`retainFlag`:	`false`
`payload`:	`22°C`
`dupFlag`:	`false`

«Subscribe»

Mit einer «Subscribe»-Nachricht kann ein Client dem *Broker* mehrere *Topics* mitteilen, die abonniert werden sollen.

Subscriber ⟶	**MQTT-Pakettyp:**	**«Subscribe»**	⟶ Broker
`packetId`:	`4712`
`topic`$_1$:	`/floor04/room04/temp`
`qos`$_1$:	`2`
`topic`$_2$:	`/floor04/room05/temp`
`qos`$_2$:	`2`
`topic`$_3$:	`/floor04/room06/temp`
`qos`$_3$:	`2`

Zu jedem *Topic*-Namen muss ein QoS-Parameter (`0`–`2`) angegeben werden. Es können beliebig viele *Topic*/QoS-Level-Paare in einer Nachricht enthalten sein.

`packetId`: Zur Prüfung des Erfolgs der Zustellung durch Rückantwort wird diese ID bei QoS-Level `1` und `2` verwendet.

`topic`$_i$: Name des i. *Topics* in dieser Nachricht, das abonniert werden soll

`qos`$_i$: QoS-Level des i. Abonnements in dieser Nachricht (`0`–`2`)

«Subscribe Bestätigung»

Bei QoS-Level `1` und `2` sendet der *Broker* eine Bestätigung an den Client zurück, der für jedes *Topic* getrennt eine Information über die maximal mögliche QoS beinhaltet.

Subscriber ⟵

MQTT-Pakettyp:	**«Subscribe»**
packetId:	`4712`
returnCode$_1$**:**	`1`
returnCode$_2$**:**	`2`
returnCode$_3$**:**	`1`

⟵ Broker

packetId: Das ist dieselbe `packetId` aus der «Subscribe»-Nachricht.

returnCode$_i$**:** das maximal mögliche QoS-Level des i. Abonnements aus der zu bestätigenden Nachricht (`0`–`2` oder `128` bei einem Fehler)

«Unsubscribe»

Subscriber-Clients können jederzeit einzelne der abonnierten *Topics* wieder abbestellen. Dazu dient dieser Nachrichtentyp, in dem analog zur *Subscription* beliebig viele *Topics* zum Abbestellen enthalten sein können.

Subscriber ⟶

MQTT-Pakettyp:	**«Unsubscribe»**
packetId:	`4713`
topic$_1$**:**	`/floor04/room04/temp`
topic$_2$**:**	`/floor04/room06/temp`

⟶ Broker

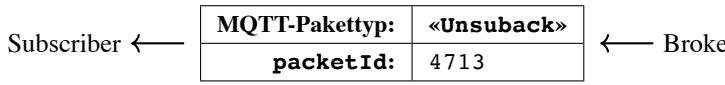
packetId: Zur Prüfung des Erfolgs der Zustellung durch Rückantwort wird diese ID bei QoS-Level `1` und `2` verwendet.

topic$_i$**:** Name des i. *Topics* in dieser Nachricht, dessen *Subscription* beendet werden soll

«Unsubscribe Bestätigung»

Als Antwort auf ein erfolgreiches «Unsubscribe» sendet der *Broker* bei QoS `1` und `2` eine Bestätigung an den Client zurück, der keine Nachrichten mehr zu den angegebenen *Topics* erhalten möchte. Sollte ein Client auf ein «Unsubscribe» keine Bestätigungsnachricht erhalten, ist das das Zeichen für den Client, dass seine Nachricht

nicht beim *Broker* angekommen ist. Der Client müsste dementsprechend versuchen die Abmeldung erneut zu senden.



`packetId`: Dieselbe `packetId` wie aus der «`Unsubscribe`»-Nachricht.

6.3 Broker

Wie bei den JMS Providern gibt es eine Reihe von Produkten, die als MQTT Server dienen können. Viele MQTT *Broker* kann man über proprietäre Schnittstellen mit eigenen Plugins erweitern (z. B. Authentifizierung von Clients, spezielle Persistenzadapter). Dafür benötigt man natürlich Zugriff auf den *Broker*.
Durch eine eindeutige Benennung des *Topics* können mehrere Anwendungen auf einem *Broker* voneinander getrennt werden. Daher muss der Zugriff auf einen MQTT *Broker* nicht exklusiv für die Clients einer Anwendung sein und so kann man durchaus statt eines dedizierten *stand alone* Servers einen gehosteten MQTT *Broker* benutzen.
Ein öffentlicher gehosteter MQTT *Broker* ist nicht das Mittel der Wahl, wenn geringe Latenzzeiten bei der Kommunikation zwischen den Knoten einer Anwendung wichtig sind. Außerdem sind die Nachrichten nicht privat, da ein fremder Client leicht alle *Topics* über das *Wildcard* «#» abonnieren kann und dann die Kommunikation anderer Anwendungen über den MQTT *Broker* «belauschen» kann. Ein öffentlicher Server ist also nicht für produktive Systeme geeignet, aber sehr nützlich um während der Entwicklung «mal eben» einen Client zu testen.
Beispiele für öffentlich gehostete MQTT-*Broker* sind:

```
tcp://iot.eclipse.org:1883
tcp://broker.mqttdashboard.com:1883
```

6.4 MQTT Clients in Java mit der Paho Library

Bei MQTT ist nur das Kommunikationsprotokoll («*Wire Protocol*») zwischen Client und *Broker* standardisiert, nicht aber die Programmierschnittstelle. Zudem gibt es *Bindings* für viele unterschiedliche Zielsprachen. Deshalb kann in diesem Abschnitt über

die Programmierung von MQTT Clients mit Paho nur eine Library beispielhaft vorgestellt werden. Da das Protokoll aber nicht sehr komplex und umfangreich ist, kann man davon ausgehen, dass andere Bibilotheken einen prinzipiell ähnlichen Zugang zum MQTT-Protokoll bereitstellen.
Im Gegensatz zu JMS sind die Client Libraries im MQTT-Umfeld nicht an einen bestimmten *Broker* gebunden, denn mit ihnen wird das *Wire*-Protokoll von MQTT erzeugt und nicht eine bestimmte proprietäre Schnittstelle des *Brokers* angesprochen. Man kann also unter Beibehaltung der MQTT Library den *Broker* auswechseln.

6.4.1 Publisher mit Paho

Zum Senden einer Nachricht wird in Paho zuerst eine Instanz der Klasse `MqttClient` erzeugt. Der Konstruktor erfordert zwei `String`-Objekte als Parameter. Der erste ist ein URL zur Adressierung des MQTT *Brokers*. Als Transportprotokoll sind TCP oder verschlüsseltes TCP (SSL/TLS) möglich, z. B.

```
tcp://localhost:1883
ssl://localhost:8883
```

Wenn der Port nicht angegeben wird, wird dafür der Default-Wert benutzt. Bei TCP ist das `1883`, bei SSL/TLS `8883`. Der zweite Parameter des Konstruktors ist ein einzigartiger Text, mit dem der Client identifiziert werden kann, wenn er sich abmeldet und später wieder anmeldet.
An dem `MqttClient`-Objekt muss die Methode `connect()` aufgerufen werden, bevor Nachrichten darüber versendet werden können. Entsprechend werden am Ende mit `disconnect()` die Ressourcen der Verbindung wieder freigegeben.
Über das `MqttClient`-Objekt kann man mit `publish(...)` `MqttMessage`-Instanzen an ein bestimmtes *Topic* versenden (s. Listing 6.1). An solchen *Message*-Objekten kann mit der Methode `setPayload(...)` der Inhalt der Nachricht eingefügt werden.

6.4.2 Subscriber mit Paho

Subscriber verwenden in Paho asynchrone Kommunikation. Dazu wird ein Callback mit der Methode `setCallback(...)` installiert (s. Listing 6.2). Das Callback-Objekt wird benutzt, wenn Nachrichten empfangen werden. Dazu muss an

dem `MqttClient`-Objekt, für das das Callback installiert worden ist, die Methode `subscribe(...)` mit einem *Topic* String aufgerufen werden. Das Callback muss das Interface `MqttCallback` implementieren und drei Methoden bereitstellen:

`void connectionLost(Throwable):` Sollte die Verbindung verlorengehen, wird diese Methode mit der Ausnahme, die dabei aufgetreten ist, aufgerufen. An dieser Stelle kann eine eigene Fehlerbehandlung durchgeführt werden. Der Grund für die Existenz dieser Methode ist, dass es sonst keine andere Möglichkeit gäbe, eine eigene Fehlerbehandlungsroutine zu definieren.

`void deliveryComplete(IMqttDeliveryToken):` Wenn eine Nachricht endgültig zugestellt ist und abhängig vom QoS-Level alle relevanten *Acknowledgement*-Nachrichten empfangen wurden, wird diese Methode mit dem `IMqttDeliveryToken`, einem Objekt, mit dem der Prozess der Zustellung einer Nachricht verfolgt werden kann, aufgerufen.

`void messageArrived(String, MqttMessage):` Diese Methode wird aufgerufen, wenn eine Nachricht erfolgreich zugestellt wurde. Erst wenn diese Methode fertig abgearbeitet ist, wird ein *Acknowledge* an den *Broker* zurückgeschickt. Die Parameter, die der Methode übergeben werden, sind das *Topic*, über das die Nachricht empfangen wurde, und eine Repräsentation der Nachricht selbst.

Listing 6.1: Grundgerüst für die *Publisher*-Funktion mit der Paho Library

```
1  MqttClient client = null;
2  String message;     //...
3  String topic;       // z. B. "4761"
4  String broker;      // z. B. "tcp://localhost:1883"
5  String clientId = MqttClient.generateClientId();
6  try {
7      client = new MqttClient(broker, clientId);
8      client.connect();
9      MqttMessage message = new MqttMessage();
10     message.setPayload(message.getBytes());
11     client.publish(topic, message);
12     client.disconnect();
13 } catch (MqttException e) {
14     // ...
15 }
```

Alternativ gibt es auch eine Variante der Callback-Klasse mit einer weiteren Methode, die aufgerufen wird, wenn die Verbindung zum *Broker* (wieder-)aufgebaut wird.

Listing 6.2: Grundgerüst für die *Subscriber*-Funktion mit der Paho Library

```
1  MqttClient client = null;
2  String topic;       // z. B. "4761"
3  String broker;      // z. B. "tcp://localhost:1883"
4  String clientId = MqttClient.generateClientId();
5  boolean stopped = false;
6  try {
7      client = new MqttClient(broker, clientId);
8      client.setCallback(new MqttCallback() {
9          @Override
10         public void connectionLost(Throwable arg0) {
11             // ...
12         }
13         @Override
14         public void deliveryComplete(IMqttDeliveryToken arg0) {
15             // ...
16         }
17         @Override
18         public void messageArrived(String topic, MqttMessage m) throws
               Exception {
19             // ... verarbeiten und ggf. stopped = true;
20         }
21     });
22     client.connect();
23     client.subscribe(topic);
24     while (! stopped) {
25         Thread.sleep(1000);
26     }
27 } catch (MqttException | InterruptedException e) {
28     // ...
29 } finally {
30     try {
31         client.disconnect();
32     } catch (MqttException e) {
33         // unrecoverable
34     }
35 }
```

Praktikum

Aktivitätsschritte:

Unter der Webadresse `http://verteiltearchitekturen.de/vol01/VAR-MQTT-solution.zip` ist eine Musterlösung zu diesem Praktikum verfügbar. In dem ZIP-File befindet sich das Eclipse-Projekt `VAR-MQTT_solution`.

6.5 Aufgabe: unterschiedliche Broker für eine prototypische MQTT-Anwendung

In diesem Praktikum entwickeln Sie mit der Paho Library prototypische MQTT Clients in Java, die als *Publisher* oder *Subscriber* über unterschiedliche *Broker* mit dem MQTT-Protokoll kommunizieren.

6.5.1 Eclipse-Projekt und Client Library

Aktivitätsschritt 6.1:
Ziel: Projekt downloaden und in Eclipse importieren
Laden Sie das ZIP-Archiv mit dem Projekt `VAR-MQTT` unter `http://verteiltearchitekturen.de/vol01/VAR-MQTT.zip` herunter. Importieren Sie das ZIP-File in Eclipse mit der Funktion *File* → *Import...* → *General* / *Existing Projects Into Workspace*. Wählen Sie dort die Option *Select Archive File* (s. Abb. 6.1). Sie sollten auch darauf achten, dass die Option *Copy Projects Into Workspace* aktiviert ist, sonst arbeiten Sie möglicherweise nur auf den Dateien in Ihrem Download-Ordner und nicht in Ihrem Eclipse Workspace.

Als Ergebnis sollte das Projekt `VAR-MQTT` in Ihrem Package Explorer in Eclipse auftauchen. Im `src/`-Ordner dieses Projektes sollten Sie das Package `var.mom.mqtt.log` wie in Abb. 6.2 sehen:

Aktivitätsschritt 6.2 (falls erforderlich):
Ziel: MQTT Client Library dem Classpath hinzufügen
Das Beispielprojekt `VAR-MQTT` enthält bereits die Paho Client Library im Folder `lib/`. Falls sie noch nicht im Klassenpfad eingebunden sein sollte, können Sie Paho in Eclipse durch rechten Mausklick auf das JAR-File (`lib/org.eclipse.paho.client.mqttv<Version>.jar`) und die Wahl von *Build Path* $rightarrow$ *Add to Build Path* im Kontextmenü Ihrem Projekt hinzufügen.

Wenn Sie ein neues MQTT-Projekt «*from scratch*» aufbauen wollen, können Sie die Paho Client Library auch direkt downloaden: `https://repo.eclipse.org/content/repositories/paho-releases/org/eclipse/paho/`

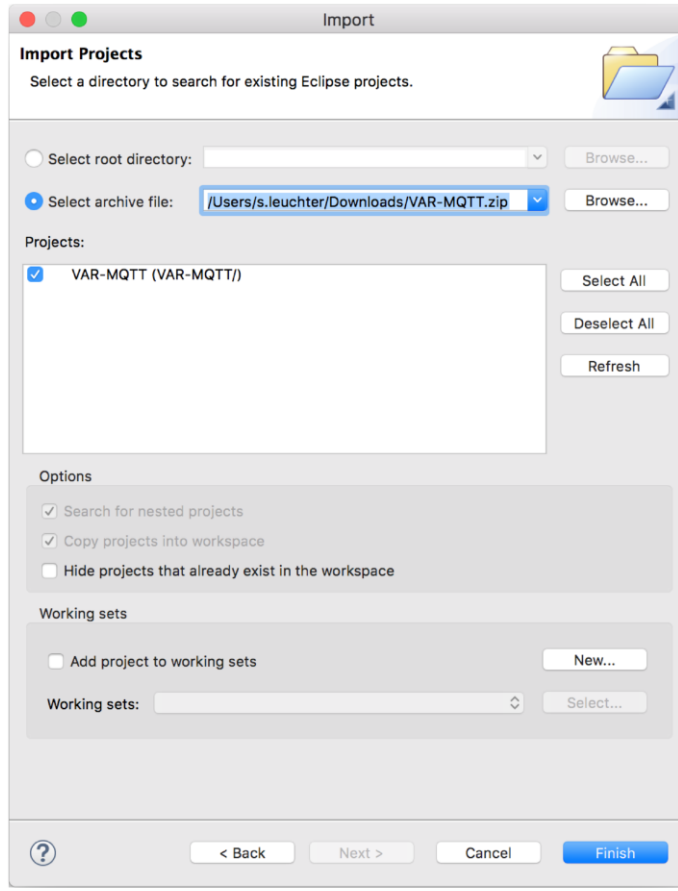


Abbildung 6.1: Einbinden des vorbereiteten MQTT-Projekts in Eclipse

```
└── var
    └── mom
        └── mqtt
            └── log
                ├── Conf.java
                ├── Publisher.java
                └── Subscriber.java
```

Abbildung 6.2: Verzeichnis- und Dateibaum für das Projekt `VAR-MQTT`

Exkurs: Die Java Library für MQTT

Paho ist eine Java Library, die das MQTT-Protokoll clientseitig verfügbar macht. Das API von Paho ist im Gegensatz zu dem von JMS nicht standardisiert. Es könnte also alternative Java Libraries geben, die anders als Paho angesprochen werden. MQTT ist für viele Plattformen und Programmiersprachen verfügbar. Die *embedded*-Knoten, die oft als *Publisher* in typischen IoT-Systemarchitekturen Verwendung finden, würden normalerweise eher nicht mit Java programmiert werden. Jedoch gibt es viele IoT-Anwendungsszenarien, in denen die Subscriber-Seite sehr wohl in Java programmiert sein könnte (z. B. im Backend, zum Logging, zur Visualisierung oder zur Datenfusion). In diesem Praktikum werden der Einfachheit halber sowohl *Publisher* als auch *Subscriber* mit Paho als Java Clients realisiert.

6.5.2 Broker

Da das MQTT-Protokoll so schmal ist, kann man es leicht implementieren. Das ist wahrscheinlich einer der Gründe dafür, dass es viele MQTT-fähige *Broker* gibt. In diesem Praktikum werden zwei unterschiedliche *Broker* benutzt, zwischen denen Sie (fast) beliebig wechseln können: Apache ActiveMQ lokal installiert und HiveMQ öffentlich gehostet.

6.5.3 ActiveMQ lokal installiert

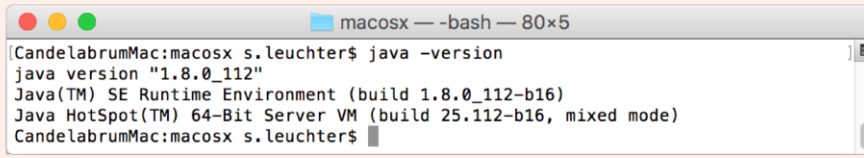
In dem Praktikum des JMS-Kapitels wurde Apache ActiveMQ bereits als *Broker* verwendet. ActiveMQ hat jedoch nicht nur Unterstützung für JMS eingebaut, sondern kann auch mit anderen Protokollen verwendet werden. Die MQTT-Unterstützung muss möglicherweise in Ihrer lokalen Installation erst noch aktiviert werden.
Falls Sie Apache ActiveMQ noch nicht haben, müssen Sie das Softwarepaket erst herunterladen und installieren.

Aktivitätsschritt 6.3:
Ziel: Feststellen, ob Java richtig für Apache ActiveMQ installiert ist
Da Apache ActiveMQ Java 8 verwendet, brauchen Sie auch ein *Java Runtime Environment* (ist Teil des *Java Development Kits*), das direkt in der Shell ausführbar ist. Es hängt vom Betriebssystem und der Installationsart ab, ob sich Java bei Ihnen «im Pfad befindet». Mit «Pfad» ist hier die Environment-Variable `$PATH` gemeint.

Diese Umgebungsvariable wird vom Betriebssystem genutzt, um den Speicherort eines ausführbaren Programms zu finden, wenn es ohne Pfadnamen gestartet wird. Um zu prüfen, ob Java bei Ihnen läuft, öffnen Sie eine Shell (unter Windows können «Win» + «R» und im Eingabefeld `cmd` eingeben). In der dann erscheinenden Windows Eingabeaufforderung bzw. Shell können Sie versuchen das Java Runtime Environment zu starten, indem Sie Folgendes eingeben:

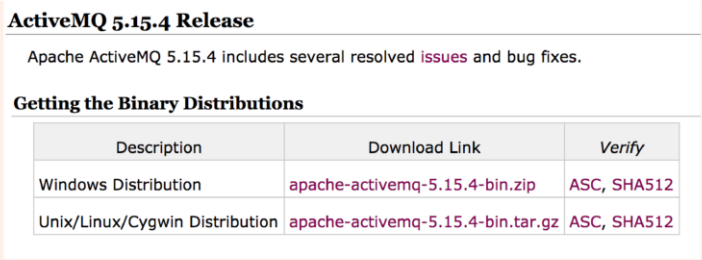
```
java -version
```

Sie sollten als Ergebnis des Aufrufs eine Meldung sehen, in der die Version Ihres Java Runtime Environments genauer beschrieben wird. Das kann beispielsweise so aussehen:



Die Versionsangabe «`1.8.0_112`» bedeutet, dass Java 8 installiert ist (JDK 1.8 entspricht Java 8).
Falls der Aufruf nicht funktioniert, ist Java wahrscheinlich nicht im Pfad. Oracle stellt eine Anleitung zum Setzen der $PATH-Umgebungsvariable für die Betriebssysteme Solaris SPARC, Red Hat Linux, SUSE Linux, Oracle Linux, Windows 10, Windows 8, Windows 7, Windows Vista, Windows XP und macOS bereit: `https://www.java.com/de/download/help/path.xml`.

Aktivitätsschritt 6.4 (falls nicht vorhanden):
Ziel: Apache ActiveMQ installieren
Laden Sie die aktuelle Version von Apache ActiveMQ von der Seite `http://activemq.apache.org/download.html` herunter («*latest stable release*»). Folgen Sie dort dem Link zur binär-Distribution. Abhängig von Ihrem Betriebssystem wählen Sie dann die passende Variante und laden Sie sie über den Download Link herunter:



ActiveMQ 5.15.4 Release

Apache ActiveMQ 5.15.4 includes several resolved issues and bug fixes.

Getting the Binary Distributions

Description	Download Link	*Verify*
Windows Distribution	apache-activemq-5.15.4-bin.zip	ASC, SHA512
Unix/Linux/Cygwin Distribution	apache-activemq-5.15.4-bin.tar.gz	ASC, SHA512

Entpacken Sie das Archiv. Verschieben Sie das entpackte Verzeichnis aus Ihrem Download-Ordner, falls es dort entpackt wurde, an einen Platz, an dem das Verzeichnis längerfristig bleiben kann. Speichern Sie den Inhalt des Archivs aber nicht direkt in Ihrem Eclipse Workspace. Bei manchen Versionen von Apache ActiveMQ und auf einigen Betriebssystemen darf **kein Leerzeichen im Pfad zum entpackten Archiv** vorkommen.

Im hier gezeigten Beispiel liegt der Inhalt des Archivs im Verzeichnis `/opt/apache-activemq-5.15.4/`. In den folgenden Ausführungen wird dieser Pfad benutzt. Sollten Sie eine andere Version von ActiveMQ installiert oder das Verzeichnis an einen anderen Ort entpackt haben, müssen Sie stattdessen Ihren Pfad einsetzen.

Aktivitätsschritt 6.5:

Ziel: Prüfen, ob das MQTT-Protokoll in Apache ActiveMQ aktiv ist

Finden Sie die Konfigurationsdatei Ihres lokal installierten ActiveMQ *Brokers* (z. B. `apache-activemq-5.15.4/conf/activemq.xml`). Prüfen Sie, ob dort ein Eintrag zu einem MQTT Transport Connector vorhanden ist. Ein entsprechender Eintrag könnte aussehen wie in Listing 6.3.

Achten Sie darauf, dass dieser Eintrag nicht innerhalb eines Kommentars (`<!– ... >`) steht (dann würde er beim Start ignoriert werden und würde nicht aktiviert), sondern «aktiv» ist. Im Eintrag könnte der Connector auch konfiguriert werden wie in Listing 6.4.

Dabei ist zu beachten, dass sowohl mit `0.0.0.0` als auch mit `localhost`, ggf. auch `127.0.0.1` jeweils der eigene lokale Rechner gemeint ist. Der Port 1883 ist der

Listing 6.3: Konfiguration von ActiveMQ für das MQTT-Protokoll

```
<transportConnectors>
  ...
  <transportConnector name="mqtt" uri="mqtt://localhost:1883"/>
  ...
</transportConnectors>
```

Listing 6.4: Erweiterte Konfiguration von ActiveMQ für das MQTT-Protokoll

```
<transportConnectors>
  ...
  <transportConnector name="mqtt" uri="mqtt://0.0.0.0:1883?
              maximumConnections=1000&amp;wireFormat.maxFrameSize
                  =104857600"/>
  ...
</transportConnectors>
```

MQTT Standard Port (TCP). Es könnte jederzeit ein anderer Port verwendet werden, sofern er noch nicht mit einem aktiven Service auf dem lokalen Rechner belegt ist.

Aktivitätsschritt 6.6 (falls erforderlich):
Ziel: MQTT-Protokoll in Apache ActiveMQ aktivieren
Sollte Ihr ActiveMQ noch nicht den MQTT *Transport Connector* aktiviert haben, ändern Sie die Konfigurationsdatei `apache-activemq-5.15.4/conf/activemq.xml` entsprechend ab wie in Listing 6.3 gezeigt.

Aktivitätsschritt 6.7 (falls erforderlich):
Ziel: Apache ActiveMQ (neu) starten
Wenn Sie die Konfiguration geändert haben oder Apache ActiveMQ noch nicht läuft, starten Sie Ihren ActiveMQ Server mit dem Kommando, das auf Ihrer Plattform dazu benötigt wird. Wechseln Sie dazu in das entsprechende Verzeichnis in `/opt/apache-activemq-5.15.4/bin`. Im Fall von Linux ist das `linux-x86-32` (für 32-Bit) oder `linux-x86-64` (für 64-Bit). Unter macOS wechseln Sie in das Verzeichnis `macosx`. Wenn Sie die Windows-Version heruntergeladen haben, bleiben Sie im Verzeichnis `/opt/apache-activemq-5.15.4/bin/` (**wechseln Sie nicht** in `win32` oder `win64`).

```
cd /opt/apache-activemq-5.15.4/bin/linux-x86-32
```

Starten Sie dort

```
./activemq start
```

bzw.

```
./activemq restart
```

Das «./» ist erforderlich, falls Sie nicht das «current working directory» («./») in der Environment-Variable `$PATH` haben (was auch nicht empfehlenswert wäre), um dem Betriebssystem mitzuteilen, dass das `activemq`-Programm aus dem aktuellen Verzeichnis gestartet werden soll.
Später kann der Server mit dem folgenden Kommando angehalten werden.

```
./activemq stop
```

Aktivitätsschritt 6.8:
Ziel: Testen, ob Apache ActiveMQ mit MQTT-Unterstützung läuft
Prüfen Sie, ob ActiveMQ tatsächlich läuft und ob das MQTT-Protokoll aktiviert ist, durch Zugriff auf die Management-Oberfläche `http://localhost:8161/admin/connections.jsp` (user: `admin`, password: `admin`).
In der Ansicht müsste es wie in Abb. 6.3 unter *Connections* einen Abschnitt «Connector mqtt» geben, in dem vorerst noch keine Verbindungen auftauchen sollten.

Aktivitätsschritt 6.9 (alternativ zu HiveMQ):
Ziel: lokalen ActiveMQ Server im Projekt verwenden
Wenn Sie in diesem Praktikum Ihren lokal installierten Apache ActiveMQ Server verwenden wollen, müssen Sie im vorbereiteten Eclipse-Projekt die Konstante `var.mom.mqtt.Conf.Broker` auf `tcp://localhost:1883` setzen, falls Ihr Apache ActiveMQ Server entsprechend konfiguriert ist.

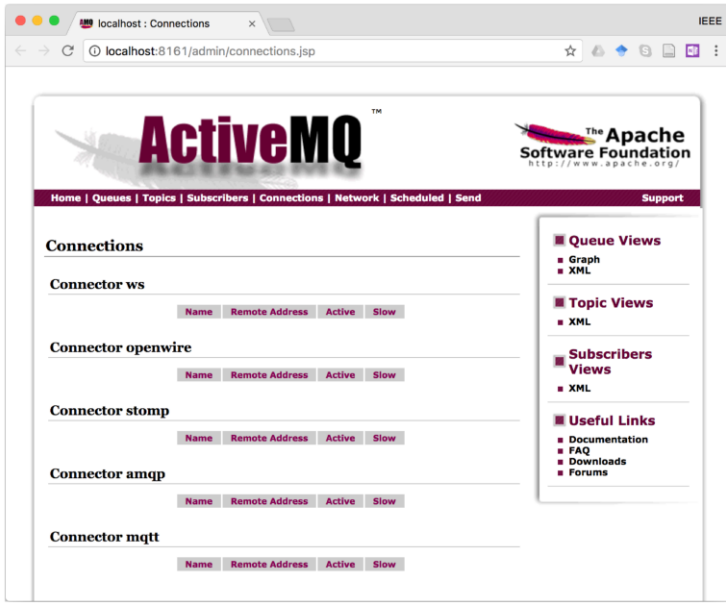


Abbildung 6.3: Die Benutzungsschnittstelle von ActiveMQ zeigt unter dem URL `http://localhost:8161/admin/connections.jsp` die aktivierten Protokolle an (Login: `admin`/`admin`)

6.5.4 HiveMQ öffentlich gehostet

Alternativ zum eigenen *Broker* können Sie auch einen öffentlich gehosteten MQTT *Broker* verwenden. Das ist natürlich keine gute Idee, wenn Sie ein kritisches System implementieren wollen. Aber wenn Sie gerade keinen eigenen Server während der Entwicklungszeit installieren wollen, ist es möglicherweise eine Alternative, solch einen öffentlich zugänglichen *Broker* zu benutzen. Bedenken Sie dabei aber, dass dessen Verfügbarkeit jederzeit eingeschränkt sein kann und dass fremde *Subscriber* jederzeit Ihre Nachrichten mitlesen können.

Aktivitätsschritt 6.10:
Ziel: öffentlich gehosteten HiveMQ Server untersuchen
Öffnen Sie die URL `http://www.mqtt-dashboard.com/`, die den aktuellen Betrieb auf dem dort öffentlich gehosteten HiveMQ anzeigt (s. Abb. 6.4). Analysieren Sie den aktuellen Kommunikationsverkehr und versuchen Sie Muster in den verwendeten Topics zu identifizieren.

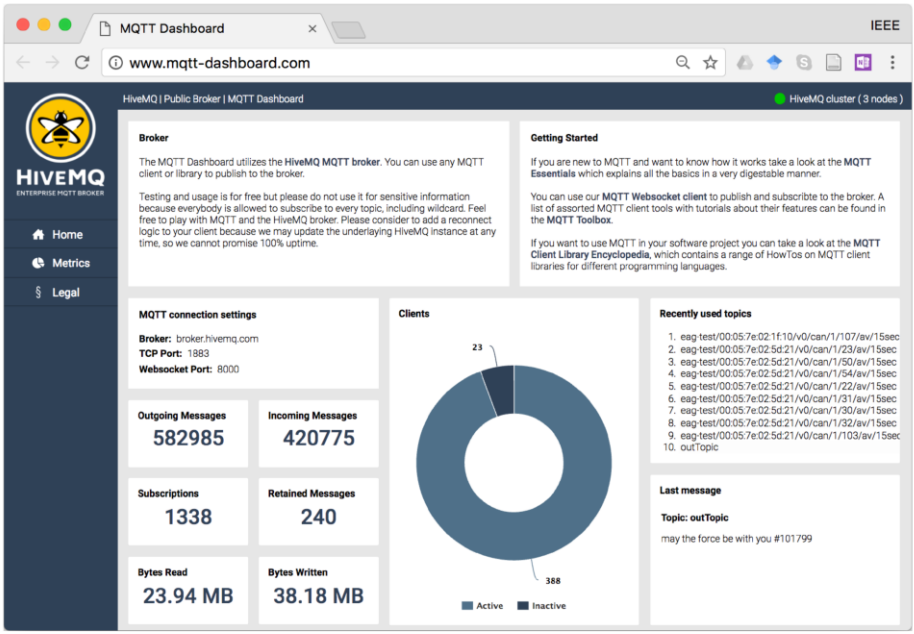


Abbildung 6.4: Die Benutzungsschnittstelle eines öffentlich gehosteten HiveMQ ist unter der URL `http://www.mqtt-dashboard.com/` verfügbar

Aktivitätsschritt 6.11 (alternativ zu ActiveMQ):
Ziel: gehosteten HiveMQ Server im Projekt verwenden
Wenn Sie in diesem Praktikum den öffentlich gehosteten HiveMQ Server verwenden wollen, müssen Sie im vorbereiteten Eclipse-Projekt die Konstante `var.mom.mqtt.Conf.Broker` auf `tcp://broker.mqttdashboard.com` setzen.

6.5.5 MQTT Clients

Aktivitätsschritt 6.12:
Ziel: vorbereitete *Publisher* und *Subscriber* laufen lassen und analysieren
Im vorbereiteten Eclipse-Projekt gibt es einen *Publisher* und einen *Subscriber*. Starten Sie zuerst den *Subscriber* und dann den *Publisher*. Beobachten Sie den Ef-

fekt beim *Subscriber*. Starten Sie den *Publisher* erneut. Analysieren Sie den Quellcode von *Publisher* und *Subscriber*.

6.6 Aufgabe: Smart-Home-Anwendung

In dieser Praktikumsaufgabe entwickeln Sie Softwarekomponenten für eine IoT-Anwendung: Es wird ein *Smart Home* simuliert, also ein Haushalt, der vielfältige Sensoren und Aktuatoren für die Steuerung der Hauselektronik (z. B. Heizung, Lüftung, Verdunklung, Zutrittskontrolle, Audio/Video-Abspielen) besitzt. MQTT bietet sich hierfür als Vernetzungstechnologie an.

6.6.1 Szenario und Beispielanwendung

In den folgenden Schritten erweitern Sie das vorgegebene *Publisher*/*Subscriber*-Paar, sodass eine Sensornetz-Anwendung entsteht, die die Funktionsweise einiger MQTT-Eigenschaften demonstriert: Es handelt sich um die Überwachung von Sensordaten in einem «Smart Home». Im fiktiven Beispiel werden die Sensoren durch kleine Java-Programme simuliert. In einem realen Szenario würden Microcontroller-gestützte Sensoren (z. B. auf der Basis von preiswerten Arduino-Boards) MQTT-Nachrichten verschicken. Einige Java *Subscriber* werden für das Logging und die Visualisierung zuständig sein.

6.6.2 Datenmodell für die Beispielanwendung

Als erstes wird das Datenmodell für die Beispielanwendung festgelegt.
Jeder Sensor bekommt ein exklusives *Topic* zugeordnet, über das neue Messwerte dieses Sensors in nicht vorher bestimmter Frequenz übermittelt werden. Damit sich die Umgebungen unterschiedlicher Aufgaben dieses Praktikums nicht gegenseitig stören, wird ein «Wurzel-*Topic Level*» verwendet, der in jedem Projekt einmalig sein muss. Im folgenden Beispiel ist das die ID `SmartHome4751`. Sie sollten einen anderen *Topic Level* Namen in Ihrem Projekt verwenden. Alle Ihre Clients müssen die *Topics* aber mit demselben «Wurzel *Topic Level*» ansprechen.
Auf den «Wurzel *Topic-Level*»-Namen folgt eine genauere Spezifikation der Messstelle (z. B. Etage und Zimmer) und der Messgröße — also beispielsweise

```
SmartHome4751/erdgeschoss/wohnzimmer/temperatur oder
SmartHome4751/dachgeschoss/badezimmer/feuchtigkeit
```

6.6.3 Publisher: Sensorsimulator

Nachdem das Datenmodell für die Beispielanwendung spezifiziert ist, können die Clients entwickelt werden. In den folgenden Aktivitätsschritten wird zuerst ein *Publisher* mit Code der vorherigen Aufgabe implementiert.

Aktivitätsschritt 6.13:
Ziel: Sensorsimulator anlegen
Legen Sie ein neues Package `var.mom.mqtt.smarthome` an. Kopieren Sie den `Publisher` in eine neue Klasse `Sensor`.

Aktivitätsschritt 6.14:
Ziel: Topic für Sensorsimulator als Kommandozeilenparameter vorgeben
Ändern Sie die `main`-Methode, sodass sie die folgenden Kommandozeilenparameter erwartet, aus denen der *Topic* `String` zusammengebaut wird.
Der «Wurzel-*Topic-Level*»-Name sollte zentral als Konstante in `var.mom.mqtt.smarthome.Conf` konfiguriert werden können. Dazu kommen dann jeweils durch «/» getrennt:

1. **Etage** (z. B. `args[0]`)
2. **Zimmer** (z. B. `args[1]`)
3. **Messgröße** (z. B. `args[2]`)
4. Als weiterer Kommandozeilenparameter soll ein Initialwert für die Messgröße übergeben werden: **Initialwert** (z. B. `args[3]`)

Aktivitätsschritt 6.15:
Ziel: Funktionalität des Sensorsimulators programmieren
Der Sensor soll nun in einer Endlosschleife zufällige Messwerte über den *Broker*

Listing 6.5: Beispiel für die Simulation des nächsten Messwerts

```
1 if (Math.random() > 0.5) {
2     messwert += 0.1;
3 } else {
4     messwert -= 0.1;
5 }
```

versenden. Dazu muss der Sensor zuerst einen `MqttClient` erzeugen und sich mit ihm verbinden. Dann werden die Messwerte generiert und gleich über den Client unter einem *Topic* gesendet, der zuerst aus dem «Wurzel-*Topic-Level*»-Namen (`String`-Konstante in `var.mom.mqtt.smarthome.Conf`) und den Kommandozeilenparametern `args[0]` bis `args[2]` zusammengebaut werden muss. Der zufällige Messwert sollte sich aus dem jeweils zuvor simulierten Messwert errechnen, wobei der initiale Messwert `args[3]` ist. Sie können dafür beispielsweise den Quellcode aus Listing 6.5 übernehmen.

Aktivitätsschritt 6.16 (wenn gewünscht und noch nicht installiert):
Ziel: Launch Groups für Eclipse
Installieren Sie das Paket «C/C++ Development Tools» in Eclipse um das Feature *Launch Groups* verwenden zu können. Öffnen Sie dazu den Dialog zur Installation von Eclipse-Komponenten: *Help* → *Install New Software*.... Wählen Sie wie in Abb. 6.5 das reguläre *Repository* (unter *Work With:*). Bei der zum Redaktionsschluss dieses Textes gerade aktuellen Version «Photon» von Eclipse ist das «Photon – `http://download.eclipse.org/releases/photon`)».

Aktivitätsschritt 6.17:
Ziel: Mehrere Sensoren starten
Starten Sie mehrere `Sensor`-Instanzen in unterschiedlichen Etagen, Räumen und Messgrößen. Wenn Sie eine entsprechend komplexere Konfiguration auf einmal starten wollen, können Sie in Eclipse eine «Launch Group» anlegen. Sie können damit bspw. alle Sensoren auf einmal starten.

