Vorlesungsscript FH Friedberg *(Hessen)*

Skript zu Rechnernetzwerke-Labor

Prof. Dr. Karim Roger Kremer

20. März 2005

Inhaltsverzeichnis

Kapitel 1

Grundlagen der TCP/IP-Netzwerkarchitektur

In diesem Kapitel werden nach der Erklärung des Aufbaus von IP-Adressen, Aspekte wie Routing und Name Service besprochen.

1.1 IP-Adressen

Nach dem IPv4-Protokollstandard haben alle IP-Adressen eine Länge von 32 Bit oder 4 Byte. Zur besseren Lesbarkeit wird die Adresse dezimal als 4 Zahlen zu je 8 Bit angegeben. Um Eindeutigkeit zu gewährleisten, werden die einzelnen Zahlen durch Punkte voneinander getrennt geschrieben, z.B. ist 111.11.22.33 von 11.111.22.33 verschieden. Jede Adresse besteht aus einem Netzwerkanteil (net-id) und einem Hostanteil (host-id). Während der Netzwerkanteil das Netzwerk eindeutig kennzeichnet, bezeichnet der Hostanteil den Rechner innerhalb des Netzwerks.

Es gibt fünf verschiedene Netzwerkklassen A, B, C, D, E; von denen D und E nur für Testzwecke genutzt werden. Die Unterscheidung der Netzwerkklassen erfolgt im ersten Byte: Je nach dem Wert des ersten Byte werden ggfs. noch ein oder zwei Byte zur Netzwerkadresse hinzugenommen. Dementsprechend verringert sich der Hostanteil der Adresse. Auf diese Weise kann man also Netzwerke mit mehr oder weniger Hosts spezifizieren.

Klasse	1. Byte	2. Byte	3. Byte	4. Byte
A	0-127 binär: 0xxx xxxx	Hostanteil	Hostanteil	Hostanteil
B	128-191 binär:10xx xxxx	Netzanteil	Hostanteil	Hostanteil
C	192-223 binär: 110x xxxx	Netzanteil	Netzanteil	Hostanteil

Die Anzahl möglicher Netze und Hosts innerhalb eines Netzes ergibt sich aus Netzwerk- und Hostanteil der verschiedenen Netzwerkklassen:

Klasse	Netzwerkanteil	Netzanzahl	Hostanteil	Hostanzahl
A	1 Bit fix + 7 Bit	128	24 Bit	16.777.214
B	2 Bit fix + 14 Bit	16.384	16 Bit	65.534
C	3 Bit fix + 21 Bit	2.097.152	8 Bit	256

Nach dem IPv4-Standard stehen also insgesamt $128 + 16.384 + 2.097.152 = 2.113.664$ Netzwerkadressen zur Verfügung. Obwohl diese Zahl auf den ersten Blick recht groß erscheint, gibt es praktisch keine freien Netzwerkadressen mehr. Deswegen wurde 1995 der Standard IPv6 oder IPng (IP next generation) verabschiedet. Dazu später mehr.

Es gibt mehrere reservierte IP-Adressen für besondere Zwecke. Die sog. Netzwerkmaske hat an den Stellen des Netzwerkanteils sämtliche Bitstellen gleich 1 und an den Stellen des Hostanteils sämtliche Bitstellen gleich 0. Wenn einem Rechner eine IP-Adresse zugewiesen wird, wird i.d.R. die Standardnetzwerkmaske automatisch generiert. Netzwerkmasken sind Filter, mit denen Rechner entscheiden, ob sich andere Rechner, an die gesendet oder von denen empfangen werden soll, im selben Netzwerk befinden oder nicht. Die Standardnetzwerkmasken lauten:

Klasse	1.Byte	2. Byte	3. Byte	4. Byte
A	255	0	0	0
B	255	255	0	0
C	255	255	255	0

Eine weitere reservierte IP-Adressenart sind die Broadcast-Adressen. Hierbei sind alle Bitstellen des Hostanteils der IP-Adresse auf 1 gesetzt. Die Broadcast-Adresse ermöglicht, Datenpakete an alle Rechner eines Netzwerks zu senden. Die Broadcast-Adresse 192.9.200.255 würde alle Rechner des C-Klasse-Netzwerks 192.9.200.0 bezeichnen.

Insgesamt ergeben sich folgende Zusammenhänge:

- Die Netzwerkadresse ergibt sich aus der bitweisen logischen Und-Verknüpfung der IP-Adresse mit der Netzwerkmaske:

IP-Adresse	192.9.200.7
Netzwerkmaske	255.255.255.0
Netzwerkadresse durch logisches Und	192.9.200.0

- Die Broadcastadresse ergibt sich aus der bitweisen logischen Oder-Verknüpfung der Netzwerkadresse mit der invertierten Netzwerkmaske:

Netzwerkadresse	192.9.200.0
invertierte Netzwerkmaske	0.0.0.255
Broadcastadresse	192.9.200.255

Eine weitere Spezialadresse ist die Loopback-Adresse, der die A-Klasse-Netzwerkadresse 127.0.0.0 zugewiesen ist. Mit Adressen dieses Netzwerks lässt sich die Netzwerkschnittstelle des eigenen Rechners überprüfen. So ist 127.0.0.1 standardmäßig die IP-Adresse des Loopback-Interface. Alle an diese Adresse gerichteten Datenpakete werden nicht nach außen gegeben, sondern von der Netzwerkschnittstelle reflektiert. Es entsteht hierdurch der Eindruck, als ob die Pakete von einem angeschlossenen Netzwerk kämen.

Im Zusammenhang mit der Netzwerkmaske steht auch das Bilden sogenannter logischer Subnetze. Unter Subnetting versteht man die Aufteilung eines größeren Netzwerks in kleinere logische Teilnetze. Gründe für dieses Subnetting können technischer oder organisatorischer Art sein, z.B.:

- Trennung von Netzwerken mit unterschiedlicher Technik,
- Trennung von Netzwerken nach Standorten oder Arbeitsbereichen,
- Abkopplung von Bereichen mit sensitiven Daten,
- Bildung logischer Arbeitsgruppen.

Im Ethernet mit CSMA/CD-Netzzugangsverfahren wächst die Wahrscheinlichkeit für Datenkollisionen mit jedem weiteren angeschlossenen Rechner (, wenn ein klassisches Ethernet-Kabel ohne Workgroup-Switch eingesetzt wird). Dieser Effekt kann durch Teilnetzbildung verringert werden. Man teilt durch Router das Netz in Teilnetze auf, um eine Datenreduktion in den Teilnetzen zu erreichen. Es ist jedoch nach wie vor möglich, über den Router Daten zwischen den Teilnetzen auszutauschen.

Technisch bedeutet Subnetting, dass man einzelne Bitstellen des Hostanteils der IP-Adresse dem Netzwerkanteil hinzufügt. Die Anzahl der Teilnetze, die so gebildet werden, wird über die Anzahl der hinzugefügten Bitstellen bestimmt. Man unterscheidet 1-Bit-, 2-Bit-, usw. Subnetting mit 2, 4, usw. Teilnetzen. Für das Subnetting ändert man in den betroffenen Hosts lediglich die Standardnetzwerkmaske ab, d.h. den Filter über den entschieden wird, welche Rechner sich in einem Netzwerk befinden.

Aus eine C-Klasse-Netzwerk mit 256 Hosts sollen 8 Teilnetze mit je 32 Hosts gebildet werden. Man verwendet also 3-Bit-Subnetting. Die Standardnetzwerkmaske 255.255.255.0 wird also zu 255.255.255.224 (224 entspricht binär 1110 0000). Ein richtig konfigurierter Router lässt nur Datenpakete aus dem Teilnetz, deren Netzwerkadresse ungleich der eigenen Netzwerkadresse ist. Außerdem lässt er nur Datenpakete in ein Teilnetz hinein, die an das Teilnetz tatsächlich gerichtet sind.

1.2 Adressauflösung

IP-Adressen müssen auf physikalische MAC-Adressen abgebildet werden, um die zugehörigen Netzwerkkarten der Rechner zu adressieren. Das Address Resolution Protocol (ARP) wird u.a. für Ethernet-MAC-Adressen verwendet. ARP arbeitet wie folgt:

Zunächst sendet ARP ein Ethernet-Broadcast-Datagramm an alle angeschlossenen Geräte des Ethernet-LAN's. Dieses Broadcast-Datagramm enthält die IP-Adresse der zu ermittelnden physikalischen Adresse. Jeder Rechner vergleicht nun seine IP-Adresse(n) mit der angefragten IP-Adresse. Bei Übereinstimmung wird die Ethernet-MAC-Adresse an den anfragenden Host in einem Ethernet-Datagramm zurückgesendet.

Die bekannten Ethernet-Adressen werden mit den IP-Adressen des LAN's pro Host in einem ARP-Cache gespeichert. Hierdurch kann eine erneute Anfrage nach der Ethernet-Adresse i.d.R. vermieden werden. Der ARP-Cache wird allerdings nach einiger Zeit für ungültig erklärt, damit Änderungen oder Störungen im Netz berücksichtigt werden.

Manchmal muss zu einer Ethernet-MAC-Adresse auch eine IP-Adresse ermittelt werden. Wenn ein Diskless-Client z.B. von einem Server aus dem Netz gebootet werden will, kennt er nur seine Ethernet-Adresse. Deshalb sendet der Client zunächst eine Broadcast-Nachricht, in der die Boot-Server aufgefordert werden, ihm seine IP-Adresse mitzuteilen. Hierzu wird das Reverse Address Resolution Protocol (RARP) verwendet. Zusammen mit dem BOOTP-Protokoll werden mit RARP unter dem Betriebssystem Linux z.B. solche Diskless-Clients gebootet.

Noch eine Bemerkung: ARP und RARP lösen natürlich nur Adressen innerhalb eines LAN's auf. Die Aufgabe, Datagramme zwischen unterschiedlichen LAN's bzw. im Internet zu vermitteln, übernimmt das Routing.

1.3 IP-Routing

Ähnlich der Briefpost verläuft das IP-Routing nach einem hierarchischen Schema. Der Brief wird zunächst zur Verteilstelle des Staates geleitet; diese leitet den Brief an das Land oder die Region weiter; von dort wird der Brief in die Stadt weitergeleitet u.s.w. Der Vorteil dieses Prinzips liegt auf der Hand: Die jeweilige Verteilstation muss nur die Richtung des Briefes kennen (z.B. den Staat), d.h. ein Teilstück des Weges. Sie muss sich nicht darum kümmern, welchen Weg der Brief nach diesem Teilstück (z.B. innerhalb des Staates) nimmt.

IP-Netzwerke sind nach dem gleichen Prinzip aufgebaut. Das Internet besteht aus einer Menge von autonomen Netzwerken. Jedes autonome Netzwerk ist für das Routing innerhalb dieses Netzes zuständig. Damit reduziert sich die Aufgabe, ein Datagramm auszuliefern, auf die Bestimmung eines Wegs zum autonomen Netzwerk des Zielrechners. Das bedeutet, sobald das Datagramm irgendeinem Rechner eines autonomen Netzwerks übergeben ist, übernimmt dieses Netzwerk die Weiterleitung.

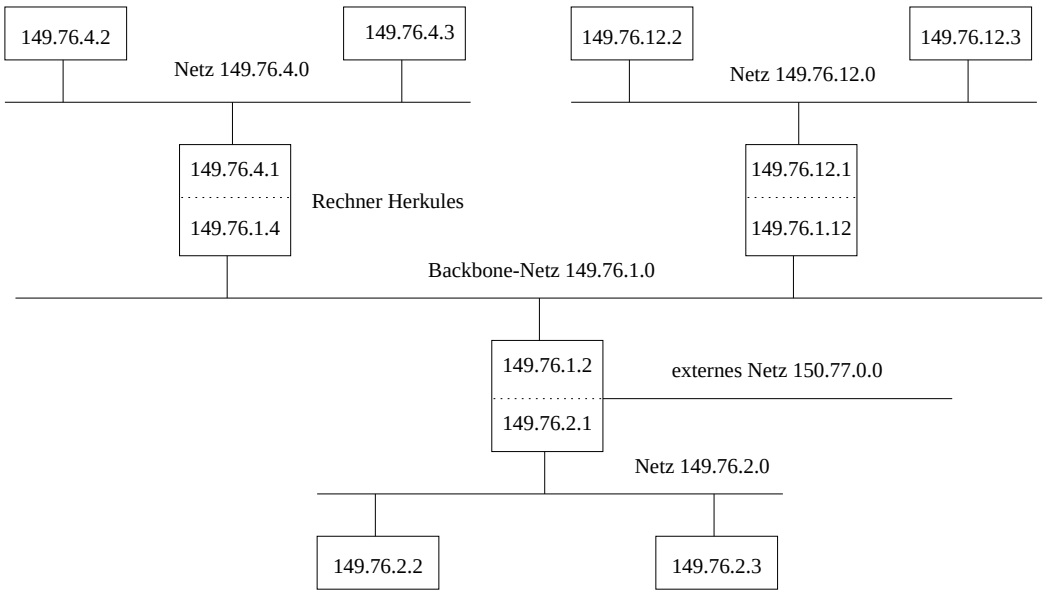
Die Sicht eines einzelnen Rechners im Internet ist also beschränkt. Die einzigen Rechner, mit denen er direkt kommunizieren kann, sind die in seinem lokalen Netz. Will er einen Rechner außerhalb des lokalen Netzes erreichen, so muss dies über einen Gateway-Rechner geschehen. Ein Gateway-Rechner ist mit zwei oder mehr physikalischen Netzen gleichzeitig verbunden. Er leitet die Datagramme zwischen den Netzen weiter. Die Netzwerksoftware des Quellrechners erkennt anhand der Ziel-IP-Adresse, dass das Zielnetz nur über einen Gateway erreicht werden kann. Das geschieht wie folgt:

- Der Netzanteil der Ziel-IP-Adresse wird mit dem eigenen Netzanteil der Quell-IP-Adresse verglichen, z.B.:

	Quell-IP-Adresse	Ziel-IP-Adresse
	149.76.4.1	149.76.1.4
Netzmaske	255.255.255.0	255.255.255.0
Netzanteil	149.76.4.0	149.76.1.0

- Ist der Netzanteil von Quelle und Ziel gleich, so befindet sich der Ziel-Rechner im lokalen Netz; ansonsten muss das Datagramm an einen Gateway weitergeleitet werden.

Nun stellt sich natürlich die Frage, wie in IP ein Gateway ausgesucht wird, der das Datagramm weiterleitet. Dazu sind zunächst die Gateways mit ihren angeschlossenen Netzen im Quellrechner verzeichnet. Dies nennt man eine Routing-Tabelle. Betrachten wir folgendes Beispiel-Netzwerk:



Die um einige Einträge erweiterte Routing-Tabelle des Rechners mit dem Namen Herkules sieht dann wie folgt aus.

Netz	Gateway
149.76.1.0	-
149.76.2.0	149.76.1.2
149.76.3.0	149.76.1.3
149.76.4.0	-
149.76.5.0	149.76.1.5
...	...
0.0.0.0	149.76.1.2

Der Eintrag zum Netz 0.0.0.0 bezeichnet einen Default-Gateway. Sollte die Netzwerkadresse des Zielrechners in der Routing -Tabelle nicht gefunden werden, so wird das Datagramm zum Default-Gateway weitergeleitet. Wenn die Zielnetzwerkadresse in der Routing-Tabelle verzeichnet ist, wird das Datagramm direkt zum Gateway des Zielnetzes weitergesendet.

Eine Routing-Tabelle kann für kleine Netze manuell erzeugt und gepflegt werden. Für größere Netze gibt es Routing-Protokolle zur automatischen Erstellung und Wartung der Routing-Tabellen. Hierbei tauschen Routing-Dämone Informationen aus, um möglichst optimale Routen zwischen den Netzwerken zu bestimmen.

Für das Routing innerhalb eines autonomen Netzwerks werden interne Routing-Protokolle, wie das Routing Information Protocol (RIP) verwendet. Das Routing zwischen verschiedenen autonomen Netzwerken wird mit externen Routing-Protokollen, wie dem External Gateway Protocol (EGP) oder dem Boarder Gateway Protocol (BGP), bewerkstelligt. RIP wählt die beste Route anhand der Anzahl von Hops, d.h. die Anzahl Gateways die ein Datagramm bis zum Zielnetz zurücklegt. Lange Routen mit mehr als 16 Hops werden als unbrauchbar verworfen.

Im Linux-Betriebssystem untersucht der Route-Dämon gated zur Boot-Zeit eines Rechners alle Netzwerkschnittstellen. Ist genau eine Schnittstelle aktiv, so empfängt gated RIP-Routing-Informationen. Bei mehr als einer aktiven Netzwerkschnittstelle sendet und empfängt gated Routing-Informationen.

Neben IP gibt es das Steuerprotokoll ICMP (Internet Control Message Protocol) im Internet. Dieses Protokoll wird verwendet, um Fehlermeldungen u.ä. zwischen Rechnern auszutauschen. Wird z.B. versucht, einen Port eines anderen Rechners zu kontaktieren, an dem kein Prozess aktiv ist, so wird vom Empfänger eine Meldung wie “Port unreachable” an den Sender zurückgegeben. Für das Routing gibt es eine interessante ICMP-Meldung, die Redirect Message. Sie wird generiert, wenn ein Host als Gateway benutzt wird, obwohl es eine kürzere Route gibt. Nehmen wir an die Routing-Tabelle des Rechners Herkules besteht z.B. nur aus folgenden Einträgen:

Netz	Gateway
149.76.1.0	-
149.76.4.0	-
0.0.0.0	149.76.1.2

Dann wird ein Datagramm für den Rechner 149.76.12.2 an den Default-Gateway gesendet. Es wäre natürlich günstiger gewesen, das Datagramm direkt an den Gateway 149.76.1.12 des Netzes 149.76.12.0 zu senden.

Wenn der Default-Gateway nun ein Datagramm für den Rechner 149.76.12.2 erhält, erkennt er, dass es eine bessere Verbindung nämlich von 149.76.1.4/(4.1) nach 149.76.1.12/(12.1) gibt, da seine Routing-Tabelle folgenden Eintrag enthält.

Netz	Gateway
149.76.12.0	149.76.1.12

Genauer gesagt, erkennt der Default-Gateway, dass der Gateway zum Netz 149.76.12.0 an das selbe Netz 149.76.1.0 wie der Absender 149.76.1.4 angeschlossen ist. Diese Routing-Information wird nun im Rechner Herkules ergänzt. Es gibt allerdings ein Problem dieser Redirect Messages, nämlich die fehlende Verifikation des Senders einer solchen Nachricht. Hierdurch werden Einbrüche unberechtigter Nutzer in Netzwerke möglich.

1.4 IP Next Generation

IPng oder IP Version 6 (IPv6) wurde als Nachfolger von IPv4 entwickelt. Version 5 wurde dem Stream Protokoll ST-II zugeteilt, das eine Komponente der Dienstintegrierten-Internet-Architektur (Internet Integrated Service Architecture, IISA) ist. Das Stream Protokoll ST-II realisiert neben RSVP (Resource Reservation Protocol) neue Netzdienste für Echtzeit- und Multimedia-Anwendungen. Da die nicht vergebenen Netze in IPv4 wegen der hierarchischen Adressenstruktur nahezu erschöpft sind, wurde in IPv6 eine neue Adressenstruktur entwickelt. Die Adresslänge wuchs von 32 auf 128 Bit. Damit sind mehr Hierarchieebenen und eine größere Anzahl adressierbarer Hosts möglich. Der IPv6 Adressraum ist 2^{96} also ungefähr 4 Milliarden mal größer als der von IPv4. Man kann berechnen, dass bei der heutigen Nutzung des IPv4 in IPv6 1564 Hosts pro m^2 der Erdoberfläche möglich wären. In dieser Hinsicht sollte der IPv6-Standard wohl genügen. IPv6 ist mit der älteren Version IPv4 kompatibel, somit haben die Hersteller von Netzwerkkomponenten und Betriebssystemen Zeit sich auf den neuen Standard umzustellen.

In IPv6 gibt es drei Arten von Adressen:

- Unicast: Adresse für eine Schnittstelle,
- Anycast: Adresse für eine Gruppe von Schnittstellen, wobei das Datagramm an die nach dem Routingmaß nächstliegende Schnittstelle gesendet wird,
- Multicast: Adresse für eine Gruppe von Schnittstellen, wobei das Datagramm an alle Schnittstellen der Gruppe gesendet wird.

Eine einzige Schnittstelle kann auch mehrere Unicast-Adressen tragen. Hierdurch hat ein Teilnehmer die Möglichkeit, für die Dienste unterschiedlicher Provider auch unterschiedliche Unicast-Adressen mit einer einzigen Netzwerkkarte zu nutzen. Unicast-Adressen werden in mehrere Gruppen unterteilt, die durch einen Präfix der Adresse gebildet werden. Einige wichtige Gruppen sind:

Gruppe	Präfix	Anteil
Provider-basiert	010	1/8
Geographisch-basiert	100	1/8
Verbindungslokal	1111 1110 10	1/1024
Standortlokal	1111 1110 11	1/1024
Multicast	1111 1111	1/256
IPv4	96 0'en	$1/2^{96}$

Provider-basierte Adressen haben folgendes Format:

010	Registry Id	Provider Id	Subscriber Id	Subnet Id	Interface Id

Die Registry Id identifiziert die Registrierungs-Institution, die den Provider Id-Teil vergibt. Die Provider Id identifiziert einen bestimmten Dienstanbieter, der die Subscriber Id vergibt. Die Subscriber Id unterscheidet die Teilnehmer des Providers. Die Subnet Id spezifiziert ein Subnetz des Teilnehmers. Die Interface Id identifiziert schließlich die Schnittstelle innerhalb des Subnetzes.