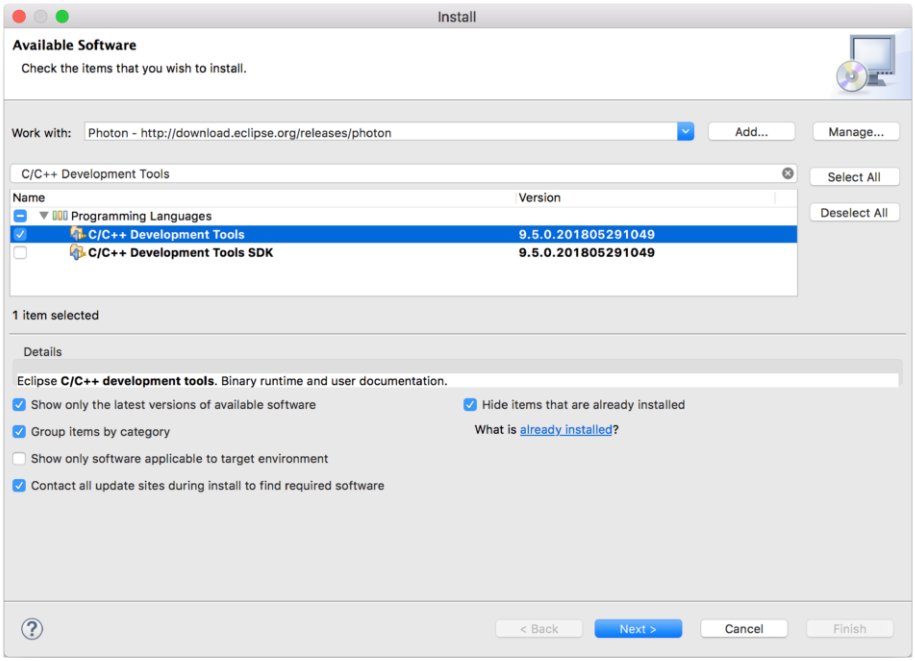


Abbildung 6.5: Installation der «C/C++ Development Tools» in Eclipse für das Feature *Launch Groups*

6.6.4 Subscriber: Alarm

Der `AlarmSubscriber` abonniert das *Topic* eines bestimmten Sensors und liest die Messungen mit. Überschreiten die Messungen einen vorher festgelegten Schwellwert soll der `AlarmSubscriber` eine Meldung ausgeben.

Aktivitätsschritt 6.18:

Ziel: `AlarmSubscriber` anlegen und Topic spezifizieren

Kopieren Sie den Beispiel-*Subscriber* aus dem Paket `var.mom.mqtt.log` in die neue Klasse `AlarmSubscriber`.

Wie schon bei der Klasse zum Simulieren eines Sensors kann das *Topic* für das Abonnement des `AlarmSubscriber` aus Kommandozeilenparametern zusammengebaut werden.

Listing 6.6: Konvertierung einer Nachricht in einen Messwert

```
1  public void messageArrived(String topic, MqttMessage m)
2      throws Exception {
3      try {
4          double observation = Double.valueOf(m.toString());
5          if (observation > threshold) {
6              // ... Trigger für Alarm
7          }
8      } catch (NumberFormatException e) {
9          // Payload der Nachricht hatte nicht Zahlenformat
10 }
```

Aktivitätsschritt 6.19:

Ziel: Callback-Methode für `AlarmSubscriber` programmieren

Erweitern Sie die Callback-Methode `messageArrived(...)` so ,dass eine Nachricht auf `System.out` ausgegeben wird, wenn der Messwert eine vorgegebene Schwelle überschreitet. Die Schwelle soll wieder als Kommandozeilenparameter übergeben werden.

In der Methode `messageArrived(...)` muss dann die Nachricht interpretiert werden. Der Einfachheit halber sollte der `Sensor`-*Publisher* lediglich einen Zahlenwert in die Nachricht als *Payload* hineinschreiben. Solch eine Nachricht kann dann nämlich ganz einfach als `Double`-Wert interpretiert werden, indem man versucht ihn zu konvertieren (s. Listing 6.6).

6.6.5 Subscriber: Logger mit Wildcard-Verwendung

Da das *Publish*/*Subscribe*-Kommunikationsmuster die Erweiterung einer Anwendung durch das Implementieren zusätzlicher *Subscriber* erlaubt, wird im nächsten Schritt ein weiterer *Subscriber* zur Protokollierung aller Nachrichten entwickelt.

Aktivitätsschritt 6.20:

Ziel: Anlegen von `LoggingSubscriber`

Kopieren Sie den Beispiel-`Subscriber` aus dem Eclipse-Beispielprojekt in eine neue Klasse `LoggingSubscriber`.

Aktivitätsschritt 6.21:
Ziel: Abonnieren aller Nachrichten im `LoggingSubscriber`
Der `LoggingSubscriber` soll alle *Topics* der Smart Home Anwendung abonnieren. Das entsprechende *Topic* muss das *Multilevel Wildcard* # verwenden.
Achtung: Dies widerspricht der *Best Practice*-Empfehlung von S. 170. Die Empfehlung ist zwar, nicht alle Nachrichten über das *Multilevel Wildcard* zu abonnieren, sondern stattdessen den *Broker* um das entsprechende Verhalten zu erweitern. In diesem Beispiel ist jedoch zum Einen der Nachrichtendurchsatz nicht sehr hoch, zum Anderen soll es möglich sein, einen öffentlichen *Broker* zu verwenden, der nicht ohne Weiteres erweitert werden kann. Es ergibt sich als Topic beispielsweise:

```
SmartHome4751/#
```

Alle empfangenen Nachrichten sollen ausgegeben werden. Für diese Praktikumsaufgabe wird die Funktionalität auf das Notwendigste reduziert. Jede empfangene Nachricht wird direkt auf der Standardausgabe (`System.out`) ausgegeben. In einem realen System würden Sie ein Logging-Framework wie *log4j* ansprechen. Es ergibt sich, dass lediglich das *Topic* geändert werden muss.

Aktivitätsschritt 6.22:
Ziel: Callback-Methode für `LoggingSubscriber` programmieren
Die Implementierung der Callback-Methode `messageArrived(...)` in der anonymen Klasse, die mit `setCallback(...)` an den Client übergeben wird, ist bereits ausreichend, wenn auf die Ausgabe von Zeitstempeln verzichtet werden kann.

6.6.6 Selbstbeschreibender Sensor mit *Retained Message* und *Last Will and Testament*

Retained Message

Eine *Retained Message* ist eine Nachricht, die zu einem *Topic* gehört und von *Publishern* dieses *Topics* gesetzt werden kann. Diese *Retained Message* wird jedem neuen *Subscriber* eines *Topics* zu Beginn des Abonnements übermittelt.

In unserem Anwendungsbeispiel hat jedes *Topic* jeweils nur einen zugeordneten *Publisher* (außer Sie lassen mehrere redundante Sensoren zur selben Messgröße im selben Raum derselben Etage laufen).

Aktivitätsschritt 6.23:
Ziel: Retained Willkommensbotschaft für neue Subscriber eines Sensor-Topics
Überarbeiten Sie die `Sensor`-Klasse, sodass eine *Retained*-Willkommensbotschaft hinterlegt wird, die den Sensor genauer beschreibt (ein Satz in natürlicher Sprache). Verwenden Sie dazu beispielsweise `args[0]`-`args[2]`, wie zuvor um das Topic zusammenzubauen.
Jeder *Subscriber*, der in Zukunft das *Topic* dieses Sensors abonniert, bekommt diese Sensorbeschreibungsnachricht zu Beginn der Verbindung übermittelt. Es gilt jeweils die letzte *Retained Message*, die über das *Topic* gesendet wurde.
Die Nachricht wird zur *Retained Mesage*, indem sie über die folgende `publish`-Methode versendet wird:

```
1 boolean retained = true;
2 client.publish(topic, payload, qos, retained);
```

Last Will and Testament (LWT)

Im Fall, dass der Sensor nicht mehr aktiv ist, würde trotzdem noch seine *Retained*-Willkommensnachricht mit der eigenen Sensorbeschreibung aktiv bleiben und neuen *Subscribern* mitgeteilt werden. Um das zu verhindern sollte im Falle des «Ablebens» des Sensorsimulatorprozesses eine *Retained*-Nachricht hinterlassen werden, die aktive und neue *Subscriber* darüber informiert, dass der Sensor in Zukunft keine Daten mehr liefert.

Aktivitätsschritt 6.24:
Ziel: Sensorabmeldung als LWT hinterlassen
Erweitern Sie die `Sensor`-Klasse um die Fähigkeit eine LWT-Nachricht zu hinterlassen. Um die LWT-Nachricht an den *Broker* zu übermitteln, wird ein

`MqttOptions`-Objekt erzeugt, befüllt und beim `connect(...)` als Parameter übergeben (s. Listing 6.7):

Die LWT-Nachricht wird nicht ausgelöst, wenn sich der *Publisher* «geordnet» über `client.disconnect()` beim *Broker* abmeldet. Falls Ihr Sensor `disconnect()` benutzt, sollten Sie dies (natürlich nur für diese Aufgabe und damit der *Publisher* «ungeordnet» abstürzen kann) auskommentieren und den laufenden Sensor dann über die Konsole abbrechen.

Listing 6.7: Setzen der LWT-Nachricht beim Verbinden des Clients

```
1 MqttConnectOptions options = new MqttConnectOptions();
2 options.setWill(topic, "ich bin weg".getBytes(),2,true);
3 client.connect(options);
```

6.7 Ausblick und Anregungen für eigene Projekte

MQTT ist einfach und kann mit der Paho Library auch problemlos aus Java-Programmen heraus benutzt werden. Es eignet sich besonders für IoT-Projekte. Die begonnene Smart-Home-Anwendung kann leicht um viele weitere Elemente erweitert werden.

6.7.1 Subscriber für Lagebilddarstellungen

Aktivitätsschritt 6.25 (fakultativ):
Ziel: Etagen Plotter
Erstellen Sie weitere *Subscriber*, die den aktuellen Zustand des überwachten Smart Homes visualisieren. Das könnten kartenbasierte Darstellungen der Positionen der Sensoren sein, bei denen Einfärbungen Alarme signalisieren oder ein Messwerte-Plotter, der die Entwicklung der Messgrößen als Kurve darstellt.
Für eine Darstellung aller gleichartigen Messwerte könnte dieses *Topic* abonniert werden:

```
SmartHome4751/#/messgroesse
```

wobei `messgroesse` entsprechend eingesetzt werden muss.
In der Callback-Methode `messageArrived(...)` kann man dann unterschieden, von welchem *Topic* jede einzelne empfangene Nachricht kam. Daraus kann dann auf den Wert für das *Wildcard* # geschlossen werden (s. Listing 6.8):

Die Nachricht muss dann daraufhin analysiert werden, ob es sich um eine Messung oder eine Willkommens- oder LWT-Botschaft handelt. Ein sehr einfacher Lösungsansatz besteht darin den Inhalt der Nachricht in eine Zahl zu konvertieren. Ergibt sich eine `Exception`, war es wohl eine Willkommens- oder LWT-Botschaft und kein Messwert. Handelt es sich um eine Nachricht mit Messwert, soll dieser als Punkt in

Listing 6.8: Matching des Wildcard-Topic Selectors

```
1 String[] topicLevels = topic.split("/");
2 String etage = topicLevels[1];
3 String raum = topicLevels[2];
```

einer Farbe, die Etage und Raum eindeutig repräsentiert, in einer Grafik dargestellt werden (y-Achse). Auf der x-Achse wird dabei die fortlaufende Zeit repräsentiert.
Der Einfachheit halber können hierbei Messwerte mit dem Zeitpunkt ihres Empfangs, und nicht mit dem Zeitpunkt der Messung, der sonst noch zusätzlich in der Nachricht repräsentiert werden müsste, angezeigt werden.

Teil III

Webbasierte Kommunikation

Hypertext Transport Protocol (HTTP)

Das World Wide Web (WWW) ist ein System von untereinander verlinkten Dateien. Es handelt sich insoweit um einen gerichteten Graphen: Knoten sind Dateien unterschiedlichen Typs. Ist der Typ Hypertext Markup Language (HTML), dann kann es Bezüge in Form von Hyperlinks (`<a href="...">`, `<img src="...">`) auf andere Dateien geben. Diese Kanten sind also gerichtet: Eine Datei «weiß nicht», welche andere Dateien auf sie selbst verweisen.

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

Exkurs: Die Erfindung des WWW

Das WWW ist vom Physiker Tim Berners-Lee ab 1989 am internationalen kernphysikalischen Großforschungszentrum CERN in Genf entwickelt worden, um wissenschaftliche Dokumente innerhalb und außerhalb der Forschungseinrichtung besser verteilen zu können. Die Entwicklung und der Betrieb des ersten Servers erfolgten übrigens auf einer der legendären NeXTcube Workstations (s. Abb. 7.1), die mit dem UNIX-artigen Betriebssystem NeXTStep ausgestattet war, welches ein Vorläufer von macOS ist.

Abbildung 7.1: Der NeXTcube im Science Museum, London, auf dem Tim Berners-Lee das WWW entwickelt hatte. Auf dem Aufkleber steht: «This machine is a server — DO NOT POWER IT DOWN!!», Bildnachweis s. S. 341

Ein dem WWW vergleichbares System war zu dieser Zeit *gopher*, das mit dem Aufkommen des WWW aber schnell an Bedeutung verloren hat. Im Vergleich zu anderen Kommunikationsprotokollen ist das dem WWW zugrundeliegende Protokoll HTTP sehr einfach und auch nicht besonders effizient. Ähnliches gilt für die im selben Zuge entwickelte Seitenbeschreibungssprache HTML. So sind Links z. B. nur unidirektional: Ein Objekt «weiß» also nicht, welche Dokumente auf es verweisen oder es einbinden. Wird eine Datei, auf die von einem anderen Dokument aus verwiesen wird (z. B. über `href` oder `src`), geändert, verschoben oder gelöscht, hat das i. Allg. ungewollte Auswirkungen auf das einbindende Dokument. Diese konzeptionelle Schwäche machte das System aber sehr einfach zu verstehen und zu implementieren. Neben der Veröffentlichung als Open Source war dies sicherlich der Hauptgrund für die schnelle Verbreitung und damit für den nachhaltigen Erfolg des WWW. Konkurrierende technisch ambitioniertere Systeme dieser Zeit wie Hyper-G/HyperWave[a], bei dem Links auf Systemebene zurückverfolgbar waren, waren viel komplexer und haben sich nicht durchsetzen können.

[a] Andrews, K., Kappe, F., & Maurer, H. (1996). The Hyper-G Network Information System. *J.UCS The Journal of Universal Computer Science, 1* (4), 206-220, doi: 10.1007/978-3-642-80350-5_20

7.1 Identifikation von Ressourcen im World Wide Web

7.1.1 MIME Type

Damit Dateien richtig angezeigt und auch sonst behandelt werden, muss ihr Typ bekannt sein. Den Typ allein aufgrund der Datei-Extension («Suffix»), also der Endung (z. B. `*.html`), zu identifizieren ist zu eingeschränkt. Ursprünglich für multimediale E-Mail-Attachements wurde deshalb ein Standard zur expliziten Kennzeichnung von Dateien entwickelt und 1996 als RFC 2046 von der Internet Engineering Task Force (IETF) veröffentlicht. Der Standard heißt *Multipurpose Internet Mail Extensions* (MIME).
Die Spezifikation des Inhaltstyps geschieht in MIME immer über einen Ober- und einen Untertyp in der syntaktischen Form:

```
Obertyp/Untertyp
```

Für den Obertyp gibt es nur eine kleine Menge von erlaubten Ausprägungen für die wichtigsten Multimedia-Arten: `text`, `image`, `audio`, `video`. Alle anderen Typen werden mit dem generischen Obertyp `application` gekennzeichnet. Speziell für E-Mail-Anwendungen gibt es noch den zusammengesetzten («*composite*») Obertypen `multipart` und `message`, die für Anwendungen im WWW keine Bedeutung haben.
Im Untertyp wird das Format genauer spezifiziert: z. B. `text/html` für HTML-Dokumente, `text/plain` für unformatierte Texte (ASCII-Text), `image/png` für Bilder im Portable Network Graphics (PNG) Format oder `audio/mp3` für MP3-Tondateien.
Im Fall von Textdateien sollte auch noch der verwendete Zeichensatz angegeben werden, denn ob 7 Bit nach der ASCII-Zuordnung (s. Tab. 7.1) oder 8 Bit ISO nach ISO 8859-1 (s. Tab. 7.2) oder 32 Bit nach Unicode zu interpretieren sind, ist ein wichtiger Unterschied, der einen Text erst lesbar machen kann.
Der Zeichensatz kann bei der MIME-konformen Spezifikation eines Text-Formates mit Semikolon getrennt und dem Schlüsselwort `charset=` angegeben werden:

```
text/html; charset=iso-8859-1
```

Tabelle 7.1: ASCII-Codierung druckbarer Zeichen (0x20-0x7F)

Hex:	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
2..	␣	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3..	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4..	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5..	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6..	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7..	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Tabelle 7.2: ISO 8859-1 Codierung (West-Europa/«Latin-1») für den Zeichenbereich von 0xA0-0xFF

Hex:	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..D	..D	..E	..F
A..		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
B..	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C..	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D..	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E..	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F..	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

7.1.2 Uniform Ressource Identifier/Uniform Ressource Locator

Die Dateien im World Wide Web sind auf unterschiedliche Rechner verteilt, die im Internet über ihre IP-Adresse oder ihren Host-Namen identifiziert werden. In den Hyperlinks, die auf Dateien im WWW verweisen, muss also mindestens der Rechnername und der Dateiname selbst enthalten sein. Das Schema, mit dem in der allgemeinsten Form Ressourcen im Internet benannt werden, heißt Uniform Resource Identifier (URI). URIs werden im RFC 3986 der IETF von 2005 spezifiziert.
Ein URI besteht generell aus den Bestandteilen:

- *Scheme*
- *Authority* (optional)
- *Path*
- *Query* (optional)
- *Fragment* (optional)

Der URI setzt sich dann zusammen aus diesen Bestandteilen und Spezialzeichen bzw. Zeichenfolgen, die die Trennung der Bestandteile markieren:

Exkurs: URL: «der» oder «die»?

URL ist die Abkürzung für «*Uniform Ressource Locator*». Die Endung -or kommt aus dem Lateinischen. Zumeist geht es um Begriffe, die ausdrücken, dass eine Person oder ein Gerät eine Handlung ausführt (es gibt Ausnahmen, die aber auch grammatikalisch anders behandelt werden, bspw. «arbor» (*fem.*)). Von vielen dieser Vokabeln gibt es Lehnwörter im Deutschen, die stets zum maskulinen Genus gehören, (z. B. der Autor, der Traktor, der Laudator, der Inkubator). Das spricht dafür, dass es «der *Uniform Ressource Locator*» und dementsprechend «**der** URL» heißen sollte.
Fälschlicherweise wird jedoch auch angenommen URL stünde für «... *Location*», was auch dem verbreiteten Sprachgebrauch entsprechen würde: «URL» wird oft mit dem Adresse oder Ort (engl. *location*) einer Ressource gleichgesetzt. Die Endung -ion ist auch in vielen Fremdwörtern aus dem Lateinischen anzutreffen. Das sind meistens Begriffe mit femininem Genus, die etwas Abstraktes (z. B. die Religion, die Nation) bezeichnen oder für die Substantivierung von lateinischen Verben (bspw. Operation, Adoption) gebraucht werden. Deshalb hört man häufig «**die** URL» im tatsächlichen Sprachgebrauch.
Wahrscheinlich ist das der Grund dafür, dass im Duden als maßgebliche Dokumentation der deutschen Sprache, beide Geschlechter für URL angegeben sind. Man kann also sowohl «der URL», als auch «die URL» sagen.

Scheme + `://` + *Authority* + *Path* + [`?` + *Query*] + [`#` + *Fragment*]
Die Trennung von *Authority* und *Path* kann erkannt werden, da *Path* mit «`/`» beginnen muss und *Authority* kein «`/`» enthalten darf.

Scheme

Wenn *Scheme* ein Zugriffsmechanismus ist (i. Allg. ein Kommunikationsprotokoll wie `http`, `ftp`, `gopher`, `rmi`, `iiop` usw.), wird der URI als Spezialform *Uniform Resource Locator* (URL) genannt. Ein Beispiel für ein URI-Schema, das im Gegensatz dazu kein Zugriffsmechanismus bezeichnet, ist doi (*Digital Object Identifier*).
Für das WWW sind die Netzwerkschemata `https`, `http` und `ftp` besonders relevant.

Authority

Authority ist der «Ort», an dem die Ressource im Internet abgerufen werden kann. Es handelt sich also um einen Rechner, der über *Host*-Namen oder IP-Adresse bezeichnet wird.

Exkurs: *Path* vs. *Ressource*

Ursprünglich war das WWW als reines Informationssystem gedacht, über das (statische) Dateien ausgetauscht werden sollten. Es hat sich gezeigt, dass sich das WWW auch als Systemarchitektur für dynamische (also beim Aufruf generierte) Informationen eignet und auch zur Verwaltung von ganz abstrakten IT-Ressourcen, die gar keine Entsprechung als Datei haben. Deshalb wird *Path* inzwischen weniger als konkreter Dateipfad angesehen, sondern viel mehr als abstrakte Spezifikation einer IT-Ressource. Das ist auch der Grund, warum der Begriff «Ressource» in URI und URL verwendet wird.

Authority → `141.19.1.171`
Authority → `www.hs-mannheim.de`
Optional kann nach dem Rechnernamen oder der IP-Adresse mit «`:`» getrennt der *Port* angegeben werden, auf dem das angegebene Kommunikationsprotokoll auf dem Rechner bereitgestellt wird. Wird die Port-Angabe weggelassen, wird der protokollspezifische Standard Port (z. B. Port 80 bei HTTP) verwendet.
Authority → `141.19.1.171:8080`
Authority → `www.hs-mannheim.de:8080`
Optional kann bei Netzwerkschemata auch eine Login-Information als Teil der *Authority*-Information angegeben werden. Dabei kann entweder nur der Login-Name oder Login-Name und Passwort angeben werden. Die Trennzeichen zum Rechnernamen der der IP-Adresse sind «`:`» zwischen Login-Name und Passwort und «`@`» zwischen Login-Information und Rechneradresse:
Authority → `s.leuchter@141.19.1.171`
Authority → `student:Turing@www.hs-mannheim.de`

Path

Der Pfad gibt den Dateinamen auf dem angefragten Rechner an. Es handelt sich aber nicht um den «richtigen» Dateinamen, sondern nur um einen Pfad relativ zu einem Wurzelverzeichnis, von dem aus der Webserver Dateien bereitstellen soll.

Beispiel

Wurzelverzeichnis → `/usr/share/public_html`
Path → `/docs/test.pdf`

Exkurs: Wurzelverzeichnis für *Path* bei Apache HTTP Server

Beim weitverbreiteten Apache HTTP Server (*httpd*) wird das Wurzelverzeichnis in der Konfigurationsdatei `conf/httpd.conf` (oft zu finden in `/etc/httpd`) durch den folgenden Direktiveneintrag gesetzt:

```
<Directory "/usr/share/public_html">
```

ausgelieferte Datei → `/usr/share/public_html` + `/docs/test.pdf`
→ `/usr/share/public_html/docs/test.pdf`

Query

Der URI-Bestandteil *Query* ist optional. Über ihn werden bei der HTTP-Methode GET (s. u.) Aufrufparameter an dynamisch generierte Ressourcen übermittelt (s. Abschnitt 7.2.2). Man kann im URI beliebig viele Parameter codieren. Jeder Parameter hat einen eindeutigen Namen und kann nach einem «**=**»-Zeichen einen Wert haben. Parameter werden durch «**&**» voneinander getrennt. Groß-/Kleinschreibung ist relevant.
Query → `Parameter1`
Query → `Parameter1=42`
Query → `vorname=Sandro&nachname=Leuchter`
Query → `a=3&b=4&c=9` (ist bedeutungsgleich zu `b=4&c=9&a=3`)
Query → `cmd=loeschen&id=4711`

Fragment

Der Fragment-Teil ist optional. Bei HTML-Dateien handelt es sich um einen Anker, der als «Ansprung»-Adresse innerhalb eines Dokuments angegeben ist (`<a name="marke">` ↔ `http://...#marke`). Allgemein kann der Fragment-Teil des URI auch für andere Zwecke genutzt werden, wenn die spezifizierte Ressource nicht im HTML-Format ist.

Schema `file://`

Für lokale Ressourcen wird das Schema `file` benutzt, das eine andere Struktur der URI impliziert als die Netzwerkschemata. *Authority* und *Query* werden hier nicht verwendet.

`file://` + *Path* + `#` + *Fragment*
Path → /usr/share/public_html/docs/test.pdf (in UNIX-artigen Umgebungen)
Path → /C:/User/public_html/docs/test.pdf (unter Windows)
`#` + *Fragment* sind optional.
URIs dürfen keine Leerzeichen oder Sonderzeichen enthalten, sondern müssen komplett aus druckbaren ASCII-Zeichen (s. Tab. 7.1) bestehen. Sollen andere Zeichen in einem URI vorkommen, müssen sie quotiert werden, also durch die Kombination mehrerer druckbarer ASCII-Zeichen repräsentiert werden. Dazu schreibt RFC 3986 vor, dass das `%`-Zeichen zusammen mit den zwei Hexadezimal-Zeichen (`0..F`) verwendet werden muss, um den Code des nicht erlaubten Zeichens auszudrücken. Daneben wird das «+»-Zeichen zur einfacheren und lesbareren Quotierung des Leerzeichens («`%20`» ist gleichwertige Quotierung neben «+») verwendet. Außerdem müssen wegen der Sonderbedeutung der Zeichen «`?`», «`&`», «=» und «#» zur Abtrennung der Bestandteile zwischen und innerhalb von *Path*, *Query* und *Fragment* auch diese Zeichen quotiert werden.
Beispiele für quotierte Zeichen in URI:
`%20` → « » (Leerzeichen), s. Tab. 7.1
`%25` → «%», s. Tab. 7.1 (da «`%`» Sonderbedeutung zum Quotieren hat)
`%D6` → «Ö», wenn `character-encoding="iso-8859-1"` (Westeuropäisch), s. Tab. 7.2
`%D6` → «ض», wenn `character-encoding="iso-8859-6"` (Arabisch)

In Java können die Klassen `java.net.URLEncoder` und `java.net.URLDecoder` verwendet werden, die die statischen Methoden `encode(String s, String enc)` bzw. `decode(String s, String enc)` anbieten, um die «`%`»-Quotierung zu in einem String für einen URI codieren bzw. die Codierung wieder zurückzuübersetzen.

7.2 Hypertext Transfer Protocol

Das Hypertext Transfer Protocol (HTTP) ist das Kommunikationsprotokoll, das ursprünglich für das WWW entwickelt wurde. Inzwischen wird das HTTP auch für andere Zwecke eingesetzt, auf die wir später in diesem Kurs noch zu sprechen kommen

werden. Der ursprüngliche Zweck des HTTP ist einem Client zu ermöglichen Dateien von einem entfernten Server abzurufen.
HTTP ist ein Anwendungsprotokoll, das über TCP/IP-Sockets abgewickelt wird. Der Standard Port von HTTP ist 80. Wird HTTP über Transport Layer Security (TLS) – oft auch als Secure Sockets Layer (SSL) genannt – abgewickelt, wird von HTTPS (HTTP Secure) gesprochen. TLS ist ein Standard, bei dem die TCP-Kommunikation verschlüsselt wird. HTTPS hat als Standard Port 443 (TCP, jedoch TLS-verschlüsselt).

7.2.1 HTTP-Protokollablauf («non persistent» und «persistent»)

Der Ablauf des HTTP ist sehr einfach: In der Basis-Version öffnet der Client eine TCP-Socket-Verbindung zum Server, schickt eine HTTP Request Nachricht, die der Server direkt (synchron) mit einer HTTP Response Nachricht beantwortet. Dann wird die TCP-Verbindung geschlossen. Dieses Kommunikationsmuster wird «*non persistent*» HTTP genannt, weil die Verbindung nicht bestehen bleibt. Da die Nutzung von HTTP im WWW aber normalerweise so aussieht, dass der Client eine ganze Reihe von Dateien kurz nacheinander von demselben Server abruft (z. B. wenn eine abgerufene HTML-Seite mehrere Grafiken einbindet, die sich auf demselben Server im selben Verzeichnis wie das HTML-Dokument befinden, s. Abb. 7.2, S. 212), ist dieses Vorgehen sehr ineffizient. Der Aufbau und der Abbau der Verbindung dauert einige Zeit, da dafür jeweils einige TCP-Pakete zwischen Client und Server ausgetauscht werden müssen.
Eine erweiterte Fassung des Protokolls («*persistent*» HTTP) sieht deshalb vor, dass die Verbindung zwischen Client und Server für einige Zeit bestehen bleibt und währenddessen mehrere HTTP Request/HTTP Response Dialoge ausgeführt werden können (s. Abb. 7.3, S. 213). Die Antwort kommt jedoch weiterhin synchron vom Server auf Anfrage vom Client.

7.2.2 HTTP Request

Die Anfrage des Clients an den HTTP Server erfolgt im Format eines HTTP *Requests*. Das ist eine Nachricht, die aus einer Kopfzeile, beliebig vielen Header-Zeilen, einer Leerzeile und einem etwaigen optionalen Inhalt besteht.

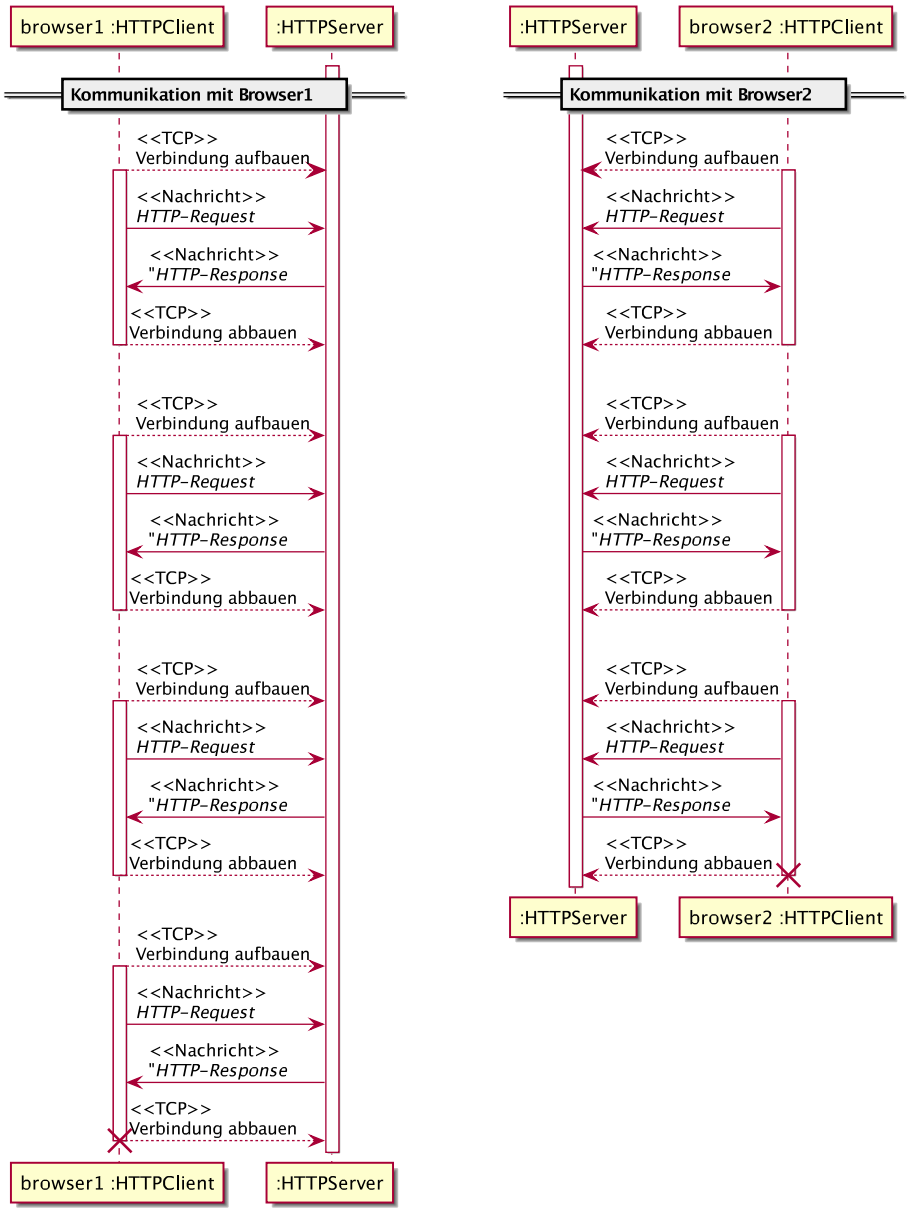


Abbildung 7.2: Prinzipieller Ablauf bei «*non persistent*» HTTP: Zwischen jedem Request-Response-Paar muss die Verbindung ab- und wieder aufgebaut werden.

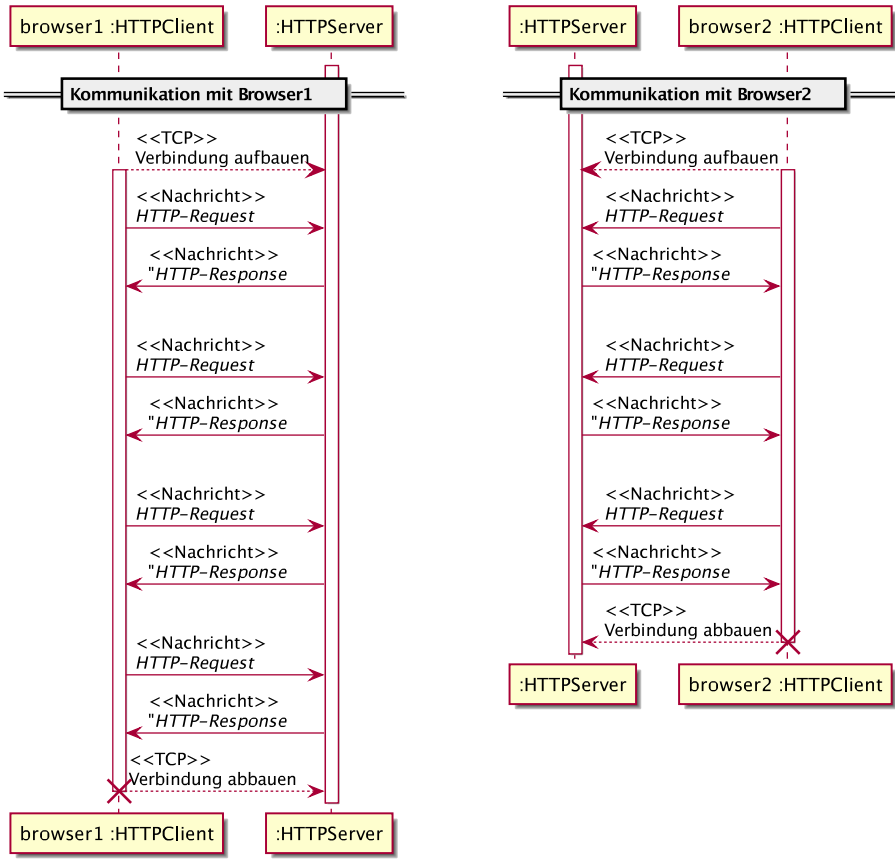


Abbildung 7.3: Prinzipieller Ablauf bei «*persistent*» HTTP: Zwischen jedem Request-Response-Paar bleibt die Verbindung bestehen und kann weiter benutzt werden.

Exkurs: HTTP-Versionen

Im Moment gibt es die Versionen HTTP/1.0 (RFC 1945 von 1996), HTTP/1.1 (RFC 2616 von 1999) und HTTP/2 (RFC 7540 von 2015). HTTP/2 unterscheidet sich von den früheren Fassungen dadurch, dass *Streams* eingeführt wurden und so nicht mehr nur ganze Datei-Objekte übertragen werden müssen. HTTP/0.9 – auch wenn dies keine offizielle Version und IETF-Spezifikation ist – war die ursprüngliche und funktional noch sehr eingeschränkte Spezifikation von Tim Berners-Lee von 1991, die heute keine praktische Bedeutung mehr hat. Verbreitete HTTP-Server wie der Apache HTTP-Server und Nginx unterstützen jedoch das HTTP/0.9 noch.

Zeilen

Zeilen gliedern dieses Nachrichtenformat. Eine Zeile wird jeweils durch die zwei nicht-druckbaren Zeichen <*Carriage-Return*> und <*Line-Feed*> beendet. <*Carriage-Return*> hat den ASCII-Code `13` (`0x0D`). In Texten wird es oft als `<CR>` wiedergegeben, in Java Strings wird es als `\r` («*return*») quotiert. <*Line-Feed*> wird entsprechend als ASCII Code `10` (`0x0A`) bzw. `<LF>` und `\n` («*new line*») wiedergegeben.

Startzeile

Die Startzeile ist die erste Zeile im HTTP Request. Sie folgt dem Format:
Kommando `<SP>`+ *Parameter* `<SP>`+ *Protokoll* `<CR><LF>`
(`<SP>`+ sind hier beliebig viele, aber mindestens ein Leerzeichen («*space*»))
Kommando kann entweder `GET`, `POST` oder `HEAD` (HTTP 1.0) bzw. `GET`, `POST`, `HEAD`, `PUT` oder `DELETE` (HTTP 1.1) sein. Diese Kommandonamen werden auch HTTP-Methoden («*method*») genannt.
Parameter sind die Teile *Path* + `?` + *Query* + `#` + *Fragment* des URL. Wie schon in dem URL sind demnach `?` + *Query* und `#` + *Fragment* optional. Der Parameter kann aufgrund der Syntax von URLs keine Leerzeichen enthalten und hat auch Sonderzeichen URL-encodiert.
Protokoll ist die Version des HTTP-Protokolls.
In der Startzeile wird also die Anfrage spezifiziert. `GET` und `POST` (zum Unterschied später mehr) sind «normale» Anfragen, bei denen eine Datei abgerufen werden soll, `HEAD` bezieht sich auf eine Anfrage *über* eine Datei. `PUT` ist eingeführt worden, um

Exkurs: Funktion der HTTP Header-Zeilen

Bei vielen Header-Zeilen geht es darum, die Fähigkeiten des Clients zu beschreiben, die einen Einfluss auf die Auswahl oder Transformation von Ressourcen auf dem Server haben können. Beispielsweise gibt es den Header `Accept-Language`, dessen Wert eine Komma-getrennte Liste von ISO 639 Sprach-Codes ist (z. B. `de` für Deutsch, `en` für Englisch oder `tlh` für Klingonisch). Der Server kann dies auswerten und ein zur konkreten Anfrage inhaltlich gleichwertiges Dokument in einer der angegebenen Sprachen ausliefern.

eine Datei vom Client auf den Server hochladen zu können und mit `DELETE` soll eine Datei auf dem Server gelöscht werden.

7.2.3 Header-Zeilen

Nach der Startzeile und bis zur ersten Leerzeile folgen beliebig viele Header-Zeilen. Jede Header-Zeile hat das Format
Header-Name + `:` + *Header-Wert*
Der Client kann beliebige Header einfügen. Die Bedeutung von einigen Header-Zeilen ist in HTTP definiert (es gibt Unterschiede zwischen den HTTP-Versionen), es können aber auch prinzipiell weitere eingeführt werden, wenn der Server oder Programme auf dem Server, die dynamische Inhalte generieren, diese interpretieren können.

7.2.4 HTTP Response

Auch die Antwort des HTTP Servers auf einen HTTP Request besteht aus `<CR><LF>` getrennten Zeilen.
Die Startzeile beinhaltet den Ergebnis-Code, mit dem der HTTP Server den Ausgang der Abarbeitung des HTTP *Requests* codiert. Außerdem ist in der Startzeile die HTTP Version des Servers vermerkt.
Wenn der Request syntaktisch korrekt war und der HTTP Server die Anfrage bearbeiten konnte, ist das Ergebnis «Erfolg». Das wird mit dem Antwort-Code `200` repräsentiert. Unterschiedliche abweichende Situationen werden durch andere Codes angezeigt. Die wichtigsten sind in Tabelle 7.3 aufgelistet.
Weiterhin kann es beliebig viele Header-Zeilen in der HTTP Response geben, die syntaktisch gleich sind zu den Header-Zeilen im Request. Zwei Header-Einträge sind jedoch mindestens erforderlich: `Content-Type`, in dem der MIME-Typ der Nutzda-

Tabelle 7.3: HTTP-Fehlercodes und ihre Bedeutung

Code	Kurzantwort	Erklärung
200	`OK`	erfolgreiche Abarbeitung des *Requests*
301	`Moved Permanently`	Die angeforderte Ressource ist unter einem anderen URL zu finden, der weiter unten in der HTTP-*Reponse* auch noch mitgeteilt wird. Der Client soll sie von dort erneut anfordern.
304	`Temporary Redirect`	Beim Header `If-modified-since` (s. u.) verwendet um zu signalisieren, dass die lokale Version des Clients noch aktuell ist.
307	`Not Modified`	Die angeforderte Ressource ist im Moment unter einem anderen URL zu finden, der weiter unten in der HTTP-*Reponse* auch noch mitgeteilt wird. Der Client soll die Ressource von dort erneut anfordern. In Zukunft gilt aber weiterhin der ursprüngliche URL.
402	`Payment Required`	Diese Antwort ist zwar schon im Standard vorgesehen, wird aber noch nicht verwendet um anzuzeigen, dass der Zugriff verweigert wurde, weil eine Zahlungspflicht noch nicht erfüllt wurde.
404	`Not Found`	Die angeforderte Ressource wurde nicht unter dem angegebenen Pfad gefunden.
414	`URI Too Long`	Der im Request enthaltene URI ist zu lang um ihn auf der Seite des Servers zu verarbeiten.
451	`Unavailable For Legal Reasons`	Der angeforderte Inhalt wird nicht ausgeliefert, weil der Zugriff auf die Ressource aufgrund einer staatlichen Anordnung nicht erlaubt ist.

Exkurs: Fehlercode 451

Im dystopischen Science-Fiction Roman «Fahrenheit 451» von Ray Bradbury aus dem Jahr 1953 geht es um eine Gesellschaft, in der der Besitz von Büchern staatlicherseits verboten ist, weil sie zu selbständigem Denken und politischem Ungehorsam führten. Sogenannte «Feuerwehrleute» sind dafür zuständig, Bücher aufzuspüren und zu verbrennen. Der Titel des Romans bezieht sich auf die Temperatur, bei der Papier anfängt zu brennen (451° F ≈ 233° C).

ten angegeben wird und der Header `Content-Length`, der die Länge der Nutzdaten in Bytes beinhaltet. Der Block der Header-Zeilen wird wiederum durch eine Leerzeile abgeschlossen.
Nach dem Header-Block kommen die Nutzdaten der Antwort. Im Fall der erfolgreichen Auslieferung einer Datei ist das der eigentliche Inhalt der Datei.

7.2.5 Sessions

Von der Protokolldefinition her ist HTTP zustandslos: Jedes Paar von HTTP Request und Response ist unabhängig von allen anderen. Für den Abruf von Dokumenten mag das ausreichend sein. Bei interaktiven Web-Anwendungen gehören jedoch mehrere Request-Response-Paare zusammen zu einer Session. Beispielsweise besteht eine typische Session beim Onlineshopping aus der Suche nach Produkten, aus dem Vormerken von Produkten für die Bestellung durch das Legen in einen digitalen Einkaufswagen und aus dem anschließenden Check-Out des Einkaufswagens an der digitalen Kasse, wobei dazu das Eingeben der Zahlungsart, das Einwilligen in Geschäftsbedingungen, das Eingeben von Liefer- und Rechnungsadresse etc. gehört.
HTTP/1.0 bietet mit dem Header «`Connection: Keep-Alive`» (also Header-Name ist `Connection` und Wert ist «`Keep-Alive`») eine Möglichkeit, mehrere HTTP-Connection-Response-Paare während einer offenen Socket-Verbindung (synchron) auszutauschen (in folgenden HTTP-Versionen ist dies das Standardverhalten). Dadurch wird der Aufwand beim Öffnen und Schließen der Socket-Verbindung für folgende Request-Response-Paare gespart. Die einzelnen Request-Response-Paare ließen sich zwar nun einem gemeinsamen Zusammenhang zuordnen – die Verbindung wird aber früher oder später wieder geschlossen (per Default in der Größenordnung von 5–30 Sekunden) und damit ist dieser Mechanismus nicht dafür geeignet, Sessions auf der Ebene einer Anwendung umzusetzen.

Es wurden einige unterschiedliche Wege entwickelt, um Zustandsinformation zwischen mehreren getrennten HTTP-Verbindungen zu bewahren bzw. mehrere getrennte HTTP-Request-Response-Paare in einen Session-Zusammenhang zu bringen. Im Folgenden wird ein wichtiger Mechanismus dargestellt, der aber von Benutzern deaktiviert werden kann. Alleine schon deshalb ist es erforderlich, auch alternative Wege dafür zur Verfügung zu stellen.

7.2.6 Cookies

Der HTTP Server kann Cookies benutzen um einzelne Benutzer über mehrere Verbindungen hinweg zu identifizieren, die zeitlich beliebig lange auseinanderliegen dürfen. Cookies werden in RFC 2109 von 1997 spezifiziert. Die IP-Adresse ist dafür nicht aussagekräftig, da beispielsweise mehrere getrennte Benutzer auf demselben Client-Rechner sein könnten (Multi-User-Betrieb). Außerdem könnten verschiedene Benutzer versuchen aus einem per *Network Address Translation* (NAT) abgeschirmten Subnetz zuzugreifen oder denselben Proxy[1] verwenden.
Stattdessen wird ein eindeutiger Textschlüssel auf dem Client gespeichert und bei späteren Zugriffen wieder zurück übertragen. Der Ablauf für Cookie-Nutzung ist folgendermaßen (s. Abb. 7.4):

1. Ein Client schickt einen HTTP Request zu einem Server, den der Benutzer vorher noch nie besucht hat.

2. Der Server legt ein neues Profil für den Benutzer in einer serverseitigen Datenbank an. Ein Teil dieses Profils ist eine eindeutige Kennung, über die der Benutzer später wieder seinem Profil zugeordnet werden soll.

3. Diese Kennung wird der HTTP Response in einem speziellen Header-Eintrag mitgeteilt. Dieser Header heißt `Set-Cookie`. Als Wert kann in diesem Header eine Liste von Cookies angegeben werden. Ein Cookie hat dabei immer einen Namen und einen Wert. Zusätzlich können Attribute angegeben werden, die z. B. die Gültigkeit näher regeln.

4. Empfängt der Client eine HTTP Response mit gesetztem `Set-Cookie` Header, soll der Client den Cookie, also den Wert des `Set-Cookie` Headers, lokal

[1] Das ist eine Zwischenstation, über die HTTP Requests und Responses transparent übermittelt werden.

speichern. Dabei muss der Client auch den Host speichern, von dem der Cookie kam.

5. Bei jeder zukünftigen Anfrage, die dieser Client in Zukunft wieder an den Server schickt, von dem ein Cookie lokal beim Client gespeichert worden ist, soll der lokal gespeicherte Wert als zusätzlicher Header in den HTTP Request aufgenommen werden. Der Header heißt `Cookie`.

6. Der Server kann nun über den Wert des Header-Feldes und Abgleich mit der Datenbank beim Server den Benutzer identifizieren. Das ist über einzelnen TCP-Socket-Verbindungen hinweg möglich. Prinzipiell ist die Zeitdauer zwischen `Set-Cookie` (in der HTTP Reponse) und Cookie im folgenden Request unbegrenzt. Je nach Anwendung wird man die Haltbarkeit eines Cookies aber begrenzen wollen. Ist nach dieser Zeitspanne ein Cookie verfallen, wird es nicht mehr als Header in den HTTP Request aufgenommen.

Über Attribute des Cookies kann der Server genauer regeln, bei welchen URLs der Cookie mitgeschickt werden soll. Es ist sogar möglich, dass der Cookie an ganz andere Server zurückgegeben wird als der Ursprung des Cookies. Solche sogenannten *Cross Domain Cookies* werden z. B. in der Online-Werbung benutzt, um interessenprofilgeleitet Werbung zu schalten. Aus datenschutzrechtlicher Sicht ist ein Nachteil an *Cross Domain Cookies*, dass anbieterübergreifend ein WWW-«Bewegungsprofil» angelegt werden kann, das missbräuchlich verwendet werden kann um Benutzer auszuspähen.
Anwendungsmöglichkeiten für Cookies sind z. B. Autorisierung, Anlegen und Verwalten eines digitalen Einkaufswagens, Empfehlungen und Werbung.
Da Cookies datenschutzrechtlich bedenklich sind und deshalb viele Benutzer die Verwendung von Cookies ablehnen oder einschränken, werden Sessions anders verwaltet. Dabei wird oft auf der Serverseite eine Datenstruktur im Speicher angelegt, die einen einzelnen Benutzer repräsentiert. Damit der Server speichermäßig nicht überlastet wird, ist es sinnvoll, Sessions nach einer gewissen Zeit der Inaktivität wieder zu löschen. Diesen Timeout kann man in der Konfiguration individuell für einzelne Web-Anwendungen oder auch den ganzen Server setzen. Eine gängige Dauer sind 30 Minuten: Erfolgt so lange kein Zugriff, wird angenommen, dass die Session beendet ist und die Datenstruktur für diese Session wieder freigegeben werden kann. Etwaige

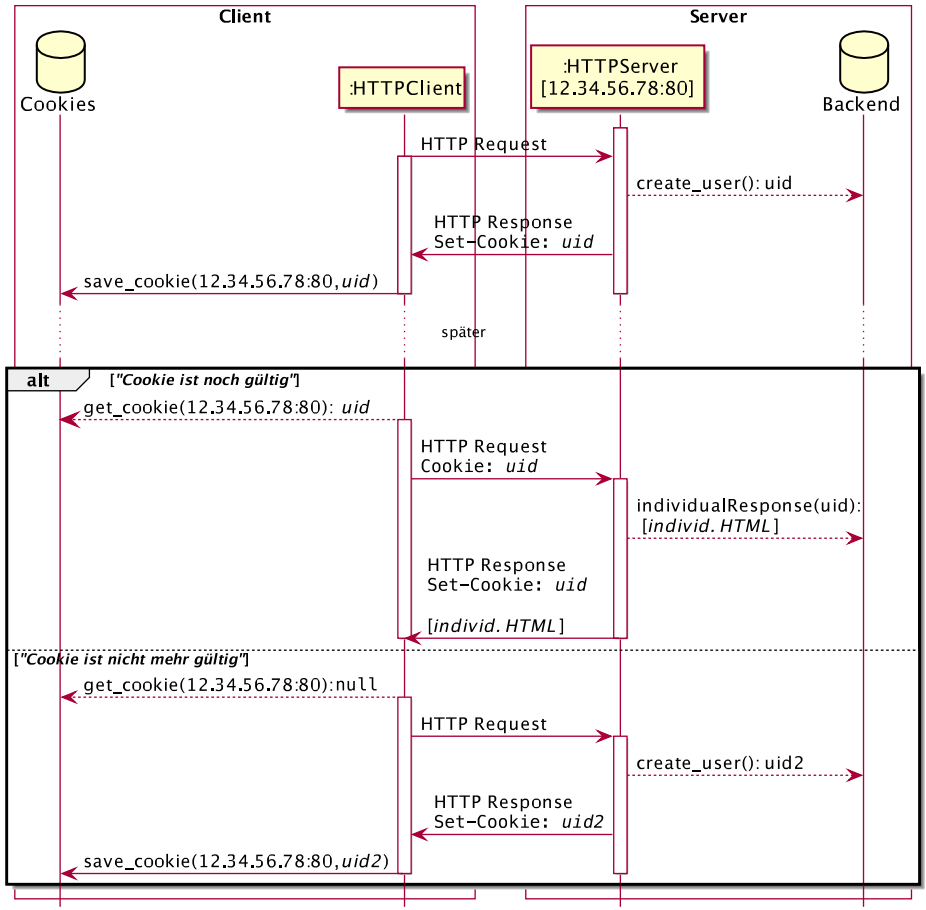


Abbildung 7.4: Prinzipieller Ablauf bei der Verwendung von Cookies zur Wiedererkennung individueller Clients

spätere Zugriffe können dann dieser Sitzung nicht mehr zugeordnet werden und starten ggf. eine neue Session.

7.2.7 Conditional GET

Viele Inhalte im WWW sind statisch und ändern sich nur selten oder sogar nie. Das trifft insbesondere auf Bilder und andere multimediale Daten zu, deren Dateien oft ziemlich groß sind. Deshalb ist es sinnvoll einmal aus dem WWW abgerufene Inhalte lokal zu speichern und beim nächsten Abruf desselben URL die lokal gespeicherte Version aus dem Cache zu verwenden anstatt die Ressource erneut beim Server abzurufen.
Damit nicht veraltete Inhalte angezeigt werden, beinhaltet HTTP eine Möglichkeit abzufragen, ob sich eine Ressource seit einem bestimmten Zeitpunkt geändert hat. Das *conditional GET* ist ein HTTP Request mit der Methode `GET`, die den Header `If-Modified-Since` aufweist. Der Wert dieses Headers ist eine Zeitstempelrepräsentation. Wird dem `GET`-Kommando so eine Bedingung wie in Abb. 7.5 beigefügt, kann die Antwort des Servers entweder `304 Not Modified` sein, wenn sich der Inhalt auf dem Server nicht von der lokal gespeicherten Version unterscheidet oder er sendet ganz normal die neuere Datei unter dem Ergebnis-Code `200 OK` zurück. In diesem Fall kann der Client die neuere Version der Datei im Cache speichern und als das letzte Datum den aktuellen Zeitpunkt vermerken. Dieses Datum wird dann zukünftig als Wert für den `If-Modified-Since` Header-Wert benutzt.
Bei dynamisch generierten Antworten ist die Wahrscheinlichkeit groß, dass sich Änderungen im Gegensatz zu früheren Aufrufen ergeben. Deshalb werden i. Allg. HTTP Responses zu URLs mit *Query*-Anteil, die am «**?**» in dem URL zu erkennen sind, nicht gecached.

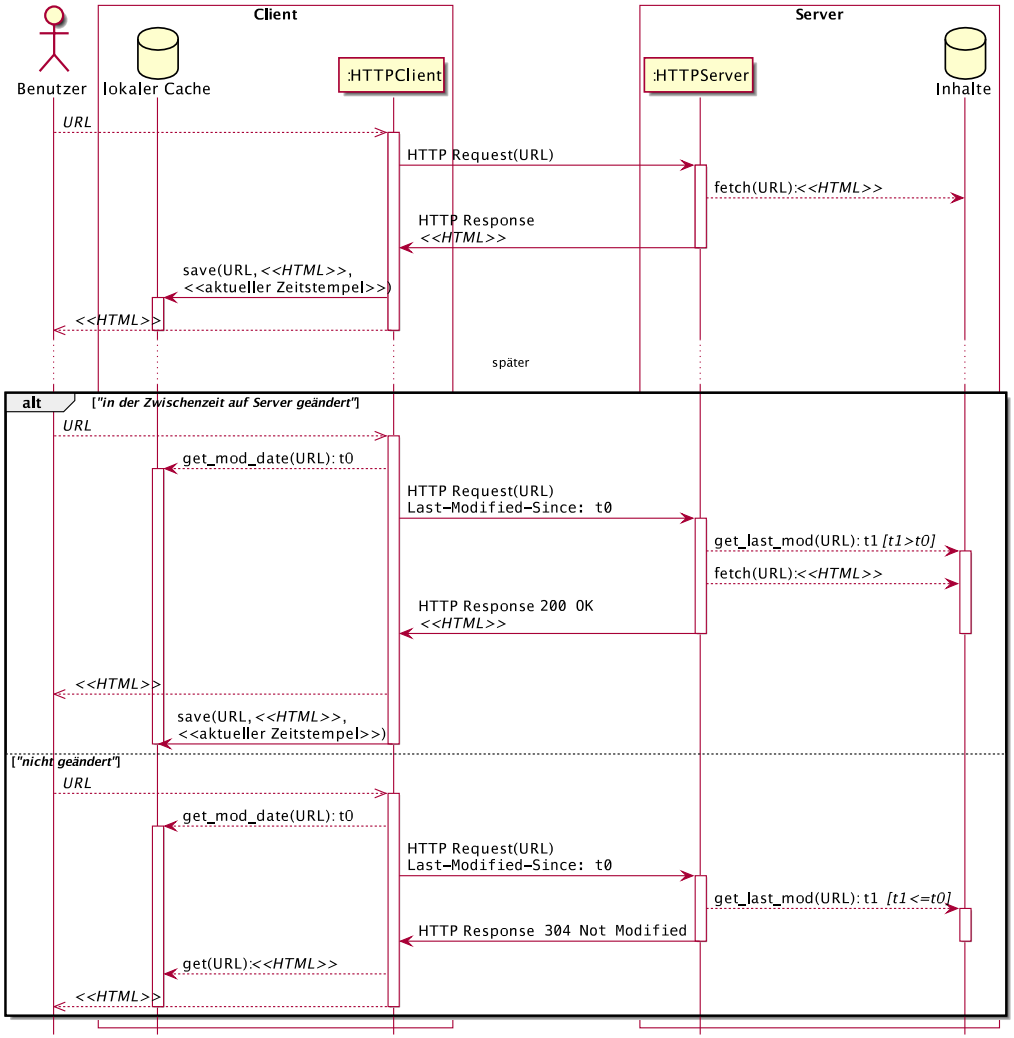


Abbildung 7.5: prinzipieller Ablauf des Protokolls zwischen HTTP Client und Server bei Verwendung von *Conditional GET*

7.3 Webserver

Der Webserver ist der Service, der Inhalte per HTTP ausliefert.

7.3.1 Statische und dynamische Inhalte

Ein Webserver muss neben der Funktion, Dateien auszuliefern auch dynamische Inhalte erzeugen können. Dazu kann man Programme einbinden, die ausgeführt werden, wenn ein bestimmter URL aufgerufen wird. Das Programm erzeugt dann eine Ausgabe (z. B. auf dem Standardausgabekanal), die dann vom HTTP Server in eine HTTP Response verpackt an den Client zurückgesendet wird.
Oft wird das einfach so gelöst, dass ein Verzeichnis im Filesystem des Webservers eine besondere Bedeutung zugewiesen bekommt: Wird eine Datei aus diesem Verzeichnis über einen HTTP Request abgerufen, behandelt der Webserver sie nicht als normale Datei, die geöffnet und in den HTTP Response Nutzdatenbereich kopiert wird, sondern führt die Datei direkt als Programm oder als interpretiertes Script aus, dessen Output in die HTTP Response kopiert wird.
Es gibt kaum einen Fall, wo es für Web-Anwendungen ausreicht, ein Programm auf immer die gleiche Art auszuführen. Stattdessen werden Aufrufparameter an das Programm übertragen. Ganz ähnlich wie die Kommandozeilenparameter beim Aufruf eines Java `main`-Hauptprogramms als `String[]` übergeben werden, werden einem Programm vom Webserver Parameter übergeben.
Diese Parameter rühren entweder aus dem aufgerufenen URL her (im Fall der HTTP-Methode `GET`), wo sie als *Query*-Anteil nach einem «**?**» codiert sind (s. Abschnitt 7.1.2), oder sie werden als Inhalt nach dem Header-Block in der HTTP-Request-Nachricht repräsentiert (Methode `POST`). Tendenziell wird `POST` bei umfangreicheren Daten benutzt (z. B. wenn der Textinhalt einer E-Mail-Nachricht an ein Web Mail Programm übertragen werden soll) oder bei Daten, die nicht im Adresseingabefeld eines Webbrowsers sichtbar sein sollen (z. B. ein Passwort).
Die Parameterwerte werden entweder (im Fall von `GET` möglich) in dem URL direkt codiert, sodass das Folgen eines Hyperlinks die Funktion inkl. Parameterübergabe auslöst, oder über ein HTML-Formular eingegeben. Im `ACTIONACTION`-Attribut des `FORM`-Elements kann der URL des aufzurufenden Programms angegeben werden. Das Attribut `METHOD` des `FORM`-Elements bestimmt die Methode (`GET` oder `POST`).

Exkurs: (Fast) Common Gateway Interface (CGI)

«CGI» steht hier für Common Gateway Interface. Das ist ein Standard, wie Webserver Programme aufrufen und die *Query*-Parameter und Header aus dem HTTP Request dem Programm zur Verfügung stellen (als Pendant zu Kommandozeilenparametern). Dies geschieht über Environment-Variablen und den Standardeingabekanal. Außerdem wird eine Schnittstelle bereitgestellt, über die das Ergebnis des Programmaufrufs dem Webserver zurück übergeben wird, so dass dieser eine passende HTTP Response-Nachricht daraus erstellen kann (über den Standardausgabekanal).
Es gibt eine effizientere Art für Webserver externe Programme zu starten als den CGI-Standard: Bei Fast-CGI wird der Interpreter-Prozess nach dem Ende des interpretierten Programms nicht beendet, sondern für zukünftige CGI-Aufrufe wiederverwendet. Dadurch starten Skripts mit Fast-CGI schneller als mit CGI, wo immer erst der Interpreter geladen und gestartet werden muss.

Wie kann aber der Webserver feststellen, dass ein URL als Programmaufruf und nicht als Dateiabruf zu interpretieren ist? Es gibt mehrere übliche Methoden, um das zu erreichen. Ein Webserver könnte so konfiguriert werden, dass immer, wenn der URL auf eine Ressource verweist, die ein spezielles Suffix (z. B. `*.php`) hat, ein bestimmter Interpreter aufgerufen wird, dem die abgerufene Datei als Programm eingegeben wird. Eine weitere Möglichkeit ist, ein ganzes Verzeichnis so zu markieren, dass alle Dateien darin als Programme ausgeführt und nicht als Dateien abgerufen werden. Eine oft anzutreffende Konvention für solch einen Verzeichnis-Namen ist `cgi-bin`. Aber das ist eine Frage der Konfiguration des Webservers.

Praktikum

Aktivitätsschritte:

Unter `http://verteiltearchitekturen.de/vol01/VAR-WebServer-solution.zip` können Sie eine Musterlösung zu diesem Praktikum herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-WebServer_solution`.

Im folgenden Praktikum untersuchen Sie HTTP Requests Ihres Webbrowsers und programmieren einen einfachen Webserver. Sie wenden dabei Techniken aus Kapitel 3 (TCP Sockets mit Java) an.

Aktivitätsschritt 7.1:
Ziel: Projekt downloaden und in Eclipse importieren
Laden Sie das ZIP-Archiv mit dem Projekt `VAR-WebServer` unter `http://verteiltearchitekturen.de/vol01/VAR-WebServer.zip` herunter. Importieren Sie das ZIP-File in Eclipse mit der Funktion *File* → *Import...* → *General* / *Existing Projects Into Workspace*. Wählen Sie dort die Option *Select Archive File*. Sie sollten auch darauf achten, dass die Option *Copy Projects Into Workspace* aktiviert ist, sonst arbeiten Sie möglicherweise nur auf den Dateien in Ihrem Download-Ordner und nicht in Ihrem Eclipse Workspace.

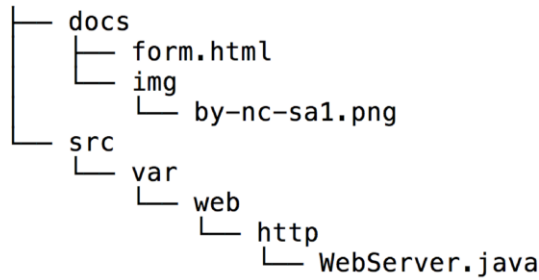


Abbildung 7.6: Verzeichnis- und Dateibaum für das Projekt `VAR-WebServer`

Als Ergebnis sollte das Projekt `VAR-WebServer` in Ihrem Package Explorer in Eclipse auftauchen. Der `src`-Ordner dieses Projektes enthält das Package:

- `var.web.http`

In dem Package ist eine Klasse namens `WebServer` bereits angelegt, die Sie in den nächsten Aktivitätsschritten zu einem einfachen, aber funktionsfähigen Webserver ausbauen werden.
In Ihrem Eclipse Workspace Directory sollte ein neues Verzeichnis für das Projekt entstanden sein. In dem Projektverzeichnis gibt es mindestens neben dem Folder `src/` noch einen Folder `docs/`. Der Webserver, den Sie in den nächsten Aktivitätsschritten entwickeln werden, soll Dateien aus diesem Ordner an die HTTP Clients ausliefern. Abb. 7.6 zeigt die Struktur unterhalb des Folders `VAR-WebServer/` (ohne das Directory `bin/`, das in Eclipse nicht angezeigt wird und möglicherweise auch erst im Build-Prozess automatisch erzeugt wird).

7.4 Aufgabe: Webserver programmieren

Der Standard Port für HTTP ist `80`. Wird dieser Port benutzt, kann die Angabe im URL entfallen (`http://verteiltearchitekturen.de/` statt `http://verteiltearchitekturen.de:80/`). Allerdings sind die Ports bis `1024` bei vielen Betriebssystemen privilegiert und man benötigt Administratorenrechte um diese Ports zu verwenden. Da dies nicht in jeder Programmierumgebung problemlos möglich ist und der Port `80` auch schon von einem tatsächlich operativen HTTP Server «besetzt» sein könnte, wird in den folgenden Codebeispielen Port `8000` benutzt. Natürlich kann auch jeder andere freie Port stattdessen verwendet werden. Traditionell

versucht man eine Ähnlichkeit zum Default-Port `80` herzustellen, also bspw. `8080`, `8888`, `8000`. Funktionell hat das aber keinen Einfluss.

7.4.1 Grundgerüst TCP-Server

Das im Eclipse-Projekt `VAR-WebServer` bereitgestellte Grundgerüst für den Webserver ist in Listing 7.1 gezeigt. Es entspricht strukturell dem `EchoServer Threaded` (Listing 3.4 in Abschnitt 3.2.3).
Im Gegensatz zum Code in TCP-Kapitel ist hier der Port fest vorgegeben (Zeile 3 in Listing 7.1. Wenn statt des Ports `8000` der HTTP Standard Port `80` benutzt werden soll, muss das Programm auf den gängigen Betriebssystemen mit Administratorrechten gestartet werden. Soll der Webserver während der Entwicklung aus Eclipse heraus gestartet werden, müsste demzufolge Eclipse mit Administratorrechten gestartet werden. Unter Windows geht das durch Rechtsklick auf das Eclipse Icon: Im Kontextmenü kann «Als Administrator ausführen» ausgewählt werden. Unter Linux/UNIX/macOS kann man das Programm `eclipse` auf der Kommandozeile durch durch «`su eclipse`» mit Super-User-Rechten starten.

7.4.2 HTTP Request prüfen

Das HTTP-Protokoll besteht im einfachsten Fall aus einer Nachricht vom Client an den Server («HTTP Request») und einer Nachricht vom Server an den Client («HTTP Response»). Nachdem dieses Nachrichtenpaar ausgetauscht wurde, kann die Verbindung vom Server geschlossen werden.
Der HTTP Request besteht aus der ersten Zeile, in der der eigentliche Request codiert ist und einer variablen Anzahl daruf folgender (nicht-leerer) Header-Zeilen. Dann folgt eine Leerzeile. Im Fall von `GET` Requests, auf die wir uns hier der Einfachheit halber beschränken wollen, ist der HTTP Request abgeschlossen.

Aktivitätsschritt 7.2:
Ziel: HTTP Request einlesen
Im ersten Schritt wird der HTTP Request bis zur ersten Leerzeile zeilenweise gelesen. Dazu wird der folgende Ausdruck in einer `while`-Schleife als Bedingung benutzt, die gleichzeitig die gerade gelesene Zeile der `String`-Variable `input` zuweist:

Listing 7.1: `var.web.http.WebServer`: Grundgerüst eines TCP-*multithreaded* Server für Port 8000

```
1  public class WebServer {
2      public void start() {
3          try (ServerSocket serverSocket = new ServerSocket(8000)) {
4              System.out.println("WebServer gestartet ...");
5              while (true) {
6                  Socket socket = serverSocket.accept();
7                  new WebThread(socket).start();
8              }
9          } catch (IOException e) {
10             System.err.println(e);
11         }
12     }
13
14     private class WebThread extends Thread {
15         private Socket socket;
16         public WebThread(Socket socket) {
17             this.socket = socket;
18         }
19         @Override
20         public void run() {
21             SocketAddress socketAddress = socket.getRemoteSocketAddress
                   ();
22             System.err.println("Verbindung zu " + socketAddress + "
                   aufgebaut");
23             try (BufferedReader in = new BufferedReader(new
                   InputStreamReader(socket.getInputStream()));
24                     PrintWriter out = new PrintWriter(socket.
                           getOutputStream(), true)) {
25                 String input;
26                 while (null != (input = in.readLine())) {
27                     System.out.println(input);
28                 }
29             } catch (Exception e) {
30                 System.err.println(e);
31             } finally {
32                 try {
33                     socket.close();
34                 } catch (IOException e) {
35                     e.printStackTrace();
36                 }
37                 System.err.println("Verbindung zu " + socketAddress + "
                        abgebaut");
38             }
39         }
40     }
41
42     public static void main(String[] args) {
43         new WebServer().start();
44     }
45 }
```

```
!"".equals(input = in.readLine())
```

Ändern Sie die Schleife zum Einlesen der Nachricht vom Client an den Server in der Methode `run` entsprechend ab.

Aktivitätsschritt 7.3:

Ziel: Test und Analyse von HTTP Requests

Testen Sie die veränderte `run`-Methode, indem Sie den Server starten und mit Ihrem Webbrowser HTTP Requests schicken. Sie müssen also einen entsprechenden URL konstruieren (z. B. `http://localhost:8000/abc/def`). Analysieren Sie die Header-Zeilen, die Ihr Browser generiert und in seinem HTTP Request mitsendet.

Aktivitätsschritt 7.4:

Ziel: Unterscheiden zwischen HTTP Request Arten

Fügen Sie Ihrem Projekt einen Aufzählungstyp namens `Type` mit den Werten `FILE` und `UNKNOWN` zur Repräsentation der unterstützen HTTP Request Arten hinzu.

```
enum Type {
    FILE, UNKNOWN;
}
```

Im Verlauf dieses Praktikums wird dieser Aufzählungstyp noch um weitere Ausprägungen erweitert.

Zur tatsächlichen Unterscheidung der HTTP-Request-Arten wird jeweils die erste Zeile der Nachrichten vom Client an den Server genauer untersucht. Der Code dafür ist in Listing 7.2 gezeigt. Er muss in die Methode `run` eingebaut werden.

Die Funktionsweise ist, dass zuerst die erste Zeile gelesen wird (bevor später wie gehabt die Header-Zeilen bis zur ersten Leerzeile eingelesen werden). Diese Zeile wird mit einem *regulären Ausdruck* auf syntaktische Korrektheit in Bezug auf das HTTP-Protokoll geprüft. Reguläre Ausdrücke können in Java u. a. über die Methode `matches` von `String` verwendet werden. Dann wird der *Path*-Bestandteil

der Zeile mit `substring` «ausgeschnitten» und in der Variable|variablefileName gespeichert.

Aus Sicherheitserwägungen heraus wäre es keine gute Idee alle Dateien des Servers über einen Webserver verfügbar zu machen. Stattdessen definiert man ein Verzeichnis im Filesystem des Servers, das die Wurzel der auszuliefernden Dateien darstellt. Anfragen mit einem Pfad werden dann immer relativ zu diesem Verzeichnis interpretiert.

Aktivitätsschritt 7.5:
Ziel: Prüfen der angeforderten Datei
Der *Path*, der im HTTP Request übermittelt wird, muss nun einer Datei im Filesystem des Servers zugeordnet werden. Die Wurzel der auszuliefernden Dateien wird hier auf das Verzeichnis `docs/` im Eclipse-Projekt gesetzt. Die `main`-Methode der Klasse `WebServer` wird in einem Verzeichnis ausgeführt, das in der *Run Configuration* festgelegt werden kann. In der Standardkonfiguration ist dies das Projektverzeichnis. Listing 7.3 zeigt, wie aus `fileName` der tatsächliche Dateiname (relativ zum Wurzelverzeichnis `docs/`) zusammengesetzt werden kann.
Mit der Methode `canRead` wird geprüft, ob die Datei lesbar ist. Sollte die Datei nicht lesbar sein, wird der Typ auf `UNKNOWN` gesetzt. Gründe dafür könnten sein, dass die Datei nicht unter diesem Namen existiert oder die Zugriffsberechtigung zum Lesen fehlt.

Listing 7.2: `var.web.http.WebServer`: erste Zeile lesen und HTTP-Kommando parsen

```
1  Type type = Type.UNKNOWN;
2  String fileName = "";
3  input = in.readLine();
4  if (input.matches("^GET (.+) HTTP/\\d\\.\\d$")) {
5      type = Type.FILE;
6      // "GET " abschneiden:
7      fileName = input.substring(4);
8      // " HTTPx.x" abschneiden:
9      fileName = fileName.substring(0, fileName.length() - 9);
10 } else {
11     type = Type.UNKNOWN;
12 }
```

Listing 7.3: `var.web.http.WebServer`: auszuliefernde Datei prüfen

```
1 File file = null;
2 if (type == Type.FILE) {
3     file = new File(System.getProperty("user.dir") + File.separator + "
          docs" + fileName);
4     if (!file.canRead()) {
5         type = Type.UNKNOWN;
6     }
7 }
```

7.4.3 HTTP Response generieren

Als Antwort wird die Datei ausgeliefert. Dazu muss lt. HTTP der Rückgabe-Code `200 OK` generiert werden, der Rückgabetyp (MIME Type) gesetzt werden und nach einer Leerzeile die Datei gesendet werden.

Aktivitätsschritt 7.6:

Ziel: Ausliefern der angeforderten Datei

In Listing 7.4, das immer noch Code für die `run`-Methode zeigt, wird zuerst anhand des Typs unterschieden, wie geantwortet wird. Im Fehlerfall (HTTP-Kommando fehlerhaft oder Datei nicht lesbar) wird ein `404 Not Found` Resultat erzeugt und über `out` an den Client zurückgesendet.

Im Erfolgsfall (`FILE`) muss noch unterschieden werden, welchen Typ die zurückzusendende Datei hat. Das wird hier einfach über den Namen des Datei-Suffix («`html`») entschieden.

Der Code zum Ausliefern einer Datei (Listing 7.5) darf nicht nur für druckbare Zeichen funktionieren, sondern muss Byte-weise lesen und schreiben. Deshalb werden hier Streams und keine Reader/Writer benutzt.

7.4.4 Anfragen, die keine Datei betreffen, behandeln

Der Server kann bereits Anfragen nach Dateien korrekt beantworten. In den folgenden Schritten soll der `WebServer` erweitert werden, so dass auch noch weitere Anfragen erkannt und behandelt werden.

Listing 7.4: `var.web.http.WebServer`: HTTP Response generieren

```
1  switch (type) {
2  case FILE:
3      out.printf("HTTP/1.1 200 OK\r\n");
4      if ("html".equalsIgnoreCase(fileName.substring(fileName.length() -
           4))) {
5          out.printf("Content-Type: text/html; charset=utf-8\r\n");
6      } else {
7          out.printf("Content-Type: application/octet-stream\r\n");
8      }
9      out.printf("\r\n");
10     // Datei ausliefern
11     break;
12 case UNKNOWN:
13     out.printf("HTTP/1.1 404 Not Found\r\n");
14     out.printf("Content-Type: text/plain; charset=utf-8\r\n");
15     out.printf("\r\n");
16     out.printf("nicht gefunden\r\n");
17     break;
18 }
```

Listing 7.5: `var.web.http.WebServer`: Datei ausliefern

```
1  FileInputStream fileIn = new FileInputStream(file);
2  OutputStream outBin = socket.getOutputStream();
3  int c;
4  while ((c = fileIn.read()) != -1) {
5      outBin.write(c);
6  }
7  outBin.close();
```

Listing 7.6: `var.web.http.WebServer`: Beispielhaft zwei weitere Anfragetypen erkennen

```
1 } else if (input.matches("^GET /cgi/choice\\?selection=Latein HTTP/\\d
      \\.\\d$")) {
2     type = Type.REQ1;
3 } else if (input.matches("^GET /cgi/choice\\?selection=HS-Mannheim HTTP
      /\\d\\.\\d$")) {
4     type = Type.REQ2;
5 }
```

Aktivitätsschritt 7.7:

Ziel: Weitere Anfragetypen erkennen

Zuerst muss der Aufzählungstyp `Type` um weitere Anfragetypen erweitert werden. Exemplarisch werden hier zwei weitere Typen hinzugefügt: `REQ1` und `REQ2`:

```
enum Type {
    REQ1, REQ2, FILE, UNKNOWN;
}
```

Zu diesen zwei neuen Anfragetypen wird jeweils ein Anfragemuster definiert und mit einem regulären Ausdruck erkannt. Fügen Sie der bisherigen Unterscheidung die zwei entsprechenden Äste wie in Listing 7.6 hinzu. Sie sind für die Erkennung der URLs `http://localhost:8000/cgi/choice?selection=Latein` und `http://localhost:8000/cgi/choice?selection=HS-Mannheim` zuständig. In diesen Fällen soll keine Datei ausgeliefert werden, sondern eine Webserver eigene Funktion ausgeführt werden, die im nächsten Aktivitätsschritt hinzugefügt wird.

Aktivitätsschritt 7.8:

Ziel: Weitere Anfragetypen beantworten

Listing 7.7 zeigt exemplarisch als Erweiterung des Codes für die `switch`-Fallunterscheidung aus Aktivitätsschritt 7.6, wie die beiden neuen Anfragetypen beantwortet werden könnten.

Listing 7.7: `var.web.http.WebServer`: Beispielhaft zwei weitere Anfragetypen behandlen

```
1  case REQ1:
2      out.printf("HTTP/1.1 200 OK\r\n");
3      out.printf("Content-Type: text/html; charset=utf-8\r\n");
4      out.printf("\r\n");
5      out.printf("<html><body>Non scholae sed vitae discimus.</body></
           html>\r\n");
6      break;
7  case REQ2:
8      out.printf("HTTP/1.1 200 OK\r\n");
9      out.printf("Content-Type: text/html; charset=utf-8\r\n");
10     out.printf("\r\n");
11     out.printf("<html><body>Die Hochschule Mannheim ist bekannt fuer
           eine praxisnahe und theoretisch fundierte Ausbildung</body></
           html>\r\n");
12     break;
```

Aktivitätsschritt 7.9:

Ziel: Weitere Anfragetypen testen

Im Eclipse-Projekt ist bereits der Ordner `docs/`, der als Wurzel für die auszuliefernden Dokumente fungieren soll, die Datei `form.html` vorbereitet. Darin ist u. a. ein HTML-Formular, das genau die beiden neu hinzugefügten Anfragetypen als HTTP Requests erzeugen kann. Analysieren Sie diese Datei und verwenden Sie sie zum Test der beiden neuen Anfragetypen: `http://localhost:8000/form.html`.

7.5 Ausblick und Anregungen für eigene Projekte

Ein eigener Webserver ist eine hervorragende Plattform für eigene Experimente.