Provider-basierte Adressen dienen der globalen Kommunikation. Für den Netzmanager sind in erster Linie Subnet Id und Interface Id interessant. Da die Länge der Felder der Provider-basierten Adressen nicht festgelegt ist, kann er z.B. die 48 Bit einer Ethernet-MAC-Adresse 1:1 als Interface Id wählen und den Rest für die Subnet Id verwenden.

Verbindungslokale und standordlokale Adressen sind im Gegensatz zu Provider-basierten Adressen nur in einem lokalen Bereich routbar. Sie werden also nur in einem Subnetz oder in einer Gruppe von Subnetzen eingesetzt.

Verbindungslokale Unicast-Adressen (Link local) haben folgendes Format:

1111 1110 10...0...	Interface Id

Standortlokale Unicast-Adressen (Site local) haben folgendes Format:

1111 111011...0...	Subnet Id	Interface Id

Solche lokale Adressen sind für die schnelle lokale Kommunikation gedacht. Außerdem können sie vor der Zuteilung einer globalen Adresse verwendet werden und erlauben einen Übergang, in dem die fehlenden Anteile der Provider-basierten Adresse anstatt der Nullen eingesetzt werden.

Multicast-Adressen haben folgendes Format:

1111 1111	000T	Scop	Group Id

Ist das T-Flag 0, so wird eine "well known" Adresse, die von der Adressvergabebehörde zugeteilt wurde, verwendet. Ist das T-Flag 1, so wird eine zeitweise zugewiesene Adresse (transient) verwendet. Scop ist ein 4 Bit-Feld zur Begrenzung des Ausdehnungsbereichs der Multicast-Gruppe:

Scop	Bereich
1	Knoten-lokal
2	Verbindungs-lokal
5	Standort-lokal
8	Organisations-lokal
14	global

Die Group Id identifiziert die Gruppe innerhalb des Bereichs.

Da nicht sofort alle Router durch IPv6-fähige Router ersetzt werden können, wird mit einem relativ langen Übergang von IPv4 zu IPv6 gerechnet. Knoten, die IPv4-kompatible IPv6-Adressen bekommen, werden als IPv6/IPv4-Knoten bezeichent. Sie unterstützen einen Mechanismus, der es erlaubt IPv6-Pakete in IPv4-Datagrammen zu verpacken, wodurch das Routing in IPv4 vorgenommen wird. Diesen Mechanismus nennt man Tunneling.

Eine IPv4-kompatible IPv6-Adresse hat folgendes Format:

IPv6-Header mit IPv4-kompatibler Adresse	Transport Layer Header	Daten

Nach dem Tunneling sieht die umgeformte IPv4-Adresse wie folgt aus:

IPv4-Header	IPv6-Header	Transport Layer Header	Daten

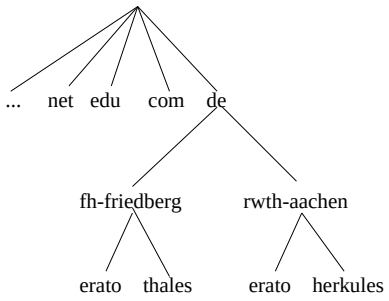
1.5 Domain Name Service (DNS)

Da man sich IP-Adressen nur schwer merken kann, hat man im Internet eine Möglichkeit geschaffen, IP-Adressen mit frei wählbaren, logischen Namen zu versehen. Die Zuordnung zwischen IP-Adressen und Rechnernamen kann in der Datei hosts vorgenommen werden. Für ein C-Klasse-Netzwerk könnte hosts z.B. folgende Einträge haben:

```
#/etc/hosts
127.0.0.1     localhost
192.168.7.1   saturn.fh-friedberg.de
192.168.7.2   jupiter.fh-friedberg.de
```

Mit der Größe des Netzwerks wächst allerdings der Aufwand für diese Datei stark an. Bis 1984 wurden alle Rechnernamen und IP-Adressen zentral in der Datei HOSTS.TXT im NIC (Network Information Center) in den USA gehalten. Jeder neue Rechner musste diese Datei installieren und nach Änderungen musste die Datei auf die Rechner neu verteilt werden. Aufgrund der steigenden Anzahl von Internet-Rechnern war dieser Aufwand aber nicht länger vertretbar, so dass ein neues Verfahren, der Domain Name Service (DNS) eingeführt wurde.

DNS organisiert Rechnernamen in einer Hierarchie von Domains ähnlich einem hierarchischen Dateisystem. Die oberste Ebene nennt sich Top-Level Domain. Darunter befinden sich Subdomains. Eine Domain ist eine Menge von Rechnern, die geographisch oder organisatorisch zusammengehören, z.B. alle Rechner eines Landes (Top-Level Domain) oder alle Rechner einer Hochschule (Subdomain). Durch die Hierarchie ist die Eindeutigkeit von Rechnernamen leicht zu erreichen: Namen müssen nur innerhalb einer Ebene eindeutig sein, d.h. es kann wie im folgenden Bild einen Rechner mit dem Namen erato sowohl an der FH Giessen-Friedberg als auch an der RWTH Aachen geben.



Die vollständigen Namen der Rechner lauten nämlich: erato.fh-friedberg.de bzw. erato.rwth-aachen.de. Man beginnt bei der Namensbildung also in der untersten Ebene bis hinauf zur Top-Level Domain, wobei die Namen durch Punkte voneinander getrennt werden. Zwischen der Schreibweise von IP-Adressen und DNS-Adressen besteht zwar eine gewisse Ähnlichkeit, denn beide bestehen gewissermaßen aus einem Hostanteil und einem Netzwerkanteil. Während eine IP-Adresse aber immer aus genau 4 Teilen zusammengesetzt ist, kann eine DNS-Adresse aus beliebig vielen Teilen bestehen. Es gibt also keine vorgeschriebene Zuordnung zwischen IP-Adressen und DNS-Adressen.

In den USA gibt es mehrere Top-Level Domains:

com	Kommerzielle Organisationen (z.B. ibm.com, sun.com)
edu	Bildungseinrichtungen (z.B. berkeley.edu)
gov	Regierungsstelle (z.B. whitehouse.gov)
mil	Militärorganisationen (z.B. army.mil, navy.mil)
net	Organisationen von Netzwerkbetreibern (z.B. nfs.net, internic.net)
org	Nicht-kommerzielle Organisationen
int	Internationale Organisationen (z.B. nato.int)

Ansonsten bilden Länderkennzeichen die Top-Level Domain, einige davon sind:

at	Österreich
au	Australien
ca	Canada
ch	Schweiz
de	Deutschland
es	Spanien
fr	Frankreich
jp	Japan
uk	Großbritanien
us	USA

Alle Top-Level Domain der USA werden vom NIC verwaltet (http://www.-internic.net). Andere Top-Level Domains werden innerhalb der Länder verwaltet, z.B. vom DE-NIC (http://www.de-nic.de) an der Universität Karlsruhe. DE-NIC verwaltet die First-Level Domains also fh-friedberg.de, rwth-aachen.de usw. Die Verwaltung der Second-Level Domains übergibt DE-NIC an den Netzwerkverwalter dieser Domains.

Insgesamt bildet DNS eine weltweit verteilte Datenbank im Internet. Eine Vielzahl von Domain Name Servern verwalten die relevanten Daten ihrer jeweiligen lokalen Domains. Der Verwaltungsbereich eines Domain Name Servers, die Zone, kann eine oder mehrere Domains umfassen. In jeder Zone sollten zur Sicherung gegen Ausfälle mindestens zwei Domain Name Server vorhanden sein, die über verschiedene Netzwerk-Routen erreichbar sind. Einer dieser Server übernimmt die Führungsrolle; er wird als Primary Domain Name Server bezeichnet. Die Konfigurationsdateien eines Primary Domain Name Server muss der Netzwerkadministrator anlegen und pflegen. Der Secondary Domain Name Server gleicht in regelmäßigen Abständen seine Informationen automatisch, mit denen des Primary Domain Name Server ab. Bei Ausfall des Primary Domain Name Server kann der Secondary Domain Name Server seine Aufgaben übernehmen.

Der Primary Name Server für die Top-Level Domain de wird von DE-NIC betrieben. Es gibt zwei weitere Secondary Domain Name Server für de, einer bei XLink (xlink1.xlink.net) der zweite bei Eunet (ns.germany.eu.net). Damit die Informationen über die Top-Level Domain de auch außerhalb Deutschlands gut verteilt sind, gibt es weitere Secondary Domain Name Server in Europa (2) und in den USA (3).

DNS ist eine Client/Server-Anwendung. Client-Programme, die auf die Dienste von Servern zugreifen, heißen Resolver. Praktisch wird der Resolver meist über Bibliotheksfunktionen in eine Applikation eingebunden, z.B. in telnet oder ftp. Wenn eine Applikation mit einem Rechner, von dem sie nur den DNS-Namen kennt, Kontakt aufnehmen will, muss sie den DNS-Namen in eine IP-Adresse umwandeln. Dazu formuliert der Resolver eine Anfrage an den in der Konfigurationsdatei resolve.conf eingetragenen Name Server.

Die Einträge in resolve.conf bestimmen die Suchstrategie. Entweder es wird von dem Root-Server der Top-Level Domain aus der Domain Name Server der Subdomain befragt (z.B. fh-friedberg.de), der die IP-Adresse zurückliefert (z.B. von thales.fh-friedberg.de) oder es wird zunächst der lokale Name Server befragt und dann ein Name Server einer höheren Domain.

DNS ist bei jedem Anschluss an das Internet erforderlich und vorgeschrieben. Man muss also auch beim Anschluss eines einzigen PC's an das Internet einen Name Server angeben oder sogar einrichten. Insbesondere muss die eigene IP-Adresse und der DNS-Name bei einem Domain Name Server eingetragen werden. Deshalb muss man schon bei der Beantragung einer Domain die Adressen von Primary und Secondary Domain Name Server angeben. Ein weiterer Secondary Domain Nameserver wird i.d.R. vom Provider gegen Entgeld betrieben. Das sorgt dafür, dass z.B. EMails beim Provider gepuffert werden können, wenn die Verbindung zum Provider gestört ist. Kleinere Firmen überlassen häufig den Providern die Installation und Pflege der DNS-Server.

Kapitel 2

Einführung in das TCP/IP-Protokoll

Dienste wie rlogin, rsh, ftp, telnet usw. sind zur Verbindung verschiedener Rechner über ein Netzwerk entstanden. Diese Netzdienste verfügen über eine bequeme Schnittstelle, die das darunterliegende Netzwerkprotokoll verbirgt. Dabei sind Netzdienste meist als Client/Server-Anwendungen realisiert. Der Server ist meist ein Hintergrundprozess oder Dämon, z.B. in UNIX telnetd oder rshd.

Um eigene verteilte Anwendungen aus Clients und Servern zu schreiben, muss man ein eigenes Anwendungsprotokoll entwerfen, d.h. inhaltlich festlegen wie Client und Server miteinander kommunizieren. Zur Übertragung der Daten benutzt man den Transportdienst des Netzwerks. Innerhalb eines Netzwerks existieren i.d.R. verschiedene Transportdienste, die man kennen muss, um den für die Anwendung am besten geeigneten anzuwenden. Daher beginnen wir mit einer Einführung in das TCP/IP-Protokoll.

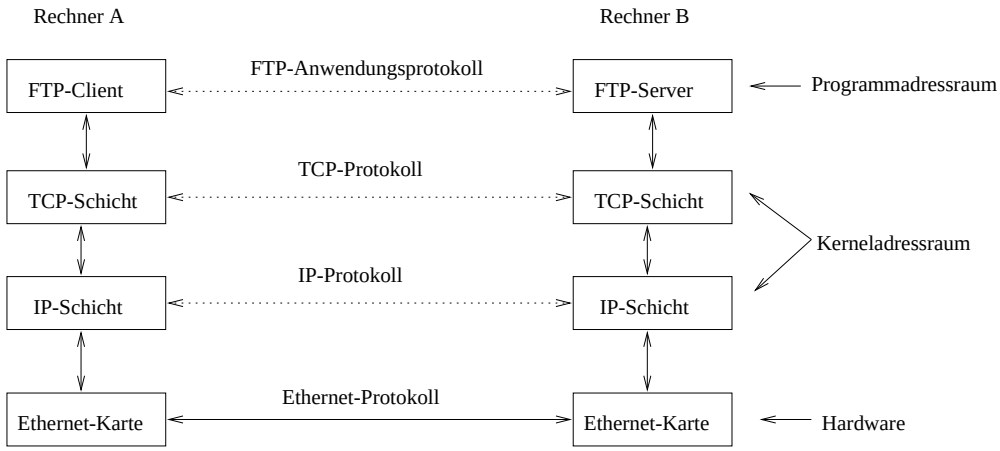
2.1 Protokollstapel

Datentransportdienste, wie TCP/IP, sind in mehreren hierarchischen Schichten organisiert, die einander nutzen und Funktionalität kapseln. Die verteilte Anwendung ftp ist über wenige Systemaufrufe und Bibliotheksroutinen des TCP-Dienstes realisiert, der meist Bestandteil des Betriebssystemkerns ist.

Die Daten wandern den Protokollstapel des Senders hinunter und den Protokollstapel des Empfängers hinauf. Konzeptionell stellt man sich allerdings vor, dass Client und Server direkt miteinander kommunizieren. Man spricht von Peer-to-Peer-Kommunikation, wenn man die Kommunikation zwischen zwei Ebenen gleichen Niveaus betrachtet.

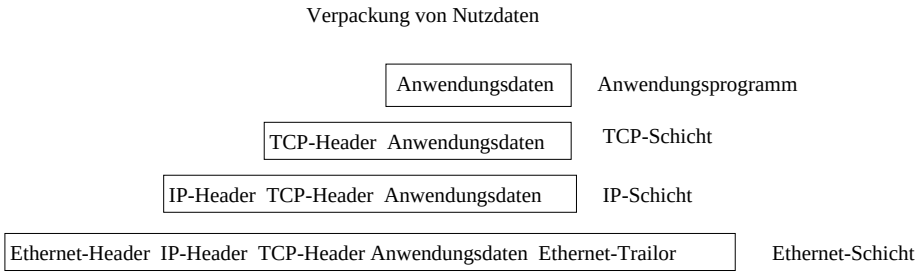
Echte Anwendungsprotokolle sind im Vergleich zur Nutzung eines Netzwerk-Transportdienstes kompliziert. Wir wollen daher erst den Transportdienst von TCP/IP kennenlernen. (Man kann muss ja auch erst lernen, wie man telefoniert, bevor man

am Telefon eine Ware verkaufen kann.) Im folgenden wollen wir uns erst mit den Aufgaben der einzelnen Schichten beschäftigen.



2.2 Konzept der Verpackung von Nutzdaten

Beim Verpacken von Nutzdaten wird von der logisch tieferen Ebene beim Senden jeweils ein Header (Kopf) oder ggfs. auch ein Trailer (Fuß) zu den Nutzdaten hinzugefügt. Header und Trailer enthalten Steuerdaten für die Kommunikation auf dieser logischen Ebene. Beim Empfang werden diese Steuerdaten wieder entfernt und die Nutzdaten an die logisch höhere Ebene weitergereicht. Das Bild zeigt als Beispiel die Verpackung von Anwendungsdaten mit TCP/IP auf einem Ethernet:



Neben Ethernet, das in UNIX-Netzen häufig eingesetzt wird, gibt es z.B. auch FDDI oder Token-Ring als physikalische Protokolle, deren Blockformat analog zu Ethernet aufgebaut ist. Keine Schicht des Protokollstapels interpretiert Daten, die sie beim Senden von einer darüber liegenden Schicht erhalten hat. Die Header- und Trailor-Daten sind nur für den entsprechenden Peer der gleichen Schicht am Empfangsende bestimmt.

2.3 Ethernet-Schicht

Die Ethernet-Schicht befasst sich mit der Übertragung von Blöcken oder Frames auf einem physikalischen Netzwerk. In einem Frame werden die Nutzdaten der übergeordneten Schicht, z.B. von IP, transportiert. Neben einer 48 Bit Zieladresse und Quelladresse enthält ein Ethernetrahmen oder Frame eine Kennzahl, die anzeigt für welches Protokoll höherer Ebene der Frame transportiert wird, z.B. hexadezimal 800 für IP und 809B für Appletalk. Durch dieses Typfeld können mehrere Protokolle unterschiedlicher Hersteller gleichzeitig auf einem einzigen physikalischen Ethernet genutzt werden. Weiterhin enthält der Rahmen die Nutzdaten, also z.B. einen Teil oder ein ganzes IP-Datagramm, und eine CRC-Prüfzahl, um die Richtigkeit der Übertragung zu überprüfen. Eine Präambel oder ein Vorspann dient zur Synchronisation des Empfängers. Die Länge eines Ethernet-Frame ist variabel.

Ethernet-Frame

Präambel	Ziel	Quelle	Typ	Nutzdaten	CRC
64 Bit	48 Bit	48 Bit	16 Bit	368- 12000 Bit	32 Bit

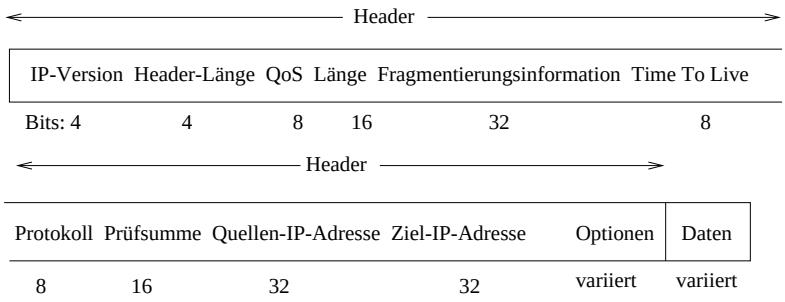
2.4 IP-Schicht

Die IP-Schicht (Internet Protocol) entspricht in etwa der Netzwerkschicht (Schicht 3) im ISO/OSI-Referenzmodell. Wie die Ethernet-Schicht ist sie für den Pakettransport zu anderen Rechnern verantwortlich, im Gegensatz zur Ethernet-Schicht jedoch zusätzlich zwischen Rechnern in unterschiedlichen Netzen. Dabei wird der Weg eines Pakets durch Netzverbindungssysteme, wie Router, bestimmt. Grundlage für dieses Routing ist nicht die Ethernet-Adresse sondern der Netzwerkanteil der IP-Adresse.

IP bietet einen verbindungslosen Transportdienst, d.h., jedes IP-Datagramm wird unabhängig von den anderen Datagrammen durch das Netz geleitet. Daher kann z.B. nicht garantiert werden, dass je zwei aufeinander folgende Datagramme denselben Weg zwischen Quelle und Ziel nehmen. Routing-Entscheidungen sind dynamisch und beruhen i.w. auf momentanen Auslastungsbeobachtungen des Netzwerks. Natürlich kann sich die Auslastungssituation anders als vom Routing angenommen ändern. Hierdurch können später gesendete früher gesendete Datagramme überholen; die Reihenfolge der Datagramme kann z.B. innerhalb einer Nachricht vertauscht sein. Der Empfänger muss die gewünschte Reihenfolge wieder herstellen. Da es keine statische Zuordnung von Ressourcen, wie Pufferspei-

cher oder Netzkapazität, gibt, können IP-Datagramme ggfs. nicht weitergeleitet werden. Man sagt dann auch, dass IP-Datagramme verloren gehen.

IP ist also bestrebt, eine möglichst gute Auslastung des Netzes zu erreichen. Man hat es mit einem Dienst nach besten Kräften (ähnlich der Briefpost) und nicht mit garantierter Zulieferung zu tun. Ein IP-Datagramm hat folgenden Aufbau:



Durch die Fragmentierungsinformation im Header können IP-Datagramme in mehrere Ethernet-Frames aufgespalten werden, wenn sie länger als ein Ethernet-Frame sind. Der Time To Live-Parameter verhindert, dass Datagramme wegen falscher Adressen endlos zwischen Gateways kreisen. Es ist ein Zähler, der pro Hop dekrementiert wird, wobei das Datagramm bei Erreichen des Zählerstandes 0 aus dem Netz entfernt wird. Das Prüfsummen-Feld bezieht sich nur auf den IP-Header. Die IP-Version bestimmt u.a. das Header Format. Die Header-Länge wird in 32 Bit Worten angegeben. Mit dem Quality of Service-Parameter (QoS) kann ein höheres Protokoll entscheiden, wie das Datagramm gehandhabt werden soll.