HTTP Version: Werten Sie bspw. die HTTP Version des Requests aus und liefern Sie HTTP Response Code «`505 HTTP Version not supported`» zurück, wenn der Client HTTP 1/1 anfordert.

Persistent HTTP: Der Server könnte mit vertretbarem Aufwand erweitert werden, so dass *persistent HTTP* unterstützt wird. Dazu müssten mehrere Request-Response-Paare innerhalb der `run`-Methode verarbeitet werden. Beachten Sie, dass der Client im

Request Header «`Connection: keep-alive`» (ab HTTP 1.0) ankündigen muss, dass er zu *persistent HTTP* in der Lage ist.

Conditional `GET`: Überprüfen Sie, ob der Client eine Datei (die ausgeliefert werden kann) nach dem ersten Zugriffsversuch anders, nämlich mit «`If-modified-since:` ...»-Header anfordert. Sie können in Java den Zeitpunkt der letzten Änderung einer Datei mit `lastModified()` an dem dazu passenden `File`-Objekt abfragen.

Cookies: Erweitern Sie Ihren Server so, dass in der HTTP Response eine Header-Zeile mit «`set-cookie:` ...» zurückgegeben wird. Der Wert des Cookies sollte bei zukünftigen Requests mitgeschickt werden. Die Reihenfolge der Header ist unbedeutend. Sie dürfen sich also nicht auf die Reihenfolge des «`cookie:` ...»verlassen, brauchen aber den «`set-cookie:` ...» auch nicht an eine bestimmte Stelle in den Header-Zeilen setzen.

Content Management System: Die aktuelle Form von `WebServer` kann nur in erster Linie Dateien ausliefern. Der Inhalt von `docs/` könnte zusätzlich durch die HTTP-Methoden `PUT` und `DELETE` manipuliert werden: Mit `PUT` könnten Dateien in `docs/` hochgeladen werden, mit `DELETE` aus `docs/` gelöscht werden. Als Client würde der HTML Editor aus Mozilla funktionieren, der auch Funktionen zum Upload bzw. zur Verwaltung von Inhalten auf solch einem Content Management System hat.

7.5.1 Mediathek mit Jugendschutz

Die Idee für diese Aufgabe ist, eine einfache Kontrollinstanz einzurichten, die Bedingungen prüft und je nach Ausgang der Prüfung entweder eine Mediendatei ausliefert oder eine Fehlermeldung produziert. Als Bedingung kommen vielfältige Tatbestände in Betracht. Bspw. könnte eine Zugriffsautorisierung nach vorheriger Benutzerauthentifizierung erfolgen. Um es nicht kompliziert zu machen, wird hier vorgeschlagen einfach eine Zeitprüfung vorzunehmen. Erlaubt wird die Auslieferung von Medien aus einem bestimmten Medienverzeichnis nur zwischen 21:00 und 06:00 Uhr. Dies könnte z. B. durch eine Bedingung wie in Listing 7.8 geprüft werden.
Als Fehlermeldung könnte der HTTP Response Code `403` («`Forbidden`») zurückgegeben werden.

Listing 7.8: Prüfung der Bedingungen für die Medienauslieferung

```
Calendar now = Calendar.getInstance();
if (now.get(Calendar.HOUR_OF_DAY) > 6
    && now.get(Calendar.HOUR_OF_DAY) < 21) {
    /* abweisen */
} else {
    /* ausliefern */
}
```

Listing 7.9: `var.web.http.WebServer`: Erkenner für die Anfrage nach zugriffsbeschränkten Inhalten

```
if (input.matches("^GET (/schutz/.*) HTTP/\\d\\.\\d$")) {
    type = Type.RESTRICTED;
}
```

Aktivitätsschritt 7.10 (fakultativ):

Ziel: `WebServer` um zeitgesteuerte Auslieferung erweitern

Erweitern Sie `WebServer`, damit Medien aus dem Verzeichnis `docs/schutz/` nur zwischen 21:00 und 06:00 Uhr ausgeliefert werden.

Sie können den Aufzählungstypen `Type` um eine Alternative `RESTRICTED` erweitern und einen regulären Ausdruck als Erkenner dafür bei der Verarbeitung des HTTP Requests wie in Listing 7.9 hinzufügen.

Aktivitätsschritt 7.11 (fakultativ):

Ziel: Entwickeln Sie eine Mediathek

Entwickeln Sie eine Mediathek zur Auslieferung von Videodateien. Sie sollten berücksichtigen, dass manche Videos aus Jugendschutzgründen nur eingeschränkt zugreifbar sein sollen. Es gibt aber auch allgemein freigegebene Dateien, bei denen ein Download jederzeit erlaubt ist. Als alternativen Zugang könnten Sie die Möglichkeit vorsehen, für autorisierte Benutzer, deren erwachsenes Alter bereits im Vorfeld verifiziert wurde, den Download eingestufter Inhalte jederzeit zu gestatten.

Servlet Container

Applikationsserver fungieren als Laufzeitumgebung für Programme. Anstatt ein Programm als Standalone-Applikation mit der JVM-Umgebung laufen zu lassen, wird in Applikationsservern eine spezielle Laufzeitumgebung mit passenden Libraries, Logging, möglicherweise auch Monitoring-Werkzeugen und Mechanismen zur Lastverteilung zwischen mehreren Servern bereitgestellt. Dafür muss das Java-Programm in Form einer Klasse eine bestimmte Schnittstelle anbieten, damit der Applikationsserver mit ihr interagieren und sie beispielsweise starten kann. Applikationsserver gibt es nicht nur für Java, sondern für ganz unterschiedliche Zielsprachen und auch Anwendungszwecke. Üblicherweise werden Applikationsserver für den Bereich *Enterprise Computing* benutzt. Es gibt aber auch vergleichbare Umgebungen in technischen und *embedded* Anwendungen (z. B. für das *Smart Home*).

Servlet Container sind spezialisierte Webserver. Sie sind oft der Teil von Applikationsservern, die zur Auslieferung dynamischer Inhalte konzipiert sind. Die Inhalte werden von Java-Klassen erzeugt. Servlet Container sind im Vergleich zu Standard-Webservern, die Programme zum Bedienen von HTTP Requests starten, besonders effizient, da nicht für jeden Aufruf ein neuer Prozess mit dem aufgerufenen Programm

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

Exkurs: Servlet Container für statische Inhalte, JSP und JSF

Servlet Container können auch problemlos statische Inhalte – also normale Dateien – ausliefern. Da viele dynamische Inhalte letztlich Webseiten mit HTML-Inhalt sind, und solche Inhalte etwas mühsam mit Java über `String`-Operationen erzeugt werden müssen, gibt es bei Servlet Containern eine besondere Methode Java-Ausdrücke in HTML-Seiten einzubetten. Solche mit Java angereicherten HTML-Seiten heißen Java Server Pages (JSP). Servlet Container übersetzen JSPs beim ersten Aufruf zuerst in Java Source Code, der dann kompiliert und anschließend wie ganz normale Servlets behandelt wird. Eine Motivation für die Einführung von JSPs war, die Arbeitsteilung zwischen Programmierern und Grafik-Designern zu verbessern. Darauf aufbauend gibt es noch Java Server Faces (JSF), die komplexere und individuell entwickelte Java-Funktionen in Libraries bereitstellen, sodass sie von Nichtprogrammierern einfacher in JSPs eingesetzt werden können.

gestartet werden muss, was Zeit kostet. Stattdessen ist ein Servlet Container ein Webserver, der in einer JVM läuft. Wird ein URL aufgerufen, wird die entsprechende Servlet-Klasse geladen, ein neues Objekt der geladenen Klasse erzeugt, und bestimmte Methoden an dem Servlet-Objekt aufgerufen. Das geht vergleichsweise schnell. Die Servlet-Objekte laufen zudem nebenläufig als Threads ab.

8.1 HttpServlet

Konkret sind Servlets Klassen, die von der Klasse `javax.servlet.http.HttpServlet` abgeleitet sind. Servlets sind integraler Bestandteil des Java EE (Enterprise Edition) Profils. Der Java Standard JSR 369 beschreibt die aktuelle Servlet-Spezifikation 4.0. Wie beim Start der statischen Methode `main` bei normalen Java-Applikationen gibt es einen festen Einstiegspunkt in `HttpServlet`-Objekte. Das ist die Methode `service` (s. Listing 8.1).

Die Aufrufparameter in Form von *Query*-Parametern und Header-Einträgen werden der `service`-Methode durch Aufrufparameter als Objekt der Klasse `HttpServletRequest` übergeben. Die Methode bekommt weiterhin ein `HttpServletResponse`-Objekt als Parameter, über das das Servlet sein Ergebnis ausgibt.

`HttpServlet` implementiert `service`. Überschreibt das eigene Servlet diese Methode nicht, werden je nach HTTP-Methode des Aufrufs vom Client, die im

`HttpServletRequest`-Objekt vorhanden ist, eine der folgenden Methoden aufgerufen, die dann in der eigenen Klasse implementiert werden müssen:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, ServletException
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
     throws IOException, ServletException
protected void doHead(HttpServletRequest req, HttpServletResponse resp)
     throws IOException, ServletException
protected void doDelete(HttpServletRequest req, HttpServletResponse
    resp) throws IOException, ServletException
protected void doPut(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, ServletException
```

Das übliche Vorgehen ist, die passenden `doXXX`-Methoden zu implementieren. Oft bedeutet dies, dass `doGet` und `doPost` dieselbe Funktionalität haben (s. z. B. Listing 8.2: `doPost` verwendet die Implementierung von `doGet`).

Listing 8.1: Analog zur `main()`-Methode ist bei Servlets `service(...)` der Einstiegspunkt

```
1 public final class HelloWorld extends javax.servlet.http.HttpServlet {
2     public void service(HttpServletRequest request, HttpServletResponse
          response) throws IOException, ServletException {
3         response.setContentType("text/html");
4         PrintWriter writer = response.getWriter();
5         writer.println("<html><body>Hello World</body></hello>");
6     }
7 }
```

Listing 8.2: Idiomatischer Gebrauch von `doPost` und `doGet`

```
1  public final class MyServlet extends javax.servlet.http.HttpServlet {
2      public void doGet(HttpServletRequest request, HttpServletResponse
           response) throws IOException, ServletException {
3          response.setContentType("text/html");
4          PrintWriter writer = response.getWriter();
5          writer.println("<html><body>Hello World</body></hello>");
6      }
7      public void doPost(HttpServletRequest request, HttpServletResponse
           response) throws IOException, ServletException {
8          doGet(request, response);
9      }
10 }
```

Exkurs: Lebenszyklus von `HttpServlet`-Instanzen

Konkrete Servlets werden als Subklasse von `HttpServlet` bereitgestellt. Bei einem passenden HTTP Request durchläuft das Servlet dann den folgenden Lebenszyklus (sollte es bereits eine initialisierte Instanz der Servlet-Klasse geben, wird bei 4 begonnen):

1. Laden der Klasse (über den Parameter `load-on-startup` im *Deployment Descriptor* kann beeinflusst werden, wann die Servlet-Klasse geladen wird – z. B. beim Start des Containers)
2. Der Konstruktoraufruf erfolgt direkt nach dem Laden. Den Konstruktor (Default) überschreibt man i. Allg. nicht, da an dieser Stelle noch nicht auf die Konfiguration (Instanz von `ServletConfig`) zugegriffen werden kann.
3. `void init(ServletConfig config) throws ServletException` (vor dem ersten Aufruf von `doService`)
4. `void doService(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`
5. Destruktor: `void destroy()`

Neben `doService` (bzw. `doGet`, `doPut`, ...) können auch `init` und `destroy` überschrieben werden, um eine spezielle Umgebung für das Servlet aufzubauen oder aufzuräumen.
Es gibt immer maximal eine Instanz jedes Servlets.

8.2 Deployment Descriptor

Der *Deployment Descriptor* ist eine Konfigurationsdatei (`web.xml`), in der die gesamte Web-Anwendung beschrieben wird. Insbesondere werden alle Servlet-Klassen mit Name und Java-Klasse aufgeführt und spezifiziert, für welche URLs das Servlet zur Beantwortung des Requests verwendet werden soll.
Ein Servlet Container kann mehrere Web-Anwendungen mit jeweils einem getrennten *Deployment Descriptor* gleichzeitig verwalten und für Clients bereitstellen. Die Web-Anwendungen laufen dann isoliert voneinander ab und haben getrennte Namensräume und können auch unterschiedliche Libraries verwenden.

8.3 HttpServlet-Zustand und Thread-Sicherheit

Sollten mehrere Clients nebenläufig auf dasselbe Servlet zugreifen, könnte sich die Abarbeitung der parallelen Anfragen in den Servlet-Methoden überlappen. Wenn in den überlappenden Anfragen gleichzeitig auf den Zustand des Servlets (also die Instanzvariablen) zugegriffen wird, kann es ungewollte «Race Conditions» geben. Der Zugriff auf Instanzvariablen ist nicht «Thread-sicher» und muss unbedingt unterbleiben oder synchronisiert erfolgen.
Eine Möglichkeit Thread-Sicherheit zu erreichen ist die Synchronisierung von kritischen Abschnitten, in denen auf Instanzvariablen zugegriffen wird, durch das `synchronized` Keyword, durch die Verwendung von `Lock`-Objekten, die Verwendung der *atomic*-Klassen und eingeschränkt auch die Thread-sicheren Collection-Implementierungen wie die *synchronized Collection Wrapper*.

Listing 8.3: Thread-sicherer Zugriff auf eine Instanzvariable über einen *intrinsic lock*

```
1  // Zugriffszähler
2  out.println("<h1>Zugriffsz&auml;hler</h1><ul>");
3  synchronized (this.counter) {
4      if (this.counter != null) {
5          this.counter++;
6      } else {
7          this.counter = 1;
8      }
9      out.println("<p>dies war Aufruf Nr. " + this.counter);
10 }
```

Listing 8.4: Thread-sicherer Zugriff auf eine `Map` über einen *synchronized Collection Wrapper*

```
1  // Instanzvariable mapSafe
2  Map<KeyType, ValType> mapSafe = Collections.synchronizedMap(new HashMap
       <KeyType, ValType>());
3
4  // ... mapSafe nebenläufig befüllen ...
5
6  // Thread-sicher über mapSafe iterieren:
7  Set<KeyType> s = this.mapSafe.keySet();
8  synchronized(this.mapSafe) {
9      while (KeyType k : s) {
10         // ... k und this.mapSafe.get(k) benutzen ...
11     }
12 }
```

Werden Variablen in der *Session* gespeichert und per `getAttribute()` und `setAttribute()` benutzt, ist zwar der jeweilige lesende und ändernde Zugriff für sich Thread-sicher. Man kann sich also sicher sein, dass das Attribut immer in einem konsistenten Zustand ist. Zumeist besteht die Verwendung einer solchen Variablen aber aus einer Transaktion, also einem Ablauf, der einen kritischen Abschnitt darstellt, während dem keine anderen Threads in den Abschnitt eintreten dürfen. In dem Fall muss die «Transaktion» durch `synchronized` oder einen `Lock` geschützt werden.

Wenn Sie Container-Datenstrukturen in Instanzvariablen speichern, sollten Sie die Thread-sicheren Collections aus `java.util.concurrent` benutzen, die Sie wie in Zeile 2 von Listing 8.4 über die *synchronized Collection Wrapper* von gewöhnlichen Collection-Objekten beziehen können. Der einzelne Lese- und Schreibzugriff ist dann zwar Thread-sicher, das Iterieren über solch eine Collection stellt aber wieder eine Transaktion dar, die wie in Zeile 8 in Listing 8.4 manuell über einen *intrinsic lock* Thread-sicher gemacht werden muss.

Exkurs: Tomcat als Servlet Container

Tomcat ist ein Servlet Container, der kostenfrei nutzbar ist und dessen Quelltext frei verfügbar ist. Die Arbeit an Tomcat wurde von Sun gestartet, der Firma, in der Java ursprünglich entwickelt wurde. Tomcat war als Referenzimplementierung für die Servlet- und JSP-Spezifikationen gedacht. Tomcat besteht aus mehreren Teilen, u. a. Coyote (einem HTTP Server für statische Inhalte), Catalina (dem eigentlichen Servlet Container) und Jasper (der JSP Engine). Außerdem unterstützt Tomcat die Java Management Extensions (JMX), eine Schnittstelle und ein Framework zur Administration von Server-Anwendungen. Beispielsweise können über JMX auf einheitliche Weise die Betriebsressourcen überwacht werden. Tomcat ist auch Teil umfassenderer Applikationsserver wie Apache Geronimo, die noch zusätzliche Funktionalitäten zum Betrieb und zur Ausführung verteilter Architekturen zur Verfügung stellen.
Tomcat kann entweder *standalone* betrieben werden – in diesem Kapitel wird das so gemacht — oder wenn mit einer hohen Frequenz von Zugriffen gerechnet wird, im Verbund mit anderen Webservern. Wenn mehrere Server im Verbund («Cluster») zum Einsatz kommen, ist das Ziel meistens einen Lastausgleich herzustellen, indem die Rechenkapazität mehrerer Hardware Server zusammengenommen wird. Beim Lastausgleich sieht die Systemarchitektur meist so aus, dass ein Rechner als Eingangspunkt dient. Die IP-Adresse dieses oft Frontend Server genannten Hosts bleibt fest und muss den Clients bekannt sein. Die weiteren Rechner des Verbunds sind den Clients nicht bekannt und deren Anzahl und örtliche Verteilung kann während des Betriebs jederzeit an die aktuelle Last angepasst werden. Auf dem Frontend Server läuft ein HTTP Server, der die HTTP Requests möglichst schnell an weitere Rechner im Verbund weiterreichen soll. Damit die Rechenkapazität des Frontend Servers dafür möglichst gut ausgenutzt wird, werden dort keine Servlets ausgeführt, da die JVM für die Abarbeitung der jeweiligen `doXXX`-Methode Rechenkapazität benötigen würde. Der Frontend Server nimmt stattdessen lediglich die HTTP Requests entgegen und verteilt sie auf die im Hintergrund parallel arbeitenden Servlet Container, die auf den Rechnern des Verbundes verteilt sind. Dadurch kann ein Lastausgleich hergestellt werden, indem mehr Hardware Server für die Abarbeitung der `doXXX`-Methoden der Servlets herangezogen werden. Als Schnittstelle zwischen dem HTTP Server auf Frontend Server und Servlet Container kann das Apache JServ Protocol (AJP) verwendet werden, für das Tomcat einen Connector bereitstellt.

Praktikum

Aktivitätsschritte:

8.4 Aufgabe: Tomcat als Servlet Container verwenden

Sie müssen zuerst «Apache Tomcat» und «Eclipse für Java EE» installieren. Zum Redaktionsschluss dieses Texts ist die Version 9.0.8 von Tomcat aktuell. Achten Sie bei

zukünftigen Versionen von Tomcat darauf, dass auch die Unterstützung in Eclipse bereits integriert ist (s. Aktivitätsschritt 8.8). Laden Sie also die letzte Version von Tomcat, für die Support in Eclipse EE integriert ist.

Aktivitätsschritt 8.1:
Ziel: Installation von Tomcat als lokale Testumgebung
Laden Sie Tomcat Version 9 (*Core*), von der Seite `http://tomcat.apache.org/download-90.cgi` herunter. Entpacken Sie das `ZIP`- oder `tar.gz`-Archiv in ein Verzeichnis Ihrer Wahl. Sie brauchen Tomcat im ersten Schritt nur lokal. Sie brauchen deshalb keine Windows-Services einrichten oder etwas in `/etc/init.d` unter Linux zu aktivieren. Das Verzeichnis, in dem Sie Tomcat entpacken, kann ein ganz normales Benutzerverzeichnis sein (muss z. B. nicht unter `C:\Programme\` liegen). Wie schon bei Apache Active MQ sollten Sie Leerzeichen, Umlaute und andere Sonderzeichen im Pfad zur Tomcat-Installation vermeiden.
Die entpackte Tomcat 9 Installation befindet sich nun im Ordner `apache-tomcat-9.X.Y` (`X` und `Y` sind hier die genauere Unterversionsnummern, der Tomcat-Fassung, die Sie heruntergeladen haben), wo das Archiv entpackt wurde. Im Verzeichnis `apache-tomcat-9.X.Y/conf` gibt es die Datei `tomcat-users.xml`. Dort können Rollen und Benutzer zur Autorisierung und Passwörter von Benutzern zur Authentifizierung eingetragen werden. Im frischen Auslieferungszustand von Tomcat sind keine Benutzer und Rollen eingetragen, bzw. alle Benutzer und Rollen sind im XML-File mit XML-Kommentaren ungültig gemacht: `<!– ... –>`.
Ändern Sie die Datei `apache-tomcat-9.X.Y/conf/tomcat-users.xml` so wie in Listing 8.5, um die neuen Rollen `manager-gui`, `manager-script`, `manager-jmx`, `manager-status`, `admin-gui` und `admin-script` zur Autorisierung von Zugriffen über das Web Interface der Administrationsoberfläche anzulegen. Der **Benutzer `student`** wird dafür mit dem **Passwort `Turing`** angelegt und diesen Rollen zugeordnet.

Aktivitätsschritt 8.2:
Ziel: Start von Tomcat und Öffnen des Web Application Managers

Apache Tomcat kann nun gestartet werden. Dazu müssen Sie in einer Shell (bzw. `cmd`-Fenster) das Shell Script `startup.bat` bzw. `startup.sh` aus dem Verzeichnis `apache-tomcat-9.X.Y/bin/` starten (wahrscheinlich müssen Sie dazu vorher die Scripte unter Linux, BSD oder macOS erst noch mit `chmod a+x apache-tomcat-9.X.Y/bin/*.sh` ausführbar machen). Möchten Sie einen so gestarteten Tomcat Server wieder stoppen, können Sie das entsprechende Script `shutdown.bat`/`shutdown.sh` starten.

Tipp: Sollten Sie beim Start Fehlermeldungen bekommen, versuchen Sie das Skript mit Administrator- oder Super-User-Rechten auszuführen (Windows: `cmd`-Fenster, in dem das `startup.bat` aufgerufen wird, als Administrator starten, Linux/BSD/macOS: `sudo startup.sh`)

Web Application Manager: Wenn die Installation geklappt hat und der Tomcat Server gestartet ist, können Sie auf die Admin-Oberfläche («Tomcat Web Application Manager») Ihrer Tomcat-Installation über den Link `http://localhost:8080/manager/html` zugreifen. Sie müssen dafür als **Login-Namen `student`** und als **Passwort `Turing`** eingeben.

In der Admin-Oberfläche (s. Abb. 8.1) werden die aktuell auf diesem Servlet Container gehosteten Web-Anwendungen angezeigt. In der Tabelle «Applications» hat dort jede Web-Anwendung eine eigene Zeile.

Listing 8.5: `conf/tomcat-users.xml` für **Benutzer `student`** mit dem **Passwort `Turing`**

```
1  <tomcat-users xmlns="http://tomcat.apache.org/xml"
2                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3                xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-
                     users.xsd"
4                version="1.0">
5   <role rolename="admin-gui"/>
6   <role rolename="admin-script"/>
7   <role rolename="manager-gui"/>
8   <role rolename="manager-script"/>
9   <role rolename="manager-jmx"/>
10  <role rolename="manager-status"/>
11  <user username="student" password="Turing" roles="manager-gui,manager-
        script,manager-jmx,manager-status,admin-gui,admin-script"/>
12 </tomcat-users>
```

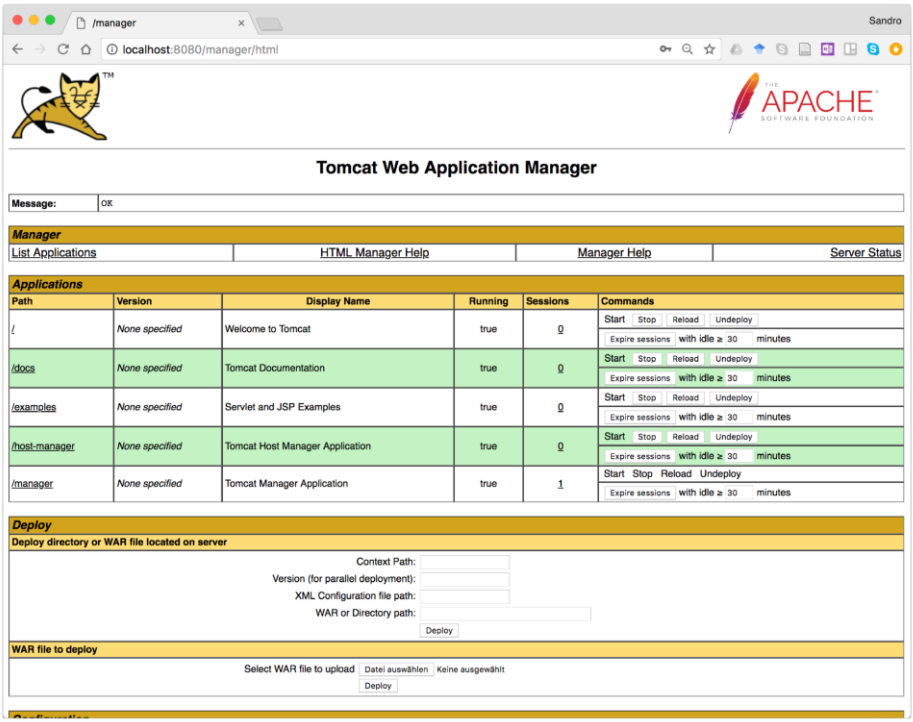


Abbildung 8.1: Screenshot der Tomcat Web-Application-Manager-Oberfläche (Tomcat 9)

Web-Anwendungen sind Zusammenstellungen von zusammengehörenden und untereinander abhängigen Servlets sowie statischen Web-Inhalten. Sie werden als JAR- bzw. ZIP-Archiv mit besonderer Struktur in einer Archivdatei mit dem Suffix `*.war` (Web Application Archive) paketiert. Wie JAR-Dateien enthalten WAR-Dateien das Verzeichnis `META-INF` mit der Datei `MANIFEST.MF` mit Informationen über den Inhalt des Archivs. Spezifisch und erforderlich in jedem WAR ist der *Deployment Descriptor* `web.xml` im Verzeichnis `WEB-INF`. Im *Deployment Descriptor* können neben anderen Informationen über die Web-Anwendung bspw. die Servlets der Web-Anwendung mit ihren URL Mappings beschrieben sein. Außerdem werden alle externen Libraries, von denen die Web-Anwendung abhängt, mit im WAR verpackt.

8.4.1 Deployment von Web-Anwendungen

Ein Servlet Container kann mehrere voneinander unabhängige Web-Anwendungen gleichzeitig betreiben. Der Beginn des *Path*-Teils der URLs wird vom Servlet Container verwendet, um zu unterscheiden, welche Web-Anwendung mit einem HTTP Request gemeint ist. Dieser URL-*Path*-Anfang wird auch Kontextpfad der Web-Anwendung genannt.
Die Inhalte im WAR sind zweckmäßigerweise so formuliert, dass sie keine absoluten Pfadnamen enthalten. WARs können deshalb auf beliebigen Servlet Containern sowie unter einem beliebigen Kontextpfad betrieben werden. Da im WAR auch alles verpackt ist, was zum Betrieb der Web-Anwendung erforderlich ist (z. B. Libraries), sind WAR-verpackte Web-Anwendungen sehr portabel.
Eine WAR-verpackte Web-Anwendung kann deshalb i. Allg. sehr leicht und über Web-Schnittstellen auf einem Servlet Container bereitgestellt werden. Bei Tomcat kann dies über den «Tomcat Web Application Manager» gemacht werden.

Aktivitätsschritt 8.3:
Ziel: Deployment über den Web Application Manager
In diesem Schritt werden Sie eine Web-Anwendung auf Ihrem Tomcat Server bereitstellen. Dieser Vorgang wird *Deployment* genannt. Laden Sie dazu zuerst das vorgefertigte WAR File `VAR-HTTP.war` von `http://verteiltearchitekturen.de/vol01/VAR-HTTP.war` herunter.

Web Application Manager: Wenn die Installation geklappt hat und der Tomcat Server gestartet ist, können Sie auf die Admin-Oberfläche («Tomcat Web Application Manager») Ihrer Tomcat-Installation über den Link `http://localhost:8080/manager/html` zugreifen. Sie müssen dafür als **Login-Namen** `student` und als **Passwort** `Turing` eingeben.
Im Abschnitt «Deploy» können Sie im Unterabschnitt «WAR file to deploy» die heruntergeladene Datei `VAR-HTTP.war` aus Ihrem lokalen Download-Ordner auswählen und über den «Deploy»-Button auf den Servlet Container Server laden. Dort wird das WAR entpackt und die Informationen aus dessen *Deployment Descriptor* werden benutzt, um die Web-Anwendung im Servlet Container zu konfigurieren. Wenn Sie über diesen Weg deployen, wird ein Standard für den Kontextpfad vergeben (`/VAR-HTTP/`, s. Abb. 8.2).

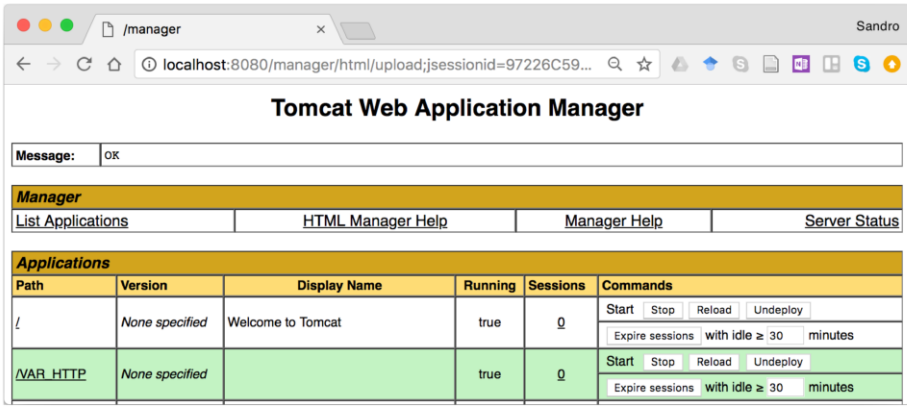


Abbildung 8.2: Ausschnitt der Tomcat-Web-Application-Manager-Oberfläche, nachdem `VAR_HTTP.war` deployt wurde, der Kontext-Pfad ist `/VAR_HTTP`

Aktivitätsschritt 8.4:

Ziel: Zugriff auf deployte Web-Anwendung

Rufen Sie die neu deployte Web-Anwendung (sie besteht nur aus einer statischen HTML-Datei (`index.html`) und einem Servlet (`var.web.servlet.hello.HelloWorld`) unter dem URL `http://localhost:8080/VAR-HTTP/index.html` auf. Im Webbrowser müsste eine Ausgabe erscheinen, die einen Link namens `hello` enthält. Das ist ein URL, der vom Servlet `var.web.servlet.hello.HelloWorld` bedient wird, das in dem WAR enthalten ist. Rufen Sie den Link mehrfach auf und inspizieren Sie die Ausgabe. Rufen Sie den Link auch mit *Query*-Parametern auf (z. B. `http://localhost:8080/VAR-HTTP/hello?ab=8711&cd=42`). Sie werden dieses Servlet in einem späteren Aktivitätsschritt noch selber programmieren.

Aktivitätsschritt 8.5:

Ziel: Management der deployten Web-Anwendung

Die neue Web-Anwendung taucht jetzt in der Liste der «Applications» unter diesem Kontextpfad `VAR-HTTP` auf (s. Abb. 8.2). Über die Buttons *Start*, *Stop*, *Reload*, *Undeploy* kann die Web-Anwendung gemanaged werden. Über «Expire Sessions ...» kann in die Konfiguration eingegriffen werden. Stoppen Sie die Web-

Anwendung und versuchen Sie erneut über den URL `http://localhost:8080/VAR-HTTP/hello` darauf zuzugreifen. Es müsste eine Fehlermeldung erscheinen. Gehen Sie zurück in die Management-Oberfläche auf `http://localhost:8080/manager/html` und «*undeployen*» Sie die Anwendung.

Servlet Container stellen eine standardisierte Ablaufumgebung für Web-Anwendungen bereit. Paketierte Web-Anwendungen können auf beliebigen Servern deployt werden. Sie sind unabhängig von allen anderen Web-Anwendungen auf demselben Servlet Container. Sie haben einen eigenen abgetrennten Namensraum, eigene URLs, die den (variablen und durch den Servlet Container vergebenen) Kontextpfad als Beginn des *Path*-Teils enthalten, und eigene Libraries — alles mit dem Ziel maximaler Portabilität: Es ist sehr einfach ein existierendes WAR auf einen Server zu deployen.

8.5 Aufgabe: «Dynamic Web Project» als Entwicklungsumgebung für Servlets in Eclipse verwenden

Während der Entwicklung einer Web-Anwendung, die aus Servlets und statischen Inhalten besteht, wäre es unpraktisch jedes Mal manuell ein WAR zu erzeugen, dieses über den «Tomcat Web Application Manager» eines beliebigen laufenden Tomcat Servers zu deployen und dann den korrekten URL im Browser einzugeben. Stattdessen gibt es für Eclipse ein Plugin, das diese Funktion (für unterschiedliche Servlet Container) automatisiert. Im Fall der Unterstützung von Tomcat wird das WAR auch nicht über das Web Interface des Servlet Containers deployt, sondern an die richtige Stelle des lokalen Filesystems kopiert und über einen lokalen Programmaufruf in die Catalina-Komponente von Tomcat eingelesen. Deshalb funktioniert das Plugin auch nur mit einer lokalen Tomcat-Installation. Nachdem die Entwicklung der Web-Anwendung abgeschlossen ist, kann manuell ein WAR erzeugt werden, das dann auf einem beliebigen entfernten Servlet Container deployt werden kann.

Unter der Webadresse `http://verteiltearchitekturen.de/vol01/VAR-HTTP-solution.zip` können Sie eine Musterlösung zu dieser und den beiden aufeinander aufbauenden folgenden Praktikumsaufgaben herunterladen. In dem ZIP befindet sich das Eclipse-Projekt `VAR-HTTP_solution`.

Aktivitätsschritt 8.6:
Ziel: Prüfen, ob Eclipse EE vorhanden ist und ggf. installieren
Prüfen Sie, ob es in Ihrer Eclipse-Variante möglich ist, ein neues Projekt vom Typ *New → Other... → Web/Dynamic Web Project* anzulegen. Falls das nicht möglich ist, müssen Sie erst noch «Eclipse IDE for Java EE Developers» installieren. Der Download ist über die Seite `https://www.eclipse.org/downloads/eclipse-packages/` möglich.

8.5.1 Serverkonfigurationen in Eclipse

Wenn ein neues Projekt vom Typ «Dynamic Web Project» angelegt wird, muss ein lokal installierter Server angegeben werden, der während der Entwicklung als Ablaufumgebung benutzt werden soll. Im *Project Explorer* von Eclipse wird dafür neben den Java- und Dynamic-Web-Projekten auch der Knoten *Servers* angezeigt (s. Abb. 8.3). Unter *Servers* werden alle installierten lokalen Ablaufumgebungen (hier: Servlet Container) mit ihrer individuellen Konfiguration angezeigt.

Aktivitätsschritt 8.7:
Ziel: Java EE Perspective in Eclipse aktivieren
Damit die passenden Menüs und Views für die Entwicklung von Servlets angezeigt werden, sollten Sie zur *Java EE Perspective* wechseln. Klicken Sie dafür im Perspektiven-Auswahl-Bereich oben rechts im Eclipse-Fenster auf «Java EE» (s. Abb. 8.4).

Nun sollte ein *View* «Servers» sichtbar sein (zu erreichen über *Window → Show View → Other... → Server/Servers*). Dort wird für jede Serverkonfiguration ein Knoten dargestellt, über den anzeigt wird, ob der Server gerade läuft und über dessen Kontextmenü der Server gemanaged werden kann (s. Abb. 8.3). Wird ein Server-Knoten im

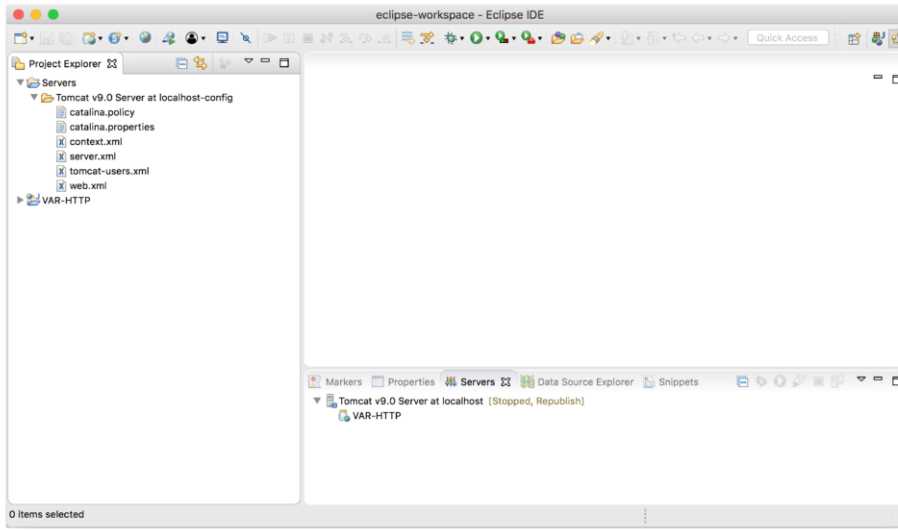


Abbildung 8.3: Screenshot von Eclipse mit konfiguriertem Tomcat 9.0 Server

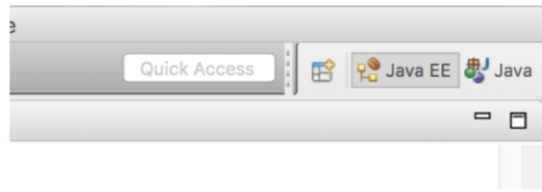


Abbildung 8.4: Screenshot vom Bereich zum Wechsel zwischen Perspektiven in Eclipse (oben rechts im Fenster)

«Server»-View aufgeklappt, werden alle «Dynamic Web Projects» angezeigt, die diesen Server als Ablaufumgebung verwenden. Die Server-Knoten bzw. deren Projektunterknoten zeigen an, ob die jeweilige Web-Anwendung, die gerade in Eclipse entwickelt wird, in der aktuellen Fassung auf dem lokalen Server deployt ist. Falls das Deployment aktuell ist, heißt der angezeigte Zustand «Synchronized», ggf. kann der Server auch anzeigen, dass er gerade einen «Restart» durchläuft.

8.5.2 Dynamic Web Project mit Server anlegen

Es gibt viele Möglichkeiten zu einer funktionierenden Entwicklungsumgebung für Web-Anwendungen in Eclipse zu kommen. Dabei können mehrere Server in unterschiedlichen Versionen oder von unterschiedlichen Herstellern oder auch derselbe Server in unterschiedlichen Konfigurationen parallel genutzt werden. Server, die bereits angelegt wurden, können für zukünftige «Dynamic Web Projects» wiederverwendet werden. In der folgenden Aufgabe legen Sie ein «Dynamic Web Project» und einen dazugehörenden Tomcat Server neu an.

Aktivitätsschritt 8.8:
Ziel: Serverkonfiguration in Eclipse EE anlegen
Starten Sie den Wizard für das Anlegen einer Serverkonfiguration über *File* → *New* → *Other...* → *Server*/*Server*).
Wählen Sie im ersten Schritt aus der Liste der von Eclipse EE unterstützten Server die Tomcat-Version aus, die Sie in Schritt 8.1 installiert hatten (im Beispiel von Abb. 8.5, links ist das Tomcat v9.0). Nach «Next» werden Sie nach dem Installationsort Ihres Tomcat Servers gefragt. Das *Tomcat installation directory*, das Sie über «Browse» angeben müssen, ist das Verzeichnis, das Sie in Aktivitätsschritt 8.1 entpackt hatten (in Abb. 8.5, rechts wäre dies `apache-tomcat-9.X.Y` – mit vollem Pfad). (s. Abb. 8.5)

Aktivitätsschritt 8.9:
Ziel: Dynamic Web Project in Eclipse EE anlegen
Legen Sie ein neues Projekt vom Typ «Dynamic Web Project» an (z. B. über *File* → *New* → *Dynamic Web Project*). Geben Sie dem Projekt einen neuen Namen (im Folgenden `VAR-HTTP`), wählen Sie als «Target runtime» die von Ihnen in Akti-

vitätsschritt 8.8 installierte Serverkonfiguration (in Abb. 8.6 links ist das Apache Tomcat v9.0).
Im nächsten Schritt des Wizards zum Anlegen eines neuen «Dynamic Web Projects» wählen Sie den Speicherort für die Java-Dateien aus, wozu hauptsächlich Ihre Servlets zählen werden.
Im dritten Schritt des Wizards wählen Sie den Pfad im Projekt aus, unter dem statische Inhalte abgelegt werden. Aktivieren Sie für diese Übungsaufgabe die Option «Generate web.xml deployment descriptor» wie in Abb. 8.6 rechts.
Die resultierende Projektstruktur, die im *Project Explorer View* in Eclipse angezeigt wird, ist in Abb. 8.7 dargestellt.

Im nächsten Schritt des Wizards zum Anlegen eines neuen «Dynamic Web Projects» wählen Sie den Speicherort für die Java-Dateien aus, wozu hauptsächlich Ihre Servlets zählen werden (s. Abb. 8.6, zweites Fenster). Im dritten Schritt des Wizards wählen Sie den Pfad im Projekt aus, unter dem statische Inhalte abgelegt werden. Aktivieren Sie für diese Übungsaufgabe die Option «Generate web.xml deployment descriptor». Die resultierende Projektstruktur, die im *Project Explorer View* in Eclipse angezeigt wird, ist in Abb. 8.7 dargestellt.

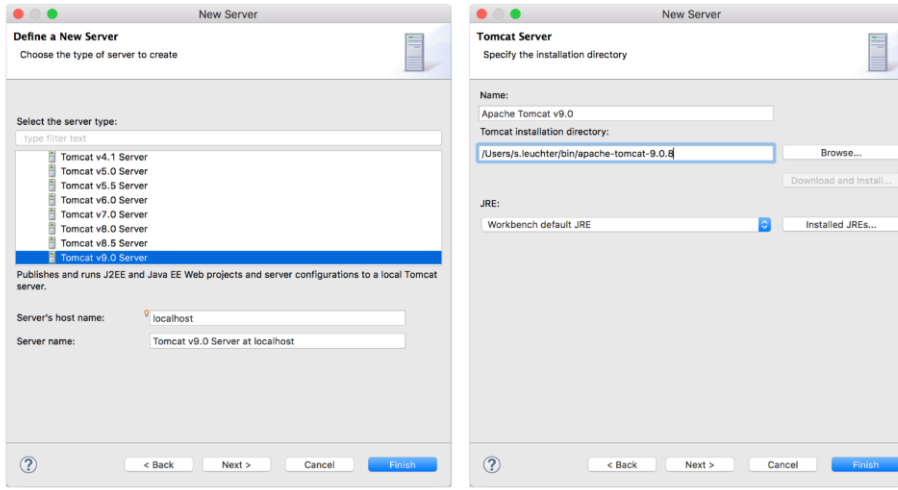


Abbildung 8.5: Screenshots der Schritte des Wizards zum Anlegen einer neuen Serverkonfiguration in Eclipse

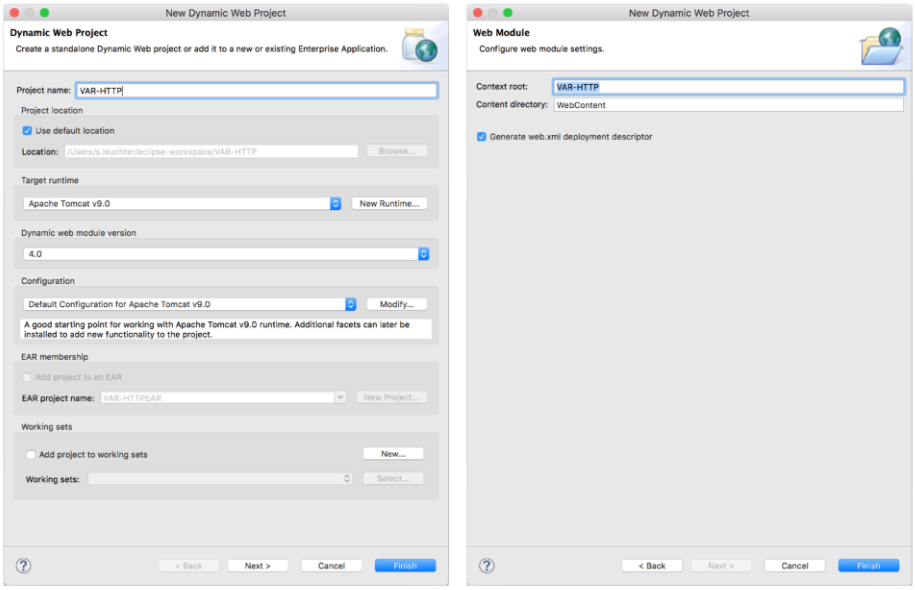


Abbildung 8.6: Screenshots der Schritte des Wizards zum Anlegen eines neuen Dynamic Web Project in Eclipse

Im Knoten *Deployment Descriptor: ...* in Abb. 8.7 sind die Konfigurationsdaten über diese Web-Anwendung übersichtlich dargestellt. Diese Informationen kommen aus der Datei `WebContent/WEB-INF/web.xml`. Statische Inhalte, die Teil dieser Web-Anwendung sein sollen, müssen im Knoten `WebContent` abgelegt werden (jedoch außerhalb der dort vorhandenen Verzeichnisse `META-INF` und `WEB-INF`). Dynamische Inhalte, also Java-Klassen, die von `HttpServlet` erben, werden im `Java Ressources/src` abgelegt (im Filesystem `workspace/VAR-HTTP/src/`, das ist das Verzeichnis, das im zweiten Schritt des Wizards angegeben wurde.

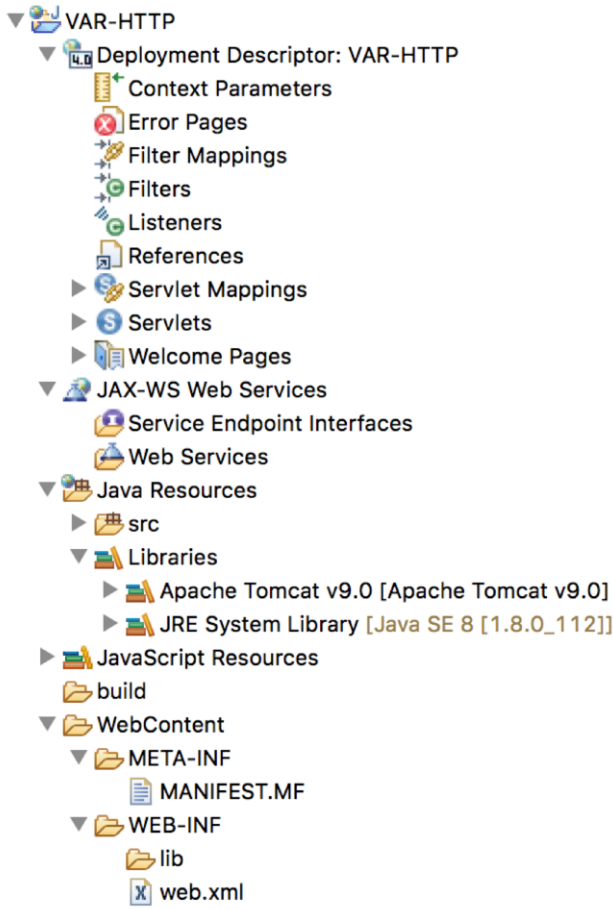


Abbildung 8.7: Screenshot des resultierenden Dynamic Web Projects im Eclipse Project Explorer View

8.6 Aufgabe: Hello World in einem «Dynamic Web Project»

Im nächsten Schritt werden Sie eine statische Datei vom Typ HTML in der Web-Anwendung erstellen und danach ein erstes Servlet.

8.6.1 Statische Inhalte in der Web-Anwendung: HTML File

Zuerst erstellen Sie eine statische HTML-Seite und konfigurieren Eclipse so, dass sie über Tomcat ausgeliefert wird. Dann fügen Sie einen TCP/IP-Monitor als Debugging-Hilfe hinzu, sodass Sie den Netzwerkverkehr zwischen Clients und Servlet Container beobachten können.

Aktivitätsschritt 8.10:
Ziel: Statische HTML-Seite neu anlegen
Erzeugen Sie über das Kontextmenü (Mausklick rechts) von `WebContent` mit *New* → *HTML File* eine neue HTML-Datei im Verzeichnis `WebContent` (s. Abb. 8.8). Nennen Sie die neue Datei `index.html`.
Als Ergebnis wird eine neue HTML-Datei namens `index.html` auf der Basis einer Vorlage in `WebContent` erstellt. Fügen Sie einige Informationen hinzu.

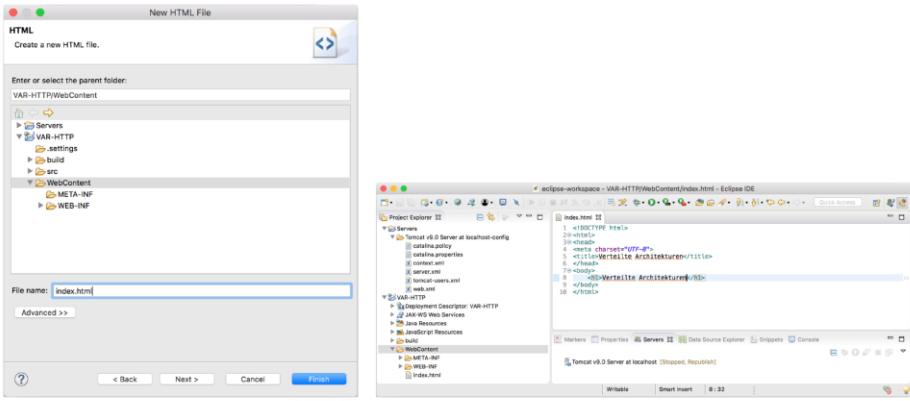


Abbildung 8.8: Dialog zum Anlegen einer neuen HTML-Datei im Verzeichnis `WebContent` einer Web-Anwendung und resultierendes Projekt

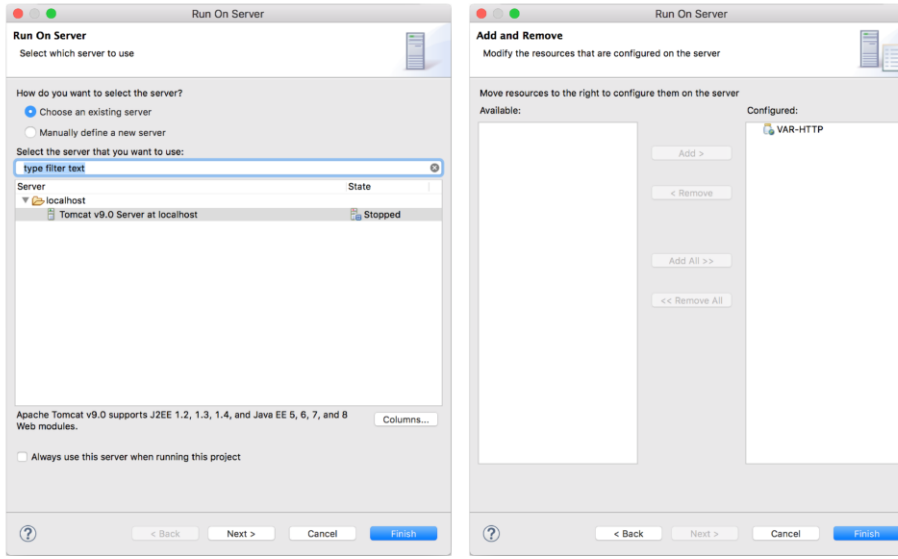


Abbildung 8.9: Screenshots vom Eclipse Wizard zum Start von Inhalt einer Web-Anwendung

Um die neue HTML-Datei «auszuprobieren», müssten Sie die Web-Anwendung, deren Teil die neue HTML-Datei ist, als WAR paketieren, diese auf einen Tomcat Server deployen und mit dem Browser vom Tomcat Server abrufen. Eclipse automatisiert diese Schritte über die neue Aktion *Run As* → *Run on Server* im Kontextmenü der HTML-Datei.

Aktivitätsschritt 8.11:
Ziel: Web-Anwendung lokal starten
Rufen Sie *Run As* → *Run on Server* im Kontextmenü der HTML-Datei auf.
Im ersten Schritt muss ausgewählt werden, auf welchem Server die Web-Anwendung deployt werden soll (s. Abb. 8.9, linke Seite).
Im zweiten Schritt (s. Abb. 8.9, rechte Seite) können dem ausgewählten Server noch weitere Web-Anwendungen («Dynamic Web Projects») hinzugefügt werden, die gerade in Eclipse geöffnet sind. Alle Web-Anwendungen in der rechten Liste werden dann in der Tomcat-Instanz, die zum Betrachten des ausgewählten Inhalts gestartet wird, verfügbar sein.

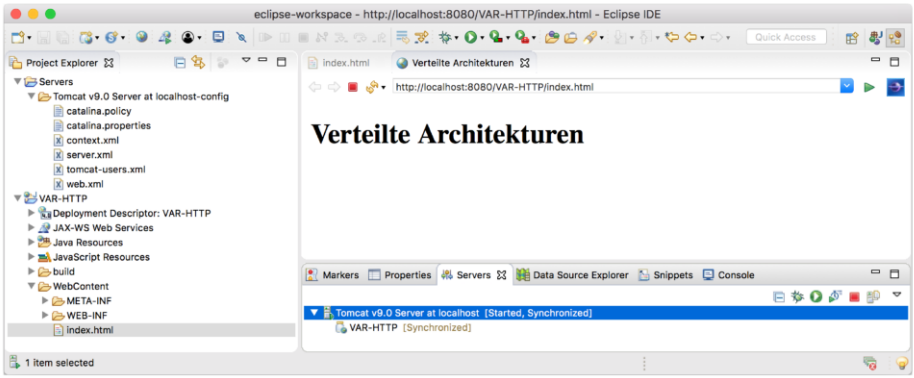


Abbildung 8.10: Screenshot der Anzeige des statischen Inhalts der Web-Anwendung im Eclipse-internen-Webbrowser

Aktivitätsschritt 8.12:

Ziel: Web-Anwendung in Eclipse EE laufen lassen und überwachen

Machen Sie sich mit den Möglichkeiten vertraut, wie der lokale Server beeinflusst werden kann (rotes Quadrat und grünes Dreieck im Browser View, s. Abb. 8.10, Kontextmenü im *Servers* View). Aktivieren Sie über das Kontextmenü Ihres Tomcat Servers im *Servers* View über *Monitoring* → *Properties* den Netzwerk-Monitor Dialog (s. Abb. 8.11, links).

Fügen Sie mit *Add...* einen neuen Monitor für den Server Port `8080` hinzu (s. Abb. 8.11, rechts). Port `8080` ist der Port von Tomcat. Geben Sie im Feld Monitor Port eine andere Port-Nummer ein, die auf Ihrem Rechner noch frei ist, z. B. `8081`. Starten Sie den neuen Monitor über den Start-Button auf der rechten Seite des Monitor-Properties-Dialogs. Der Status des Monitors muss zu *Started* wechseln (s. Abb. 8.12).

Wenn Sie nun einen HTTP Request an den Monitor-Port (z. B. `8081`) schicken, wird der geloggt und unverändert an den Server Port (z. B. `8080`) weitergeleitet.

Öffnen Sie den View TCP/IP-Monitor (*Window* → *Show View* → *Other...* → *Debug/TCP/IP-Monitor*, s. Abb. 8.13).

Benutzen Sie zusätzlich auch einen Webbrowser außerhalb von Eclipse, um auf die lokal deployte Web-Anwendung zuzugreifen (Sie können den URL einfach aus dem Eclipse-internen Browser View kopieren).

Tipp: Sollten keine HTTP Requests und Responses im TCP/IP-Monitor angezeigt werden, starten Sie Eclipse neu.

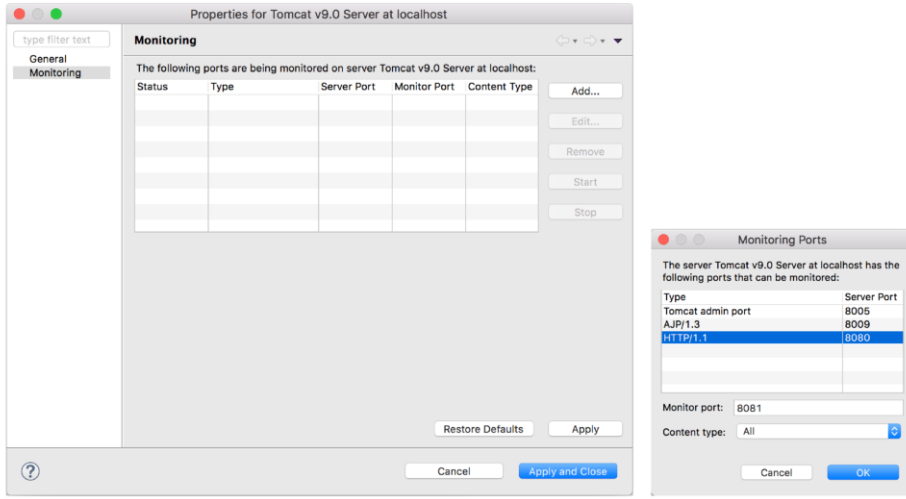


Abbildung 8.11: Screenshot von Eclipse mit Monitoring Properties

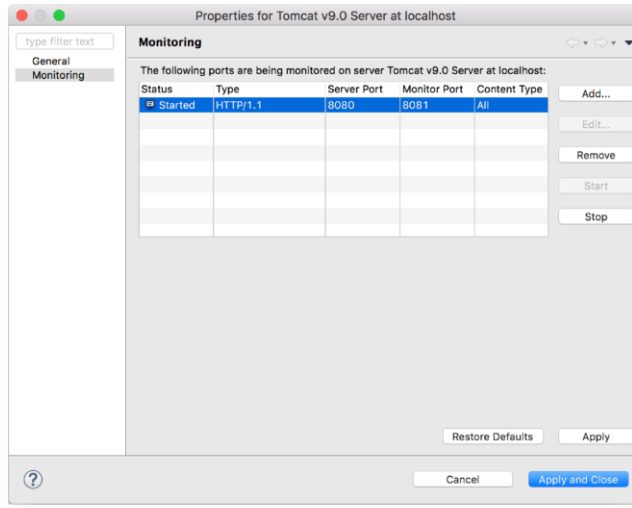


Abbildung 8.12: Screenshot von Eclipse mit Monitoring Properties (Fortsetzung)

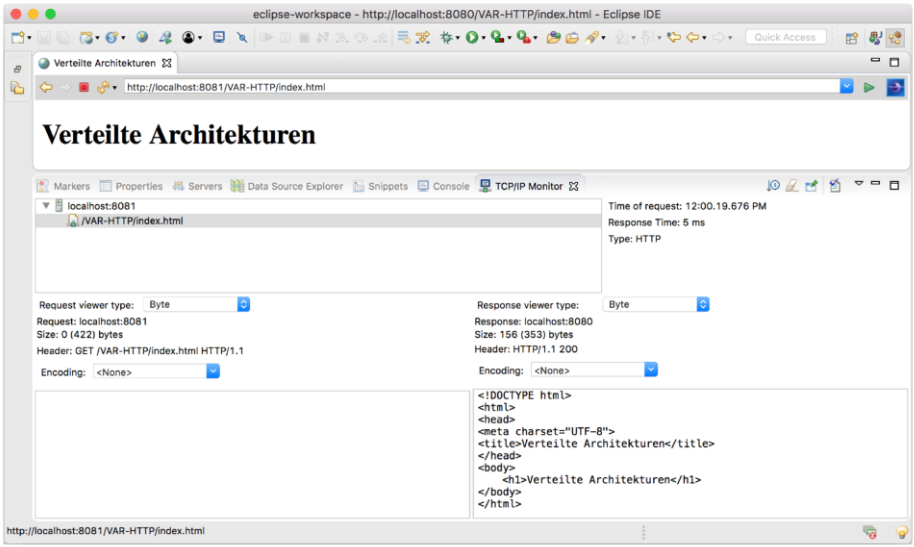


Abbildung 8.13: Screenshot von Eclipse mit angezeigtem TCP/IP-Monitor (nun erfolgt der Zugriff vom internen Browser aus über Port 8081)

8.7 Aufgabe: Dynamische Inhalte in der Web-Anwendung: HttpServlet

In dieser Übung werden Sie ein Servlet programmieren, das über Tomcat ausgeliefert wird. Dabei lernen Sie die Schnittstellen und Fähigkeiten von `HttpServlet` und dazugehörenden Klassen kennen.

Aktivitätsschritt 8.13:
Ziel: Package für Servlet im Dynamic Web Project anlegen
Legen Sie im «Dynamic Web Project» in Eclipse ein neues Servlet an. Gehen Sie dazu wie auf der linken Seite in Abb. 8.14 gezeigt im Project Explorer in Eclipse auf den Knoten Java Ressources/src und legen Sie dort ein neues Package an (z. B. `var.web.servlet.hello`). Finden Sie heraus, wo im Dateisystem auf Ihrer Festplatte das Verzeichnis für das Package angelegt wurde.

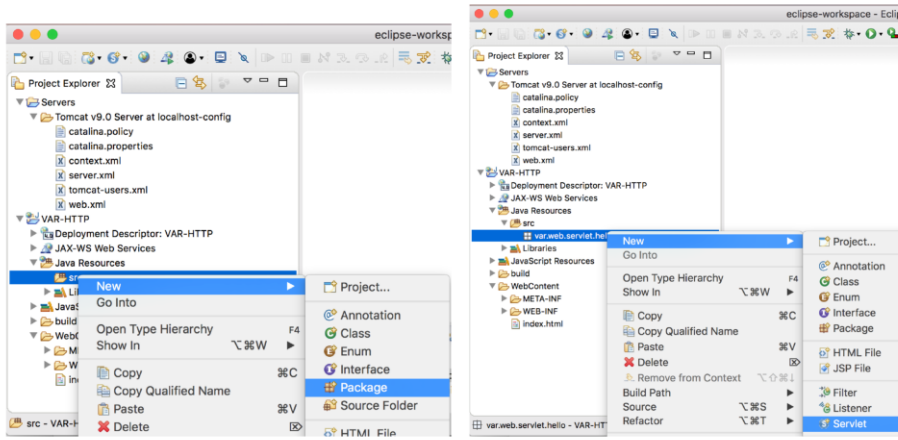


Abbildung 8.14: Anlegen eines neuen Packages (linke Seite) und einer neuen HttpServlet-Klasse (rechte Seite) in einer Dynamic Web Application in Eclipse

Aktivitätsschritt 8.14:

Ziel: Servlet im Dynamic Web Project anlegen

Erzeugen Sie dann eine neue HttpServlet-Klasse in diesem Package über das Kontektmenü mit *New Servlet* (s. Abb. 8.14, rechte Seite). In einem Wizard wird die grundlegende Konfiguration erfragt. Im ersten Schritt spezifizieren Sie die neue Java-Klasse; u.a. geben Sie den Namen der neuen Servlet-Klasse (z.B. `Hello World`) an (s. Abb. 8.15, linke Seite). Dann geben Sie im zweiten Schritt Informationen zum Deployment an (s. Abb. 8.15, rechts). Die Beschreibung des Servlets kann z.B. in einer Managementoberfläche angezeigt werden. Besonders wichtig ist, dass mindestens ein URL-Mapping angegeben wird. Das ist der letzte Teil des URL, der nach dem Kontextpfad (der die gesamte Web-Anwendung identifiziert) das konkrete Servlet anfragt. Es ist möglich, mehrere Mappings anzugeben. Das Servlet wird dann über mehrere URLs aufrufbar. Fügen Sie mehrere URL Mappings hinzu. Sie sollten mit «/» beginnen.

Im letzten Schritt des Wizards wird ausgewählt, welche Code-Teile als Gerüst in die Klasse generiert werden sollen. Das Ergebnis kann dann aussehen wie in Listing 8.6.

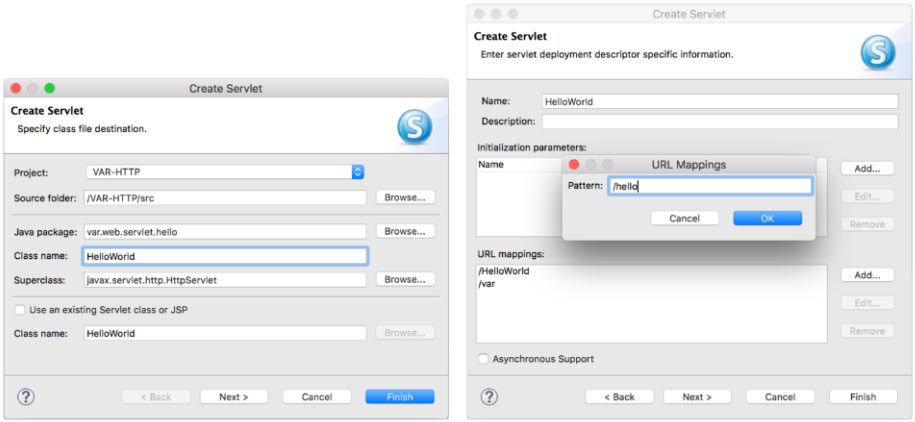


Abbildung 8.15: Screenshots vom Eclipse-Wizard zum Erzeugen eines neuen Servlets (Schritt 1-3)

Aktivitätsschritt 8.15:

Ziel: Position des Servlets im «Dynamic Web Project» und im Filesystem

Finden Sie heraus, wo im Dateisystem auf Ihrer Festplatte die neue Klasse als Datei angelegt wurde.

Aktivitätsschritt 8.16:

Ziel: Deployment des `HelloWorld`-Servlets

Deployen Sie das Servlet wie bei der statischen HTML-Seite über *Run As* → *Run on Server*. Dies bewirkt das Deployen und dass sich ein Eclipse-internes Browserfenster zum (ersten) Servlet Mapping öffnet.

Das Servlet kann nun aufgrund der Annotation @WebServlet unter den folgenden URLs aufgerufen werden:

- `http://localhost:8080/VAR-HTTP/HelloWorld`
- `http://localhost:8080/VAR-HTTP/hello`
- `http://localhost:8080/VAR-HTTP/var`

Probieren Sie auch aus, Ihr Servlet über einen Webbrowser anzusprechen.

Listing 8.6: Servlet `var.web.servlet.hello.HelloWorld`

```
1  package var.web.servlet.hello;
2
3  import java.io.IOException;
4  import javax.servlet.ServletException;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.HttpServlet;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 /**
11  * Servlet implementation class HelloWorld
12  */
13 @WebServlet({ "/HelloWorld", "/var", "/hello" })
14 public class HelloWorld extends HttpServlet {
15     private static final long serialVersionUID = 1L;
16
17     /**
18      * @see HttpServlet#HttpServlet()
19      */
20     public HelloWorld() {
21         super();
22         // TODO Auto-generated constructor stub
23     }
24
25     /**
26      * @see HttpServlet#doGet(HttpServletRequest request,
            HttpServletResponse response)
27      */
28     protected void doGet(HttpServletRequest request,
           HttpServletResponse response) throws ServletException,
           IOException {
29         // TODO Auto-generated method stub
30         response.getWriter().append("Served at: ").append(request.
               getContextPath());
31     }
32
33     /**
34      * @see HttpServlet#doPost(HttpServletRequest request,
            HttpServletResponse response)
35      */
36     protected void doPost(HttpServletRequest request,
           HttpServletResponse response) throws ServletException,
           IOException {
37         // TODO Auto-generated method stub
38         doGet(request, response);
39     }
40
41 }
```

Exkurs: Deployment Descriptor als `WEB-INF/web.xml` oder über Annotationen

Bei früheren Fassungen des Servlet- bzw. Java EE-Standards wurden alle Informationen des *Deployment Descriptors* in der Datei `WEB-INF/web.xml` repräsentiert. Seit der Servlet-Spezifikation 3.0 können die Servlet-spezifischen Informationen als Annotationen direkt im Servlet gespeichert werden.

Listing 8.7: Code-Fragment zur Ausgabe aus einem Servlet

```
1  protected void doGet(HttpServletRequest request, HttpServletResponse
       response) throws ServletException, IOException {
2      response.setContentType("text/html");
3      PrintWriter out = response.getWriter();
4      out.println("<html><head>");
5      out.println("<title>HTTPRequest Inhalte ausgeben</title>");
6      out.println("</head><body>");
7      // ... Inhalt der Seite ...
8      out.println("</body>");
9      out.flush();
10 }
```

8.7.1 HttpRequest

Im nächsten Schritt wird das Servlet ausgebaut.

Aktivitätsschritt 8.17:

Ziel: Ausgabe des Servlets über den HttpResponse-Parameter erzeugen

Bauen Sie die Methode `doGet` etwas um, damit Sie analog zu `System.out` über den Ausgabe-Strom des Servlets Ausgaben machen können. Orientieren Sie sich dabei am Code-Fragment in Listing 8.7.

Lassen Sie das Servlet dann laufen und inspizieren Sie die Ausgabe.

Aktivitätsschritt 8.18:

Ziel: Zugriff auf Header-Zeilen über den `HttpRequest`-Parameter

Fügen Sie an die Stelle // ... Inhalt der Seite ... in Listing 8.7 eine Ausgabe aller Header-Zeilen des HTTP Requests hinzu. Die Aufruf-Header können Sie über den `request`-Parameter erreichen, der die Methoden `getHeaderNames()` und

`getHeader(String)` anbietet. Zum Zugriff auf die Header-Zeilen des HTTP Requests können Sie die Struktur im Code-Fragment in Listing 8.8 und die dort benutzten Methoden verwenden.
Sollte der Wert des Header ein Datum oder eine Zahl sein, kann direkt über die Varianten `getDateHeader(String)`, `getIntHeader(String)` zugegriffen werden, um die Umwandlung zu sparen.
Lassen Sie das Servlet laufen und inspizieren Sie die Ausgabe.

Aktivitätsschritt 8.19:
Ziel: Zugriff auf Parameter über den `HttpRequest`-Parameter
Fügen Sie zusätzlich zur Ausgabe der Header an die Stelle // ... Inhalt der Seite ... eine Ausgabe aller Parameter des HTTP Requests hinzu. Die Parameter können analog zu den *Headern* über den `request`-Parameter erreicht werden (s. Listing 8.9). Die Methoden dafür heißen `getParameterNames()` und `getParameter(String)`.
Lassen Sie das Servlet laufen und inspizieren Sie die Ausgabe.

Listing 8.8: Code-Fragment zur Inspektion der Header-Zeilen des HTTP Requests

```
1 out.println("<h1>Header</h1><ul>");
2 Enumeration<String> headerNames = request.getHeaderNames();
3 while (headerNames.hasMoreElements()) {
4     String headerName = headerNames.nextElement();
5     String headerValue = request.getHeader(headerName);
6     out.println("<li>" + headerName + ": " + headerValue);
7 }
8 out.println("</ul>");
```

Listing 8.9: Code-Fragment zum Zugriff auf Parameter des HTTP Requests

```
1 out.println("<h1>Aufruf-Parameter</h1><ul>");
2 Enumeration<String> parameterNames = request.getParameterNames();
3 while (parameterNames.hasMoreElements()) {
4     String parameterName = parameterNames.nextElement();
5     String parameterValue = request.getParameter(parameterName);
6     out.println("<li>" + parameterName + ": " + parameterValue);
7 }
8 out.println("</ul>");
```

Listing 8.10: Inhalt der Datei `form1.html`

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Formular 1</title>
    </head>
    <body>
        <form action="hello" method="POST">
            <input type="hidden" name="Parameter1 (unsichtbar)" value="
                4711">
            <input type="hidden" name="Parameter2 (unsichtbar)" value="
                4712">
            Eingabe: <input type="text" name="Parameter3"><br />
            <input type="text" name="Parameter4" value="
                voreingetragener Wert"><br />
            <input type="submit" name="submitButton" value="OK">
        </form>
    </body>
</html>
```

8.7.2 Aufruf Parameter für Servlets

Wahrscheinlich werden noch keine Parameter ausgegeben. Das liegt daran, dass das Servlet im Moment noch ohne *Query* aufgerufen wird. Dafür wird eine Aufrufumgebung in Form eines HTML-Formulars gebraucht. In den nächsten Aufgaben erstellen Sie mehrere statische HTML-Dateien auf die gleiche Weise wie im Aktivitätsschritt «statische HTML-Seite neu anlegen».

Aktivitätsschritt 8.20:

Ziel: Erstellen eines HTML-Formulars für `POST`

Erstellen Sie eine HTML-Datei in `WebContent` namens `form1.html`. Der Inhalt von `form1.html` soll aus einem `FORM`-Element mit mehreren `INPUT`-Elementen bestehen. Die konkrete Gestaltung und welche Felder Sie genau vorsehen, ist für die kommenden Aktivitätsschritte nicht wichtig. Wichtig ist, dass als `ACTION`-Attribut des `FORM`-Elements ein URL für das Servlet gewählt wird und dass das `ACTION`-Attribut den Wert `POST` hat.. Da sich das Servlet (bzw. sein URL-Mapping) und die `form1.html`-Datei relativ auf derselben Ebene befinden, führt der Aufruf `hello` zum `HelloWorld`-Servlet. Ein Beispiel für solch eine Datei ist in Listing 8.10.

Sind Formular und Servlet auf derselben Ebene, haben Sie in diesem Beispiel auch denselben Kontextpfad:

Formular: `http://localhost:8080/VAR-HTTP/form1.html`
Servlet: `http://localhost:8080/VAR-HTTP/hello`

Aktivitätsschritt 8.21:
Ziel: Erstellen eines HTML-Formulars für `GET`
Erstellen Sie ein zweites Formular unter anderem Namen im selben Verzeichnis wie `form1.html`. Diesmal soll die Methode aber `GET` sein und nicht `POST`. Untersuchen Sie (auch unter Zuhilfenahme des TCP/IP-Monitors), wie sich die Aufrufe bei `GET` und `POST` unterscheiden.

Aktivitätsschritt 8.22:
Ziel: Untersuchen relativer Pfade in der Web-Applikation
Erstellen Sie ein neues Verzeichnis (*New* → *Folder*) in `WebContent` namens `dir/`. Kopieren Sie `form1.html` in das neue Verzeichnis. Ändern Sie das `ACTION`-Attribut so, dass das `HelloWorld`-Servlet weiterhin die Eingaben von diesem Formular empfängt.

Aktivitätsschritt 8.23:
Ziel: Zustand des Servlets über eine Instanzvariable beobachten
Führen Sie eine Instanzvariable zum Zählen der Aufrufe im Servlet ein, um die Beliebtheit Ihrer Seite zu überwachen oder um eine Maßeinheit zur Abrechnung gegenüber einem Kunden daraus abzuleiten. Bei jedem Aufruf von `getXXX(...)` soll der Zähler auf der Seite ausgegeben und auch um `1` erhöht werden. Analysieren Sie das beobachtbare Verhalten und folgern Sie, wie viele Instanzen des Servlets existieren. Versuchen Sie auch von anderen Rechnern zuzugreifen und beobachten Sie das Verhalten.

Aktivitätsschritt 8.24:
Ziel: Synchronisierung des Zugriffs auf die Instanzvariable im Servlet
Ändern Sie den Zugriff auf den Zähler aus der vorigen Aufgabe in eine Thread-sichere Variante ab. Verwenden Sie z. B. *intrinsic locks* wie in Listing 8.3.

8.8 Ausblick und Anregungen für eigene Projekte

Servlet Container stellen eine umfassende Technologie für Client-/Server-Architekturen bereit. Durch die Integration von Java-Klassen können serverseitig komplexe Funktionen realisiert werden.

8.8.1 Mediathek mit Jugendschutz

Die Idee für diese Aufgabe ist, eine einfache Kontrollinstanz einzurichten, die Bedingungen prüft und je nach Ausgang der Prüfung entweder eine Mediendatei ausliefert oder eine Fehlermeldung produziert. Als Bedingung kommen vielfältige Tatbestände in Betracht. Bspw. könnte eine Zugriffsautorisierung nach vorheriger Benutzerauthentifizierung erfolgen. Um es nicht kompliziert zu machen, wird hier vorgeschlagen einfach eine Zeitprüfung vorzunehmen. Erlaubt wird die Auslieferung von Medien aus einem bestimmten Medienverzeichnis nur zwischen 21:00 und 06:00 Uhr. Dies könnte z. B. durch eine Bedingung wie in Listing 8.11 geprüft werden.
Um eine Datei (vom `File`-Objekt `mediaFile`) auszuliefern, muss von der auszuliefernden Datei ein `InputStream` geöffnet werden und bspw. Byte-weise auf den Ausgabekanal wie in Listing 8.12 geschrieben werden (`out` ist der `OutputStream` der `HttpResponse`).

Listing 8.11: Prüfung der Bedingungen für die Medienauslieferung

```
1 Calendar now = Calendar.getInstance();
2 if (now.get(Calendar.HOUR_OF_DAY) > 6
3     && now.get(Calendar.HOUR_OF_DAY) < 21) {
4     /* abweisen */
5 } else {
6     /* ausliefern */
7 }
```

Listing 8.12: Datei, die ausgeliefert werden soll, öffnen und Byte-weise auf den Ausgabekanal `out` schreiben

```
1  try (InputStream in = new BufferedInputStream(new FileInputStream(
       mediaFile))) {
2      int character;
3      while ((character = in.read()) != -1) {
4          out.write((byte) character);
5      }
6  } catch (Exception ex) {
7      // Fehlerbehandlung
8  }
```

Als Fehlermeldung könnte der HTTP Response Code `403` (`HttpServletResponse.SC_FORBIDDEN`) zurückgegeben werden.

Aktivitätsschritt 8.25 (fakultativ):

Ziel: Servlet zur zeitgesteuerten Auslieferung von Files erstellen

Implementieren Sie ein Servlet, das Medien aus dem Verzeichnis `media/` nur zwischen 21:00 und 06:00 Uhr ausliefert. Es wird vorgeschlagen, den Namen und Pfad über den *Path* des `HttpRequests` zu übergeben. Das kann man leicht erreichen, wenn als URL Pattern für das Servlet «`/media/*`» angegeben wird. Dann wird das Servlet jedesmal aufgerufen, wenn eine Datei unterhalb von `/media/` angefordert wird.

```
@WebServlet(urlPatterns = "/media/*")
```

Auf den angeforderten Dateinamen kann man mit der Methode `getPathInfo()` am `HttpRequest` (hier über den Parameter `request`) zugreifen:

```
File mediaFile = new File(getServletContext().getRealPath("/media/"
    + request.getPathInfo()));
```

Unter `http://verteiltearchitekturen.de/vol01/VAR-Servlets-Filter.zip` können Sie eine Lösungsskizze zu dieser Aufgabe herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-Servlets_Filter_solution`.