2.5 Transportschicht

Die Transportschicht (ISO/OSI-Ebene 4) ist für Anwendungsprogrammierer am interessantesten. Üblicherweise wird eins von zwei alternativen Transportprotokollen benutzt entweder das User Datagram Protocol (UDP) oder das Transmission Control Protocol (TCP). Während UDP einen verbindungslosen Transportdienst realisiert, arbeitet TCP verbindungsorientiert. UDP ist vergleichbar mit der Briefpost; TCP mit dem Telefondienst.

Es folgen einige Eigenschaften von UDP mit ihren Analogien zur Briefpost:

- Jeder Brief hat eine Empfängeradresse. Jedes Datagramm enthält die Adresse des Transportendpunktes.
- Man weiß nicht, ob und wann ein Brief am Ziel ankommt. UDP hat keinen Mechanismus, um verloren gegangene Datagramme neu zu senden oder dem Sender den Empfang von Datagrammen zu bestätigen.

- Die Post gewährleistet nicht, dass Briefe in derselben Reihenfolge ankommen, wie sie aufgegeben wurden. UDP liefert Datagramme nicht zwingend in der abgesendeten Reihenfolge beim Empfänger ab.

- Mehrere Briefe an einen Empfänger verschmelzen nicht automatisch zu einer Einheit. Der Empfänger muss die Briefe passend zusammenfügen. UDP-Empfänger empfangen Datagramme und keine Nachrichten.

TCP arbeitet analog zum Telefondienst mit folgenden Eigenschaften:

- Die Telefonnummer muss nur einmal beim Verbindungsaufbau angegeben werden. Während des Anrufs weiß das Netz, dass die Worte zu einem bestimmten Empfänger gelangen müssen. Bei TCP gibt der Client zum Aufbau einer Verbindung eine Server-Adresse an und schließt die Verbindung später wieder. Alle Datagramme können mit dem einmaligen Verbindungsaufbau gesendet werden.

- Die Person am anderen Telefon hört die Worte und Sätze in der vom Sender gesprochenen Reihenfolge. TCP garantiert, dass alle Zeichen in der Reihenfolge ankommen, in der sie abgeschickt wurden.

- Die Telefonverbindung arbeitet in beiden Richtungen gleichzeitig. TCP ist bidirektional.

- Die Telefonverbindung liefert ein kontinuierliches Audiosignal. Das Signal wird nicht in Datenpakete zerlegt. TCP ist stromorientiert (stream oriented), d.h. wenn man Teile von z.B. 100 Byte, 100 Byte und 50 Byte absendet, erscheinen diese nicht als Teile beim Empfänger, sondern als kontinuierlicher Strom von 250 Byte. Zwischen Sender und Empfänger muss es daher Vereinbarungen über die Länge der Nachrichten geben (z.B. Längenfeld oder binäre 0 als Nachrichtenende).

2.6 User Datagram Protocol (UDP)

UDP leistet nur wenig mehr als IP. Die wichtigste Funktion von UDP besteht darin, Datagramme von einem Prozess im Sender-Rechner an einen Transportendpunkt im Empfänger-Rechner zu senden und von diesem Transportendpunkt zum Empfänger-Prozess weiterzuleiten. Transportendpunkte heißen Ports und sind durch eine 16 Bit Portnummer definiert. Im Rahmen der Briefpost-Analogie entspricht IP dem Briefträger, der Post für eine Firma zentral abliefert, und UDP dem Zustelldienst der Post innerhalb der Firma. UDP erbt fast alle Nachteile von IP,

also z.B. die fehlende Empfangssicherung und das Reihenfolgeproblem der Datagramme. Ein UDP-Paket hat folgenden Aufbau:

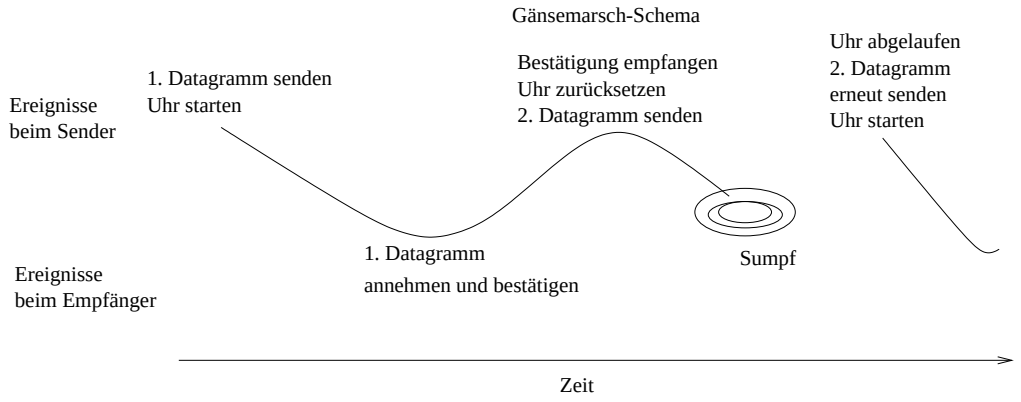
Absendeport	Empfangsport	Länge in Bytes	Prüfsumme des Headers	Anwendungsdaten
Bits: 16	16	16	16	variiert

2.7 Transmission Control Protocol (TCP)

TCP arbeitet wie UDP mit Portnummern und stellt dem Empfänger eine Kontrollsumme zur Verfügung, um den richtigen Empfang zu verifizieren. Damit enden aber die Ähnlichkeiten zu UDP.

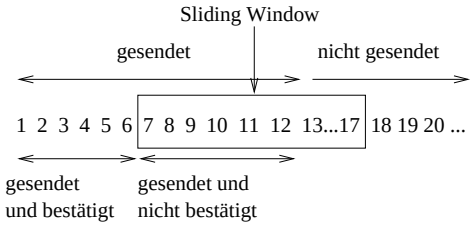
TCP bietet einen zuverlässigen Datentransport in richtiger Reihenfolge der Datagramme. Die angewendete Technik heißt positive Bestätigung mit Übertragungswiederholung. Immer wenn mit TCP ein Datagramm gesendet wird, startet der Sender eine interne Uhr (timer). Erhält der Empfänger das Datagramm, so schickt er eine Bestätigung an den Sender. Wenn der Sender die Bestätigung erhält, setzt er die Uhr wieder zurück. Geht das Datagramm oder die Bestätigung durch den Transport über IP verloren, so läuft die Zeit im Sender ab (timeout). Der Sender wiederholt daraufhin seine Sendung.

Dieses Gänsemarsch-Schema, bei dem auf die Bestätigung jedes Datagramms gewartet wird, bevor das nächste Datagramm gesendet wird, lastet die Datenverbindung i.d.R. nicht aus. Die Analogie wäre ein Dialog mit Postkarten, wobei die Bestätigung der Postkarte ebenfalls eine Postkarte ist. Beide Postkarten sind je einen Tag unterwegs, so dass nur alle zwei Tage eine neue Postkarte geschickt werden kann, obwohl der Briefdienst viel mehr bewältigen kann.



Zur Lösung dieses Problems setzt TCP eine Technik ein, die sich Sliding Window Protocol (gleitendes Fenster Protokoll) nennt. Hierbei können mehrere unbestätigte Datagramme gleichzeitig im Netz vorhanden sein. Bytes, die sich im Fenster befinden, dürfen gesendet werden und sind noch nicht bestätigt. Bytes vor dem Anfang des Fensters wurden übertragen und bestätigt. Bytes hinter dem Ende des Fensters dürfen noch nicht gesendet werden. Während der Datenübertragung gleitet das Fenster über den Byte-Strom, daher stammt der Name Sliding Window.

Im TCP-Header gibt es drei Felder zur Kontrolle des Gleitfensters. Die Sequenznummer ist der Bytestartpunkt eines Datagramms im Bytestrom. Sie wird vom Sender gepflegt. Der Empfänger kann anhand dieser Sequenznummer die richtige Reihenfolge der Datagramme bestimmen. Die Bestätigungsnummer zeigt dem Sender an, welches Datagramm bestätigt wurde. Bestätigungs-Datagramme haben dasselbe Format wie Daten-Datagramme aber natürlich keine Daten. Das dritte Feld enthält die Größe des Gleitfensters. Der Empfänger gibt dem Sender hiermit an, wieviele Zeichen er akzeptieren kann, bevor er eine weitere Bestätigung schickt. Im Bild hat der Empfänger Byte 1 bis 6 bestätigt und eine Fenstergröße von 11 Byte mit der letzten angekommenen Bestätigung an den Sender zurückgegeben. Der Sender darf also Byte 7 bis 17 senden.



Mit dem Sliding Window Protocol hat TCP die Flusskontrolle des Datenstroms. Die Algorithmen zur Anpassung der Fenstergröße stellen sicher, dass die Netzbandbreite gut genutzt wird und dass der Empfänger keine Daten fallen lassen muss.

Der Aufbau eines TCP-Datagramms ist wie folgt:

Absendeport	Empfangsport	Sequenznummer	Bestätigungsnummer	Markierungen
Bits: 16	16	32	32	16

Fenstergröße	Prüfsumme	Wichtigkeitszeiger	Optionen	Anwendungsdaten
16	16	16	variiert	variiert

Die Eigenschaftskombinationen von UDP und TCP sind natürlich von den Designern gewählt worden. Andere Kombinationen wären möglich, z.B. ein ver-

bindungsloser Datagramm-Dienst, der die Reihenfolge der Datagramme einhält aber die Richtigkeit der Daten nicht garantiert.

Kapitel 3

C-Programmierung mit Sockets

3.1 Sockets und Ports

Die ursprüngliche Programmierschnittstelle zu TCP und UDP stammt aus der UNIX-Version BSD 4.2. Sie bestand aus 8 Systemaufrufen und wurde mit Sockets bezeichnet (Socket bedeutet Steckdose). Ein Socket ist ein Kommunikationsendpunkt, d.h. die Stelle an der sich das Anwendungsprogramm und der Transportdienst des Netzwerks treffen.

Sockets sind nicht nur für TCP und UDP des Internet sondern auch für andere Protokolle wie Xerox NS spezifiziert. Sockets haben daher eine Adressfamilie z.B. AF_INET für die Internet-Protokolle TCP und UDP, AF_UNIX für Protokolle auf ein- und demselben UNIX-Rechner (kein Netzwerk) und AF_NS für die Xerox NS-Protokolle.

Weiterhin haben Sockets einen Typ, der den mit ihnen verbundenen Transportdienst anzeigt. Die wichtigsten Socket-Typen sind RAW-, DATAGRAM- und STREAM-Sockets. RAW-Sockets wollen wir hier nicht weiter besprechen, sie erlauben für Spezialfälle einen direkten Zugriff auf das Netzwerkprotokoll. Für die beiden anderen Socket-Typen werden unterschiedliche Datenstrukturen z.B. in C definiert:

UNIX-Domain-Socket-Adresse:

```
struct sockaddr_un {
       short sun_family;    /* Tag: AF_UNIX */
       char  sun_path[108]; /* path-name */
};
```

Internet-Domain-Socket-Adresse:

```
struct sockaddr_in {
       short   sin_family; /* Tag: AF_INET */
       u_short sin_port; /* Port number */
       struct  in_addr sin_addr;
       /* IP-address */
       char    sin_zero[8]; /* Padding */
};
```

In der Ursprungsmarkierung, dem Tag, muss bei beiden Strukturen der passende Wert eingetragen werden. Das muss sein, damit Funktionen wie bind(), die nur einen Zeiger auf die Struktur übergeben bekommen, erfahren können um welchen Adresstyp es sich handelt. Die Struktur sockaddr_in beinhaltet die Struktur in_addr, die eine Internet-Adresse enthält.

```
struct in_addr {
       u_long s_addr;
};
```

Es existiert auch eine generische Socket-Adress-Struktur, die verwendet wird, um beide Socket-Adress-Typen an Funktionen weiterzugeben:

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

Wenn eine Socket-Adresse an einen Systemaufruf weitergegeben wird, muss zusätzlich ein weiterer Parameter übergeben werden, der die Länge der Socket-Adresse enthält. Ein bind()-Aufruf sieht wie folgt aus:

```
struct sockaddr_in server;
int sock;
bind(sock, (struct sockaddr *)&server,
     sizeof(server));
```

Hierbei wird der cast-Operator des zweiten Parameters benutzt, um den Compiler davon zu überzeugen, dass ein generischer Socket-Adressen-Zeiger übergeben wird. Außer der Adressfamilie besitzen Sockets einen Typ, der die Art des Transportprotokolls bestimmt:

```
SOCK_STREAM /* TCP-Socket */
SOCK_DGRAM /* UDP-Socket */
SOCK_RAW /* direkter Zugriff auf die IP-Schicht */
```

3.2 Verbindungsorienterte Clients und Server

3.2.1 Verbindungsaufbau und Datentransfer

Die Client/Server-Beziehung ist asymmetrisch, d.h. der Server wartet passiv auf Arbeit und er weiß nicht von welchem Client die nächste zu erledigende Arbeit kommt. Ein Client geht aktiv auf die Suche nach einem Server, der seine Arbeit erledigt. Diese Asymmetrie spiegelt sich auch in den Socket-Systemaufrufen wider. Vom Server werden folgende Aktionen durchgeführt:

- Ein Socket der benötigten Adressfamilie und des benötigten Typs wird angelegt:

```
int sock;
sock = socket(AF_INET, SOCK_STREAM, 0);
```

Das dritte Argument des socket-Systemaufrufs teilt mit, welches Protokoll vom Socket benutzt werden soll. Neben TCP und UDP gibt es z.B. ICMP in der Internet-Protokollfamilie. 0 bedeutet hier, dass das System das zugehörige Protokoll selbst auswählt.

- Es wird eine Adresse an den Socket gebunden. Beim Adresstyp AF_INET besteht sie aus einer IP-Adresse und einer Portnummer. Alle drei Felder müssen in die Struktur sockaddr_in gesetzt werden. Der Programmausschnitt dazu sieht z.B. wie folgt aus:

```
#define SERVER_PORT 4321
struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(SERVER_PORT);
bind(sock,(struct sockaddr *)&server,
     sizeof(server));
```

Die Portnummer ist frei wählbar; sie muss aber mit dem Client abgestimmt sein. Portnummern kleiner als 1024 können in UNIX nur von Prozessen mit Superuser-Privilegien benutzt werden. Dies sind sogenannte reservierte Ports, die als eine Art Verifizierer angesehen werden. Der Client wird über reservierte Ports nur offizielle Server erwarten. Durch diesen Mechanismus ist natürlich nur sichergestellt, dass der Server ein Superuser-Prozess ist.

Allgemein sollten für benutzerdefinierte Server die eingetragenen Portnummern in der Datei services vermieden werden, da diese von den Standard-Internet-Servern wie telnetd, ftpd usw. verwendet werden. Die Funktion htons wandelt die

Portnummer von der Host- zur Netzwerk-Byte-Ordnung um. Es gibt vier verschiedene Makros zur Umwandlung von short- oder long-integer Zahlen vom Host- in das Netzwerk-Format und umgekehrt:

```
u_short htons(u_short hostshort):
        host to network short (16 Bit)
u_long  htonl(u_long  hostlong) :
        host to network long  (32 Bit)
u_short ntohs(u_short netshort) :
        network to host short (16 Bit)
u_long  ntohl(u_long  netlong)  :
        network to host long  (32 Bit)
```

Auf den meisten Systemen tun diese Makros nichts, da die Byte-Anordung von Host und Netzwerk übereinstimmend big endian sind. Auf einem Rechner mit little endian-Darstellung würde allerdings ohne diese Funktionen ein Fehler entstehen. Das Netzwerkprotokoll verwendet in seinen Headern nur Integer-Zahlen und keine Gleitkomma-Zahlen, deren Speicherung auf unterschiedlichen Rechner-Architekturen noch verschiedenartiger als die von Integer-Zahlen ist. Der IP-Adresse des Socket wird eine Art Metazeichen mit der Konstanten INADDR_ANY zugewiesen. Dies bedeutet, dass das Socket von jeder Netzwerkkarte des Rechners eingehende Verbindungsanfragen akzeptiert. Bei einem Gateway mit mehreren Netzwerkkarten, kann es auch sinnvoll sein, eine IP-Adresse einer speziellen Netzwerkkarte anzugeben, über die dann ausschließlich kommuniziert wird. Diese angegebene IP-Adresse hat dann natürlich nichts damit zu tun, von welchen Clients Verbindungen akzeptiert werden.

- Als nächstes informiert man den Kernel davon, dass Verbindungsanfragen am Socket akzeptiert werden sollen:

  ```
  listen(sock, 5);
  ```

Das zweite Argument legt fest, wieviele anstehende Verbindungsanfragen in eine Warteschlange gestellt werden können, bevor Verbindungsanfragen abgewiesen werden. D.h. solche Verbindungsanfragen werden gepuffert, wenn der Server mit einem anderen Client kommuniziert.

- Der letzte Schritt des Servers besteht aus dem Warten auf eine Verbindungsanfrage und dem Akzeptieren der Anfrage des Client:

  ```
  struct sockaddr_in client;
  int fd, client_len;
  client_len = sizeof(client);
  fd = accept(sock, &client, &client_len);
  ```

Der Aufruf von accept() stoppt den Server-Prozess, bis ein Client Verbindung aufgenommen hat. Das zweite Argument zeigt auf eine Struktur, in der die Socket-Adresse des anfragenden Client zurückgegeben wird. Hierdurch kann der Server die IP-Adresse des Client und dessen Port prüfen. Der Server kann hierbei z.B. einen DNS-Server befragen, welcher Name sich hinter der IP-Adresse verbirgt und anschließend auswerten, ob der Rechner mit diesem Namen und der Port als Client zugelassen ist. Der NFS-Server wertet auf diese Weise mit Hilfe der Datei exports, in der die Client-Rechner eingetragen sind, mount-Versuche der anderen Rechner aus und lehnt sie ggfs. ab.

Das dritte Argument ist ein Wert-Rückgabe-Parameter (value return). Es wird die Länge der sockaddr_in-Struktur an die Funktion accept() übergeben und accept() gibt die Anzahl eingetragener Bytes des zweiten Arguments im dritten Argument zurück. Der Rückgabewert von accept() ist ein neuer Deskriptor, der sich wie ein herkömmlicher Dateideskriptor verhält, d.h. auf ihn können z.B. read- und write-Aufrufe ausgeführt werden.

Es gibt also zwei Deskriptoren eines Socket, einen Rendezvous-Deskriptor, der Verbindungen akzeptiert (hier: sock) und einen Verbindungs-Deskriptor für die Kommunikation mit einem Client. Die Verbindung zwischen Client und Server ist also durch 5 Merkmale bestimmt:

1. das verwendete Protokoll,
2. die IP-Adresse des Client,
3. die Portnummer des Client,
4. die IP-Adresse des Server,
5. die Portnummer des Server.

Ist irgendeins dieser Merkmale unterschiedlich, so handelt es sich um eine andere Verbindung. Ein zweiter Client kann mit dem Server Kontakt aufnehmen, wenn er auf einem anderen Rechner arbeitet (IP-Adresse des Client) oder wenn er eine andere Portnummer als der erste Client verwendet.

Nach dem Aufbau der Verbindung verhält sich der Deskriptor fd wie jeder andere Dateideskriptor. Man kann auf ihm Operationen wie read(), write(), dup(), close() usw. ausführen. Um einen stdio-Strom zu erhalten, kann fdopen() auf den Socket-Deskriptor fd aufgerufen werden. Anschließend kann sogar mit Funktionen zum formatierten Lesen und Schreiben gearbeitet werden, wie fscanf() und fprintf(). Allerdings muss man an die Auswirkungen der Pufferung in der stdio-Bibliothek denken (Funktionen setbuf(), fflush()).

Der Dialog zwischen Client und Server ist vom Anwendungsprotokoll abhängig. Üblicherweise werden Anfragen des Client mit Anworten des Servers bedient. Häufig beinhaltet das Protokoll einen Hinweis darauf, dass der Dialog abgeschlossen ist. Der Client liefert eine Kennung dafür, dass er keine Anfragen an den Server mehr hat. Schließt der Client oder der Server die Verbindung hingegen ohne Vorwarnung mit close(), so erhält der Server oder der Client beim nächsten Lesen auf dem Deskriptor eine EOF-Meldung (EOF = End of File).

Ein Server kann mehrere Clients nacheinander bedienen, indem er in einer Schleife die Verbindung zu einem Client schließt und den nächsten Client bedient:

```
struct sockaddr_in client;
int fd, client_len;
char inbuf [1024];
char outbuf[1024];
while (1) {
      fd = accept(sock, &client, &client_len);
      while (read(fd, inbuf, 1024) > 0) {
      /* Daten aus inbuf bearbeiten und
         Antwort in outbuf schreiben.
       */
          ...
          write(fd, outbuf, 1024);
      }
      /* Wenn der Client seinen Endpunkt
         mit close() geschlossen hat,
         schließt der Server auch fd,
         sonst würden ggfs. zu viele offene
         Deskriptoren im Server entstehen.
      */
      close(fd);
}
```

Dieser Servertyp wird als iterativer Server bezeichnet. Für Server, die nur kurze Verbindungen zu Clients aufbauen ist ein iterativer Ansatz vernünftig. Server, die länger mit Clients ununterbrochen zusammenarbeiten, müssen jedoch mehrere Clients parallel bedienen können.

Üblicherweise braucht man spezielle Client-Programme für die Arbeit mit einem Server. Bestehende Clients wie telnet reichen hier nicht aus. Um ein Client-Programm zu schreiben, sind folgende Schritte notwendig:

- Wie im Server wird im Client zunächst ein Socket erzeugt:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
```

- Im Gegensatz zum Server muss der Client nicht explizit eine Adresse an den Socket binden. In diesem Fall vergibt das System automatisch eine Portnummer an den Socket. Das geht natürlich nicht, wenn ein bestimmter Port gebunden werden soll, z.B. eine reservierte Portnummer:

```
#define CLIENT_PORT 1000
struct sockaddr_in client;
client.sin_family = AF_INET;
client.sin_addr.s_addr = INADDR_ANY;
client.sin_port = htons(CLIENT_PORT);
bind(sock, (struct sockaddr *)&client,
     sizeof(client));
```

- Im nächsten Schritt wird der Socket des Client mit dem Socket des Server verbunden. Dazu dient der Systemaufruf connect(). Einer der Parameter ist eine sockaddr_in-Struktur mit der Adresse des Servers, die aus folgenden Bestandteilen zusammengesetzt ist:
  - Adresstyp-Kennzeichen AF_INET
  - Portnummer des Server
  - IP-Adresse des Server

Während die Portnummer in das Client-Programm meist konstant eingebaut werden kann, ist dies für die IP-Adresse des Servers meist nicht sinnvoll, da die meisten Serverprozesse alternativ auf verschiedenen Rechnern arbeiten können. (Bei konstanter IP-Adresse müsste es dann für jeden Server-Rechner ein spezielles Client-Programm geben.) Daher liest der Client normalerweise den Namen des Server-Rechners ein. Dieser Rechnername wird dann mit der Datei hosts oder durch einen DNS-Server auf die IP-Adresse abgebildet. Hierzu dient die Funktion gethostbyname(), die aus einem Rechnernamen u.a. dessen IP-Adresse ermittelt. Genau genommen liefert gethostbyname() einen Zeiger auf eine Struktur vom Typ hostent oder im Fehlerfall NULL zurück:

```
struct hostent {
  char *h_name; /* offizieller Hostname */
  char **h_aliases; /* Alias-Namen-Liste */
  int h_addrtype; /* z.B. AF_INET */
  int h_length; /* Byte-Länge der Adresse */
  char **h_addr_list; /* Hostadressen-Liste */
};
```

Normalerweise braucht man nur die erste Hostadresse der Liste. Zu diesem Zweck gibt es das Makro:

```
#define h_addr h_addr_list[0];
```

Analog zu den Rechnernamen werden auch gängige Services, durch Servicenamen bezeichnet. Diese Namen sind mit den zugehörigen Portnummern der Server in einer Datei oder in der NIS-Diensttabelle, beide mit dem Namen services, verzeichnet. Die Funktion getservbyname() liefert analog zu gethostbyname() eine Struktur zurück, deren wichtigster Bestandteil die Portnummer des Servers ist:

```
struct servent {
  char *s_name; /* offizieller Servicename */
  char **s_aliases; /* Alias-Namen-Liste */
  int s_port; /* Portnummer des Service */
  char *s_proto; /* Protokoll des Service */
}
```

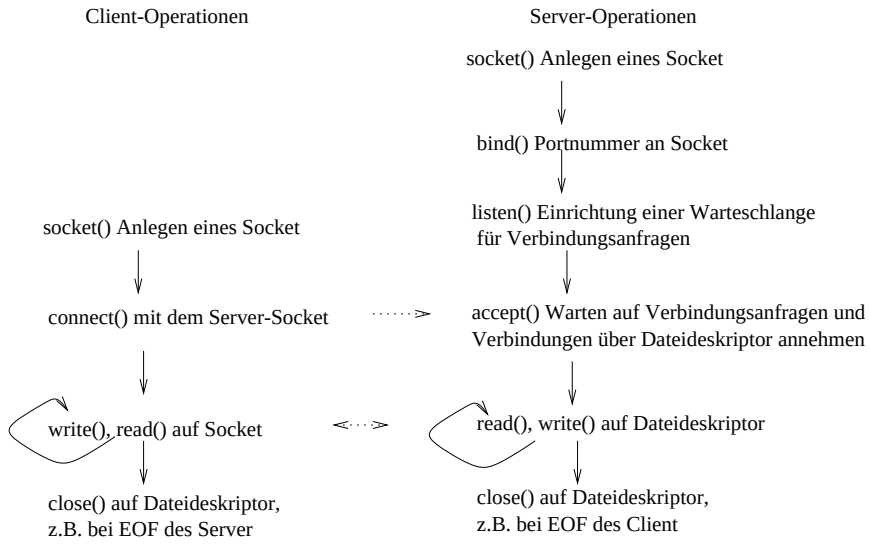
Der folgende Programmausschnitt verbindet den Client schließlich mit dem Server:

```
struct hostent *host_info;
struct servent *serv_info;
struct sockaddr_in server;
if ((host_info = gethostbyname(argv[1]))
     == NULL) {
   fprintf(stderr, "Rechner %s nicht
           gefunden\n", argv[1]);
   exit(1);
}
memcpy(&server.sin_addr, host_info->h_addr,
       host_info->h_length);
if ((serv_info=getservbyname(argv[2],"tcp"))
     == NULL) {
   fprintf(stderr, "Service %s nicht
           gefunden.\n", argv[2]);
   exit(2);
}
server.sin_port = serv_info->s_port;
if (connect(sock,&server,sizeof(server))<0) {
   fprintf(stderr, "Fehler beim connect.\n");
   exit(3);
}
```

Die Hostadresse h_addr wird auf die Stelle &server.sin_addr kopiert und zwar mit memcpy(), damit das Programm auch für andere Adresslängen als 4-Byte-IP-Adressen funktioniert. Es werden nämlich genau h_length Bytes kopiert. Für IPv4-Adressen genügt auch die Anweisung:

```
server.sin_addr = *(long *)host_info->h_addr;
```

Zusammenfassend sind also folgende Operationen in Client und Server zum Aufbau einer verbindungsorientierten Kommunikation notwendig:



Während der Server also über mindestens zwei Deskriptoren verfügt, den Rendezvous-Deskriptor und mindestens einen Verbindungs-Deskriptor, besitzt der Client genau einen Deskriptor.

Programmausschnitt im Client:

```
int sock;
connect(sock, (struct sockaddr*)&server,
        sizeof(server));
count = read(sock, inbuf, MAXLEN);
...
write(sock, outbuf, strlen(outbuf));
```

Programmausschnitt im Server:

```
int fd, client_len;
client_len = sizeof(client);
fd = accept(sock, &client, &client_len);
write(fd, outbuf, strlen(outbuf));
...
count = read(fd, inbuf, MAXLEN);
```

3.2.2 Parallele Mehrprozess-Server

Parallele Server können mehrere Clients simultan bedienen. Solche parallelen Server können wahlweise über einen oder mehrere Prozesse realisiert sein. Im letzteren Fall sprechen wir von parallelen Mehrprozess-Servern.

Mehrprozess-Server sind z.B. in UNIX einfach zu realisieren, da die Zustandsinformation pro Client in einem separaten Prozess gehalten werden kann. In einem ersten naiven Ansatz erzeugt der Server für jeden seiner Clients einen separaten Kindprozess. Der Elternprozess nimmt nur Verbindungen mit accept() entgegen und übergibt sie dann an den neu erzeugten Kindprozess. Da ein Kindprozess die gesamte Umgebung des Elternprozesses, also insbesondere auch die Dateideskriptoren, erbt, ist dies sehr einfach zu implementieren:

```
sock = socket(...);
bind(sock,...);
listen(sock, 2);
while (1) {
  fd = accept(sock, ...);
  if (fork() == 0) {
  /* Kindprozess bearbeitet den Request */
     ...
     exit(0);
  } else close(fd); /* Elternprozess */
}
```

Der Elternprozess schließt den Verbindungs-Deskriptor, nachdem die Verbindung des Client an den Kindprozess übergeben wurde, damit er im Laufe der Zeit nicht zu viele offene Deskriptoren enthält.

Dieses naive Server-Programm hat jedoch einen Fehler: Wenn man sich z.B. mit dem Befehl ps nach dem Ende einiger Client-Sitzungen die Prozesstabelle des Server-Rechners anschaut, sieht man, dass Zombie-Prozesse entstanden sind. Das liegt daran, dass der Elternprozess keinen wait()-Systemaufruf auf einen beendeten Kindprozess macht. Ein solcher wait()-Aufruf, mit dem der Elternprozess die exit-Zustände der Kindprozesse auswertet, ist in UNIX jedoch vorgeschrieben.

Das Problem ist nur, dass wait() den Server-Prozess blockieren würde. Dies darf jedoch nicht sein, da der Server jederzeit Verbindungsanfragen entgegen nehmen muss.

Zombie-Prozesse belegen zwar nur Platz in der Prozesstabelle des Betriebssystems, jedoch hat diese eine feste Größe, so dass keine neuen Prozesse mehr erzeugt werden können, wenn sie vollständig belegt ist. Erst wenn der Elternprozess der Kindserver beendet wird, erbt der Prozess mit dem Namen init alle noch auf Rückgabe ihrer Exit-Zustände wartenden Kindprozesse und beendet die Kindprozesse durch wait()-Aufrufe.

Um solche Zombie-Zustände der Kindprozesse zu vermeiden, muss ein asynchrones wait() realisiert werden, dass nur bei Beendigung eines Kindprozesses aufgerufen wird, so dass der Server weiterarbeiten kann. Hierfür gibt es je nach UNIX-Version zwei Möglichkeiten:

- In BSD-UNIX kann ein Interrupt-Handler für das Signal SIGCHLD installiert werden, das beim Beenden eines Kindprozesses an den Elternprozess gesendet wird. Der Interrupt-Handler ruft dann wait() auf den Kindprozess auf und reinstalliert sich. Allerdings gibt es noch ein Problem, da Systemaufrufe wie accept(), read() usw. ebenfalls durch Signale unterbrochen werden. Sie liefern dann den Fehlercode EINTR in der Variablen errno zurück, worauf dann ein erneutes accept(), read() usw. durchgeführt werden kann. Hier die Erweiterungen des Server-Programms:

```
#include <signal.h>
void waiter() { /* Signal handler */
  wait(0);
  signal(SIGCHLD, waiter); /* Reinstallation */
}
main() {
  ...
  signal(SIGCHLD, waiter); /* Installation */
  while ((fd = accept(...)) < 0) {
    if (errno != EINTR) {
      /* exit wird nur aufgerufen, wenn kein
         Signal angekommen ist.
       */
      perror("accepting connection");
      exit(1);
    }
  }
}
```

- In UNIX System V.4 ist es möglich, einen vordefinierten Signal-Handler zu benutzen, der alle obigen BSD-Operationen ersetzt:

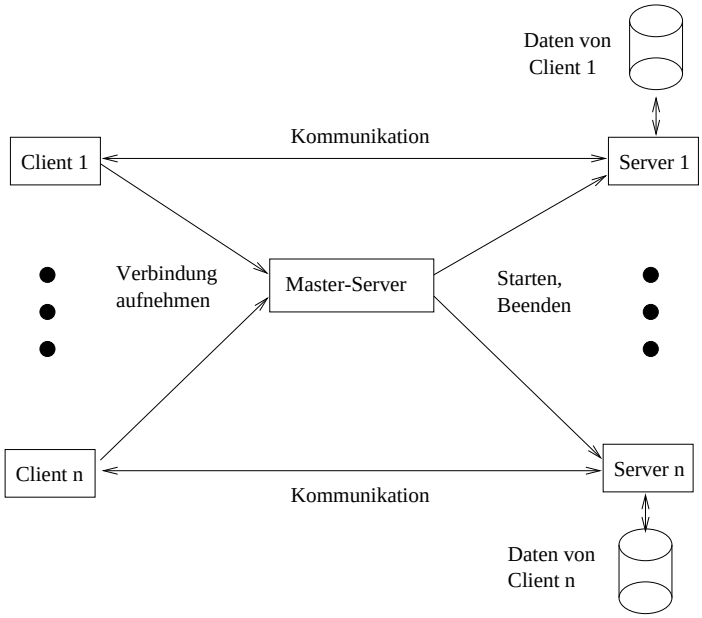
```
#include <signal.h>
signal(SIGCHLD, SIG_IGN);
```

3.2.3 Parallele Einprozess-Server

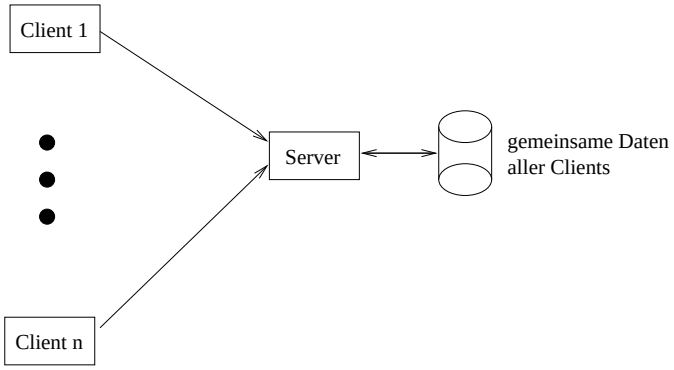
Der Vorteil der Einprozess-Lösung gegenüber der Mehrprozess-Lösung besteht darin, dass weniger Systemressourcen verwendet werden. Der Hauptnachteil ist, dass in einem Prozess sämtliche Statusinformationen aller Clients gehalten werden müssen, was einen erheblichen Programmieraufwand bedeuten kann.

Je nach der Anwendung kann es aber vorteilhafter sein, mehrere Clients durch einen Serverprozess oder durch mehrere Serverprozesse zu bedienen. Dabei können folgende Fälle unterschieden werden.

- Die Clients haben nur private und keine gemeinsamen Daten im Server. Dies ist der klassische Anwendungsfall für einen Mehrprozess-Server, da zwischen den Kindservern kein Datenaustausch notwendig ist.



- Die Clients haben keine privaten sondern nur gemeinsame Daten im Server. Dies ist der klassische Fall für einen Einprozess-Server, da ein Mehrprozess-Server hier erheblichen Zusatzaufwand erfordert, um mit mehreren Server-Prozessen auf gemeinsame Daten z.B. über shared memory zuzugreifen. Operationen müssen atomar sein, d.h. angefangen Operationen müssen ohne Störung z.B. durch andere Operationen zu Ende gebracht werden. Dies muss z.B. durch Semaphore oder Dateisperren realisiert werden. Dieser Aufwand kann bei der Einprozess-Lösung eingespart werden, da der Serverprozess die Operationen der Clients nacheinander auf den globalen Daten ausführen kann.



- In der Praxis ist der Fall der Mischung von globalen und lokalen Daten der Clients in den Servern häufig anzutreffen. Dann muss abgewogen werden, welche der Lösungen günstiger ist.

Für parallele Einprozess-Server gibt es den Systemaufruf select(), um zwischen den von Clients empfangenen Nachrichten zu multiplexen. Der Einsatz von select() prägt den Kontrollfluss im Server. Nach jeder Interaktion mit einem Client muss in die Hauptschleife zurückgekehrt werden. Die Syntax von select() ist wie folgt:

```
select(int nfds, fd_set *readfds,
       fd_set *writefds, fd_set *exceptfds,
       struct timeval *timeout);
```

Beim Aufruf blockiert select() solange, bis irgendeiner der Deskriptoren eines vordefinierten Deskriptorensatzes von readfds lesebereit, writefds schreibbereit oder wenn an einem Deskriptor von exceptfds eine Ausnahmebedingung anhängig

ist. nfds ist der größte zu überprüfende Deskriptor. Mit dem timeout-Parameter kann eine Zeit gesetzt werden, nach der der Aufruf von select() auf jeden Fall aus der Blockade zurückkehrt. Ist timeout = NULL, so gibt es keinen Timeout des select().

Die genaue Anzahl verfügbarer Deskriptoren ist systemabhängig; sie werden daher im Typ fd_set gekapselt. (Die Anzahl der Dateideskriptoren kann mit der Funktion getrlimit() zur Laufzeit des Programms ermittelt werden.) Die zu überprüfenden Deskriptoren werden an select() übergeben und dazu mit einem Makro FD_SET in die Menge der zu übergebenden Deskriptoren aufgenommen:

```
FD_SET(fd, &fdset);
```

Wenn der Aufruf von select() zurückkehrt, sind die Deskriptoren gesetzt, an denen ein Ereignis zur Bearbeitung anliegt. Mit dem Makro FD_ISSET kann überprüft werden, an welchem Deskriptor bzw. welchen Deskriptoren Ereignisse anliegen:

```
if (FD_ISSET(fd, &fdset)) {...}
```

Zur Überprüfung auf Ereignisse werden durch select() die übergebenen Deskriptorensätze vor der Rückkehr von select() überschrieben. Einmal aufgebaute Verbindungen zu Clients müssen daher in einem getrennten Feld von Deskriptoren als Kopie gehalten werden, da sie sonst durch select() zerstört werden können:

```
fd_set permanent_set, ready_set;
memcpy(&ready_set, &permanent_set,
       sizeof(permanent_set));
select(max_fd, &ready_set, NULL, NULL, NULL);
```

Es folgt ein kleines Programmbeispiel, in dem die Deskriptoren 3 und 4 des readfds gesetzt werden.

```
#include <sys/types.h>
fd_set readfds;
/* Put descriptors 3 and 4 into readfds. */
FD_ZERO(&readfds);
FD_SET(3, &readfds);
FD_SET(4, &readfds);
/* Control descriptors 0 to 4. */
select(5, readfds, NULL, NULL, NULL);
if (FD_ISSET(3, &readfds)) {
  /* Read from descriptor 3. */
  ...
```

```
}
if (FD_ISSET(4, &readfds)) {
  /* Read from descriptor 4. */
  ...
}
```

Um das Hauptprogramm des Einprozess-Servers zu schreiben, braucht man noch zwei weitere Makros zur Manipulation der Deskriptor-Sätze:

```
/* Löschen aller Deskriptoren in fdset */
FD_ZERO(&fdset);
/* Löschen des Deskriptors fd in fdset */
FD_CLR(fd, &fdset);
```

Hier nun der neue Teil des Hauptprogramms:

```
max_fd = sock;
while (1) {
  memcpy(&ready_set, &permanent_set,
         sizeof(permanent_set));
  select(max_fd+1,&ready_set,NULL,NULL,NULL);
  if (FD_ISSET(sock, &ready_set)) {
     /* Verbindungsanfrage eines Client */
     client_len = sizeof(client);
     fd = accept(sock, &client, &client_len);
     FD_SET(fd, &permanent_set);
     if (fd > max_fd) max_fd = fd;
  }
  for (fd = 0; fd <= max_fd; fd++)
    if ((fd != sock) &&
        FD_ISSET(fd, &ready_set)) {
       /* Request an fd bearbeiten */
       ...
    }
}
```

Wenn ein Client sich abmeldet wird sein Deskriptor aus dem permanent_set entfernt und geschlossen:

```
close(fd);
FD_CLR(fd, &permanent_set);
```

3.3 Verbindungslose Clients und Server

Verbindungslose Clients und Server mit dem TCP-Protokoll UDP werden hauptsächlich innerhalb von lokalen Netzen eingesetzt. Hier gibt es keine Reihenfolgeprobleme von Datagrammen durch Routing und i.d.R. auch keine verlorenen oder falschen Daten. Verliert oder verfälscht ein LAN Daten, so ist kein Protokoll mit Übertragungswiederholung notwendig, sondern eine Mitteilung an den Netzwerkadministrator, das Problem zu finden und zu beheben.

Die Asymmetrie der Socketaufrufe zwischen Client und Server ist im verbindungslosen Fall viel weniger auffällig als beim verbindungsorientierten Fall. Wer Client und Server ist, wird durch das Anwendungsprotokoll bestimmt. Der Client sendet ein Datagramm, auf das der Server antwortet.

Verbindungslose Clients und Server beginnen mit dem Anlegen eines Datagramm-Sockets:

```
int sock;
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

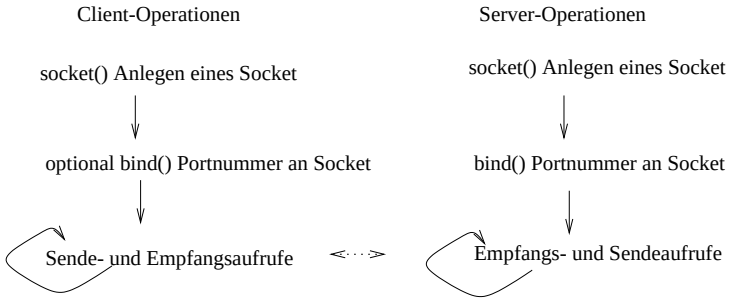
Während der Server eine dem Client wohl bekannte (well known) Portnummer an den Socket binden muss, ist dies für die Clients nicht ausdrücklich gefordert. Wenn ein Client versucht, ein Datagramm zu übertragen, und keinen gebundenen Socket vorfindet, so bindet das Betriebssystem automatisch eine Portnummer an den Socket. Dies ist allerdings nur sinnvoll, wenn vom Server nicht eine bestimmte Portnummer, z.B. ein reservierter Port, gefordert wird.