Exkurs: Das Interface `javax.servlet.Filter`

Seit der Servlet-Spezifikation 2.3 gibt es das `Filter`-Interface. `Filter` Implementierungen transformieren `HttpRequest` und/oder `HttpResponse`. Sie können verkettet werden und liefern damit wiederverwendbare Bausteine für Aufgaben wie Authentifizierung/Autorisierung, Ver-/Entschlüsselung, Transformation von Datenformaten, Zugriffsprotokollierung und Datenkompression.
Das Listing 8.13 zeigt exemplarisch, wie ein `Filter` ähnlich einem `HttpServlet` implementiert wird. Analog zu den `HttpServlet`-Methoden `doXXX(...)` bzw. `service(...)` wird bei `Filter`-Implementierungen die Methode `doFilter(...)` überschrieben. In ihr wird dann über eine `FilterChain` zum nächsten Filter weitergeleitet.

Listing 8.13: Grundsätzliche Struktur bei der Implementierung eines `javax.servlet.Filter`

```
1  public final class MyFilter implements Filter {
2      private FilterConfig filterConfig = null;
3      @Override
4      public void init(FilterConfig filterConfig) throws ServletException
           {
5          this.filterConfig = filterConfig;
6      }
7      @Override
8      public void destroy() {
9          this.filterConfig = null;
10     }
11     @Override
12     public void doFilter(ServletRequest req, ServletResponse resp,
           FilterChain chain) throws IOException, ServletException {
13         if (filterConfig == null) {
14             return;
15         }
16         // Request untersuchen
17         // Request wrappen und/oder Response wrappen
18         chain.doFilter(req, resp); // bzw. jeweils gewrappte req/resp
19     }
20 }
```

Aktivitätsschritt 8.26 (fakultativ und optional):

Ziel: Zeitgesteuerte Auslieferung von Files mit `Filter`

Bauen Sie das Servlet aus Aktivitätsschritt 8.25 so um, dass ein `Filter` zur Zeitkontrolle verwendet wird. Sie müssen dazu eine neue Klasse wie in Listing 8.13 implementieren, die das `Filter`-Interface umsetzt.

Aktivitätsschritt 8.27 (fakultativ):

Ziel: Entwickeln Sie eine Mediathek

Entwickeln Sie eine Mediathek zur Auslieferung von Videodateien. Sie sollten berücksichtigen, dass manche Videos aus Jugendschutzgründen nur eingeschränkt zugreifbar sein sollen. Es gibt aber auch allgemein freigegebene Dateien, bei denen ein Download jederzeit erlaubt ist. Als alternativen Zugang könnten Sie die Möglichkeit vorsehen, für autorisierte Benutzer, deren erwachsenes Alter bereits im Vorfeld verifiziert wurde, den Download eingestufter Inhalte jederzeit zu gestatten.

Deployment auf einer Cloud-basierten Plattform

In diesem Kapitel geht es darum, Web-Anwendungen professionell in der Cloud – konkret als Java Servlet in der Google App Engine – zu *hosten*, also netzbasiert zu betreiben. Sie werden einen zunächst kostenfreien Account für das Google Cloud Angebot einrichten sowie lokal für die Zielumgebung Google App Engine eine Web-Anwendung mit Servlets entwickeln und testen. Anschließend wird die Web-Anwendung über Googles Cloud Plattform öffentlich zugänglich gemacht und mit einem eigenen Domain-Namen verbunden.

9.1 Cloud Computing

Cloud Computing bedeutet, dass Ressourcen von Rechenzentren genutzt werden, die nicht unter der direkten Kontrolle des Nutzers stehen, sondern von Drittanbietern betrieben werden. Solche Ressourcen (z. B. Speicher, Rechenleistung, Anwendungen oder Backup-Fähigkeiten) werden über das Internet bereitgestellt. Im Regelfall werden Cloud-Computing-Anbieter ihre Ressourcen mehreren Kunden bereitstellen, so dass

S. Leuchter, *Internet-Kommunikation*, DOI: 10.5281/zenodo.1042168

jeder Nutzer nur einer von mehreren Mietern ist. Beim Cloud Computing geht es also in erster Linie darum, die Nutzer vom Betrieb eines Rechenzentrums und von eigenen Investitionen in Hardware gegen Mietgebühren zu entlasten.
Dabei steht zumeist auch die flexible Nutzung im Vordergrund. Kunden wollen in der Lage sein, ohne lange Vorlaufzeiten Hardwareressourcen aufzustocken um bspw. Lastspitzen abzufangen. Gleichzeitig soll die dafür erforderliche Hardware in Zeiten geringerer Auslastung nicht ohne Verwendung vorgehalten werden müssen.
Um eine rechtliche Handhabe gegenüber den Betreibern von Cloud-Angeboten zu haben, schließen Kunden im Regelfall Verträge über die zu erbringende Dienstgüte mit dem Anbieter. Solche «Service Level Agreements» (SLA) definieren messbare Größen wie die Verfügbarkeit von Ressourcen und die Reaktionszeiten beim Bereitstellen von zusätzlichen Ressourcen in Spitzenlastzeiten.
Standardisierte Schnittstellen und Plattformen erlauben den Kunden ihre Cloud-Anbieter zu wechseln oder auch einen Mix aus mehreren Cloud-Angeboten bei mehreren Anbietern zusammenzustellen. Da es vielen Nutzern erforderlich erscheint, ein gewisses Maß an Kontrolle zu behalten, gibt es in der Praxis auch hybride Cloud-Computing-Lösungen, bei denen die Grundlast durch ein eigenes Rechenzentrum bereitgestellt wird und nur in Spitzenlastzeiten Kapazitäten bei externen Cloud-Computing-Anbietern hinzugekauft werden. Der eigene Betrieb von Rechenzentrumsressourcen und ihre Bereitstellung kann über die gleichen Internet-Protokolle und über SLAs gegenüber Fachabteilungen erfolgen. Wird eine Anwendung selber betrieben, spricht man auch von «*on premise*».

9.2 Cloud-Service-Modelle

Beim Betrieb von Anwendungen in der Cloud werden drei Cloud-Service-Modelle unterschieden und vom «*on premise*»-Betrieb abgegrenzt.

- *Software as a Service* (SaaS)
- *Platform as a Service* (PaaS)
- *Infrastructure as a Service* (IaaS)

In Abb. 9.1 werden die Cloud-Service-Modelle SaaS, PaaS und IaaS mit selbst gehosteten Anwendungen verglichen. In jedem der vier Fälle gibt es eine andere Aufteilung

der Verantwortlichkeit zwischen Cloud-Betreiber und Nutzer. Im Extremfall SaaS werden alle Aufgaben vom Cloud-Betreiber übernommen. Nutzer greifen nur noch über eine Netzwerkverbindung (im Regelfall HTTP) auf beim Cloud-Anbieter laufende Anwendungen zu und haben keine Möglichkeit in den Betrieb einzugreifen. «*on premise*» ist über die Abstufungen PaaS und IaaS der gegenteilige Pol. Bei «*on premise*» wird der Betrieb von Anwendung, Plattform, Infrastuktur und Hardware komplett beim Nutzer («in dessen Geschäftsräumen», engl. «*on premise*») verantwortet.

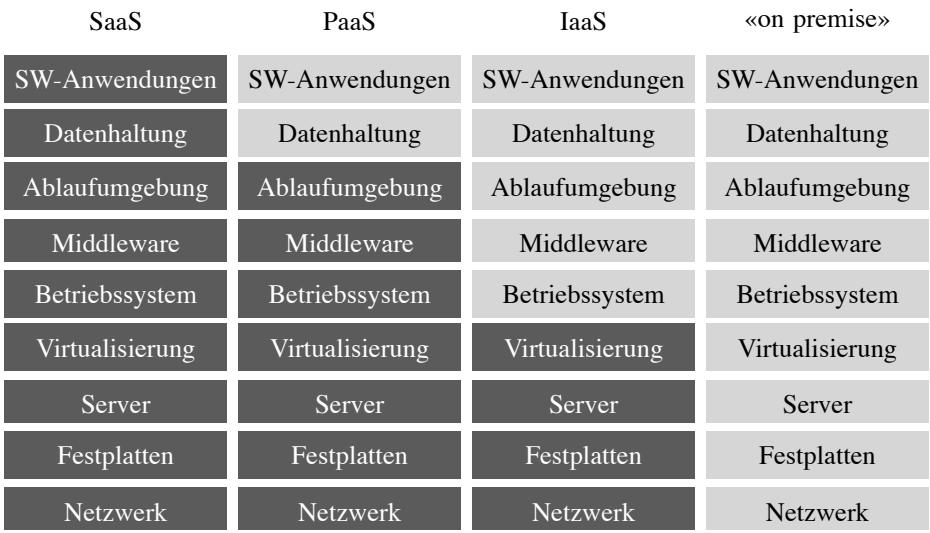


Abbildung 9.1: Die Cloud-Service-Modelle SaaS, PaaS und IaaS im Vergleich mit selbst gehosteten Anwendungen: *dunkel* ≙ vom Cloud-Anbieter bereitgestellt, *hell* ≙ muss vom Nutzer betrieben werden

Exkurs: «Serverless»-Backend-Architektur

Ein aktueller Trend sind «Serverless»-Backend-Architekturen. Im Prinzip wird wie beim Cloud-Service-Modell PaaS eine gemanagte Ablaufumgebung für Code des Nutzers bereitgestellt. Der Unterschied liegt in der Granularität. Beim Modell PaaS wird im Regelfall eine komplette Anwendung für den Cloud-Betrieb entworfen, die aus vielen untereinander abhängigen Teilen besteht.
Das Abrechnungsmodell orientiert sich dementsprechend an laufenden Rechnerinstanzen, die jeweils die gesamte Anwendung ausführen. Im Gegensatz dazu ist die Granularität von «serverless» bereitgestellten Anwendungen feiner: Bei «serverless» werden nur noch einzelne Funktionen bzw. Services in der Cloud bereitgestellt. Die Abrechnung erfolgt pro Funktionsaufruf.
Dieses Modell wird auch *Function as a Service* (FaaS) oder *Backend as a Service* (BaaS) genannt, weil beim Entwurf verteilter Architekturen einzelne Funktionen so von einem leichtgewichtigen mobilen Frontend in die Cloud als Backend ausgelagert werden können.

9.3 Platform as a Service

Die Google App Engine ist ein Cloud-Hosting-Angebot, das eine PaaS anbietet. PaaS ist ein Cloud-Service-Modell, bei dem der Cloud-Anbieter den Nutzern eine Umgebung in Form eines Applikationsservers bereitstellt. Der Applikationsserver läuft auf Hardware im Rechenzentrum des Cloud-Anbieters. Der Vorteil von PaaS im Vergleich zu IaaS und selbst gehosteten Anwendungen ist, dass Nutzer der Plattform vom Betrieb der Hardware und der Infrastruktur (Betriebssystem, Applikationsserver, Middleware, Backup, Updates) entlastet werden. Diese Aufgaben übernimmt der Cloud-Anbieter.
Der Nutzer konzentriert sich im Gegensatz zur Betriebsseite darauf, die eigenen Funktionen in Form von auf der Plattform ablauffähigem Code zu entwickeln. PaaS-Angebote beinhalten immer auch Werkzeuge für das Management einer deployten Anwendung, also z. B. eine Oberfläche um Komponenten zu starten oder neue Versionen einzuspielen. In diesem Praktikum wird das PaaS-Angebot in der Cloud Infrastruktur von Google, das unter dem Namen «Google App Engine» vermarktet wird, auf diese Art verwendet.
PaaS-Anbieter sind besonders stark darin, Lastverteilung und Ausfallsicherheit zu gewährleisten, ohne dass der Cloud-Kunde dafür besondere Vorkehrungen treffen müsste. In der Google App Engine kann man als Kunde zwischen unterschiedlichen *Service*

Exkurs: Google App Engine

In diesem Praktikum wird mit der Google App Engine dieselbe Infrastruktur verwendet, die auch professionelle Anbieter (z. B. Snapchat, das komplett auf Google's App Engine gehostet wird) für die Bereitstellung geschäftskritischer Anwendungen benutzen. Als Applikationsserver stehen bei Google App Engine (im *standard environment*) im Moment Umgebungen für Python 2.7, Java 7/8 (Java Servlets), PHP 5.5 und Go 1.6/1.8/1.9 zur Verfügung. Außerdem ist es im *flexible environment* zusätzlich möglich Docker Container mit Programmen in Node.js, Ruby und .NET zu benutzen.

Levels wählen, anhand derer sich bestimmt, wie viele parallellaufende Instanzen des Applikationsservers zur Verfügung stehen, um Requests abzuarbeiten.

9.4 Skalierung

Einer der Vorteile von PaaS ist die prinzipielle Fähigkeit zur Skalierung. Die Infrastruktur kann die Anzahl der Client-Zugriffe auf die über die Plattform gehosteten Web-Anwendungen überwachen und bei erhöhten Zugriffsaufkommen automatisch mehr Instanzen der Ablaufumgebung auf mehr Rechnern starten und möglicherweise auch intern Netzwerke umkonfigurieren, so dass die Ressourcen für die aktuelle Last optimiert werden.
Dadurch kann es passieren, dass mehrere Instanzen eines Servlets (auf mehreren Rechnern) parallel laufen. Es ist aber ebenso möglich, dass laufende Instanzen des eigenen Servlets beendet werden. Servlets, die auf einer PaaS deployt werden, dürfen also keinen Zustand besitzen.
Beim PaaS-Angebot von Google, der Google App Engine, müssen Daten über spezielle Google-Services persistiert werden und auch auf vorher persistierte Daten zugreifen. Aus diesem Grund (und auch aus kommerziellen Erwägungen heraus) ist die Ablaufumgebung im Vergleich zum Standard Servlet Container eingeschränkt. Externe Libraries können nicht ohne Weiteres verwendet werden, da Google die eigenen Services kostenpflichtig bereitstellt.

9.5 Statische Inhalte in PaaS Servlet Containern

An PaaS-Angeboten, wie der Google App Engine, sind der automatische Lastausgleich und die sehr einfache Administrierbarkeit attraktiv.
Es ist möglich, den Servlet Container eines PaaS-Angebots als Plattform für komplett statische Websites zu «zweckentfremden». Im Fall der Google App Engine kann über das Eclipse Plugin einfach ein Projekt angelegt werden, das keine Java-Anteile beinhaltet, sondern komplett aus HTML-Seiten und Mediendateien besteht. Wird solch ein Projekt in der App Engine deployt, erhält man eine statische Website mit hoher Verfügbarkeit, optional eingebautem Lastausgleich und einfacher Administrationsoberfläche.

Praktikum

Aktivitätsschritte:

Unter `http://verteiltearchitekturen.de/vol01/VAR-GAE-solution.zip` können Sie eine Musterlösung zu diesem Praktikum herunterladen. Darin befindet sich das Eclipse-Projekt `VAR-googleappengine_solution`.

9.6 Aufgabe: Web-Anwendung für Wahl in Google App Engine deployen

Aktivitätsschritt 9.1 (falls nicht bereits vorhanden):
Ziel: Google-Account anlegen

Voraussetzung für die Bearbeitung der folgenden Praktikumsaufgaben ist ein Google-Account, z. B. für *gmail.com*, den Sie erst noch anlegen müssen, falls Sie noch keinen haben. Mit diesem Account kann man sich dann in der *Develo-*

per Console einloggen: `http://appengine.google.com/`. Alternativ kann man auch direkt dort über «Konto erstellen» einen Zugang zur Nutzung anlegen. Sie müssen jedoch noch durch einen Identifizierungsprozess gehen.

Aktivitätsschritt 9.2:
Ziel: Download des Software Development Kits (SDK)
Google App Engine SDK und Google Cloud SDK sind Voraussetzung zur Nutzung der Plattform.
Die Java-Variante des Google App Engine SDK kann kostenfrei unter `https://cloud.google.com/appengine/docs/standard/java/download` heruntergeladen werden.
Wählen Sie in jedem Fall den Download des Java SDK für das «*standard environment*». Das «*flexible environment*» ist hingegen für IaaS Deployments.
Folgen Sie den Download- und Installationsanweisungen für Ihr Betriebssystem.
Das Cloud SDK kann unter `https://cloud.google.com/sdk/` heruntergeladen werden. Folgen Sie auch hier den Download- und Installationsanweisungen für Ihr Betriebssystem.

Das SDK enthält alle Schnittstellen und Libraries um die zu hostende Web-Anwendung zu entwickeln und lokal auf dem eigenen Rechner in einer Sandbox (das ist ein funktionales Abbild der Google-App-Engine-Plattform) zu testen. Die Sandbox bringt dazu alle Services der Google App Engine mit. Außerdem enthält das SDK alles, um eine fertig entwickelte Web-Anwendung zu verpacken und in der gehosteten PaaS-Infrastruktur von Google unter Ihrem Google Account zu deployen. Die weitere Konfiguration Ihrer Web-Anwendung (z. B. Einstellungen für Lastausgleich) nehmen Sie in der Web-Oberfläche (`https://appengine.google.com/`) vor.

Aktivitätsschritt 9.3:
Ziel: Integration des SDK in Eclipse
Das SDK für die Google App Engine kann über Eclipse verwendet werden. Dazu muss über das Menü *Help* → *Install New Software*... zuerst das Google Repository eingebunden werden. Von diesem Repository können dann die erforderlichen Eclipse-Komponenten nachgeladen werden.

Fügen Sie das «Google Cloud Platform for Eclipse»-Repository mit der URL `https://dl.google.com/eclipse/google-cloud-eclipse/stable/` über den Button *Add...* bei *Work with:* hinzu (s. Abb. 9.2). Wählen Sie dann wie in Abb. 9.3 dargestellt das Eclipse Plugin «Google Cloud Platform for Eclipse» (falls noch nicht installiert) zur Installation aus. Bestätigen Sie diesen Schritt mit dem *Next*-Button des Installations-Wizards.
Im Ergebnis werden eine Reihe Komponenten in Eclipse installiert, die für die Integration der Google Cloud Platform erforderlich sind (s. Abb. 9.4).
In den folgenden Schritten müssen die Lizenz akzeptiert und dann Eclipse neu gestartet werden.

9.6.1 Serverseitiges Projekt in der Google App Engine anlegen

Nachdem die Voraussetzungen für die Nutzung der Google App Engine hergestellt wurden, muss im nächsten Schritt ein serverseitiges Projekt angelegt werden.

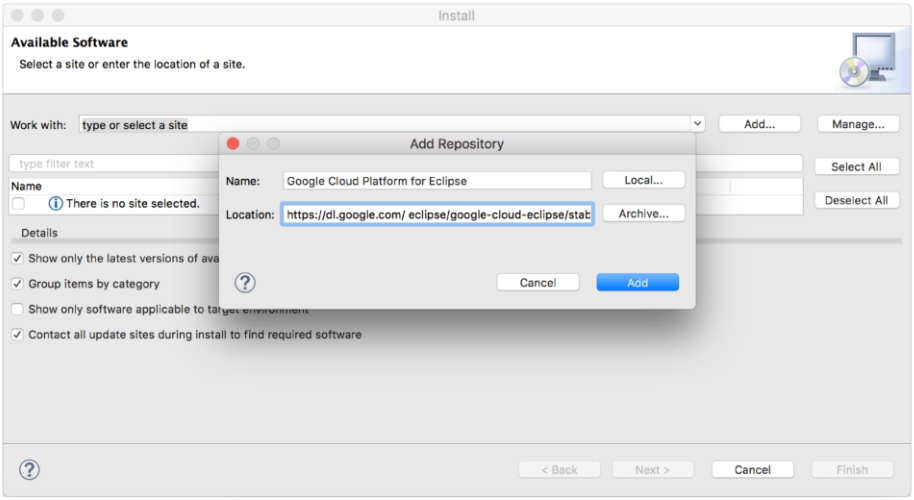


Abbildung 9.2: Einbinden des Google Repositories `https://dl.google.com/eclipse/google-cloud-eclipse/stable/` in Eclipse über den Button *Add...* bei *Work with:*

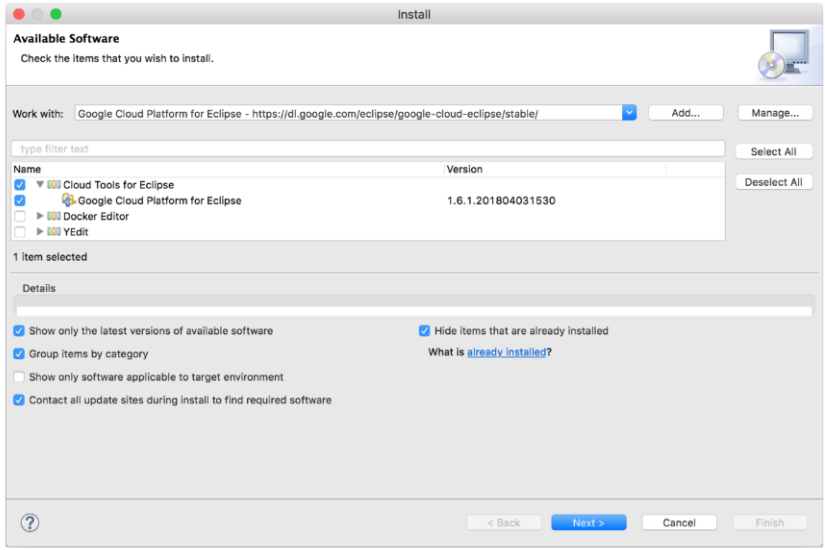


Abbildung 9.3: Installieren der Google Cloud Platform for Eclipse

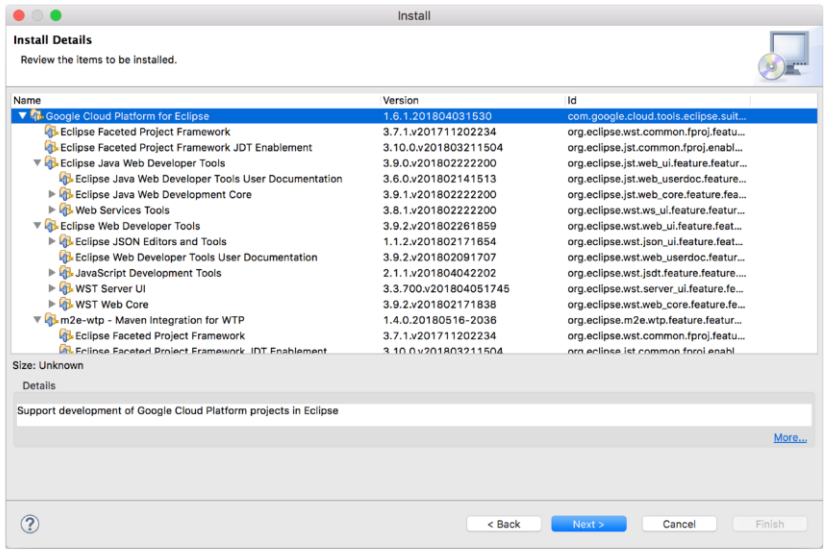


Abbildung 9.4: Installieren der Google Cloud Platform for Eclipse: Liste der zu installierenden Komponenten

Aktivitätsschritt 9.4:

Ziel: Serverseitiges Projekt in Google App Engine anlegen

Loggen Sie sich in die Entwickler-Konsole (s. Abb. 9.5) der Google Cloud Platform ein: `https://console.cloud.google.com/cloud-resource-manager`. Sie brauchen dazu einen Google-Account, also z. B. `vorname.nachname@gmail.com` (s. Aktivitätsschritt 9.1).

Google hat ein umfassendes Angebot zu unterschiedlichen Aspekten des Cloud-Computing. Die App Engine ist ein Teil davon.

Anfangs sollte die Liste der serverseitigen Projekte leer sein (s. Abb. 9.6). Legen Sie über die Funktion «Create Project» ein neues serverseitiges Projekt an. Dessen ID wird in den nächsten Schritten benötigt. Die Screenshots im Folgenden gehen vom Projektnamen «`VAR-GoogleAppEngine`» aus. Da es zu Namenskollisionen mit anderen Entwicklerinnnen und Entwicklern kommen könnte, wird von Google zu dem Projektnamen automatisch eine eindeutige Projekt-ID erstellt. Es kann sein, dass die Projekt-ID sogar identisch mit dem Projektnamen ist. Im folgenden Beispiel wurde von Google die Projekt-ID «`citric-sol-205517`» zum Projektnamen «`VAR-GoogleAppEngine`» vergeben (s. Abb. 9.5 und 9.7).

Die ID wird später auch im URL Ihres öffentlich zugänglichen Deployments verwendet, falls Sie keinen eigenen Domain-Namen kaufen. Er sollte Ihnen also zusagen, wenn Sie keinen eigenen Domain-Namen benutzen.

Zu jedem Projekt gehört ein Name und eine Projekt-ID, die in späteren Schritten noch gebraucht wird. Als erweiterte Option konnte in früheren Versionen der Google Cloud Platform an dieser Stelle bereits der Ort des Rechenzentrums gewählt werden, wo die Anwendung gehostet werden soll. Wenn Sie die Möglichkeit dazu haben, solch eine Einstellung vorzunehmen, sollten Sie vorzugsweise die «App Engine location» auf `europe-west` stellen. In der aktuellen Version der Entwickler-Konsole der Google Cloud Platform ist es erst in einem späteren Schritt (beim Deployment, s. Aktivitätsschritt 9.10) möglich die *Location* einzustellen. In Abb. 9.8 ist der *Settings*-Dialog zu sehen, der hier noch keine *Location*-Einstellungen erlaubt.

Die Parameter, die das Hosting der Web-Anwendung beeinflussen, werden über das «Dashboard» (s. Abb. 9.10), erreichbar über einen URL mit der ID des neuen Projekts: `https://console.cloud.google.com/home/dashboard?project=citric-sol-205517` oder das Menü, das in Abb. 9.9 zu sehen ist, konfiguriert. In Abschnitt 9.6.4 geht es weiter mit Einstellungen zum serverseitigen

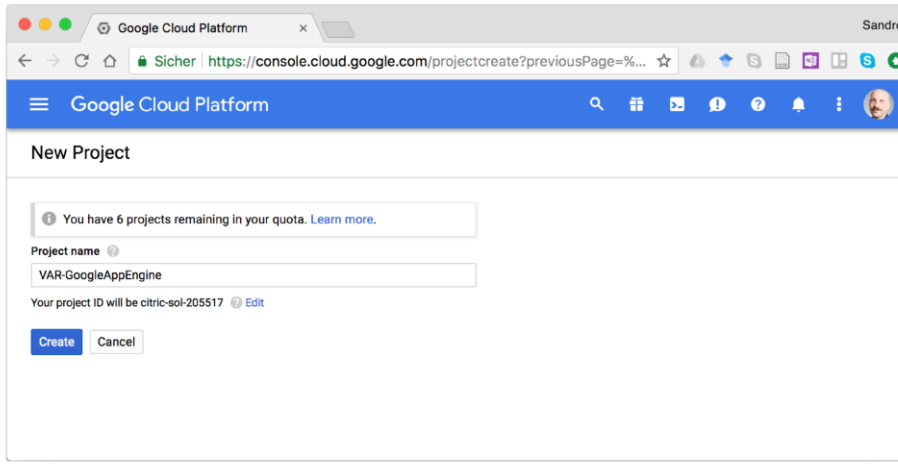


Abbildung 9.5: Anlegen des ersten serverseitigen Projekts «`VAR-GoogleAppEngine`» in der Entwickler-Konsole der Google Cloud Platform

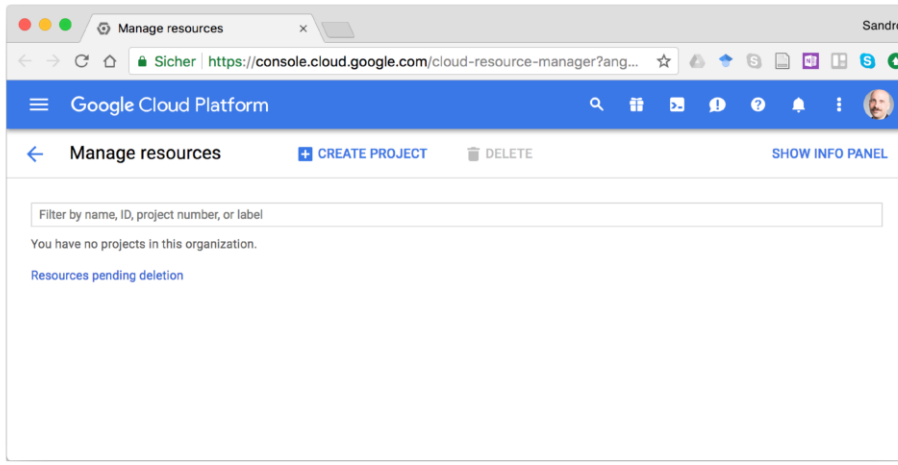


Abbildung 9.6: Ansicht der anfangs leeren Liste der serverseitigen Projekte in der Entwickler-Konsole der Google Cloud Platform beim Anlegen des Projektes «`VAR-Google AppEngine`»

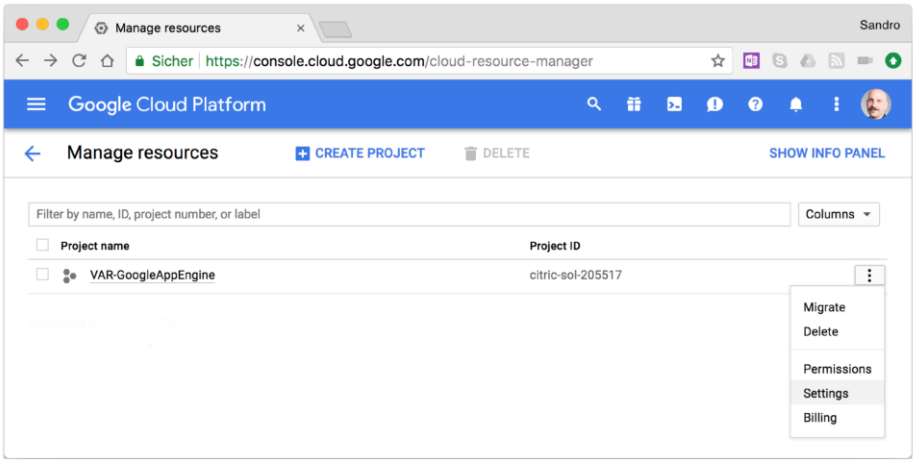


Abbildung 9.7: Ansicht der Liste der serverseitigen Projekte in der Entwickler-Konsole der Google Cloud Platform beim Anlegen des Projektes «`VAR-GoogleAppEngine`»

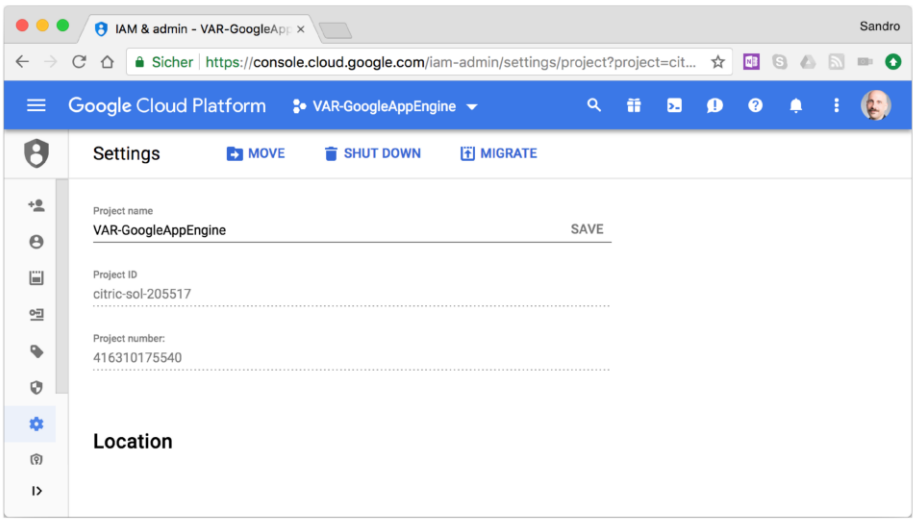


Abbildung 9.8: Ansicht der «Settings» des neu angelegten serverseitigen Projekts «`VAR-GoogleAppEngine`» in der Entwickler-Konsole der Google Cloud Platform

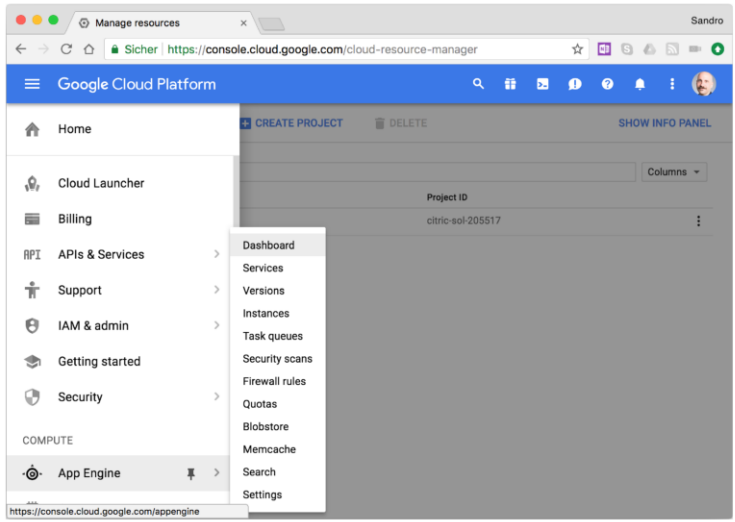


Abbildung 9.9: Aufruf des Menüs um zum Dashboard der Google App Engine in der Entwickler-Konsole zu gelangen

Projekt, die über das Dashboard vorgenommen werden. Zunächst soll aber erst ein lokal lauffähiges Projekt in Eclipse entwickelt werden.

9.6.2 Lokales (Eclipse-seitiges) Projekt anlegen

Um die Programmierung der Web-Anwendung zu beginnen, muss lokal ein neues Projekt der Art *Google App Engine Standard Java Project* in Eclipse angelegt werden. Man kann die Anmeldung mit dem eigenen Google Account bis zum späteren Deployment verschieben (Login wird erst in Aktivitätsschritt 9.10 benötigt).

Aktivitätsschritt 9.5:
Ziel: Google App Engine Standard Java Project in Eclipse anlegen
Erzeugen Sie in Eclipse ein lokales *Google App Engine Standard Java Project* über den Wizard *File* → *New* → *Other...* → *Google Cloud Platform/Google App Engine Standard Java Project* (s. Abb. 9.11).
Das serverseitige Projekt muss einen Namen erhalten (im Beispiel in Abb. 9.12 «`VAR-googleappengine`»). Dieser Projektname muss dabei nicht mit dem

Namen oder der ID des serverseitigen Projekts (s. Aktivitätsschritt 9.4) identisch sein.

Das resultierende Eclipse-*Google App Engine Standard Java Project* kann analog zum *Dynamic Web Project* von Eclipse für Java EE mit statischen Webseiten (unter `src/main/webapp/`) und Klassen (unter `VAR-Wahl.zipmain/java`) befüllt werden. Das Beispielprojekt (s. Abb. 9.13) besteht aus den Dateien `var.web.cloud.poll.BallotBox` (eine gewöhnliche Java-Klasse), dem `HttpServlet` `var.web.cloud.poll.BallotBoxServlet`, das `BallotBox` verwendet, und der statischen HTML-Seite `wahl.html`, die ein `FORM`-Element enthält, das das `BallotBoxServlet` über seine relative URL `/vote` als `ACTION`-Attribut referenziert.

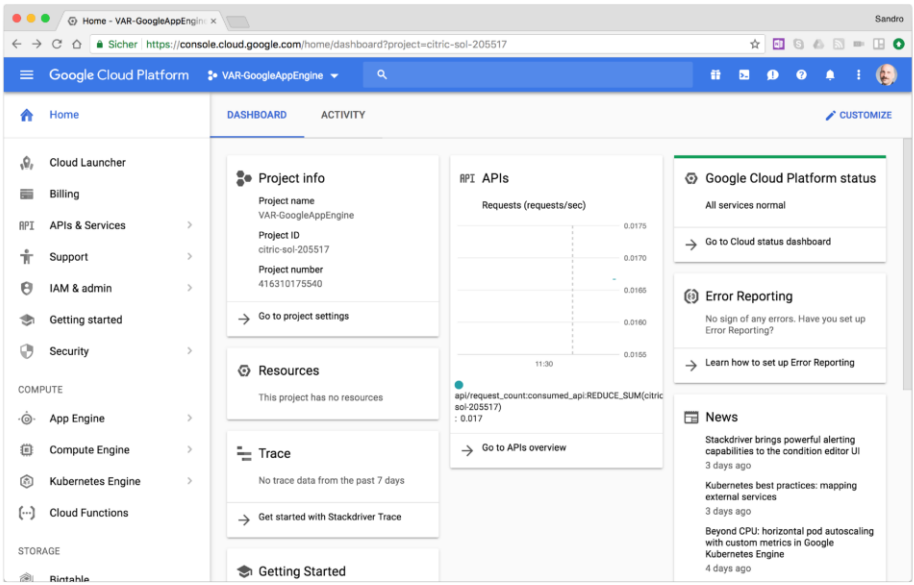


Abbildung 9.10: Screenshot des Dashboards des neu angelegten serverseitigen Projekts, das noch nicht deployed wurde

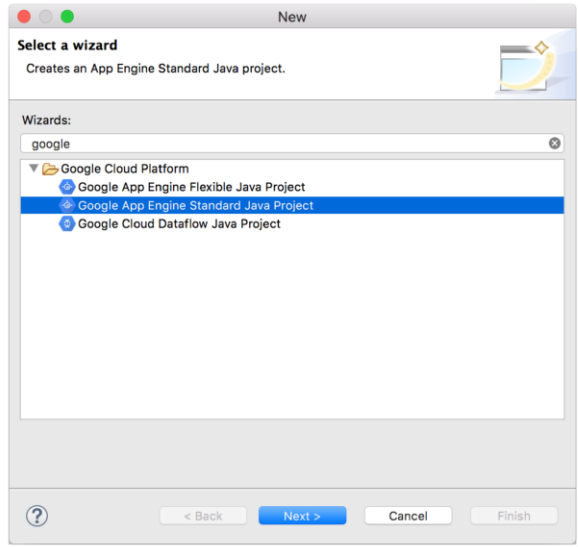


Abbildung 9.11: Anlegen eines *Google App Engine Standard Java Project* in Eclipse

Aktivitätsschritt 9.6:

Ziel: Web-Anwendung von Tomcat nach *Google App Engine* portieren

Befüllen Sie Ihr Projekt mit den Inhalten eines existierenden *Dynamic Web Projects*. Als Beispiel können Sie das Projekt `VAR-Servlets_Wahl` verwenden, das Sie unter `http://verteiltearchitekturen.de/vol01/VAR-Servlets-Wahl.zip` herunterladen können.