Verbindungslose Sockets können nicht mit read()- und write()-Aufrufen zusammenarbeiten. Es gibt spezielle Funktionen, die bei jedem Aufruf als Parameter die Zieladresse, also Port- und IP-Nummer, des Empfängers verwenden. Die am klarsten strukturierten Aufrufe sind sendto() und recvfrom():

```
sendto(sock,buffer,count,flags,&addr,addrlen);
/* addr ist die Adress-Struktur
   des Empfängers,
   z.B. vom Typ sockaddr_in.
 */
recvfrom(sock,buffer,count,flags,&addr,&addrlen);
/* addr ist die Adress-Struktur des Senders,
   z.B. vom Typ sockaddr_in.
 */
```

Bei recvfrom() wird die Adresse eines Senderprozesses vom Empfänger eingetragen und gewartet, bis der Sender etwas geschickt hat. Natürlich gibt es für die Senderadresse auch Wildcards.

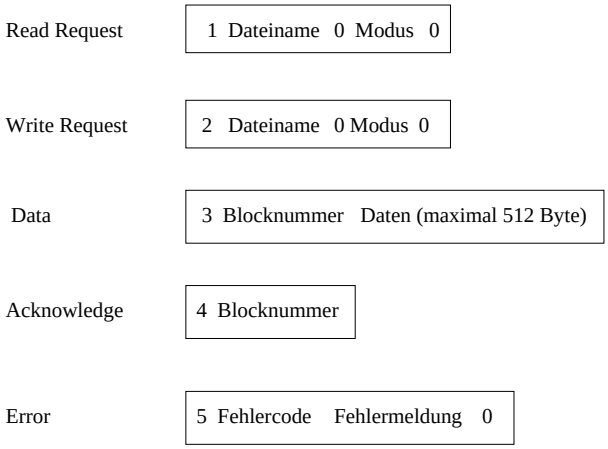
Neben sendto() und recvfrom() gibt es zwei weitere Aufrufe, bei denen die Adresse des Kommunikationspartners weggelassen werden kann, weil sie vorher mit einem connect()-Aufruf vereinbart wurde. Die Client- und Server-Operationen für die verbindungslose Kommunikation sehen prinzipiell wie folgt aus:



Als Beispiel für einen verbindungslosen Dienst werden wir eine externe Version des UNIX-Befehls cat implementieren, das wie folgt aufgerufen wird:

```
rcat <IP-Name> <Datei>
Beispiel: rcat monet /etc/passwd
```

rcat ist ein Client, der den tftp-Server nutzt, d.h. das tftp-Protokoll verwendet (trivial file transfer protocol). Das tftp-Protokoll definiert 5 Datenpakettypen:

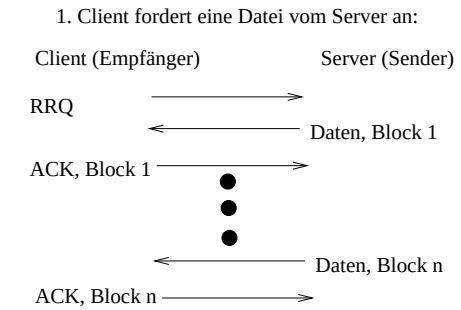


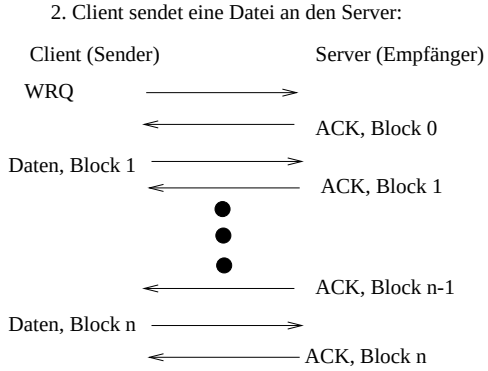
Jedes Datenpaket beginnt mit einem 2-Byte-Operations-Code, dessen Wert den Pakettyp anzeigt. Ein Datentransfer wird vom Client ausgelöst. Er sendet einen Lese- oder Schreibaufruf. Neben dem mit einem binären 0-Byte beendeten

Dateinamen enthält ein Read- oder Write-Request einen Modus für den Datentransfer.

Es gibt zwei Formate für Dateiübertragung netascii und octet. Bei netascii wird zwischen Client und Server der Standard-ASCII-Zeichensatz verwendet. Das Konvertieren in und aus dem netascii-Format übernehmen Client und Server. Für viele IBM-Systeme muss eine Datei z.B. von EBCDIC nach netascii konvertiert werden. Das octet-Format wird für die Übertragung binärer Dateien benutzt. Die beiden Hauptanwendungen für binären Dateitransfer sind Rechnerarchitekturen mit gleichen Binärformaten und Dateiserver, die den Inhalt von Client-Dateien nicht zu interpretieren brauchen.

Es gibt zwei vorgesehene Operationsarten von tftp, die ohne Berücksichtigung der Fehlerbehandlung wie folgt arbeiten:





Verlorene Datenpakete werden vom Sender nach einer Maximalzeit, nach der kein Acknowledge-Paket angekommen ist, erneut gesendet. Je nach Operationsmodus kann der Client oder der Server Sender sein. Geht nur das Acknowledge-Paket verloren und wurde das Datenpaket richtig übermittelt, so stellt der Empfänger anhand der Blocknummer fest, dass es sich bei dem erneut gesendeten Datenpaket um ein Duplikat handelt. Das Duplikat wird ignoriert und das Acknowledge-Paket erneut gesendet.

Da es keine Authentifikation eines Client beim tftp-Server gibt, müssen die Dateizugriffsrechte des tftp-Servers restriktiv verwaltet werden. Normalerweise wird der tftp-Server unter einem Nutzer, der sehr geringe Rechte hat ausgeführt, z.B. nobody. Er hat dann in UNIX i.w. nur die others Berechtigung bzgl. Lese- und Schreibzugriffen.

Als Kommunikationsprotokoll verwendet tftp meist UDP; es gibt jedoch auch TCP-Implementationen. Da wenige Dateien so klein sind, dass sie über ein einziges Datenpaket übermittelt werden können, muss auch in UDP ein Weg zwischen Client und Server eingerichtet werden. Dies geschieht in folgender Weise:

Mehrere Clients können mit dem tftp-Server über den wohl bekannten Port 69 Kontakt aufnehmen. Der tftp-Server ist also als paralleler Server implementiert. Dabei wird Port 69 nur für das Rendezvous von Client und Server verwendet. Für den Dialog mit dem Client legt der Server ein neues Socket mit einer anderen Portnummer an. Die neue Portnummer wird mit dem Empfang des ersten Datenpaketes mit recvfrom() vom Server an den Client in der Server-Struktur übergeben.

Bei einer Leseanfrage sendet der Server als Antwort eine Reihe von Datenpaketen an den Client. Jedes Datenpaket enthält entweder genau 512 Byte Daten, nur das letzte Datenpaket enthält 0 bis 511 Byte. Dieses kürzere Datenpaket signalisiert das Ende des Datenstroms. Der Client muss jeden Block durch das Senden eines Acknowledge-Pakets bestätigen.

Einen trivialen tftp-Client wie rcat zu implementieren macht mit diesen Kenntnissen wenig Schwierigkeiten: rcat schickt ein RRQ-Paket und erhält daraufhin Datenpakete, die es mit Bestätigungspaketen an den Server beantwortet. Der einzige schwierigere Teil ist der Aufbau des RRQ-Pakets. Hier kann wegen der variablen Längen der Zeichenketten keine Struktur eingesetzt werden, sondern es müssen Kopieroperationen in einen Puffer benutzt werden:

```
char buffer[600], *p;
*(short *)buffer=htons(OP_RRQ); /* OP-Code */
p = buffer + 2;
strcpy(p, argv[2]); /* Dateiname */
p += strlen(argv[2]) + 1;
strcpy(p, MODE); /* Modus */
p += strlen(MODE) + 1;
count = sendto(sock,buffer,p-buffer,0,
              &server,sizeof(server));
```

Der triviale tftp-Client rcat trifft keine Vorkehrungen, um verlorene Datagramme wiederherzustellen. Wenn eine Bestätigung vom Client an den Server oder ein Datenpaket vom Server an den Client verloren geht, wird der Client für immer

blockieren. Es muss also eine Zeitüberwachung mit Wiederholungsmöglichkeit des Sendens implementiert werden. Um das Prinzip auch in einem intakten LAN zu zeigen, werden wir künstlich 25% aller Bestätigungspakete verlieren:

```
if (rand() % 4) sendto(...);
```

Ein Ansatz, die Datagramm-Leseoperation in ein Zeitlimit laufen zu lassen, sieht folgendermaßen aus: Der Client-Socket, über den die Kommunikation mit dem Server abläuft, wird mit folgendem Systemaufruf als nicht-blockierend (non-blocking) initialisiert:

```
fcntl(sock, F_SETFL, FNDELAY);
```

Die Konstanten sind in <sys/fcntl.h> definiert. Wenn dies geschehen ist, kehrt recvfrom() sofort mit dem Returnwert -1 zurück, wenn kein Datagramm angekommen ist. Dabei ist die globale Variable errno auf EWOULDBLOCK gesetzt. Man kann nun z.B. in einer Schleife eine maximale Anzahl von recvfrom()-Aufrufen mit diesem Fehler akzeptieren, bevor man erneut das Datagramm an den Server sendet. Dieses Verfahren liefert jedoch kein exaktes Zeitlimit und es ist sehr rechenzeitaufwendig. Deswegen implementieren wir ein anderes Verfahren:

Wir lassen uns sozusagen spätestens nach einer Timeout-Zeit wecken, d.h. präziser gesagt, der blockierende recvfrom()-Aufruf wird durch ein Signal spätestens nach dem Timeout unterbrochen. Die Funktion, die den Wecker startet, heißt alarm() und das betreffende Signal SIGALARM. Wenn count = recvfrom() zurückkehrt kann das nun folgende Ursachen haben:

- ein gültiges Datagramm ist angekommen: count > 0,
- ein Timeout ist aufgetreten: count == -1 und errno == EINTR,
- ein anderer Fehler ist aufgetreten: count == -1 und errno != EINTR.

Folgender Programmausschnitt überwacht das Zeitlimit und wiederholt das Senden des Read-Requests oder eines Acknowledge des rcat-Clients. Der Signal-Handler braucht im Falle des SIGALARM einfach nur zum Prozess zurückzukehren.

```
signal(SIGALARM, alarm_catcher);
alarm(TIMEOUT); /* Alarm setzen */
count = recvfrom(...);
alarm(0); /* Alarm zurücksetzen */
if (count == -1) {
   if (errno == EINTR) {
```

```
            sendto(...); /* erneut senden */
         } else {
            perror(...); /* anderer Fehler */
            exit(1);
         }
      }
      /* normaler Empfang */
      ...
```

Kapitel 4

Remote Procedure Call (RPC)

4.1 Einführung

Die Programmierung verteilter Algorithmen mit Client und Server beinhaltet bei der Socket-Schnittstelle die direkte Programmierung der Transportschicht. Weiterhin musste der Client die Adresse des Server-Programms herausfinden (gethostbyname(), getservbyname()), um mit ihm in Kontakt treten zu können. Ein weiteres Problem ist die unterschiedliche Darstellung von Daten auf unterschiedlichen Rechnerarchitekturen.

Die RPC-Schnittstelle (Remote Procedure Call) vereinfacht die Programmierung verteilter Client-/Server-Applikationen erheblich und löst diese Probleme. Dabei nutzt das RPC-Paradigma die Struktur der meisten (auch nicht verteilten) Programme:

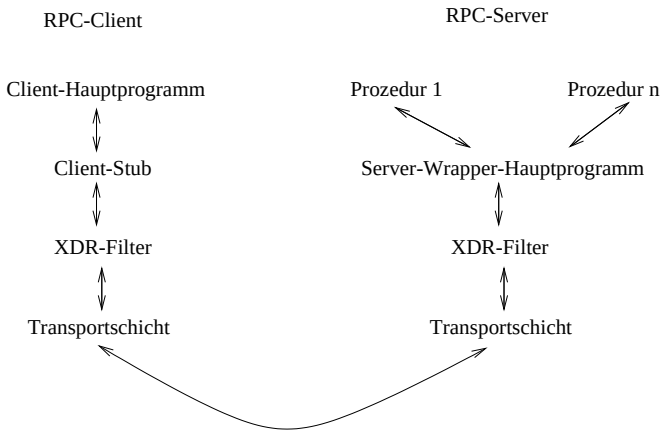
- Steuerungsfunktionen entscheiden, was getan werden muss (Caller, Client).
- Ausführungsfunktionen erledigen die zugeordneten Arbeiten (Callee, Server).

RPC's bewahren diese Schichtstruktur und erlauben Teilen der Programme, auf verschiedenen Rechnern zu arbeiten.

Weiterhin verbirgt die RPC-Schnittstelle die Interprozess-Kommunikation hinter Funktionsaufrufen (Procedure Calls). Für die Datenübertragung zwischen Client und Server wird eine standardisierte Darstellung, die External Data Representation (XDR), gewählt. Mit Hilfe von Funktionsaufrufen aus der XDR-Bibliothek werden Daten vom Rechnerformat zum XDR-Format und umgekehrt konvertiert. Die wichtigsten RPC-Implementationen sind die ONC-RPC's (Open Network Computing, SUN) und die NCS-RPC's (Network Computing System, Appollo, OSF/DCE).

RPC's arbeiten prinzipiell wie folgt:

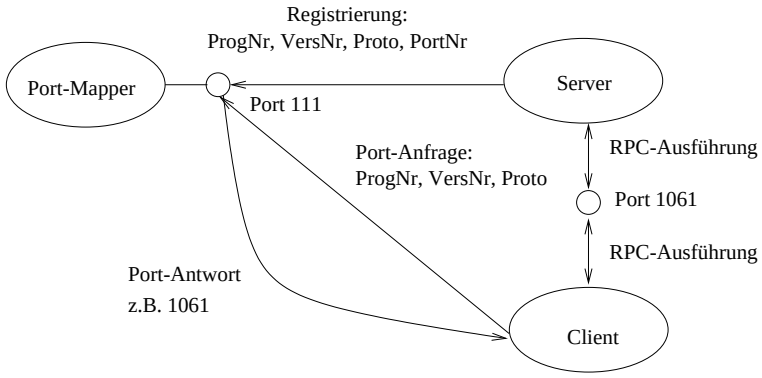
- Das Client-Programm ruft eine Client-Stummel-Prozedur (Client-Stub), die die Argumente der Prozedur zu einer Nachricht für den Server zusammenbaut.
- Die Datentypen des Client-Progamms werden in die XDR-Repräsentation mit Hilfe von XDR-Filtern konvertiert. Solche XDR-Filter gibt es für alle gängigen C-Datentypen.
- Der Transportdienst des Netzwerks überträgt die Nachricht zu einem Server-Wrapper-Programm.
- Der Server-Wrapper zerlegt die Nachricht und konvertiert sie wieder in das plattformabhängige Datenformat. Anschließend ruft er die passende Server-Prozedur mit den Parametern auf.
- Nachdem die Server-Prozedur beendet ist, verpackt der Server-Wrapper die Ergebnisdaten in eine XDR-Nachricht.
- Der Transportdienst des Netzwerks überträgt die Ergebnisnachricht zum Client-Stub.
- Der Client-Stub zerlegt die Ergebnisnachricht und konvertiert sie in das plattformabhängige Format.



Für die Identifikation einer Server-Prozedur gibt es ein Verwaltungsprogramm für die RPC-Server eines Hosts, den Port-Mapper. Die Kontaktaufnahme eines RPC-Clients mit einem RPC-Server funktioniert wie folgt:

- Jede externe Prozedur wird über drei ganze Zahlen identifiziert: Programm-Nummer, Versions-Nummer und Prozedurnummer.

- Die beiden Zahlen Programm- und Versionsnummer identifizieren ein Serverprogramm.
- Die Prozedurnummer identifiziert die Prozedur im Server.
- Wenn ein Server startet, trägt er seinen Transportendpunkt im Port-Mapper unter einer Programm- und Versionsnummer ein.
- Der Client nimmt zunächst Kontakt mit dem Port-Mapper des fremden Rechners auf. Dabei übergibt er Programm- und Versionsnummer sowie den benötigten Transportdienst an den Port-Mapper des Rechners, der immer unter Port 111 erreichbar ist.
- Der Port-Mapper gibt die Portnummer des Servers an den Client zurück.
- Nun ruft der Client mit dem Server-Wrapper über diesen Port und die Prozedurnummer den zugehörigen RPC auf.



Das Kommando rpcinfo -p zeigt den Inhalt der Port-Mapper-Tabelle an. Der Port-Mapper-Mechanismus hat mehrere Vorteile:

- Die Portnummer ist frei wählbar; sie kann bei unterschiedlichen Systemen verschieden sein.
- Ein und derselbe Dienst kann mit mehreren Protokollen an unterschiedlichen Ports realisiert sein, z.B.:
  - ypbind, version 3, udp, 1029,
  - ypbind, version 3, tcp, 1027.

- Es können Server-Versionen für ältere Clients eingetragen werden. Hierbei kann ein Programm mehrere Server-Versionen am selben Port vereinigen oder mehrere Programme realisieren die Server-Versionen an je einem Port:
  - mountd, version 1, udp, 1056,
  - mountd, version 2, udp, 1056,
  - primserv, version 1, udp, 1057,
  - primserv, version 2, udp, 1058.

Der Service, der bei rpcinfo -p angezeigt wird, ist ein symbolischer Name für die Programmnummer. Die Zuordnung von Service und Programmnummer ist in der Datei rpc festgehalten, z.B.:

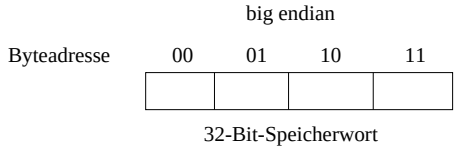
```
#service     progno       alias service names
portmapper   100000       portmap sunrpc
rstatd       100001       rstat rstat_svc rup
ruserd       100002       rusers
nfs          100003       nfsprog
ypserv       100004       ypprog
mountd       100005       mount showmount
...
```

Die Programmnummer ist eine 32-Bit-Integer-Zahl, die in folgende Bereiche aufgeteilt ist:

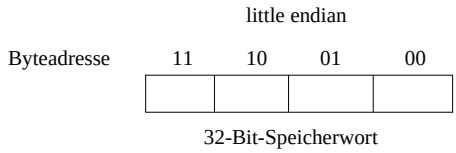
- 0x00 00 00 00 - 0x1f ff ff ff entspricht dezimal 536.879.911, werden von SUN verwaltet.
- 0x20 00 00 00 - 0x3f ff ff ff entspricht dezimal 1.073.741.823, werden von Anwendern definiert.
- 0x40 00 00 00 - 0x5f ff ff ff entspricht dezimal 1.610.612.735, sind temporäre Programmnummern.
- 0x60 00 00 00 - 0xff ff ff ff entspricht dezimal 4.294.967.295, sind für zukünftige Zwecke reserviert.

4.2 External Data Representation

Unterschiedliche Rechnerarchitekturen benutzen i.a. auch verschiedene Darstellungen binärer Daten. Einer der häufigsten Darstellungsunterschiede betrifft die Byteanordnung. Die meisten Architekturen transferieren 32-Bit-Worte zwischen Hauptspeicher und Prozessor, d.h. der Speicher wird in 4-Byte-Worten adressiert, wobei die unteren 2 Bit einer Adresse zur Identifikation eines Byte innerhalb eines Speicherwortes dienen. Die normale (big endian) Anordnung speichert die höherwertigen Bytes einer Integer-Zahl zu niedrigeren Adressen, z.B. bei den Prozessor-Familien Motorola 68K und bei SUN-Sparc:



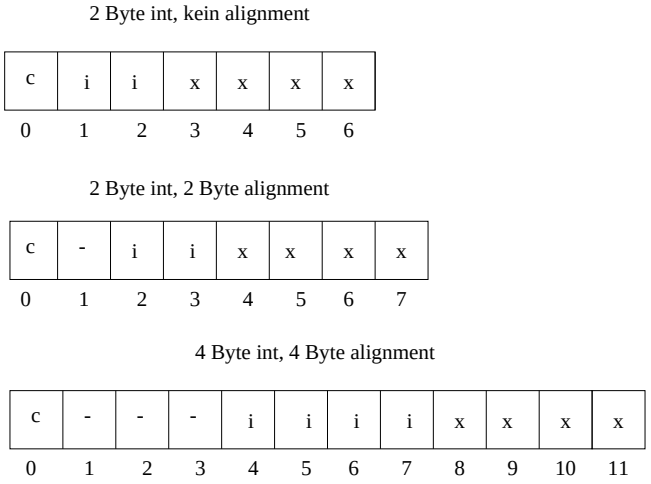
Im Gegensatz dazu speichert die little endian Anordnung die höherwertigen Bytes zu höheren Adressen, z.B. bei den Prozessor-Familien Intel x86, DEC-VAX und INMOS-Transputer:



Natürlich sind auch die Gleitkommazahlen-Darstellungen in verschiedenen Rechnern unterschiedlich.