Die Java-Klassen (inkl. Servlets) kommen in das Verzeichnis `src/main/java/` Ihres neu angelegten *Google App Engine Standard Java Project*. Sollten Ihre Klassen Packages verwenden, müssen Sie die Package-Struktur durch Verzeichnisse unterhalb von `src/main/java` abbilden. Statische Dateien (z. B. HTML-Dateien oder Grafiken) kommen in das Verzeichnis `src/main/webapp/`. Sollte der Pfad zu einer solchen statischen Datei Verzeichnisse enthalten, müssen diese Verzeichnisse als Hierarchie unterhalb von `src/main/webapp/` angelegt werden.

Der *Deployment Descriptor* ist die Datei `src/main/webapp/WEB-INF/web.xml` (s. Listing 9.1). Darin wird ein sogenanntes *Welcome File* angegeben. Das ist ein Dateiname (ohne Pfad), der optional bei Anfrage-URLs weggelassen werden kann:

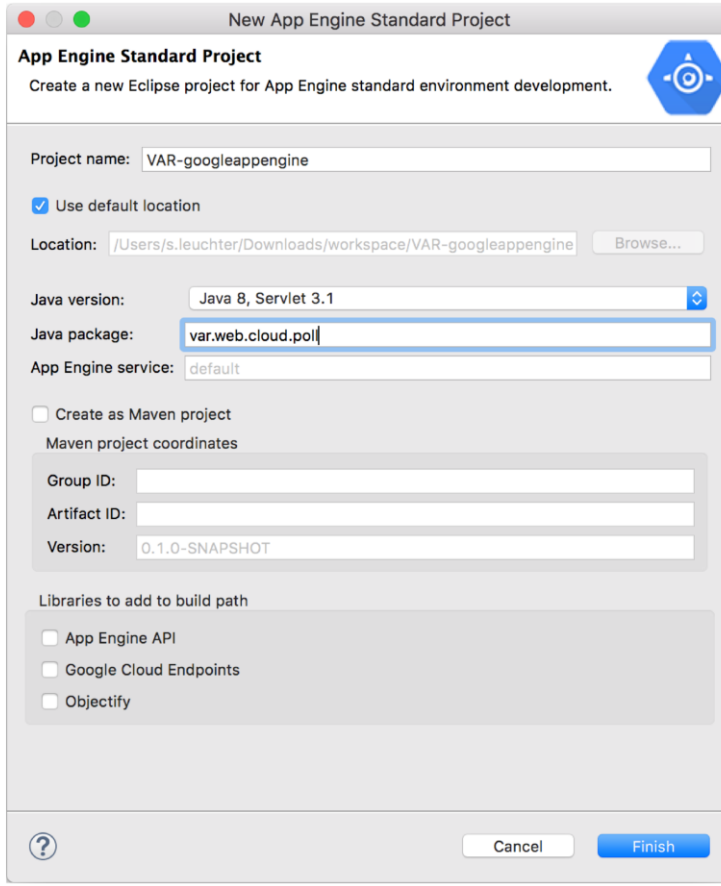


Abbildung 9.12: Konfiguration des *Google App Engine Standard Java Project* in Eclipse

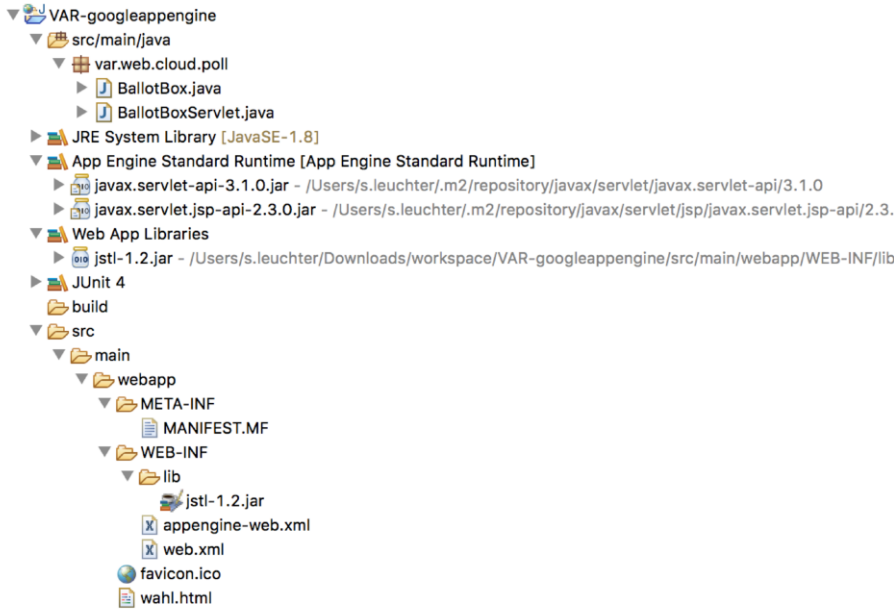


Abbildung 9.13: Screenshot vom Projektbaum des *Google App Engine Standard Java Project* in Eclipse

Wird ein unvollständiger URL abgerufen, wird versucht durch Anhängen des *Welcome Files* einen existierenden URL zu generieren.

Beispiel Zur Anfrage...

`http://klassensprecherwahlen.de/`

... wird keine Ressource gefunden. Stattdessen wird...

Listing 9.1: *Deployment Descriptor* der Beispielanwendung in der Datei `src/main/webapp/WEB-INF/web.xml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="
      http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://
      xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web
      -app_3_1.xsd" version="3.1">
3   <welcome-file-list>
4     <welcome-file>wahl.html</welcome-file>
5   </welcome-file-list>
6 </web-app>
```

`http://klassensprecherwahlen.de/wahl.html`
... zurückgeliefert, das gefunden werden konnte.
Man kann mehrere *Welcome Files* angeben, die i. Allg. im Kontext einer ganzen Web-Anwendung gelten. Oft verwendete Namen für *Welcome Files* sind `index.html`, `index.htm` und `default.htm`.

Aktivitätsschritt 9.7 (falls gewünscht):
Ziel: Deployment Descriptor anpassen
Passen Sie den automatisch generierten *Deployment Descriptor* Ihres lokalen *Google App Engine Standard Java Project* in `src/main/webapp/WEB-INF/web.xml` an ein für Ihre Web-Anwendung passendes *Welcome File* an.

Die Datei `src/main/webapp/WEB-INF/appengine-web.xml` in Listing 9.2 dient weiterhin als Konfiguration des Deployments auf der Google App Engine.

Listing 9.2: Google App Engine-Konfiguration der Beispielanwendung in der Datei `src/main/webapp/WEB-INF/appengine-web.xml`

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
3
4   <threadsafe>true</threadsafe>
5   <sessions-enabled>false</sessions-enabled>
6   <runtime>java8</runtime>
7
8 </appengine-web-app>
```

9.6.3 Deployment in lokaler Sandbox

Um die so beschriebene Web-Anwendung während der Programmierung lokal zu testen, kann die Sandbox, die Teil des SDK ist, benutzt werden. Durch die Installation des Plugins in Eclipse gibt es als neuen «Run As»-Menüeintrag «App Engine» (s. Abb. 9.14).
Mit dieser Option ist eine spezielle Art der Eclipse *Run Configuration* unter dem neu angelegten Typ *App Engine Local Server* verbunden. Für das lokale Projekt `VAR-googleappengine` ist in Abb. 9.15 und Abb. 9.16 gezeigt, wie die Parameter der Tab-Sektionen «Server» und «Cloud Platform» belegt sind.

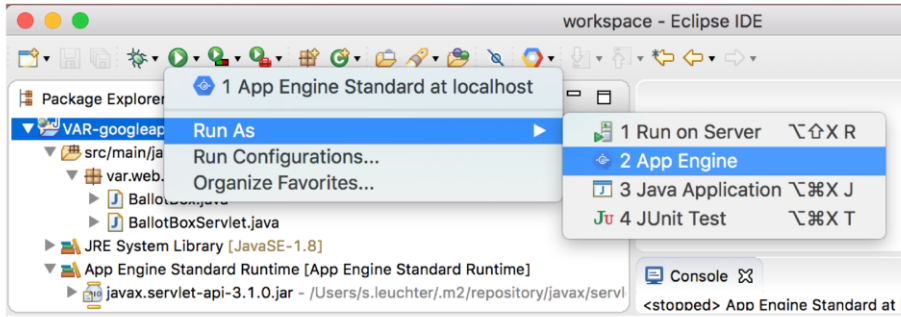


Abbildung 9.14: Starten des Projekts als *App Engine*

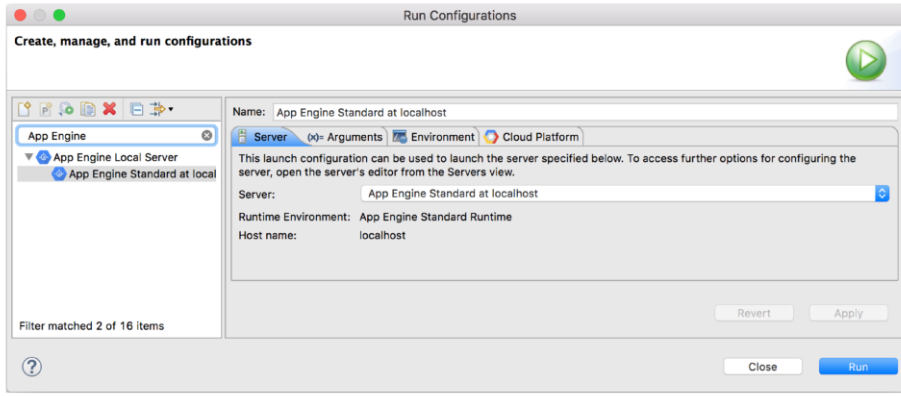


Abbildung 9.15: Tabulator «Server» der Eclipse *Run Configuration* für *App Engine Local Server* in der lokalen Sandbox

Die Umgebung verwendet intern den Jetty Servlet Container und HTTP Server der Eclipse Foundation.

Im Reiter «Server» (s. Abb. 9.16) brauchen keine Angaben für den lokalen Servlet Container Jetty gemacht werden. Der Kontextpfad ist in der Standardkonfiguration «/», so dass die Beispielanwendung in der lokalen Sandbox über den folgenden URL erreichbar ist: `http://localhost:8080/`

Nach dem Start dieser *Run Configuration* kann das Projekt unter `http://localhost:8080/` erreicht und ausprobiert werden. Auf ein Servlet mit URL Mapping `/test` könnten Sie demnach unter `http://localhost:8080/test` zugreifen und auf die Datei `wahl.html` im Verzeichnis `src/main/webapp/` über `http://localhost:8080/wahl.html`. Sollte `wahl.html` im *Deploy-*

ment Descriptor als *Welcome File* definiert worden sein, dann können Sie auch über `http://localhost:8080/` darauf zugreifen. Der Kontextpfad ist also standardmäßig «/».

Aktivitätsschritt 9.8:
Ziel: lokaler Start und Test der Web-Anwendung
Starten Sie Ihr Projekt über *Run as...* → *App Engine* (s. Abb. 9.14) und testen Sie sie mit einem Webbrowser unter `http://localhost:8080/`, bis sie fehlerfrei arbeitet.

9.6.4 Projekt in Google App Engine deployen

Wenn die Anwendung fertig entwickelt ist, muss sie in den Betrieb überführt werden, also öffentlich verfügbar gemacht werden. Dazu wird sie in der Google App Engine *deployt*. Abb. 9.17 zeigt, wie die Funktion zum serverseitigen Deployment in Eclipse erreichbar ist. Voraussetzung für das serverseitige Deployment ist, dass die serverseitige Projekt-ID angelegt wurde (s. Abb. 9.5).

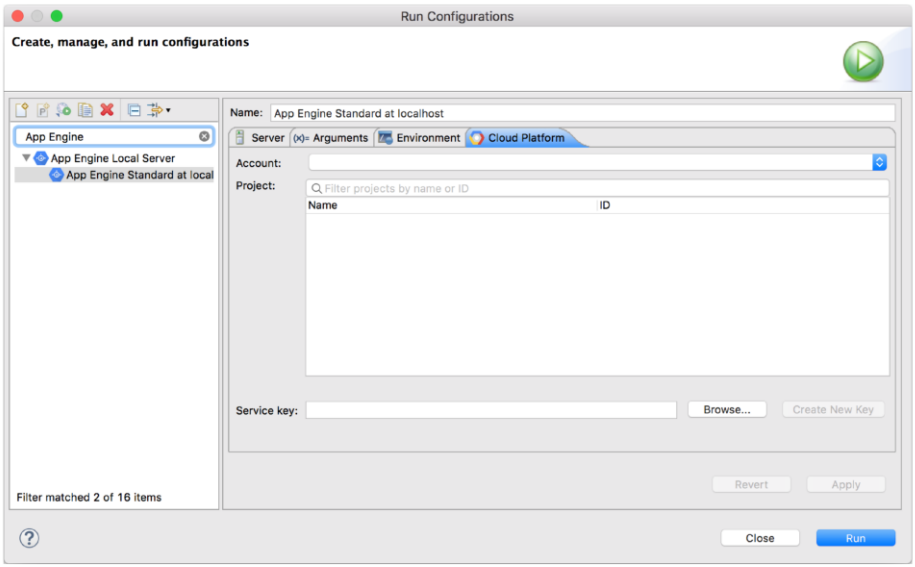


Abbildung 9.16: Tabulator «Cloud Platform» der Eclipse *Run Configuration* für *App Engine Local Server* in der lokalen Sandbox

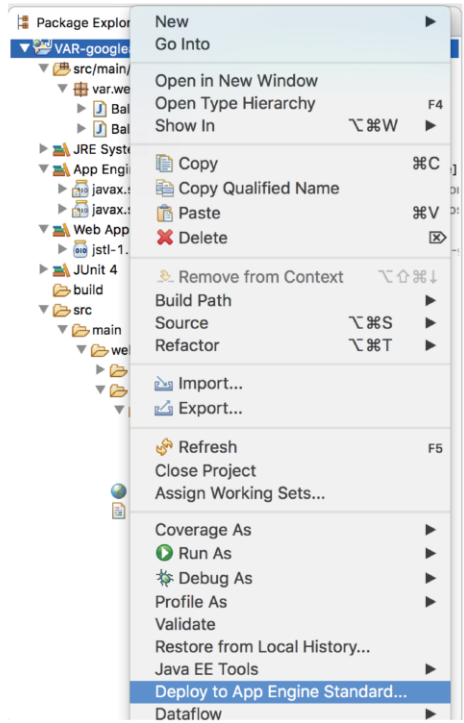


Abbildung 9.17: Deployment der Web-Anwendung des lokalen Projekts im serverseitigen Projekt in der Google App Engine

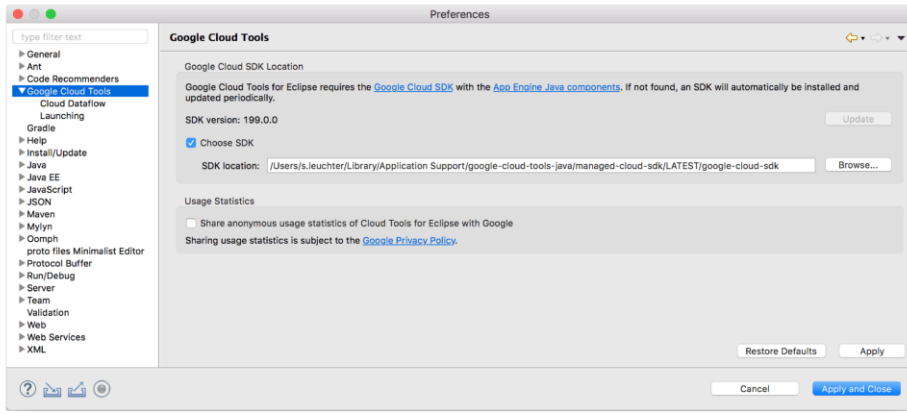


Abbildung 9.18: Dialog zur Konfiguration der *Google Cloud Tools* in den Eclipse-Einstellungen

Aktivitätsschritt 9.9:

Ziel: Google Cloud Tools konfigurieren und Eclipse bei Google anmelden

Voraussetzung für den Deployment-Vorgang ist, dass die *Google Cloud Tools* korrekt in Eclipse eingebunden sind. Dazu muss in den Einstellungen von Eclipse wie in Abb. 9.18 der Dateipfad zur lokalen Installation des *Google Cloud SDK* angegeben werden.

Sie müssen sich nun auch aus Eclipse heraus bei Google anmelden. Entweder geschieht dies vor dem Deployment über die Funktion *Sign in to Google...* aus dem Google Cloud Platform-Menü (s. Abb. 9.20) oder während des Deployment-Vorgangs. In dem Dialogfenster, das nach «Deploy to App Engine Standard ...» erscheint, muss dazu der Google-Account ausgewählt werden, unter dem das serverseitige Projekt angelegt wurde. In Abb. 9.19 ist noch kein Account ausgewählt. Wird hier «Sign into another account» ausgewählt, öffnet sich in einem Browser-Fenster eine Abfrage nach dem Google Account.

Falls Sie mehrere Google Accounts haben wie im Fall in Abb. 9.21, müssen Sie denjenigen auswählen, unter dem das serverseitige Projekt angelegt wurde. Daraufhin erscheint eine Berechtigungsabfrage für die *Cloud Tools for Eclipse* wie in Abb. 9.22. Am Ende des Login-Vorgangs sollte im Browser eine Benachrichtigung über die erfolgte Autorisierung wie in Abb. 9.23 angezeigt werden.

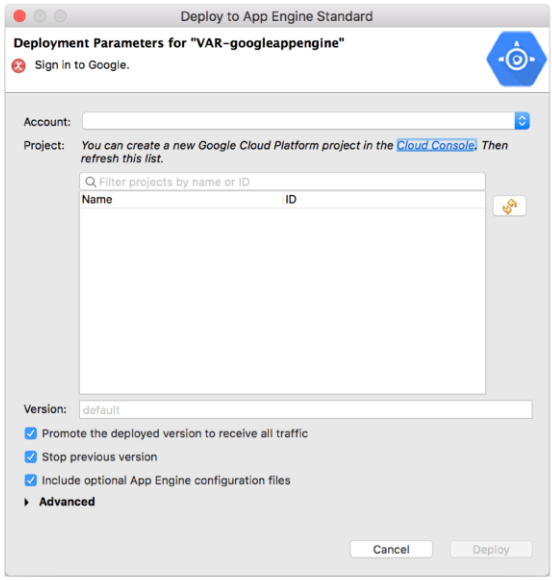


Abbildung 9.19: Dialog zum Deployment des Eclipse Projekts «`VAR-google appengine`» in der Google App Engine (Eclipse ist noch nicht in einen Google Account eingeloggt)

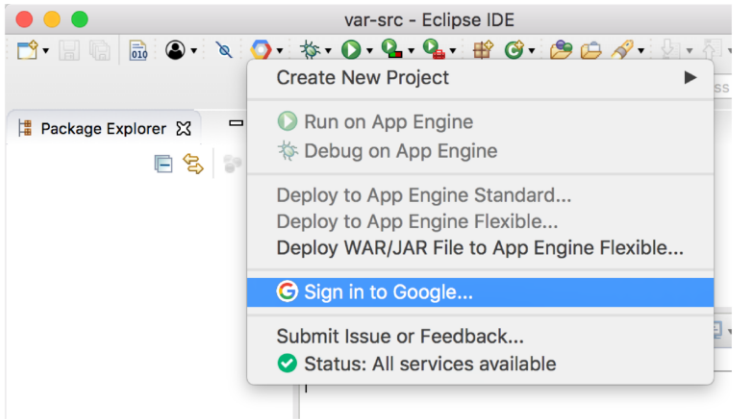


Abbildung 9.20: Aufruf der Funktion zum Einloggen von Eclipse in einen Google-Account

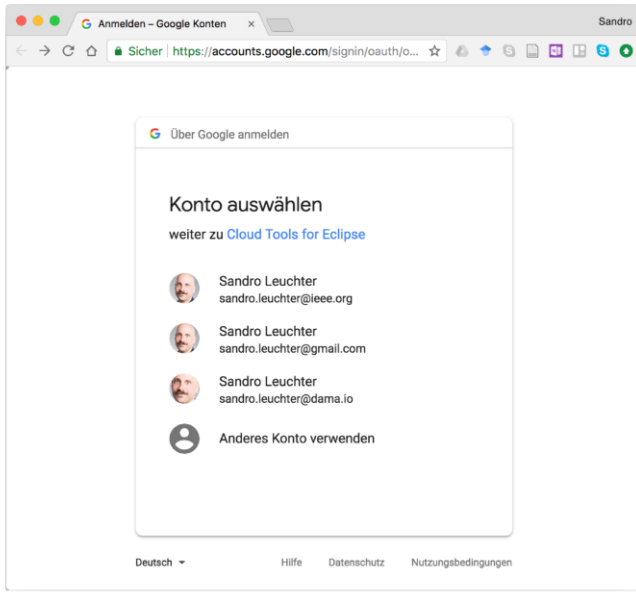


Abbildung 9.21: Dialog zur Auswahl des Google-Accounts für die Cloud Tools for Eclipse

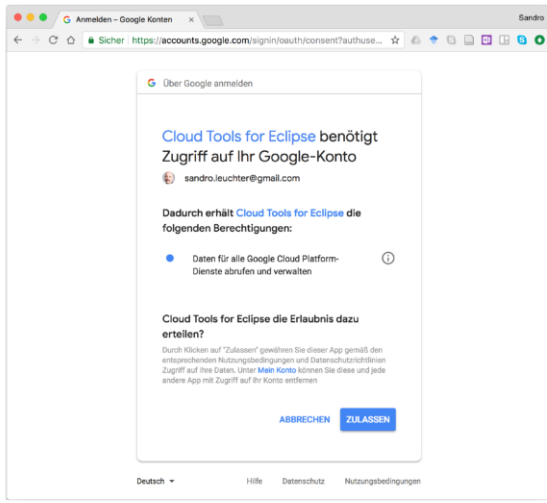


Abbildung 9.22: Berechtigungsabfrage für die Cloud Tools for Eclipse

Aktivitätsschritt 9.10:

Ziel: Projekt in Google App Engine deployen

Deployen Sie dann Ihre Anwendung über «Deploy to App Engine Standard ...» wie in Abb. 9.17. Sie müssen dabei das serverseitige Projekt auswählen, in das die Web-Anwendung «hinein-deployt» werden soll. Im Beispiel in Abb. 9.24 ist dies das serverseitige Projekt «`VAR-GoogleAppEngine`».

Im Regelfall muss vorher noch im serverseitigen Projekt eine «Google App Engine Application» angelegt werden. In Abb. 9.24 ist das beim servserseitigen Projekt «`VAR-GoogleAppEngine`» der Fall: Man sieht dort die Meldung *This project does not have an App Engine application which is required for deployment*. Die erforderliche «Google App Engine Application» im serverseitigen Projekt «`VAR-GoogleAppEngine`» kann einfach über den dort angegebenen Link erzeugt werden.

Dabei kann auch der Ort des Rechenzentrums gewählt werden, in dem die Anwendung gehostet werden soll. Sie sollten hier wie in Abb. 9.25 vorzugsweise die «App Engine location» auf `europe-west` stellen.

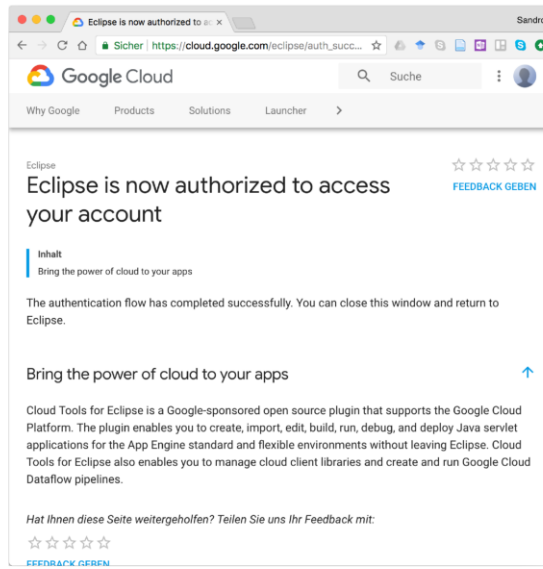


Abbildung 9.23: Mitteilung über die erfolgreiche Autorisierung für die Cloud Tools for Eclipse

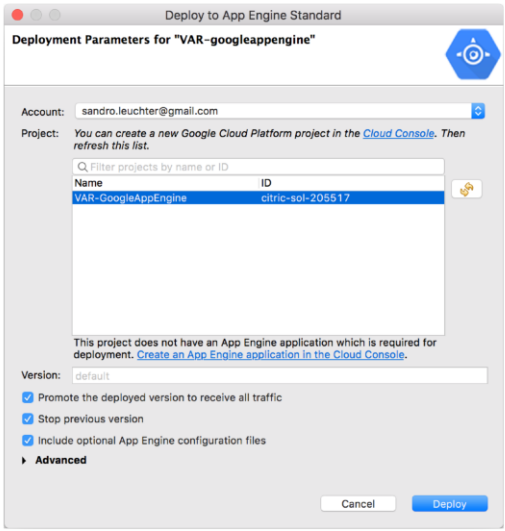


Abbildung 9.24: Dialog zum Deployment des Eclipse-Projekts «`VAR-google appengine`» in der Google App Engine (im Gegensatz zu Abb. 9.19 ist Eclipse nun eingeloggt und die Auswahl eines serverseitigen Projekts ist möglich)

> Testen Sie, ob Sie auf Ihre Anwendung unter dem erwarteten URL zugreifen können.

Nachdem das Deployment erfolgt ist, ist die Anwendung öffentlich erreichbar. Der URL entspricht dem serverseitig angelegten Projektnamen, bzw. dessen Project-ID (`app-id`).

`http://app-id.appspot.com/`

Die Beispielanwendung ist somit unter `http://citric-sol-205517.appspot.com/` erreichbar (s. Abb. 9.26).

Sollten Sie Probleme haben, den richtigen URL zu Ihrer deployten Anwendung zu finden, können Sie auch über den «Serving»-Link in der Statuszeile Ihrer Anwendung in der «Versions»-Übersicht des Dashboards zugreifen (s. Abb. 9.27). Sollte dort ein anderer Status angezeigt werden, hat das Deployment nicht einwandfrei funktioniert.

Im App Engine Dashboard der Google Cloud Platform (s. Abb. 9.27) können noch Parameter angepasst und überwacht werden, die die Art des Hostings der deployten Anwendung beeinflussen.

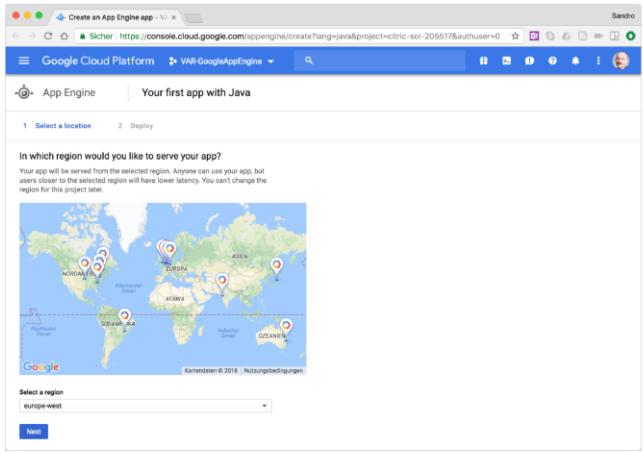


Abbildung 9.25: Dialog zur Auswahl des Ortes des Rechenzentrums, in dem das Deployment des Eclipse-Projekts «`VAR-googleappengine`» stattfinden soll

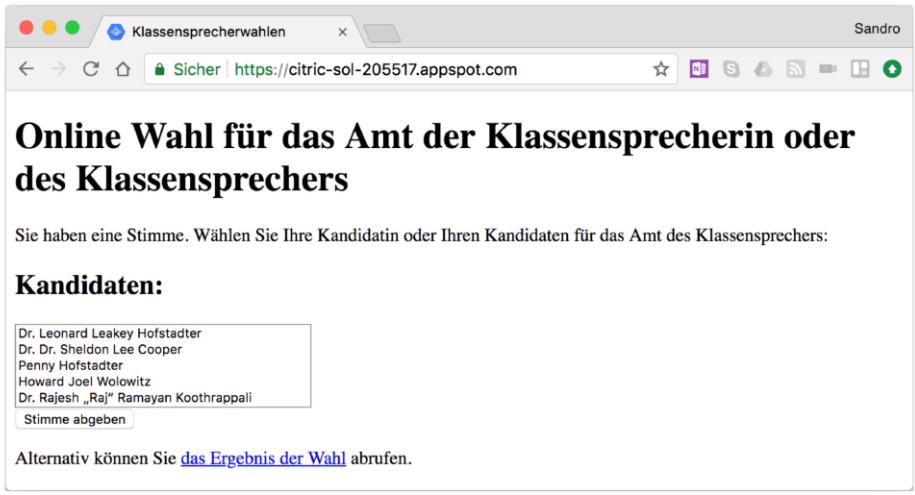


Abbildung 9.26: Aufruf der deployten Anwendung

Exkurs: Parallele Sandboxes: Versionierung in Google App Engine

Die Anwendung kann parallel in unterschiedlichen Versionen deployt werden (die Versionsnummer kann im Properties-Dialog des lokalen Projektes oder in der Web-Oberfläche des Dashboards gesetzt werden). Die parallelen Versionen können durch *Traffic Splitting* unterschiedlichen Nutzern zugänglich gemacht werden. So kann eine neue Version im Beta Test beispielsweise nur einem kleinen Prozentsatz der Nutzer zugänglich gemacht werden, um den Regelbetrieb nur gering zu beeinflussen.
Das Benennungsschema der *Host*-Namen enthält die Versionsnummer an der Stelle `version`:
`http://version-dot-app-id.appspot.com/`, also z. B.
`http://1-dot-citric-sol-205517.appspot.com/`

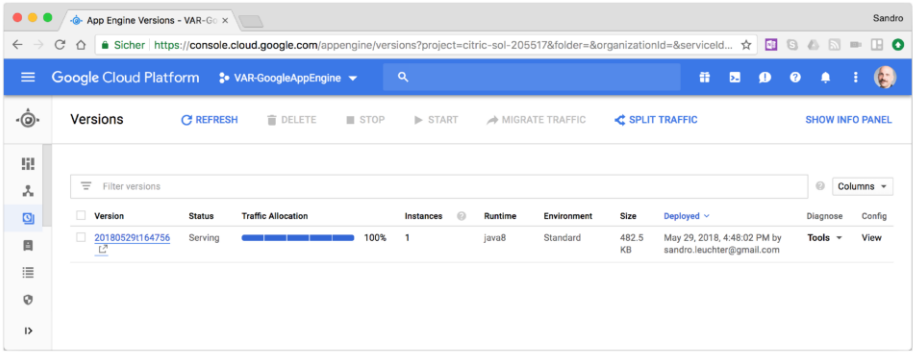


Abbildung 9.27: Sektion «Versions» im Dashboard (Konfiguration der Versionen der deployten Anwendung)

Exkurs: Netzwerkverkehr auf Versionen aufteilen

Wenn es mehrere parallel gehostete Versionen der Anwendung gibt (s. Exkurs «parallele Sandboxes: Versionierung in Google App Engine» auf S. 301), dann kann unter der Sektion *Versions* im Dashboard (s. Abb. 9.27) verwaltet werden, wie der Netzwerkverkehr auf die Versionen aufgeteilt wird («*Split Traffic*») und ob unterschiedliche Leistungsparameter genutzt werden sollen.

Exkurs: Quotas

Unter der Sektion *Quotas* kann überwacht werden, in welchem Maß die Quota – also die Limits unterschiedlicher zur Verfügung gestellter Ressourcen – von der deployten Anwendung ausgeschöpft wurde (s. Abb. 9.28).

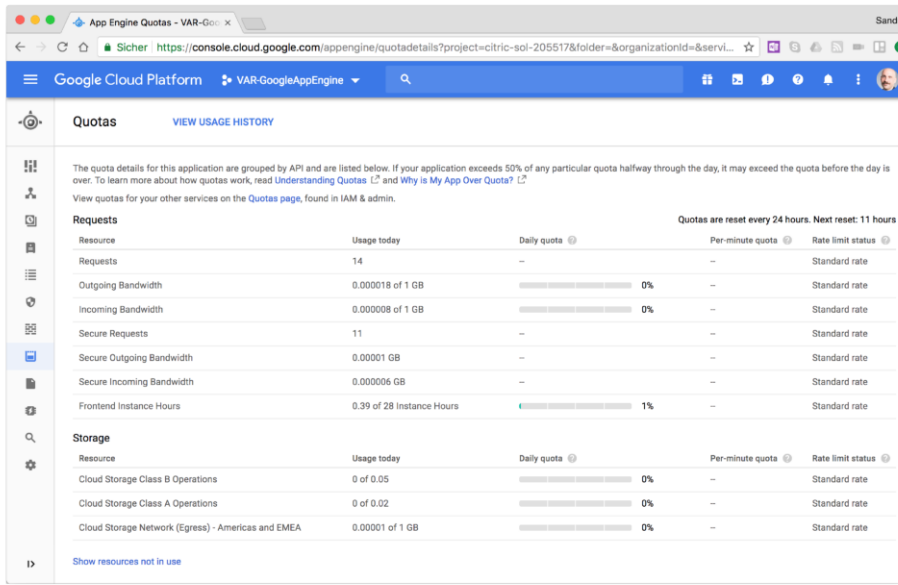


Abbildung 9.28: Sektion «Quotas» im Dashboard (Überwachung der Ressourcennutzung)

9.6.5 Erreichbarkeit unter eigenem Domain-Namen

Im letzten Schritt soll die deployte Anwendung unter einem eigenen Domain-Namen erreichbar gemacht werden. Das ist prinzipiell ein optionaler Schritt, denn die Anwendung ist schließlich bereits unter der `http://appspot.com/`-Domain öffentlich erreichbar.
Als erstes muss dafür ein eigener Domain-Name registriert werden. Das geht leider nicht kostenfrei. Es gibt zahlreiche Anbieter, die das übernehmen.
Es geht dabei aber nur darum, einige Einträge im DNS vorzunehmen. Ressourcen wie Speicherplatz, ein HTTP Server bzw. Servlet Container oder Übertragungsbandbreite brauchen nicht gekauft werden, sondern kommen in der hier verwendeten Ausbaustufe kostenfrei von der Google App Engine.

Domain-Registrierung

In diesem Beispiel wird ein Angebot von Strato zur Registrierung einer `.de`-Domain für 2,40 EUR im ersten Jahr (danach 14,40 EUR) benutzt. Dabei ist auch E-Mail enthalten (wird in diesem Praktikum nicht gebraucht). In den folgenden Abb. 9.29 und 9.30 wird beispielhaft gezeigt, wie die Domainregistrierung bei Strato erfolgt. Es gibt einen Identifikationsschritt, der z. B. ohne langen Verzug über eine SMS an eine angegebene Mobilfunknummer erfolgen kann.

Aktivitätsschritt 9.11:
Ziel: Registrieren Sie einen Domain-Namen
Wählen Sie einen Internet Service Provider wie Strato[a], HostEurope[b] oder GoDaddy[c]. Folgen Sie den jeweiligen Schritten, um eine Domain zu registrieren.
Sie brauchen nur den Domain-Namen. Web-Hosting, E-Mail, WordPress oder Website-Baukästen sind nicht erforderlich.
Im Folgenden wird mit dem Beispiel `klassensprecherwahlen.de` weitergearbeitet. Diese Domain ist nun bereits belegt, so dass Sie einen eigenen Domain-Namen wählen müssen. Für die Wahl der TLD gibt es keine Einschränkungen.

[a] `https://www.strato.de/`
[b] `https://www.hosteurope.de/`
[c] `https://de.godaddy.com/`

Abbildung 9.29: Screenshot während der Suche nach einem Domain-Namen

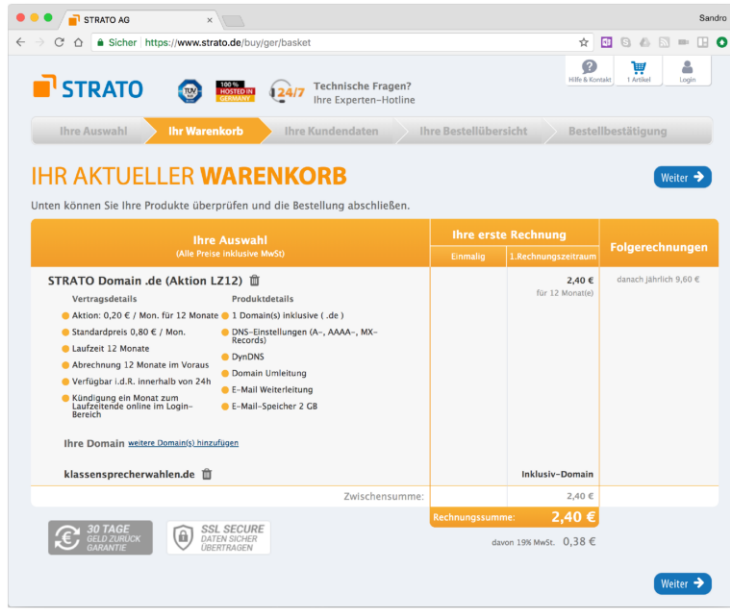


Abbildung 9.30: Beim Bestellen im Webshop: Domain muss noch hinzugefügt werden

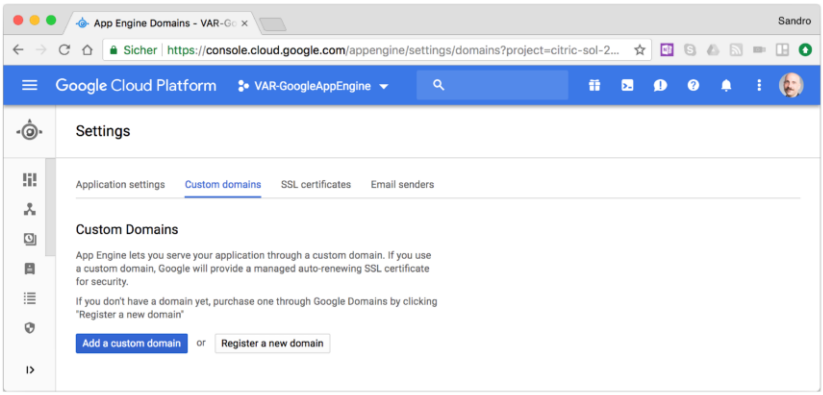


Abbildung 9.31: Start des Registrierungsprozesses für eine eigene Domäne

Konnektierung der eigenen Domain mit der in Google App Engine deployten Web-Anwendung

Aktivitätsschritt 9.12:

Ziel: Registrierung als Domain-Verantwortlicher bei Google

Wenn die Domain im DNS verfügbar («*konnektiert*») ist, können Sie sich bei Google als dafür verantwortlich registrieren lassen. Sie müssen dafür allerdings nachweisen, dass Sie in Besitz der Kontrolle über die Domain sind. Dazu wird ein an sich funktionsloser Eintrag im DNS vorgenommen, über den das Google-System ermittelt, dass man tatsächlich DNS-Einträge für diese Domain einfügen bzw. verändern darf, also der «de facto»-Eigentümer der Domain ist. Die Registrierung ist nicht nur für die Google Cloud Platform von Belang, sondern ermöglicht es auch eine Web-Präsenz zur Indizierung für die Google-Suchmaschine anzumelden.