Weitere Unterschiede können durch Ausrichtungsregeln (alignment rules) der Rechnerarchitekturen entstehen. Eine typische Ausrichtungsregel kann z.B. lauten: 32-Bit-Ganzzahlen müssen auf einer Byte-Adresse liegen, die ein Vielfaches von 4 ist. Solche Regeln zwingen den Compiler u.U., Löcher in Datenstrukturen zu erzeugen, z.B.:

```
struct demo {
  char c;
  int  i;
  long x;
}
```



Ein weiteres Problem ist die Übermittlung von Zeigern zwischen Rechnern unterschiedlicher Architekturen. Zeiger weiterzugeben macht natürlich nur Sinn, um eine Art von Graphstruktur an einen anderen Rechner zu übermitteln, z.B. einen Baum oder eine lineare Liste. Dazu später mehr.

Für XDR-Formate gelten folgende Regeln:

- Ganzzahlen werden big endian abgespeichert.
- Alle XDR-Daten sind ein Vielfaches von 4 Byte lang, ggfs wird mit 0-Bytes aufgefüllt.
- Datentypen werden nicht mitgesendet, sondern müssen dem Empfänger implizit bekannt sein.
- XDR geht davon aus, dass der Byte-Strom in der Reihenfolge des Aussendens beim Empfänger ankommt. Wie Bits innerhalb eines Bytes übertragen werden, wird dem Netzwerk überlassen.

Funktionen zum Ver- und Entschlüsseln in das und aus dem XDR-Format werden XDR-Filter genannt. Für jeden Grunddatentyp gibt es einen gesonderten XDR-Filter. XDR-Filter arbeiten mit zwei Parametern. Der erste Parameter ist eine Referenz im XDR-Strom. Der zweite Parameter ist ein Zeiger auf die zu kodierenden Daten oder auf die Stelle, an der die zu dekodierenden Daten abgelegt werden. Der XDR-Datenstrom wird normalerweise mit einem Speicherpuffer assoziiert; er kann aber auch z.B. zu Demonstrationszwecken mit dem stdio-Strom verbunden werden. Es folgt ein Einsatz von XDR-Filtern ohne RPC's. Dies kann z.B. sinnvoll sein, um Daten auf ein Magnetband zu schreiben, das von einem anderen Rechner eingelesen werden soll. Die Hauptanwendung von XDR-Filtern ist jedoch die Übertragung von Argumenten und Rückgabewerten bei RPC's.

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
main() {
  int i= 0x12345678;
  XDR handle; /* XDR-Referenz */
  xdrstdio_create(&handle,stdout,XDR_ENCODE);
  /* Kodierrichtung: XDR-handle auf stdout */
  xdr_int(&handle, &i);
}
```

xdr_int() wirkt hier ähnlich wie fprintf() jedoch in binärer Datendarstellung.

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
main() {
  int i;
  XDR handle; /* XDR-Referenz */
  xdrstdio_create(&handle,stdin,XDR_DECODE);
  /* Dekodierrichtung: XDR-handle auf stdin */
  xdr_int(&handle, &i);
  printf("%x\n", i);
}
```

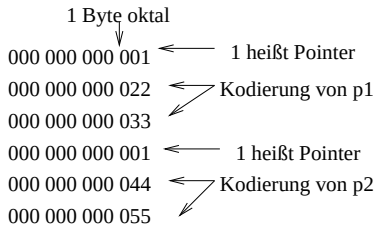
Die Übertragung von Datenstrukturen mit Zeigern ist ein besonderer Fall, da die Adressräume verschiedener Rechner unterschiedlich sind. XDR stellt einen speziellen Filter xdr_pointer() zur Verfügung, der auch Zeigerketten verfolgen kann. Damit kann das Ergebnis beim Empfänger auch als Zeigerkette rekonstruiert werden. Die XDR-Deklaration ähnelt der C-Deklaration für Zeiger. Ein Rechteck kann z.B. durch folgende Struktur in der Datei rectangle.x definiert werden:

```
struct point {
  int x;
  int y;
};
struct rectangle {
  point *topleft;
  point *botright;
}
```

Wenn man diese Struktur mit XDR versendet, werden implizit xdr_pointer()-Aufrufe durchgeführt. Folgendes Programm veranschaulicht die XDR-Kodierung auf stdout:

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
#include "rectangle.h"
point p1 = {022, 033};
point p2 = {044, 055};
rectangle r = {&p1, &p2};
int main(void) {
  XDR handle;
  xdrstdio_create(&handle,stdout,
                  XDR_ENCODE);
  xdr_rectangle(&handle, &r);
}
```

Wird die Ausgabe dieses Progamms über eine Pipe an das Programm od -b übergeben, so werden oktal folgende kodierte Werte ausgegeben:



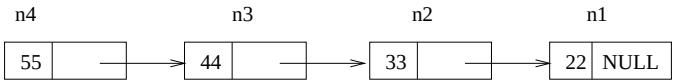
Diese Ausgabe kann folgendermaßen erklärt werden:

1. xdr_rectangle() ruft xdr_pointer() auf, um die topleft-Einheit zu kodieren.
2. xdr_pointer() schreibt eine 1 heraus und ruft xdr_point() auf, um die Einheit, auf die gezeigt wird, zu kodieren, also hier *topleft.
3. xdr_point() ruft zweimal xdr_int() auf und kodiert die x- und y-Einheiten der Struktur.
4. Das Programm kehrt zu xdr_rectangle() zurück und ruft analog für die botright-Einheit xdr_pointer() auf.
5. ist analog zu 2.

6. ist analog zu 3.

Rekursive Datenstrukturen, wie lineare Listen und Bäume, können ebenfalls mit XDR versendet werden, solange keine Zykeln in der Datenstruktur vorhanden sind, z.B. rückwärts-verzeigerte oder zyklisch-verzeigerte Listen. Als Beispiel betrachten wir eine einfach verzeigerte Liste:

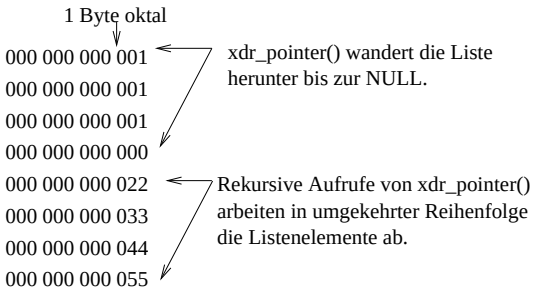
```
struct intlist {
  struct intlist *next;
  int data;
};
```



Betrachten wir dazu folgendes Programm:

```
#include <stdio.h>
#include <sys/types.h>
#include <rpc/rpc.h>
#include "intlist.h"
intlist n1 = {NULL, 022}, n2 = {&n1, 033},
        n3 = {&n2, 044},  n4 = {&n3, 055};
main() {
  XDR handle;
  xdrstdio_create(&handle,stdout,
                  XDR_ENCODE);
  xdr_intlist(&handle, &n4);
}
```

Die Ausgabe bei intlist | od -b lautet wie folgt:



Die Ausgabe kann wie folgt erklärt werden: Der Aufruf von xdr_intlist() ruft zunächst xdr_pointer() auf, um den ersten next-Zeiger zu kodieren. Da next nicht gleich NULL ist, wird eine 1 in den Strom kodiert. Da die Einheit, auf die next zeigt wiederum vom Typ intlist ist, wird xdr_intlist() erneut rekursiv aufgerufen. Dann ruft xdr_intlist() wieder xdr_pointer() usw.

Es ergibt sich also eine wechselseitige Rekursion der Aufrufe von xdr_intlist() und xdr_pointer(), wobei zunächst die gesamte Pointerkette mit xdr_pointer() durchgegangen wird. Erst wenn der Zeiger next beim letzten Listenelement NULL ist, wird eine 0 in den Strom ausgegeben und xdr_intlist() nicht mehr aufgerufen.

Anschließend wird der letzte Aufruf von xdr_intlist() durch die Kodierung von data mit xdr_int() fertig gestellt. Es folgt die Kodierung des vorletzten Elements usw., so dass die Listenelemente von hinten nach vorne ausgegeben werden.

4.3 Aufbau von Client und Server mit rpcgen

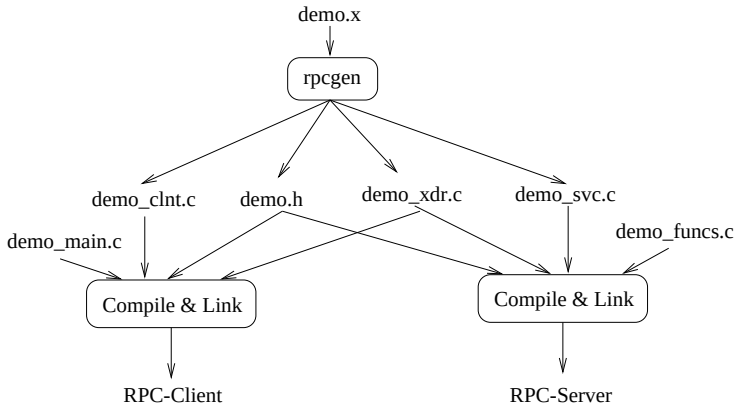
Die Hauptarbeit beim Aufbau von Clients und Servern auf RPC-Basis übernimmt das Dienstprogramm rpcgen. rpcgen verwendet als Eingabe drei Dateien:

- eine Datei aus Protokollspezifikationen mit XDR-Datentypen und Prototypen der Serverfunktionen, z.B. demo.x,
- eine Datei mit Serverfunktionen, z.B. demo_funcs.c,
- eine Datei mit dem Clientprogramm, z.B. demo_main.c.

Aus den Protokollspezifikationen erzeugt rpcgen:

- den Quellcode des Client-Stub, z.B. demo_clnt.c,
- den Quellcode des Server-Wrapper, z.B. demo_svc.c,
- den Quellcode der XDR-Filter, z.B. demo_xdr.c
- die Headerdatei für die XDR-Deklarationen, z.B. demo.h.

Mit Hilfe dieser Quelldateien und den Quelldateien des Clientprogramms und der Serverfunktionen werden der RPC-Client und RPC-Server compiliert und gebunden:



Als Beispiel für eine RPC-Anwendung werden wir ein Primzahlsieb nach dem Algorithmus des Eratosthenes von Kyrene aufbauen. Der Algorithmus arbeitet wie folgt:

- Man streicht nach der 2 als Primzahl jede durch 2 teilbare Zahl aus dem zu untersuchenden Zahlenbereich $1 \leq i \leq n$.
- Die nächste nicht gestrichene Zahl $i = 3$ ist prim. Man streicht Vielfache von 3. Dies braucht man nur ab $3 * 3$ zu tun, da die $3 * 2$ als Vielfaches von 2 schon gestrichen wurde.
- Die nächste nicht gestrichene Zahl $i = 5$ ist prim. Man streicht Vielfache von 5. Dies braucht man nur ab $5*5$ zu tun, da $5*2$, $5*3$, $5*4$ als Vielfache von 2 und 3 schon gestrichen wurden.
- Der Algorithmus endet, wenn die zu untersuchende Zahl $i^2 > n$ erfüllt, da dann keine weiteren Zahlen aus dem zu untersuchenden Bereich mehr gestrichen werden.

Ein triviales Programm zur Berechnung von Primzahlen lautet wie folgt:

```
main(){
  int i, how_many, primes[1000];
  how_many = find_primes(1, 1000, primes);
  for (i = 0; i < how_many; i++)
    printf("%d ist prim.\n", primes[i]);
}

int find_primes(int min,int max,int *array){
  int i, count = 0;
```

```
  for (i = min; i <= max; i++)
    if (isprime(i)) array[count++] = i;
  return count;
}

int isprime(int n){
  int i;
  for (i = 2; i * i <= n; i++)
    if ((n % i) == 0) return 0;
    /* n ist nicht prim */
  return 1;
  /* n ist prim */
}
```

Dieses Programm ist natürlich nicht laufzeit-effizient, da bei der Ermittlung der Primeigenschaft einer Zahl n vorher ermittelte Ergebnisse nicht wiederverwendet werden. Weiterhin ist auch eine parallele Lösung für dieses Problem denkbar, die dann aber statt mit RPC's mit Sockets , PVM (Parallel Virtual Machine) oder MPI (Message Passing Interface) programmiert wird.

Zur Demonstration der Verteilung einer Anwendung mit RPC's können wir dieses Programm jedoch verwenden. Der einzige Grund zur Verteilung wäre ein schneller externer Rechner zur Berechnung der Primzahlen. Wir entscheiden uns, das Primzahlprogramm zu verteilen, indem wir find_primes() und deren Dienstfunktion isprime() extern ausführen lassen. Das Protokoll zwischen dem Client und dem Server muss nun in XDR-Sprache spezifiziert werden. Dabei stoßen wir auf zwei Probleme:

- Die Funktion find_primes() hat zwei Eingabeparameter. SUN-RPC's können aber nur ein Argument entgegennehmen. Deswegen muss eine Struktur gebildet werden, in der die Argumente gemeinsam übergeben werden.
- Die Primzahlenliste wird als Ausgabeparameter von find_primes() zurückgeliefert. Diese Call-by-Reference-Technik funktioniert bei RPC's nicht. Daher wird ein Ganzzahlenfeld mit dynamischer Länge als XDR-Datentyp für die Rückgabe vereinbart.

Den ersten Teil der primes.x-Datei bilden also folgende XDR-Deklarationen:

```
const MAXPRIMES = 1000;
struct prime_request {
  int min;
  int max;
```

```
};
struct prime_result {
  int array<MAXPRIMES>;
};
```

Diese XDR-Beschreibung ermöglicht rpcgen, die XDR-Filter zu den Datentypen zu erzeugen. Den zweiten Teil der primes.x-Datei bilden die Server-Programm- und Versionsnummer und die Prototypen der Serverfunktionen:

```
program PRIMEPROG {
  version PRIMEVERS {
    prime_result FIND_PRIMES(prime_request)=1;
  } = 1; /* Versionsnummer */
} = 0x2000009a; /* Programmnummer */
```

Diese Schreibweise erscheint in diesem Beispiel unnötig kompliziert; sie ist aber flexibel, wenn mehrere Dienstfunktionen und Programmversionen vereinbart werden. Wenn der Primzahl-Server z.B. mit einer zusätzlichen Prozedur COUNT_-PRIMES() weiterentwickelt wird, so kann hierfür eine weitere Programmversion eingefügt werden:

```
program PRIMEPROG {
  version PRIMEVERS1 {
    prime_result FIND_PRIMES(prime_request)=1;
  } = 1; /* Versionsnummer 1 */
  version PRIMEVERS2 {
    prime_result FIND_PRIMES(prime_request)=1;
    int          COUNT_PRIMES(prime_request)=2;
  } = 2; /* Versionsnummer 2 */
} = 0x2000009a; /* Programmnummer */
```

Der Lauf von rpcgen erzeugt die Dateien primes.h, primes_clnt.c, primes_svc.c, primes_xdr.c. Aus primes.h verwenden wir Deklarationen für die Serverfunktionen und das Client-Programm. Die Serverfunktion wird aus der Funktion find_-primes() entwickelt:

```
/* primes_funcs.c */
/* RPC-Server Version 1 */
#include <rpc/rpc.h>
#include "primes.h"
prime_result *find_primes_1_svc(prime_request
                                *request){
```

```
/* Der Speicher für die Rückgabewerte muss
   statisch vereinbart werden, da sie über
   Referenzen nach Rückkehr der Funktion
   an den XDR-Filter übergeben werden
   (xdr_prime_result()) und sonst vom
   Stapelspeicher gelöscht werden.
 */
  static prime_result result;
  static int prime_array[MAXPRIMES];
  int i, count = 0;
  for (i=request->min; i <= request->max; i++)
    if (isprime(i)) prime_array[count++] = i;
  /* Zusammenstellen der Rückgabestruktur:
     Bei dem Feld variabler Länge muss die
     Elementzahl zurückgegeben werden
     (siehe primes.h).
   */
   result.array.array_len = count;
   result.array.array_val = prime_array;
   return &result;
   /* result wird von xdr_prime_result()
      weiterverarbeitet
    */
}
```

Der Client wird aus dem ursprünglichen Hauptprogramm main() abgeleitet. Zur Kommunikation mit dem RPC-Server muss der Client eine Client-Kennung haben, die er durch den Aufruf clnt_create() erhält. Die Syntax von clnt_create() lautet:

```
CLIENT *cl;
cl=clnt_create(host,prog,version,transport);
```

- host ist der Name des Server-Rechners,
- prog, version sind ganzzahlige Programm- und Versionsnummern,
- transport ist der Name des benötigten Transportdienstes wie in der Datei netconfig definiert.

Hier folgt nun das Client-Programm:

```
/* primes_main.c */

#include <rpc/rpc.h>
#include “primes.h”

main(int argc, char *argv[]) {
  int i;
  CLIENT *cl; /* Client handle */
  prime_result *result;
  prime_request request;
  if (argc != 4) {
    fprintf(stderr, "usage: %s host
            min max\n", argv[0]);
    exit(1);
  }
  /* Client mit dem RPC-Server verbinden. */
  cl = clnt_create(argv[1], PRIMEPROG,
                   PRIMEVERS, "tcp");
  if (cl == NULL) {
    clnt_pcreateerror(argv[1]);
    /* genaue Fehlerausgabe */
    exit(2);
  }
  /* Request-Struktur bilden. */
  request.min = atoi(argv[2]);
  request.max = atoi(agrv[3]);
  /* Aufruf des RPC */
  result = find_primes_1(&request, cl);
  if (result == NULL) {
    clnt_perror(cl, argv[1]);
    exit(3);
  }
  /* Ausgabe der Ergebnisse */
  for (i=0; i<result->array.array_len; i++)
    printf("%d ist prim.\n",
           result->array.array_val[i]);
  printf("Anzahl Primzahlen: %d\n",
          result->array.array_len);
  /* Freigabe des Speichers für result, der
     durch xdr_prime_result() angelegt wurde.
   */
```

```
    xdr_free(xdr_prime_result, result);
}
```

Anmerkungen:

- Wegen der typedefs in primes.h werden prime_result und prime_request ohne struct bei der Deklaration verwendet.
- Es können Fehler beim Verbinden mit dem Server entstehen, wenn der Rechner unbekannt ist oder das Server-Programm also Programm- und Versionsnummer nicht im Port-Mapper eingetragen ist.
- Wenn die externe Funktion nicht erreicht werden kann, z.B. bei einem fehlerhaften Client-Stub tritt ein Fehler auf (bei Verwendung von rpcgen unwahrscheinlich).
- Anwendungsfehler, z.B. Vertauschen von max und min sind Sache der Anwendung. Hier kann z.B. ein Fehlercode als Teil der Rückgabestruktur vom Server zurückgesendet werden.

Nach dem Start des Server-Programms, kann der Status im Port-Mapper mit dem Kommando rpcinfo eingesehen werden. Hierbei ist es bequem für die Programmnummer einen Namen in der Datei rpc zu vergeben:

```
primes 536871066
```

Hierbei entspricht 536871066 hexadezimal 0x20 00 00 9a. Mit rpcinfo -p [Rechner] kann nun überprüft werden, ob sich der Server im Port-Mapper registriert hat.

4.4 Authentifizierung

Methoden zur Authentifizierung beruhen meistens auf einem Nachweis und einem Verifizierer, die beide vom Client an den Server weitergegeben werden. Der Nachweis wird zur Identifikation des Client übergeben, z.B. der Name des Client-Rechners und die Benutzerkennung. Der Verifizierer ist nur dem echten Client bekannt; er wird zur Validierung des Client im Server benutzt. Verifizierer sind z.B. ein Passwort, ein Fingerabdruck oder ein Foto.

Manchmal muss auch die Identität des Servers vom Client überprüft werden. Ein Beispiel für einen Nachweis des Servers bei TCP/IP ist der Servername; der Verifizierer ist dann ein reservierter Port, an den der Server angeschlossen ist,

z.B. der rlogind-Dämon. Häufig werden Nachweis und Verifzierer als Teil der Anwendungsebene übergeben, z.B. bei rexec.

Für RPC's ist das Konzept der reservierten Ports unbrauchbar, da die Portnummern dynamisch vergeben werden. Das RPC-Protokoll definiert keinen Authentifizierungs-Mechanismus. Die meisten Implementationen unterstützen allerdings drei verschiedene Authentifikationsformen. Dabei gibt es RPC-Routinen, die Authentifikationsdaten im Client zusammenstellen und im Server wieder abholen. Folgende Stufen (flavours) sind üblich:

AUTH_NONE: Weder Nachweis noch Verifzierer werden weitergegeben.

AUTH_UNIX: Als Nachweis werden Rechnername, UserID und GroupID des Client übermittelt. Verifizierer gibt es nicht.

AUTH_DES: Nachweise sind Netznamen mit Benutzerkennung und Rechnername. Der Verifizierer ist eine Zeitmarkierung, die mit DES (Data Encryption Standard) kodiert wird.

Als Beispiel soll die AUTH_UNIX-Authentifizierung in einem RPC-Aufruf vorgeführt werden. Auf der Client-Seite muss dafür lediglich die Authentifikationsstruktur in die Client-Struktur mit einer Systemfunktion eingetragen werden:

```
cl = clnt_create(...);
cl->cl_auth = authunix_create_default();
```

Auf dem Server kann die Dienstfunktion die Nachweise über einen zweiten Parameter auswerten, der bisher ignoriert wurde:

```
prime_result *find_primes_2(prime_request *req,
             struct svc_req *rp)
```

Die Struktur svc_req hat folgendes Aussehen:

```
struct svc_req {
  /* service program number */
  u_long rq_prog;
  /* service program version */
  u_long rq_vers;
  /* desired procedure */
  u_long rq_proc;
  /* raw credentials */
  struct opaque_auth rq_cred;
```

```
  /* read only credentials */
  caddr_t rq_clntcred;
  /* transport */
  SVCXPRT *rq_xprt;
}
```

Interessante Felder dieser Struktur sind rq_cred, das die Authentifikationsstufe enthält und rq_clntcred, das die stufenspezifischen Daten zur Authentifizierung enthält. Für die AUTH_UNIX-Authentifizierung ist eine Struktur für die Parameter in <rpc/auth.h> definiert:

```
struct authsys_parms {
  u_long autp_time;
  /* Client Host name */
  char *aup_machname;
  /* Client user id */
  uid_t aup_uid;
  /* Client group id */
  gid_t aup_gid;
  /* Length of group list */
  u_int aup_len;
  /* List of groups */
  gid_t *aup_gids;
}
```

Der folgende Progammausschnitt zeigt, wie der Server die Authentifikationsdaten empfängt:

```
int *my_service_proc_1(my_rquest *r,
                       struct svc_req *rp) {
struct authunix_parms *ucred;
if (rp->rq_cred.oaflavor == AUTH_UNIX) {
 ucred=(struct authunix_parms*)
       (rp->rq_clntcred);
 ...
}
```

Nun kann die eigentliche Validierung auf dem Server erfolgen, z.B. durch Abbildung der übermittelten UserID auf einen Nutzernamen in der Datei passwd und Vergleich mit einer Liste zugelassener Nutzer. Eine Funktion zur Nutzervalidierung könnte z.B. wie folgt aussehen:

```
int validate_user(struct svc_req *rp) {
  struct authunix_parms *ucred;
  struct passwd *pwent;
  char *client_name;
  FILE *fd;
  char name_in_file[50];

  if (rp->rq_cred.oa_flavor != AUTH_UNIX)
    return 0;
  ucred=(struct authsys_parms*)
        (rp->rq_clntcred);
  pwent=getpwuid(ucred->aup_uid);
  if (pwent == NULL) return 0;
  /* nicht validiert */
  client_name = pwent->pw_name;
  if ((fd=fopen("/etc/validusers", "r"))==NULL)
    return 0;
    /* nicht validiert */
  while (fscanf(fd, %s, name_in_file) != EOF)
    if (strcmp(client_name, name_in_file)==0){
      fclose(fd);
      return 1; /* validiert */
  }
  fclose(fd);
  return 0; /* nicht validiert */
}
```

Kapitel 5

Java-Programmierung mit Sockets

Die Socket-Schnittstelle in Java erlaubt es, über eine TCP/IP-Verbindung Daten zwischen einer Client- und einer Server-Applikation auszutauschen. Sie stellt damit den Basiskommunikations-Mechanismus für Java dar. Höher abstrahierende Modelle wie RMI (Remote Method Invocation) basieren auf dem Socket-Mechanismus.

5.1 Client-Sockets

5.1.1 IP-Adressen und Ports

Zur Adressierung von Rechnern im Netz wird die Klasse InetAddress des Pakets java.net verwendet. Ein InetAddress-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners. Die IP-Adresse kann mit der Methode getHostAdress und der Rechnername mit getHostName erfragt werden. getAddress liefert die IP-Adresse als byte-Array mit vier Elementen. Es folgen Methoden der Klasse java.net.InetAddress:

```
String getHostName()
String getHostAddress()
byte[] getAddress()
```

Um ein InetAddress-Objekt zu generieren, stehen die beiden statischen Methoden getByName und getLocalHost der Klasse java.net.InetAddress zur Verfügung:

```
public static InetAddress
getByName(String host)
throws UnknownHostException
public static InetAddress
getLocalHost()
throws UnknownHostException
```

getByName erwartet einen String mit der IP-Adresse oder dem Namen des Hosts als Argument. getLocalHost liefert ein InetAddress-Objekt für den eigenen Rechner. Beide Methoden lösen eine Ausnahme des Typs UnknownHostException aus, wenn die Adresse nicht ermittelt werden kann. Das ist der Fall, wenn kein DNS-Server zur Verfügung steht, der die gewünschte Namensauflösung erledigen könnte oder der Name des Rechners in keinem DNS-Server verzeichnet ist.

Das folgende Listing zeigt ein einfaches Programm, das zu einer IP-Adresse den symbolischen Namen des zugehörigen Rechners ermittelt und umgekehrt:

```
import java.net.*;

public class ipaddress {

public static void main(String[] args) {
 if (args.length != 1) {
  System.err.println("Usage: java ipaddress <host>");
  System.exit(1);
 }
 try {

 //Get requested address

 InetAddress addr =
   InetAddress.getByName(args[0]);
 System.out.println(addr.getHostName());
 System.out.println(addr.getHostAddress());
 } catch (UnknownHostException e) {
  System.err.println(e.toString());
  System.exit(1);
 }
}
```

5.1.2 Lesen vom Socket

Als Socket bezeichnet man eine Programmierschnittstelle zur Kommunikation zweier Rechner im Netz. Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt und mit BSD-UNIX eingeführt. Sockets gibt es für unterschiedliche Netzwerkprotokolle u.a. auch TCP/IP. Das Übertragen von Daten über eine Stream Socket-Verbindung ähnelt dem Zugriff auf eine Datei:

- Zunächst wird eine Verbindung aufgebaut.

- Dann werden Daten gelesen und/oder geschrieben.
- Schliesslich wird die Verbindung wieder abgebaut.

Die Socket-Programmierung in Java ist gegenüber der Socket-Programmierung in C weniger mühsam. Im wesentlichen sind dazu die beiden Klassen Socket und ServerSocket erforderlich. Sie repräsentieren Sockets aus der Sicht einer Client- bzw. Server-Anwendung. Nachfolgend wollen wir uns mit den Client-Sockets beschäftigen, die Klasse ServerSocket wird im nächsten Abschnitt behandelt.

Die Klasse java.net.Socket besitzt verschiedene Konstruktoren, mit denen ein neuer Socket erzeugt werden kann. Die wichtigsten von ihnen sind:

```
public Socket(String host, int port)
throws UnknownHostException, IOException
public Socket(InetAddress address, int port)
throws IOException
```

Beide Konstruktoren erwarten als erstes Argument die Übergabe des Rechnernamens, zu dem eine Verbindung aufgebaut werden soll. Dieser kann entweder als Domainname in Form eines Strings oder als Objekt des Typs InetAddress übergeben werden. Soll eine Adresse mehrfach verwendet werden, ist es besser, die zweite Variante zu verwenden. In diesem Fall kann das übergebene InetAddress-Objekt wiederverwendet werden, und die DNS-Adressauflösung muss nur einmal erfolgen. Wenn der Socket nicht geöffnet werden konnte, gibt es eine Ausnahme des Typs IOException bzw. UnknownHostException.

Der zweite Parameter des Konstruktors ist die Portnummer. Sie dient sie dazu, den Typ des Servers zu bestimmen, mit dem eine Verbindung aufgebaut werden soll. Die wichtigsten Standard-Portnummern sind in der Datei /etc/services aufgelistet.

Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden getInputStream und getOutputStream je ein Stream zum Empfangen und Versenden von Daten beschafft werden:

```
public InputStream getInputStream()
throws IOException
public OutputStream getOutputStream()
throws IOException
```

Diese Streams können entweder direkt verwendet oder mit Hilfe der Filterstreams in einen bequemer zu verwendenden Streamtyp geschachtelt werden. Nach Ende der Kommunikation sollten sowohl die Eingabe- und Ausgabestreams als auch der Socket selbst mit close geschlossen werden.

Als erstes Beispiel wollen wir uns ein Programm ansehen, das eine Verbindung zum DayTime-Service auf Port 13 herstellt. Dieser Service läuft auf fast allen UNIX-Maschinen und kann gut zu Testzwecken verwendet werden. Nachdem der Client die Verbindung aufgebaut hat, sendet der DayTime-Server einen String mit dem aktuellen Datum und der aktuellen Uhrzeit und beendet dann die Verbindung.

```
import java.net.*;
import java.io.*;
public class daytime {

public static void main(String[] args) {
 if (args.length != 1) {
  System.err.println(
   "Usage: java daytime <host>");
  System.exit(1);
 }
 try {
  Socket sock = new Socket(args[0], 13);
  InputStream in = sock.getInputStream();
  int len;
  byte[] b = new byte[100];
  while ((len = in.read(b)) != -1) {
    System.out.write(b, 0, len);
  }
  in.close();
  sock.close();
 } catch (IOException e) {
  System.err.println(e.toString());
  System.exit(1);
 }
}
```

Das Programm erwartet einen Hostnamen als Argument und gibt diesen an den Konstruktor von Socket weiter, der eine Verbindung zu diesem Host auf Port 13 erzeugt. Nachdem der Socket steht, wird der InputStream beschafft. Das Programm gibt dann so lange die vom Server gesendeten Daten aus, bis durch den Rückgabewert -1 angezeigt wird, dass keine weiteren Daten (End of File) gesendet werden. Nun werden der Eingabestream und der Socket geschlossen und das Programm beendet. Die Ausgabe des Programms ist beispielsweise:

```
Sat Nov 7 22:58:37 2002
```

Um den Socket alternativ mit einem InetAddress-Objekt zu öffnen, wäre folgender Code notwendig:

```
InetAddress addr = InetAddress.getByName(args[0]);
Socket sock = new Socket(addr, 13);
```

5.1.3 Schreiben auf den Socket

Nachdem wir jetzt wissen, wie man lesend auf einen Socket zugreift, wird in diesem Abschnitt der schreibende Zugriff vorgestellt. Dazu schreiben wir ein Programm, das eine Verbindung zum ECHO-Service auf Port 7 herstellt. Das Programm liest so lange die Eingaben des Anwenders und sendet sie an den Server, bis das Kommando QUIT eingegeben wird. Der Server liest die Daten zeilenweise ein und sendet sie unverändert an unser Programm zurück, von dem sie auf dem Bildschirm ausgegeben werden. Um Lese- und Schreibzugriffe zu entkoppeln, verwendet das Programm einen separaten Thread, der die eingehenden Daten liest und auf dem Bildschirm ausgibt. Dieser läuft unabhängig vom Vordergrund-Thread, in dem die Benutzereingaben abgefragt und an den Server gesendet werden.

```
import java.net.*;
import java.io.*;
public class EchoClient {

public static void main(String[] args) {

 if (args.length != 1) {
  System.err.println("Usage:
  java EchoClient <host>");
  System.exit(1);
 }

 try {
  Socket sock = new Socket(args[0], 7);
  InputStream in = sock.getInputStream();
  OutputStream out = sock.getOutputStream();

  //Timeout setzen

  sock.setSoTimeout(300);
```

```
 //Ausgabethread erzeugen

 OutputThread th = new OutputThread(in);
 th.start();

 //Schleife für Benutzereingaben

 BufferedReader conin = new BufferedReader(
 new InputStreamReader(System.in));
 String line = "";

 while (true) {

 //Eingabezeile lesen

  line = conin.readLine();
  if (line.equalsIgnoreCase("QUIT")) {
   break;
  }

 //Eingabezeile an ECHO-Server schicken

  out.write(line.getBytes());
  out.write('\r');
  out.write('\n');

 //Ausgabe abwarten

  th.yield();
 }

//Programm beenden

System.out.println(
  "terminating output thread...");

th.requestStop();
th.yield();
try {
 Thread.sleep(1000);
```

```
 } catch (InterruptedException e) { }
 in.close();
 out.close();
 sock.close();
} catch (IOException e) {
 System.err.println(e.toString());
 System.exit(1);
}
} // main
} // class

class OutputThread
extends Thread {
 InputStream in;
 boolean stoprequested;
 public OutputThread(InputStream in) {
  super();
  this.in = in;
  stoprequested = false;
 }

public synchronized
void requestStop() {
 stoprequested = true;
}

public void run() {
 int len;
 byte[] b = new byte[100];

 try {
  while (!stoprequested) {
   try {
    if ((len = in.read(b)) == -1) {
     break;
    }
    System.out.write(b, 0, len);
   } catch (InterruptedIOException e) {
     //nochmal versuchen
   }
  }
```

```
 } catch (IOException e) {
  System.err.println("OutputThread: " +
      e.toString());
 }
}
}
```

Wie im vorigen Beispiel wird zunächst ein Socket zu dem als Argument angegebenen Host geöffnet. Das Programm beschafft dann Ein- und Ausgabestreams zum Senden und Empfangen von Daten. Der Aufruf von setSoTimeout gibt die maximale Wartezeit bei einem lesenden Zugriff auf den Socket an (300 ms.). Wenn bei einem read auf den InputStream nach Ablauf dieser Zeit noch keine Daten empfangen wurden, terminiert die Methode mit einer InterruptedIOException; wir kommen darauf gleich zurück. Nun erzeugt das Programm den Lesethread und übergibt ihm den Eingabestream. In der nun folgenden Schleife werden so lange Eingabezeilen gelesen und an den Server gesendet, bis der Anwender das Programm mit QUIT beendet.

Das Programm wurde auf einer LINUX-Version entwickelt, die noch kein präemptives Multithreading unterstützt. Die verschiedenen Aufrufe von yield dienen dazu, die Kontrolle an den Lesethread zu übergeben. Ohne diesen Aufruf würde der Lesethread gar nicht zum Zuge kommen und das Programm würde keine Daten vom Socket lesen. Auf Systemen, die präemptives Multithreading unterstützen, sind diese Aufrufe nicht notwendig.

Die Klasse OutputThread implementiert den Thread zum Lesen und Ausgeben der Daten. Da die Methode stop der Klasse Thread im JDK 1.2 als deprecated markiert wurde, müssen wir mit Hilfe der Variable stoprequested etwas mehr Aufwand treiben, um den Thread beenden zu können. stoprequested steht normalerweise auf false und wird beim Beenden des Programms durch Aufruf von requestStop auf true gesetzt. In der Hauptschleife des Threads wird diese Variable periodisch abgefragt, um die Schleife bei Bedarf abbrechen zu können.

Problematisch an dieser Technik ist lediglich, dass der Aufruf von read normalerweise so lange blockiert, bis weitere Zeichen verfügbar sind. Steht das Programm also in der Zeile,

```
if ((len = in.read(b)) == -1) {
```

so hat ein Aufruf requestStop zunächst keine Wirkung. Da das Hauptprogramm die Streams und den Socket schliesst, würde es zu einer SocketException kommen. Unser Programm verhindert das durch den Aufruf von setSoTimeout. Dadurch wird ein Aufruf von read nach spätestens 300 ms mit einer InterruptedIOException beendet. Diese Ausnahme wird abgefangen, um anschliessend vor dem nächsten Schleifendurchlauf die Variable stoprequested erneut abzufragen.

5.1.4 Client zum Zugriff auf einen Web-Server

Die Kommunikation mit einem Web-Server erfolgt über das HTTP-Protokoll. Ein Web-Server läuft normalerweise auf TCP-Port 80 (manchmal läuft er zusätzlich auch auf dem UDP-Port 80) und kann wie jeder andere Server über einen Client-Socket angesprochen werden. Wir wollen an dieser Stelle nicht auf Details eingehen, sondern nur die einfachste und wichtigste Anwendung eines Web-Servers zeigen, nämlich das Übertragen einer Seite. Ein Web-Server ist in seinen Grundfunktionen ein recht einfaches Programm, dessen Hauptaufgabe darin besteht, angeforderte Seiten an seine Clients zu versenden. Kompliziert wird er vor allem durch die Vielzahl der mittlerweile eingebauten Zusatzfunktionen, wie beispielsweise Logging, Server-Scripting, Server-Side-Includes, Security- und Tuning-Features usw.

Fordert ein Anwender in seinem Web-Browser eine Seite an, so wird diese Anfrage vom Browser als GET-Transaktion an den Server geschickt. Um beispielsweise die Seite http://paris.fh-friedberg.de/index.html zu laden, wird folgendes Kommando an den Server paris.fh-friedberg.de gesendet:

```
GET /index.html
```

Der erste Teil gibt den Kommandonamen an, dann folgt die gewünschte Datei. Die Zeile muss mit einer CRLF-Sequenz abgeschlossen werden, ein einfaches '\n' reicht nicht aus. Der Server versucht nun die angegebene Datei zu laden und überträgt sie an den Client. Ist der Client ein Web-Browser, wird er den darin befindlichen HTML-Code interpretieren und auf dem Bildschirm anzeigen. Befinden sich in der Seite Verweise auf Images, Applets oder Frames, so fordert der Browser die fehlenden Seiten in weiteren GET-Transaktionen von deren Servern ab.

Die Struktur des GET-Kommandos wurde mit der Einführung von HTTP 1.0 etwas erweitert. Zusätzlich werden nun am Ende der Zeile eine Versionskennung und wahlweise in den darauffolgenden Zeilen weitere Headerzeilen mit Zusatzinformationen mitgeschickt. Nachdem die letzte Headerzeile gesendet wurde, folgt eine leere Zeile (also ein alleinstehendes CRLF), um das Kommandoende anzuzeigen. HTTP 1.0 ist weit verbreitet, und das obige Kommando würde von den meisten Browsern in folgender Form gesendet werden (jede der beiden Zeilen muss mit CRLF abgeschlossen werden):

```
GET /index.html HTTP/1.0
```

Wird HTTP/1.0 verwendet, ist auch die Antwort des Servers etwas komplexer. Anstatt lediglich den Inhalt der Datei zu senden, liefert der Server seinerseits einige Headerzeilen mit Zusatzinformationen, wie beispielsweise den Server-Typ,

das Datum der letzten Änderung oder den MIME-Typ der Datei. Auch hier ist jede Headerzeile mit einem CRLF abgeschlossen, und nach der letzten Headerzeile folgt eine Leerzeile. Erst dann beginnt der eigentliche Dateiinhalt.

Das folgende Programm kann dazu verwendet werden, eine Datei von einem Web-Server zu laden. Es wird mit einem Host- und einem Dateinamen als Argument aufgerufen und lädt die Seite vom angegebenen Server. Das Ergebnis wird auf dem Bildschirm angezeigt.

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

public class httpget {

 public static void main(String[] args) {
  if (args.length != 2) {
   System.err.println(
     "Usage: java httpget <host> <file>");
   System.exit(1);
  }
  try {
   Socket sock = new Socket(args[0], 80);
   OutputStream out = sock.getOutputStream();
   InputStream in = sock.getInputStream();

   //GET-Kommando senden

   String s = "GET " + args[1] +
      " HTTP/1.0" + "\r\n\r\n";
   out.write(s.getBytes());

   //Ausgabe lesen und anzeigen

   int len;
   byte[] b = new byte[100];
   while ((len = in.read(b)) != -1) {
    System.out.write(b, 0, len);
   }

   //Programm beenden

   in.close();
```

```
      out.close();
      sock.close();
     } catch (IOException e) {
       System.err.println(e.toString());
       System.exit(1);
     }
    }
   }
```

5.2 Server-Sockets

Der wesentliche Unterschied zwischen Java-Client- und Java-Server-Programmen liegt in der Art des Verbindungsaufbaus, für den es eine spezielle Klasse ServerSocket gibt. Diese Klasse stellt Methoden zur Verfügung, um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgtem Verbindungsaufbau einen Socket zur Kommunikation mit dem Client zurückzugeben. Bei der Klasse java.net.ServerSocket sind im wesentlichen der Konstruktor und die Methode accept von Interesse:

```
public ServerSocket(int port)
throws IOException
public Socket accept()
throws IOException
```

Der Konstruktor erzeugt einen ServerSocket für einen bestimmten Port, also einen bestimmten Typ von Serveranwendung. Anschliessend wird die Methode accept aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten. accept blockiert so lange, bis sich ein Client bei der Serveranwendung anmeldet also einen Verbindungsaufbau zu unserem Host unter der Portnummer, die im Konstruktor angegeben wurde, initiiert. Ist der Verbindungsaufbau erfolgreich, liefert accept ein Socket-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann. Anschliessend steht der ServerSocket für einen weiteren Verbindungsaufbau zur Verfügung oder kann mit close geschlossen werden.