1. Im ersten Schritt muss dazu im Google Cloud Platform Dashboard der App Engine unter der Sektion «Settings» beim Tabulator-Reiter «Custom Domains» *Add a custom domain* gewählt werden (s. Abb. 9.31).

2. Im nächsten Schritt muss der Domain-Name angegeben werden, auf den die Anwendung registriert werden soll (s. Abb. 9.32).

3. Dann wird man zur Webmaster-Zentrale weitergeleitet, wo der Besitz bestätigt werden muss (s. Abb. 9.33).

4. Nachdem der Domain-Registrar (in diesem Beispiel «Sonstiges» als möglichst generischen Fall) ausgewählt wurde, wird eine zufällig generierte Zeichenfolge angezeigt, die als zusätzlicher DNS-Eintrag für die eigene Domain registriert werden muss. Je nach Registrar kann die Methode abweichen, ist aber prinzipiell ähnlich (s. Abb. 9.34).

5. Beim Verwalter der eigenen Domäne (im Beispiel `http://strato.de/`) muss nun der DNS-Eintrag vorgenommen werden. Dazu muss man sich zunächst dort einloggen (s. Abb. 9.35).

6. Über den Link «verwalten» hinter der eigenen Domäne gelangt man zu den Funktionen zur Änderung der DNS-Einträge (s. Abb. 9.36 und 9.37).

7. In der DNS-Verwaltung gibt es mehrere Optionen, von denen in diesem Schritt die Verwaltung des TXT-Records ausgewählt werden muss (s. Abb. 9.37).

8. In der dann erscheinenden Ansicht wird der Eintrag aktiviert und der in der Google Webmaster-Zentrale angezeigte Text (s. Abb. 9.34) als benutzerdefinierter Datensatz eingetragen (s. Abb. 9.38).

9. Als Ergebnis wird in der Domänenübersicht von `http://strato.de/` der neue Eintrag angezeigt (s. Abb. 9.39). Mit `nslookup` oder `dig` kann man sich davon überzeugen, dass der TXT-Eintrag tatsächlich dem eigenen DNS-Eintrag hinzugefügt wurde (s. Abb. 9.40).

10. Nun kann wieder zurück in die Webmaster-Zentrale von Google gewechselt werden. Nach einiger Zeit wird die DNS-Änderung dort registriert worden sein (s. Abb. 9.41). Das kann einige Zeit dauern, da Änderungen im DNS verteilt werden müssen.

Aktivitätsschritt 9.13:
Ziel: Konnektierung der eigenen Domäne mit gehosteter Web-Anwendung
Nachdem die Inhaberschaft der eigenen Domäne erfolgreich nachgewiesen wurde, kann die eigene Domäne in der «Custom Domains»-Ansicht der «Settings» Sektion des Google App Engine Dashboards ausgewählt werden (s. Abb. 9.42).

Nun werden DNS-Einträge angezeigt, die für die eigene Domäne registriert werden sollen, damit der eigene Domänenname auf die App Engine gehostete Web-Anwendung aufgelöst wird. In der Abb. 9.43 werden jeweils vier Einträge für IPv4- und IPv6-Adressen angezeigt.

Wenn dieser Schritt abgeschlossen ist, werden auf der Google App Engine Einstellungsseite des Projekts «`VAR-GoogleAppEngine`» wie in Abb. 9.44 alle konnektierten Domain-Namen angezeigt.

Aktivitätsschritt 9.14:
Ziel: Ändern der DNS-Einträge der eigenen Domäne
In der Strato-Domänenverwaltung müssen Sie wieder zur DNS-Verwaltung wechseln (s. Abb. 9.37). Diesmal muss die Funktion für die Verwaltung der A («Address») Einträge (analog AAAA für IPv6-Adressen) gewählt werden. Bei Strato kann nur ein Eintrag vorgenommen werden. Von den vier IPv4-Adressen von Google (s. Abb. 9.43) wird einer ausgewählt und als Eintrag übernommen (s. Abb. 9.45).

Der neue Eintrag wird wieder durch das verteilte DNS propagiert und nach einer gewissen Zeit ist der eigene Domänenname mit der in der Google App Engine gehosteten

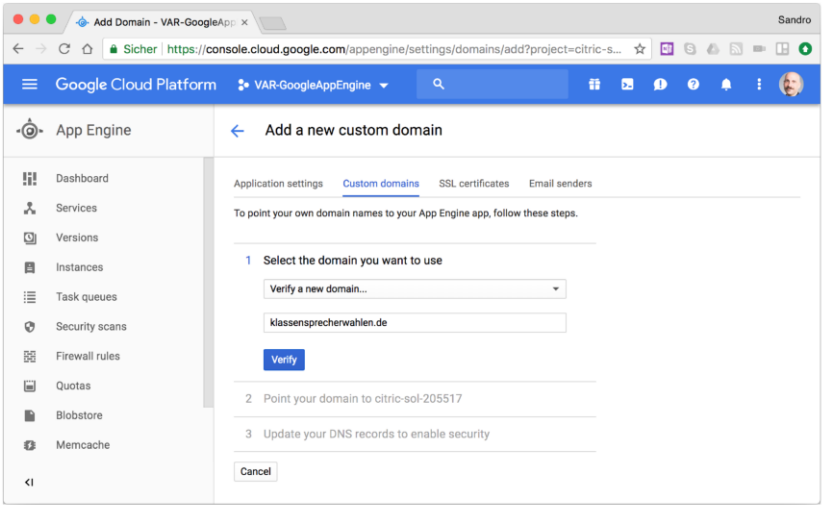


Abbildung 9.32: Domänennamen angeben, auf den die Anwendung registriert werden soll

Anwendung verbunden (s. Abb. 9.46). Je nach lokalem DNS kann es sein, dass die eigene Domäne auf dem einen Rechner schon konnektiert ist und auf einem anderen noch auf die Strato Default-Seite verweist.

9.6.6 Ausblick: Zustandslosigkeit und Instanzen der Web-Anwendung

Die Beispielanwendung zur Klassensprecherwahl ist zustandsbehaftetet, denn das aktuelle Wahlergebnis (die abgegebenen Stimmen) wird in einer Variablen, also im RAM, gespeichert. Nach jeder Stimmabgabe ist das Resultat von `http://klassensprecherwahlen.de/vote?action=print` anders. Dies widerspricht der Anforderung von S. 278, dass Google App Engine Applikationen keinen Zustand haben dürfen. Sollten sehr viele HTTP Requests in kurzer Zeit eintreffen, werden nämlich weitere Instanzen der Web-Anwendung gestartet um die erhöhte Rechenlast abzufangen. Jede Instanz hätte dann ihren eigenen Zustand, so dass unterschiedliche Nutzer möglicherweise verschiedene Wahlergebnisse sehen würden. Wird hingegen eine Instanz länger nicht genutzt (treffen also keine HTTP Requests ein), wird die Instanz beendet. Es kann durchaus sein, dass zu einem Zeitpunkt gar keine Instanz der Web-Anwendung in der Google App Engine aktiv läuft. Erst wenn ein neuer

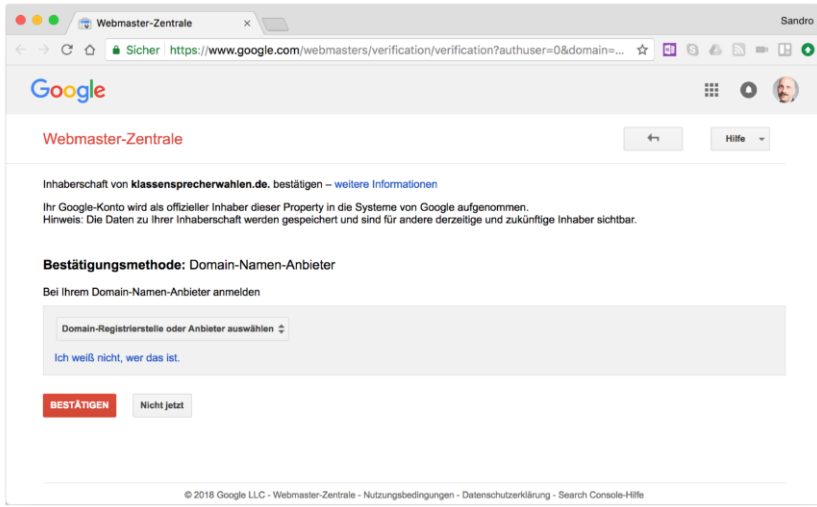


Abbildung 9.33: Beginn der Prozedur zur Bestätigung der Inhaberschaft der Domäne in der Webmaster-Zentrale

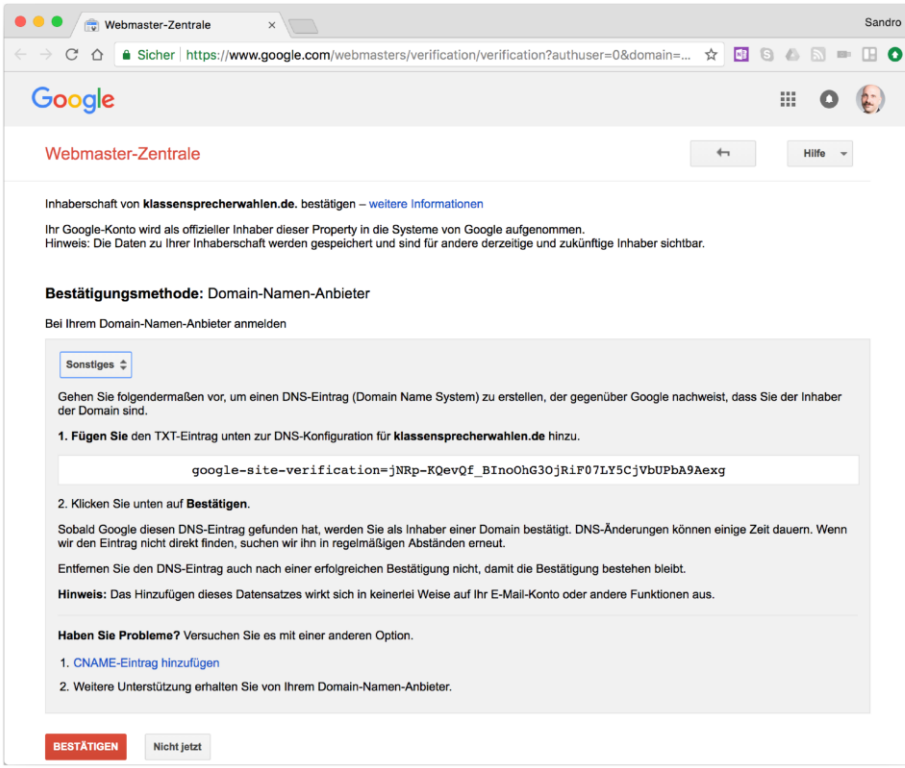


Abbildung 9.34: Anzeige des TXT-Datensatzes, der zur Inhaberbestätigung für die eigene Domäne im DNS vermerkt werden muss

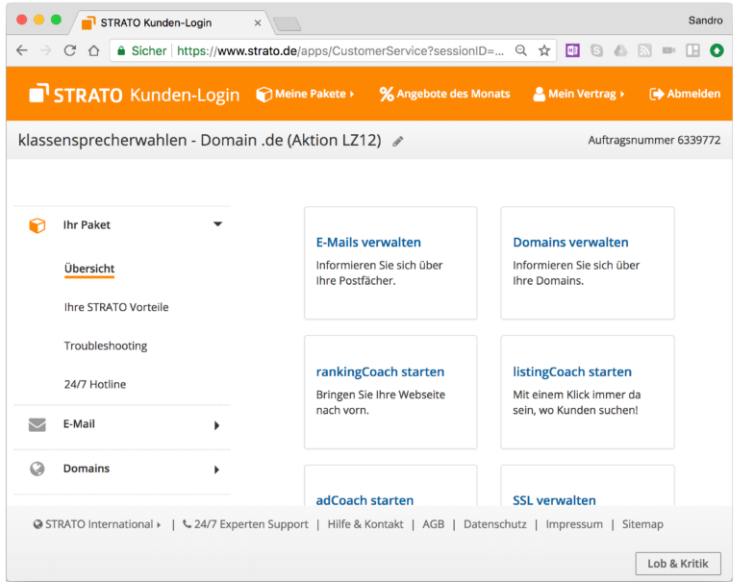


Abbildung 9.35: Login beim Verwalter der eigenen Domäne (hier `http://strato.de/`)

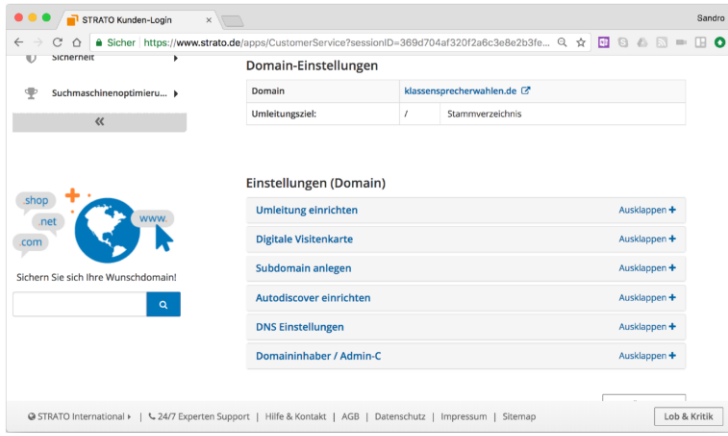


Abbildung 9.36: Überblick über die Domänenverwaltung bei `http://strato.de/`

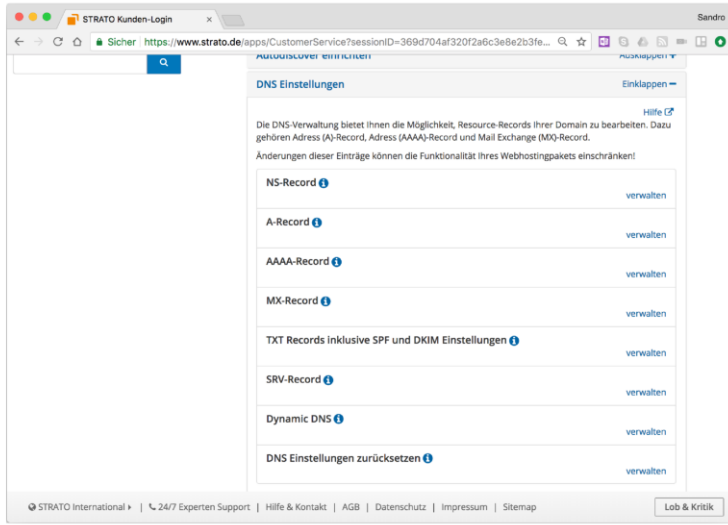


Abbildung 9.37: DNS-Verwaltung in der Domänenverwaltung bei `http://strato.de/`

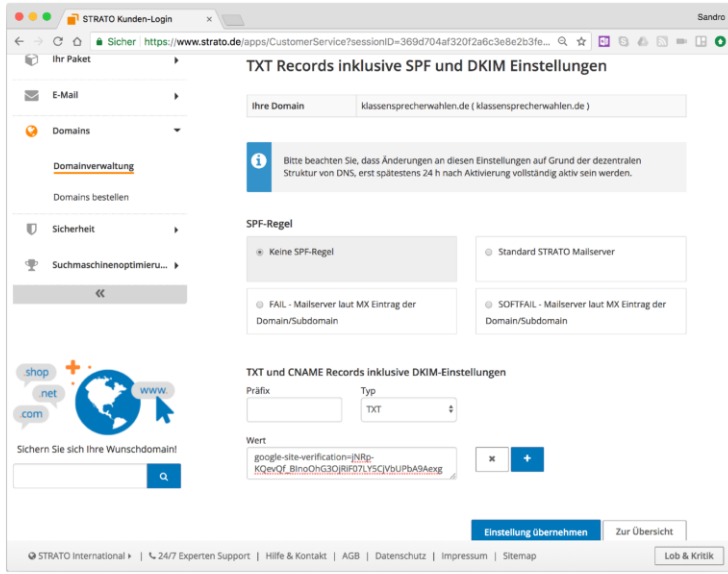


Abbildung 9.38: Domänenverwaltung bei `http://strato.de/`: Eintrag des TXT-Datensatzes ins DNS für die eigene Domäne

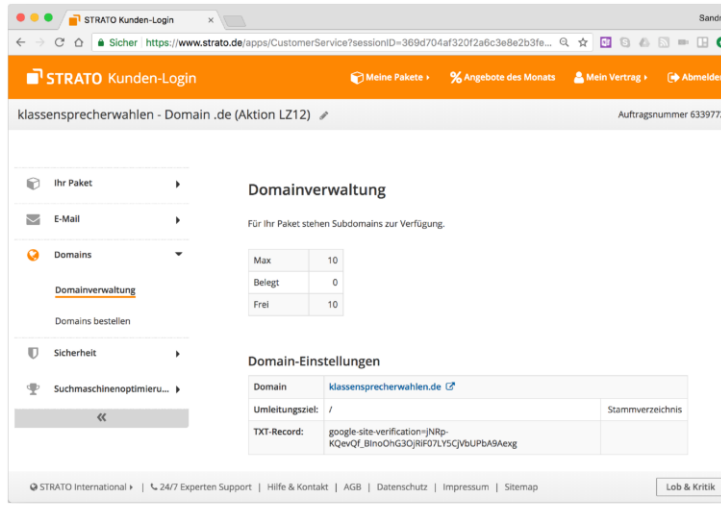


Abbildung 9.39: Anzeige des neuen TXT-Eintrags im DNS für die eigene Domäne bei der Domänenverwaltung von `http://strato.de/`

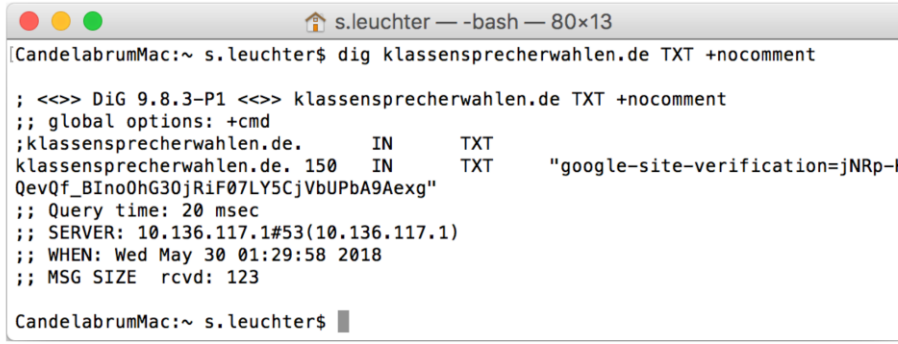


Abbildung 9.40: Anzeige des neuen TXT-Eintrags im DNS für die eigene Domäne mit `dig`

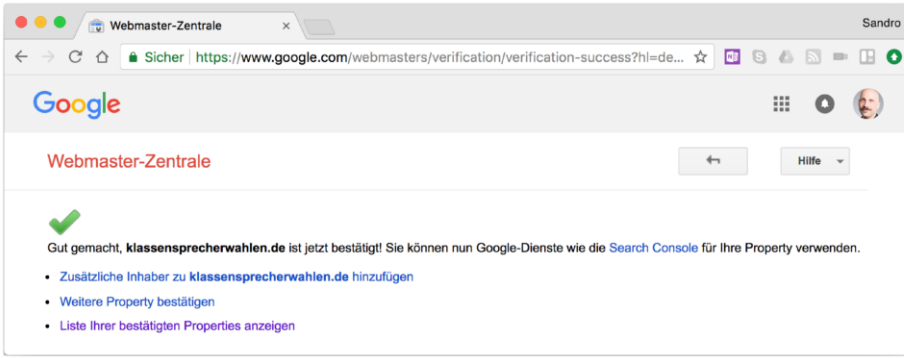


Abbildung 9.41: Feedback in der Google Webmaster-Zentrale, dass die Inhaberschaft der eigenen Domäne bestätigt wurde

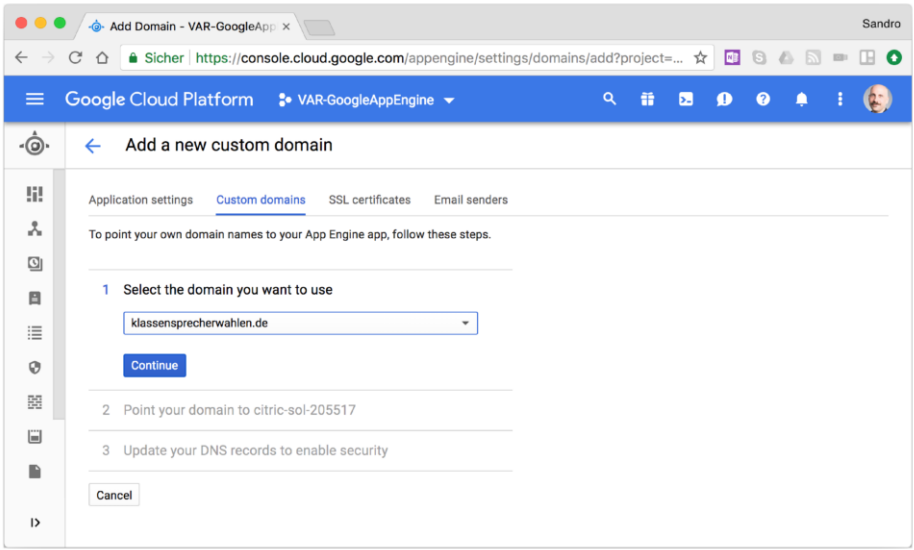


Abbildung 9.42: Auswahl der eigenen Domäne für das App-Engine-Projekt

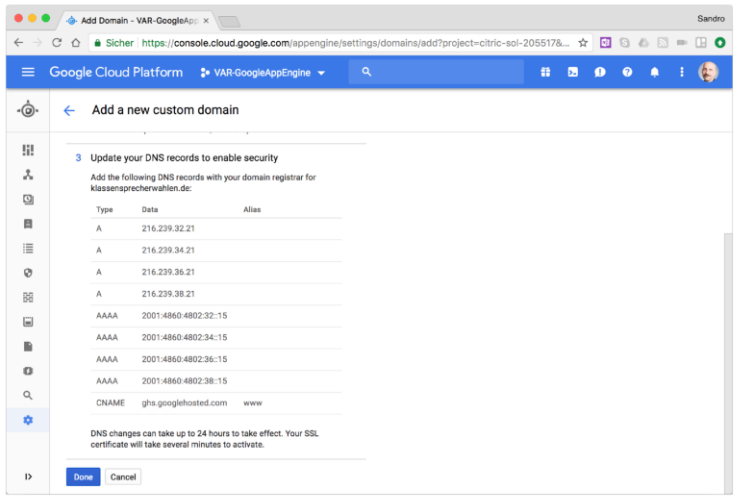


Abbildung 9.43: Anzeige der erforderlichen DNS-Einträge für die Auflösung der eigenen Domäne zu IP-Adressen für die in der Google App Engine gehostete Anwendung

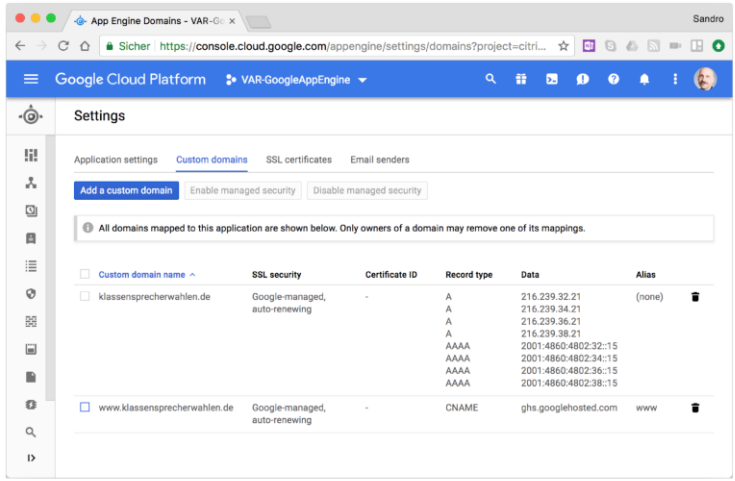


Abbildung 9.44: Abgeschlossene Konfiguration der eigenen Domäne für die in der Google App Engine gehostetete Web-Anwendung

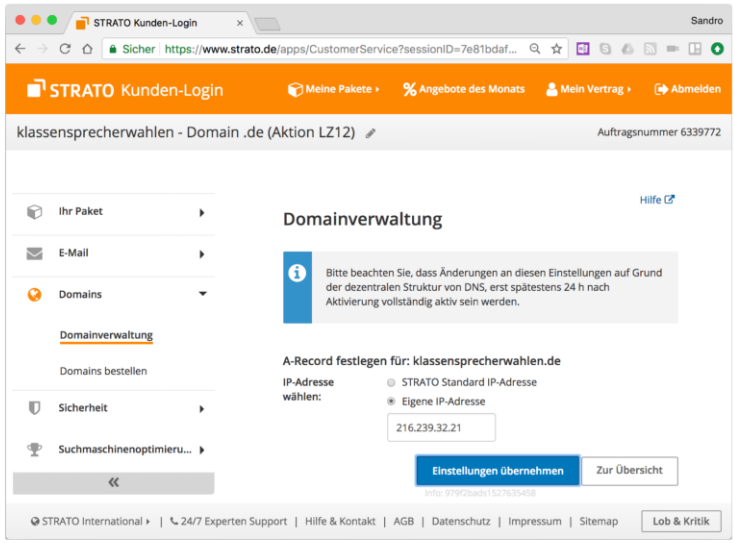


Abbildung 9.45: Eintragen eines DNS A-Record bei `http://strato.de/`

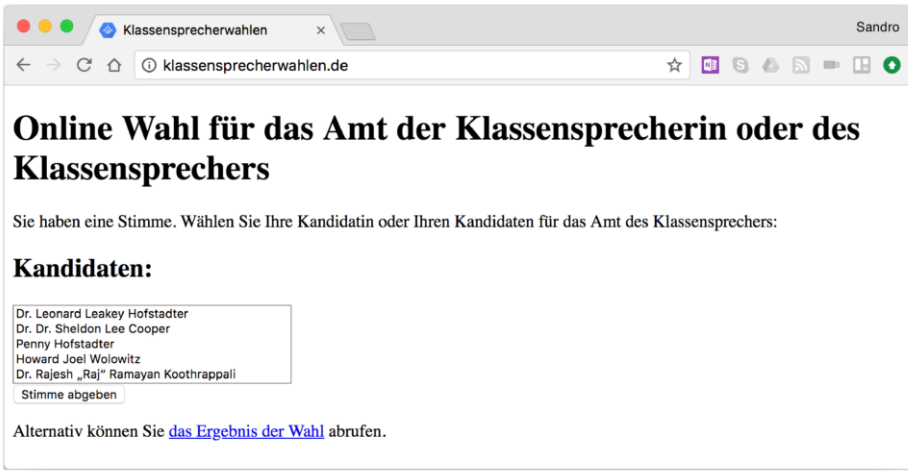


Abbildung 9.46: Die eigene Domäne wird nun so zu IP-Adressen aufgelöst, dass sie mit der in der Google App Engine gehosteten Web-Anwendung verbunden ist.

HTTP Request für diese Anwendung im Google-Rechenzentrum eintrifft, wird dann wieder eine neue Instanz für diese Web-Anwendung gestartet. Das kann beim *standard environment* einige Sekunden dauern (bei Web-Anwendungen, die das *flexible environment* benutzen, dauert es hingegen Minuten, bis eine neue Instanz einsatzbereit ist).

Aktivitätsschritt 9.15 (fakultativ):
Ziel: Analyse des Instanziierungsverhaltens der Google App Engine
Wenn Ihre Anwendung zustandsbehaftet ist, können Sie das Verhalten der Google App Engine in Hinblick auf den automatischen Start und die Beendigung von Instanzen analysieren. Wird Ihre Web-Anwendung bspw. eine Zeit lang nicht benutzt, wird die laufende Instanz beendet. Versuchen Sie danach erneut auf Ihre Web-Anwendung zuzugreifen, wird eine neue Instanz gestartet, deren Zustand anfangs den anwendungsspezifischen Initialwert hat (im Beispiel der Klassensprecherwahl keine abgegebene Stimme). Daran kann der Neustart erkannt werden.
Um zu verhindern, dass ein etwaiger Zugriff von Dritten auf Ihre Anwendung das Ergebnis verfälscht, können Sie «Traffic Splitting» (s. Abb. 9.27) auf getrennte Versionen benutzen.

Tipp: Mit Apache JMeter[a] können Sie Testabläufe für Web-Anwendungen, speziell Lasttests, automatisieren.

[a] `https://jmeter.apache.org/`

Teil IV

Anhang

Index

Abkürzungsverzeichnis

AJP	Apache JServ Protocol
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARPANET	Advanced Research Projects Agency Network
ASCII	American Standard Code for Information Interchange
BaaS	Backend as a Service
BelWue	Baden-Württembergs extended LAN
ccTLD	Country Code Top Level Domain
CGI	Common Gateway Interface
DARPA	Defense Advanced Research Projects Agency
Datagram	Data Telegram
DDNS	Dynamisches DNS

DDoS	Distributed Denial of Service
DDS	Data Distribution Service
DE-CIX	Deutscher Commercial Internet Exchange
DENIC	Deutsches Network Information Center
DFN	Deutsches Forschungsnetz
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DOI	Digital Object Identifier
DSL	Digital Subscriber Line
EAI	Enterprise Application Integration
ECTS	European Credit Transfer System
FaaS	Function as a Service
GWT	Google Web Toolkit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IaaS	Infrastructure as a Service
IANA	Internet Assigned Numbers Authority
ICANN	Internet Corporation for Assigned Names and Numbers
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol

IPv4	Internet Protocol, Version 4
IPv6	Internet Protocol, Version 6
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
IT	Informationstechnik
Java EE	Java Enterprise Edition
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JSF	Java Server Faces
JSP	Java Server Pages
JSR	Java Specification Request
JVM	Java Virtual Machine
LTE	Long Term Evolution
LWT	Last Will and Testament
MAC	Medium Access Control
MIME	Multipurpose Internet Mail Extensions
MOM	Message Oriented Middleware
MQ	Message Queue
MQTT	Message Queue Telemetry Transport
NAT	Network Address Translation
OASIS	Organization for the Advancement of Structured Information Standards

OMG	Object Management Group
oss	Open Source Software
P/S	Publish/Subscribe
PaaS	Platform as a Service
PNG	Portable Network Graphics
PTP	Point To Point
QoS	Quality of Service
R/R	Request/Reply
RFC	Request for Comments
RMI	Remote Method Invocation
RZ	Rechenzentrum
SaaS	Software as a Service
SDK	Software Development Kit
SEO	Search Engine Optimization
SLA	Service Level Agreement
SMS	Short Message Service
SSL	Secure Sockets Layer
SW	Software
TCP	Transmission Control Protocol
TLD	Top Level Domain
TLS	Transport Layer Security
TTL	Time To Live
UDP	User Datagram Protocol

URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VPN	Virtual Private Network
WAR	Web Application Archive
WWW	World Wide Web
ZugErschwG	Gesetz zur Erschwerung des Zugangs zu kinderpornographischen Inhalten in Kommunikationsnetzen

Bildnachweise

Buchumschlag und S. 344, Band 1: Foto von Michael Warren (mikewarren bei Flickr), lizensiert unter CC BY 2.0 (`https://creativecommons.org/licenses/by/2.0/`). Bezug via Flickr (`https://www.flickr.com/photos/57389319@N00/10917370`, letzter Zugriff: 14.10.2018).

S. 3, 47 und 75, jeweils oben: Foto von Martinelle (`https://pixabay.com/en/users/Martinelle-495565/`), lizensiert unter CC0 (`https://creativecommons.org/publicdomain/zero/1.0/`). Bezug via pixabay (`https://pixabay.com/en/network-cable-ethernet-computer-1572617/`, letzter Zugriff: 14.10.2018).

S. 5, Abb. 1.1 Darstellung von Jon Postel. Die Darstellung ist ein Werk eines Mitarbeiters der Streitkräfte der Vereinigten Staaten oder des Verteidigungsministeriums der Vereinigten Staaten, aufgenommen oder hergestellt während seiner offiziellen Anstellung. Als amtliches Werk der Bundesregierung der Vereinigten Staaten ist dieses Bild lt. Wikimedia Commons gemeinfrei (`https://commons.wikimedia.org/wiki/Template:PD-USGov-Military`). Bezug via Wikimedia Commons (`https://commons.wikimedia.org/wiki/File:Internet_map_in_February_82.jpg`, letzter Zugriff: 14.10.2018).

Netzwerkstruktur des Arpanet als Vorläufer des Internet, Stand Februar 1982.

S. 16, Abb. 1.3 Foto aus einem Tweet von Negatif Pollyanna @FindikKahve bei Twitter (`https://twitter.com/FindikKahve/status/446961277739749376`, letzter Zugriff: 14.10.2018).

Information zum Google-DNS `8.8.8.8` an einer Wand im Stadtteil Kadıköy von Istanbul zum Umgehen der Twitter-Sperre 2014 in der Türkei.

S. 117, 123 und 167, jeweils oben: NOAA Photo Library: nssl0026, by OAR/ERL/National Severe Storms Laboratory (NSSL), lizensiert unter CC BY 2.0 (http://creativecommons.org/licenses/by/2.0). Bezug via Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Nssl0026_-_Flickr_-_NOAA_Photo_Library.jpg, letzter Zugriff: 14.10.2018).

S. 203, 237 und 273, jeweils oben: Fotografie von Carol M. Highsmith [reproduction number: LC-DIG-highsm-11604] (lt. http://www.loc.gov/rr/print/res/482_high.html gemeinfrei). Bezug via Library of Congress, Prints & Photographs Division (http://www.loc.gov/pictures/resource/highsm.11604/, letzter Zugriff: 14.10.2018).

Lesesaal der Forschungsbibliothek *Library of Congress* der Vereinigten Staaten von Amerika in Washington, D.C.

S. 204, Abb. 7.1 Aufnahme und Upload auf von Binary Koala, lizensiert unter CC BY SA 2.0 (https://creativecommons.org/licenses/by-sa/2.0/). Bezug via Flickr (http://flickr.com/photos/35632769@N00/16707532898, letzter Zugriff: 14.10.2018).

NeXTcube, auf dem anfangs die Software für das Workd Wide Web von Tim Berners Lee entwickelt wurde.

S. 319, 335 und 341, jeweils oben: Ausschnitt aus einem Foto von Florian Richter («florianric» bei Flickr), lizensiert unter CC BY 2.0 (http://creativecommons.org/licenses/by/2.0). Bezug via Flickr (https://www.flickr.com/photos/florianric/7263382550/, letzter Zugriff: 14.10.2018).

S. 344, Band 2: Foto von Miraz092, lizensiert unter CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/). Bezug via Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Table_Tennis_Net.jpg, letzter Zugriff: 14.10.2018).

S. 344, Band 3: Grafik von Cody Hofstetter, lizensiert unter CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/). Bezug via Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Carna_Botnet_March-December_2012.png, letzter Zugriff: 14.10.2018).

Es handelt sich um eine Snapshot der Visualisierung von tageszeitlich aktiven Hosts nach geographischer Position, die über das Carna Botnet von März bis Dezember 2012 im Rahmen des Hacks «*Internet Census of 2012*» ermittelt wurde.

S. 344, Band 4: Foto von Russss, lizensiert unter CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/) oder GFDL (https://www.gnu.org/licenses/fdl.html). Bezug via Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Redsandsforts.jpg, letzter Zugriff: 14.10.2018).

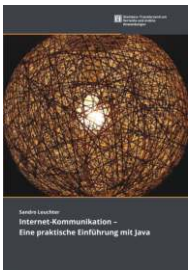
Es handelt sich um die befestigten Plattformen «Red Sands», die im Zweiten Weltkrieg vom britischen Militär zur Flugabwehr vor der englischen Ostküste errichtet wurden. Die ähnliche Festung «Roughs Tower» wurde 1967 besetzt und (einseitig) zum unabhängigen Fürstentum «Sealand» erklärt.

S. 344, Band 5: Grafikausschnitt einer Briefmarke aus Ecuador, gemeinfrei lt. Wikimedia Commons (https://commons.wikimedia.org/wiki/Commons:Stamps/Public_domain#Ecuador). Bezug via Don Hillger & Garry Toth über *Regional and Mesoscale Meteorology*

Branch der *Colorado State University* (`http://rammb.cira.colostate.edu/dev/hillger/Ecuador.748A.jpg`, letzter Zugriff: 14.10.2018).

Auf der Briefmarke aus Ecuador von 1966 ist Syncom (*Synchronous Orbit Communications Satellite*) 2, der erste funktionstüchtige Satellit auf einer geosynchronen Umlaufbahn abgebildet. Mit ihm wurden erstmalig Telefonverbindungen sowie experimentell auch Fernsehbilder (s/w, niedrige Bandbreite und ohne Ton) wie z. B. von den XVIII. Olympischen Spiele aus Tokio in die USA übertragen.

Verteilte Architekturen



Band 1: Internet-Kommunikation (2018)

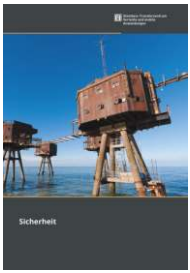
- Internet-Adressierung, DNS, DHCP, UDP, TCP, Socket-Programmierung
- Message Broker, JMS, ActiveMQ, MQTT, Paho
- Web, HTTP, Servlets, Tomcat, Cloud Computing, Google App Engine

Band 2: Aufruf verteilter Unterprogramme (geplant 2019)

- Webservices, XML/JSON-RPC, SOAP/JAX-WS, REST/JAX-RS, Swagger, CoAP
- Serialisierungsformate, XML, JSON, BSON, ProtocolBuffers
- WebSockets, HTTP/2, gRPC, verteilte Objekte, RMI, CORBA

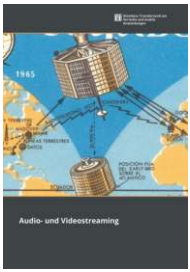
Band 3: Skalierbare Systemarchitekturen (geplant 2020)

- Reactive Architekturen, Cluster, JGroups, DDS, Kafka, Akka
- Microservices, Resource Injection, JNDI, Spring Boot, Spring Cloud, Docker, Kubernetes
- Service Discovery, Serverless Architekturen



Band 4: Sicherheit (in Planung)

- Verschlüsselung, PKI, TLS
- Authentifizierung, Challenge/Response, HTTP, OAuth, SAML, SSO
- Autorisierung, Blockchain



Band 5: Audio- und Videostreaming (in Planung)

- Disovery: Zeroconf/Bonjour, multicast DNS, UPnP, SSDP, DLNA
- On demand Streaming: Jitter, Live Streaming: RTP, RTCP, RTSP; VoIP: SIP, SIMPLE, WebRTC
- Peer to Peer, Streaming Server, multi room, Multicast, Overlay-Netze, CDN