5.2.1 Echo-Server für einen Client

Wir wollen uns die Konstruktion von Servern an einem Beispiel ansehen. Dazu soll ein einfacher ECHO-Server geschrieben werden, der auf Port 7000 auf Verbindungswünsche wartet. Alle eingehenden Daten sollen unverändert an den

Client zurückgeschickt werden. Zur Kontrolle sollen sie ebenfalls auf die Konsole ausgegeben werden:

```
import java.net.*;
import java.io.*;
public class SimpleEchoServer {
 public static void main(String[] args) {
  try {
   System.out.println("Warte auf Port 7000");
   ServerSocket echod = new ServerSocket(7000);
   Socket socket = echod.accept();
   System.out.println("Verbindung hergestellt");
   InputStream in = socket.getInputStream();
   OutputStream out = socket.getOutputStream();
   int c;
   while ((c = in.read()) != -1) {
    out.write((char)c);
    System.out.print((char)c);
   }
   System.out.println("Verbindung beenden");
   socket.close();
   echod.close();
  } catch (IOException e) {
   System.err.println(e.toString());
   System.exit(1);
  }
 }
}
```

Wird der Server gestartet, kann via Telnet oder mit dem EchoClient auf den Server zugegriffen werden:

```
telnet localhost 7
```

Wenn der Server läuft, werden alle eingegebenen Zeichen direkt vom Server zurückgesendet und als Echo in Telnet angezeigt. Läuft er nicht, gibt es beim Verbindungsaufbau eine Fehlermeldung.

5.2.2 Echo-Server für mehrere Clients

Wir wollen das vorgestellte Programm nun in mehrfacher Hinsicht erweitern:

- Der Server soll mehr als einen Client gleichzeitig bedienen können.
- Die Clients sollen zur besseren Unterscheidung durchnumeriert werden.
- Beim Verbindungsaufbau soll der Client eine Begrüssungsmeldung erhalten.
- Für jeden Client soll ein eigener Thread angelegt werden.

Um diese Anforderungen zu erfüllen, verändern wir das obige Programm ein wenig. Im Hauptprogramm wird nun nur noch der ServerSocket erzeugt und in einer Schleife jeweils mit accept auf einen Verbindungswunsch gewartet. Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung nicht mehr im Hauptprogramm, sondern es wird ein neuer Thread mit dem Verbindungs-Socket als Argument erzeugt. Dann wird der Thread gestartet und erledigt die gesamte Kommunikation mit dem Client. Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet. Das Hauptprogramm braucht sich nur noch um den Verbindungsaufbau zu kümmern und ist von der eigentlichen Client-Kommunikation vollständig befreit.

```
import java.net.*;
import java.io.*;
public class EchoServer {
 public static void main(String[] args) {
  int cnt = 0;
  try {
   System.out.println("Warte auf Port 7000");
   ServerSocket echod = new ServerSocket(7000);
   while (true) {
    Socket socket = echod.accept();
    (new EchoClientThread(++cnt, socket)).start();
   }
  } catch (IOException e) {
   System.err.println(e.toString());
   System.exit(1);
  }
 }
}

class EchoClientThread extends Thread {
 private int name;
 private Socket socket;
```

```
public EchoClientThread(int name, Socket socket) {
 this.name = name;
 this.socket = socket;
}

public void run() {
 String msg = "EchoServer: Verbindung " + name;
 System.out.println(msg + " hergestellt");
 try {
  InputStream in = socket.getInputStream();
  OutputStream out = socket.getOutputStream();
  out.write((msg + "\r\n").getBytes());
  int c;
  while ((c = in.read()) != -1) {
   out.write((char)c);
   System.out.print((char)c);
  }
  System.out.println("Verbindung "
      + name + " wird beendet");
  socket.close();
 } catch (IOException e) {
  System.err.println(e.toString());
 }
}
}
```

Zur besseren Übersicht werden alle Client-Verbindungen durchnumeriert und als erstes Argument an den Thread übergeben. Unmittelbar nach dem Verbindungsaufbau wird diese Meldung auf der Server-Konsole ausgegeben und an den Client geschickt. Anschliessend wird in einer Schleife jedes vom Client empfangene Zeichen an diesen zurückgeschickt, bis er von sich aus die Verbindung unterbricht. Man kann den Server leicht testen, indem man mehrere Telnet-Sessions zu ihm aufbaut. Jeder einzelne Client sollte eine Begrüssungsmeldung mit einer eindeutigen Nummer erhalten und autonom mit dem Server kommunizieren können. Der Server sendet alle Daten zusätzlich an die Konsole und gibt sowohl beim Starten als auch beim Beenden eine entsprechende Meldung auf der Konsole aus.

5.2.3 Prototyp eines Webserver

Die Kommunikation zwischen einem Browser und einem Web-Server entspricht etwa folgendem Schema:

- Der Web-Browser baut eine Verbindung zum Server auf.
- Er schickt eine Seitenanforderung und ein paar zusätzliche Informationen in Form eines Requests gemäss HTTP-Spezifikation.
- Der Server analysiert den Request und schickt die gewünschte Datei bzw. eine Fehlermeldung an den Browser.
- Der Server beendet die Verbindung.
- Hat der Browser auf diese Weise eine HTML-Seite erhalten, interpretiert er den HTML-Code und zeigt die Seite formatiert auf dem Bildschirm an.
- Enthält die Datei IMG-, APPLET- oder ähnliche Elemente, werden diese in derselben Weise vom Server angefordert und in die Seite eingebaut.

Die wichtigste Aufgabe des Servers besteht also darin, eine Datei an den Client zu übertragen. Wir wollen uns zunächst das Listing ansehen und dann auf Details der Implementierung eingehen:

```
import java.io.*;
import java.util.*;
import java.net.*;

/************************************
 Dieser einfache Webserver ist in der
 Lage, Seitenanforderungen lokal zu dem
 Verzeichnis, aus dem er gestartet wurde,
 zu bearbeiten. Wurde der Server z.B. im
 Verzeichnis /tmp gestartet, so würde eine
 Seitenanforderung http://localhost/xy.htm
 die Datei /tmp/xy.htm laden. Die Dateitypen
 .htm, .html, .gif, .jpg und .jpeg werden
 erkannt und mit korrekten MIME-Headern
 übertragen, alle anderen Dateien werden als
 "application/octet-stream" übertragen.
 Jeder Request wird durch einen eigenen
 Client-Thread bearbeitet, nach Übertragung
 der Antwort schliesst der Server den Socket.
 Antworten werden mit HTTP/1.0-Header
 gesendet.
****************************************/
```

```
public class ExperimentalWebServer {

 public static void main(String[] args) {

  if (args.length != 1) {
   System.err.println("Usage: java
   ExperimentalWebServer <port>");
   System.exit(1);
  }
  try {
   int port = Integer.parseInt(args[0]);
   System.out.println("Listening on port "
    + port);
   int calls = 0;
   ServerSocket httpd =
    new ServerSocket(port);
   while (true) {
    Socket socket = httpd.accept();
    (new BrowserClientThread(++calls,
     socket)).start();
   }
  } catch (IOException e) {
   System.err.println(e.toString());
   System.exit(1);
  }
}
}

/**************************************
  Die Thread-Klasse für die Client-
  Verbindung.
**************************************/

class BrowserClientThread
extends Thread {

static final String[][] mimetypes = {
{"html", "text/html"}, {"htm", "text/html"},
{"txt", "text/plain"}, {"gif", "image/gif"},
{"jpg", "image/jpeg"}, {"jpeg", "image/jpeg"},
{"jnlp", "application/x-java-jnlp-file"}
```

```
};

private Socket socket;
private int id;
private PrintStream out;
private InputStream in;
private String cmd;
private String url;
private String httpversion;

/*****************************************
 Erzeugt einen neuen Client-Thread mit der
 angegebenen id und dem angegebenen Socket.
*****************************************/

public BrowserClientThread(int id,
  Socket socket) {
 this.id = id;
 this.socket = socket;
}

/*****************************************
 Hauptschleife für den Thread.
*****************************************/

public void run() {
 try {
  System.out.println(id +
    ": Incoming call...");
  out =
   new PrintStream(socket.getOutputStream());
  in = socket.getInputStream();
  readRequest();
  createResponse();
  socket.close();
  System.out.println(id + ": Closed.");
 } catch (IOException e) {
  System.out.println(id + ": " +
   e.toString());
  System.out.println(id + ": Aborted.");
 }
```

```
}

/***********************************
 Liest den nächsten HTTP-Request vom
 Browser ein.
***********************************/

private void readRequest()
throws IOException {

 //Request-Zeilen lesen

 Vector request = new Vector(10);
 StringBuffer sb = new StringBuffer(100);

 int c;
 while ((c = in.read()) != -1) {
  if (c == '\r') { //ignore }
  else if (c == '\n') {

   //line terminator

   if (sb.length() <= 0) { break; }
   else {
    request.addElement(sb);
    sb = new StringBuffer(100);
   }
  } else {
   sb.append((char)c);
  }
 }

 //Request-Zeilen auf der Konsole ausgeben

 Enumeration e = request.elements();
 while (e.hasMoreElements()) {
  sb = (StringBuffer)e.nextElement();
  System.out.println("< " + sb.toString());
 }

 //Kommando, URL und HTTP-Version extrahieren
```

```
 String s = ((StringBuffer)request.
  elementAt(0)).toString();
 cmd = "";
 url = "";
 httpversion = "";

 int pos = s.indexOf(' ');
 if (pos != -1) {
  cmd = s.substring(0, pos).toUpperCase();
  s = s.substring(pos + 1);
  //URL
  pos = s.indexOf(' ');
  if (pos != -1) {
   url = s.substring(0, pos);
   s = s.substring(pos + 1);
   //HTTP-Version
   pos = s.indexOf('\r');
   if (pos != -1) {
    httpversion = s.substring(0, pos);
   } else {
    httpversion = s;
  }
 } else {
  url = s;
 }
}
}

/**************************************
Request bearbeiten und Antwort erzeugen.
**************************************/

private void createResponse() {
 if (cmd.equals("GET") || cmd.equals("HEAD")) {
  if (!url.startsWith("/")) {
   httpError(400, "Bad Request");
  } else {

   //MIME-Typ aus Dateierweiterung bestimmen
```

```
    String mimestring = "application/octet-stream";
    for (int i = 0;
         i < mimetypes.length; ++i) {
     if (url.endsWith(mimetypes[i][0])) {
      mimestring = mimetypes[i][1];
      break;
     }
    }

    //URL in lokalen Dateinamen konvertieren

    String fsep = System.getProperty(
     "file.separator", "/");
    StringBuffer sb = new StringBuffer(
     url.length());
    for (int i = 1; i < url.length(); ++i) {
     char c = url.charAt(i);
     if (c == '/') {
      sb.append(fsep);
     } else {
      sb.append(c);
    }
  }
  try {
   FileInputStream is = new
    FileInputStream(sb.toString());

   //HTTP-Header senden

   out.print("HTTP/1.0 200 OK\r\n");
   System.out.println("> HTTP/1.0 200 OK");
   out.print("Server: WebServer \r\n");
   System.out.println("> WebServer");
   out.print("Content-type: " + mimestring
    + "\r\n\r\n");
   System.out.println("> Content-type: " +
    mimestring);
   if (cmd.equals("GET")) {

    //Dateiinhalt senden
```

```
    byte[] buf = new byte[256];
    int len;
    while ((len = is.read(buf)) != -1) {
     out.write(buf, 0, len);
    }
   }
   is.close();
  } catch (FileNotFoundException e) {
   httpError(404, "Error Reading File");
  } catch (IOException e) {
   httpError(404, "Not Found");
  } catch (Exception e) {
   httpError(404, "Unknown exception");
  }
  }
} else { httpError(501, "Not implemented");
}
}

/**************************************
 Eine Fehlerseite an den Browser senden.
**************************************/
private void httpError(int code,
 String description) {
 System.out.println("> ***" + code + ": " +
  description + "***");
 out.print("HTTP/1.0 " + code + " " +
   description + "\r\n");
 out.print("Content-type:
   text/html\r\n\r\n");
 out.println("<html>");
 out.println("<head>");
 out.println("<title>
  WebServer-Error</title>");
 out.println("</head>");
 out.println("<body>");
 out.println("<h1>HTTP/1.0 " +
  code + "</h1>");
 out.println("<h3>" +
  description + "</h3>");
 out.println("</body>");
```

```
 out.println("</html>");
 }
}
```

Der Web-Server besteht aus den beiden Klassen ExperimentalWebServer und BrowserClientThread. Nachdem in ExperimentalWebServer eine Verbindung aufgebaut wurde, wird ein neuer Thread erzeugt und die weitere Bearbeitung des Requests an ein Objekt der Klasse BrowserClientThread delegiert. Der in run liegende Code beschafft zunächst die Ein- und Ausgabestreams zur Kommunikation mit dem Socket und ruft dann die beiden Methoden readRequest und createResponse auf. Anschliessend wird der Socket geschlossen und der Thread beendet.

In readRequest wird der HTTP-Request des Browsers gelesen, der aus mehreren Zeilen besteht. In der ersten wird die eigentliche Dateianforderung angegeben, die übrigen liefern Zusatzinformationen wie den Typ des Browsers, akzeptierte Dateiformate und ähnliches. Alle Zeilen werden mit CRLF abgeschlossen, nach der letzten Zeile des Requests wird eine Leerzeile gesendet.

Ein typischer Request könnte etwa so aussehen:

```
GET /ansisys.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (Win95; I)
Host: localhost:80
Accept: image/gif, image/x-xbitmap,
image/jpeg, image/pjpeg, image/png, */*
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
HTTP/1.0 200 OK Server: ExperimentalWebServer
Content-type: text/html
```

Unser Web-Server liest den Request zeilenweise in den Vector request ein und gibt alle Zeilen zur Kontrolle auf der Konsole aus. Anschliessend wird das erste Element extrahiert und in die Bestandteile Kommando, URL (Dateiname) und HTTP-Version zerlegt. Diese Informationen werden zur weiteren Verarbeitung in den Membervariablen cmd, url und httpversion gespeichert.

Nachdem der Request gelesen wurde, wird in createResponse die Antwort erzeugt. Zunächst prüft die Methode, ob es sich um ein GET- oder HEAD-Kommando handelt (HTTP kennt noch mehr). Ist das nicht der Fall, wird durch Aufruf von httpError eine Fehlerseite an den Browser gesendet. Andernfalls fährt die Methode mit der Bestimmung des Dateityps fort. Der Dateityp wird mit Hilfe der Arraykonstante mimetypes anhand der Dateierweiterung bestimmt und in einen

passenden MIME-Typ konvertiert, der im Antwortheader an den Browser übertragen wird. Der Browser entscheidet anhand dieser Information, was mit der nachfolgend übertragenen Datei zu tun ist z.B. Anzeige als Text, Anzeige als Grafik, Speichern in einer Datei. Wird eine Datei angefordert, deren Erweiterung nicht bekannt ist, sendet der Server sie als application/octet-stream an den Browser, damit dieser dem Anwender die Möglichkeit geben kann, die Datei auf der Festplatte zu speichern.

Der Mime-Typ application/x-java-jnlp-file wird für den Betrieb von Java Web Start benötigt. Dieses seit dem JDK 1.4 verfügbare Werkzeug dient zum Laden, Aktualisieren und Starten von Java-Programmen über Internet-Verbindungen .

Nun wandelt der Server den angegebenen Dateinamen gemäss den Konventionen seines eigenen Betriebssystems um. Dazu wird das erste "/" aus dem Dateinamen entfernt (alle Dateien werden lokal zu dem Verzeichnis geladen, aus dem der Server gestartet wurde) und alle "/" innerhalb des Pfadnamens werden in den lokalen Pfadseparator konvertiert (unter MS-DOS ist das der Backslash). Dann wird die Datei mit einem FileInputStream geöffnet und der HTTP-Header und der Dateiinhalt an den Client gesendet. Konnte die Datei nicht geöffnet werden, wird eine Ausnahme ausgelöst und der Server sendet eine Fehlerseite.

Der vom Server gesendete Header ist ähnlich aufgebaut wie der Request-Header des Clients. Er enthält mehrere Zeilen, die durch CRLF-Sequenzen voneinander getrennt sind. Nach der letzten Headerzeile folgt eine Leerzeile, also zwei aufeinanderfolgende CRLF-Sequenzen. HTTP 1.0 und 1.1 spezifizieren eine ganze Reihe von (optionalen) Headerelementen, von denen wir lediglich die Versionskennung, unseren Servernamen und den MIME-Bezeichner mit der Typkennung der gesendeten Datei an den Browser übertragen. Unmittelbar nach dem Ende des Headers wird der Dateiinhalt übertragen. Eine Umkodierung erfolgt dabei normalerweise nicht, alle Bytes werden unverändert übertragen.

Unser Server kann sehr leicht getestet werden. Am einfachsten legt man ein neues Unterverzeichnis an und kopiert die übersetzten Klassendateien und einige HTML-Dateien in dieses Verzeichnis. Nun kann der Server wie jedes andere Java-Programm gestartet werden. Beim Aufruf ist zusätzlich die Portnummer als Argument anzugeben:

```
java ExperimentalWebServer 80
```

Nun kann ein normaler Web-Browser verwendet werden, um Dateien vom Server zu laden. Befindet sich beispielsweise eine Datei index.html im Server-Verzeichnis und läuft der Server auf derselben Maschine wie der Browser, kann die Datei über die Adresse http://localhost/index.html im Browser geladen werden. Auch über das lokale Netz des Unternehmens oder das Internet können leicht Dateien geladen werden. Hat der Host, auf dem der Server läuft, keinen Nameserver-Eintrag, kann statt dessen auch direkt seine IP-Adresse im Browser angegeben werden.

Auf einem UNIX-System darf ein Server die Portnummer 80 nur verwenden, wenn er Root-Berechtigung hat. Ist das nicht der Fall, kann der Server alternativ auf einem Port grösser 1023 gestartet werden:

```
java ExperimentalWebServer 7777
```

Im Browser muss die Adresse dann ebenfalls um die Portnummer ergänzt werden:

```
http://localhost:7777/index.html
```

Kapitel 6

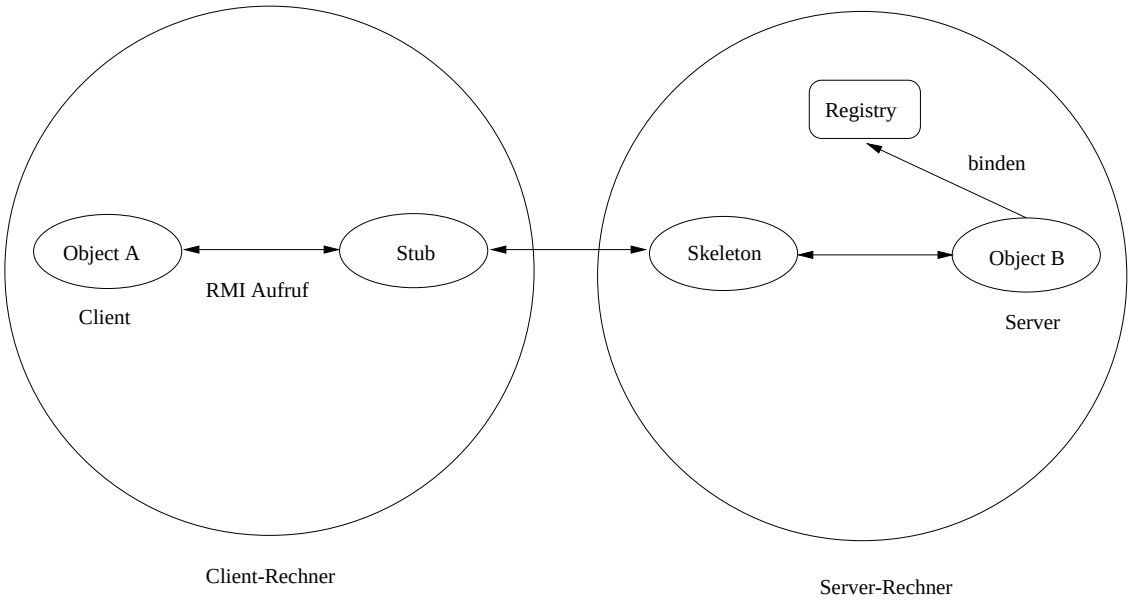
Remote Method Invocation (RMI)

Als Sun die Java-Sprache entwickelt hatte, war die Socket-Kommunikation der einzige Kommunikationsmechanismus für verteilte Objekte. Sun erkannte bald, dass ein Mechanismus auf einer höheren Abstraktionsstufe gebraucht wurde. Deshalb wurde das Objekt Modell durch RMI erweitert. RMI erlaubt Objekten, auf ihrem Abstraktionsniveau durch den Aufruf von entfernten Methoden zu kommunizieren.

Die Grundidee von RMI ist das Konzept eines entfernt bereitgestellten Objekts (remote enabled object). Wenn ein Objekt entfernt bereitgestellt ist, können irgendwelche Objekte auf anderen virtuellen Maschinen im Netz mit ihm interagieren.

Wenn ein Client-Objekt A mit einem Server-Objekt B kommuniziert und beide auf verschiedenen virtuellen Maschinen laufen, interagiert A mit einer entfernten Schnittstelle von B (remote interface von B). Die entfernte Schnittstelle definiert die Methoden von B, die von nicht lokalen Objekten verwendet werden können. A interagiert nie direkt mit B, sondern mit dem Interface von B. Durch einen Namensdienst (naming service), der sich Rebistry nennt, erhält A eine Referenz auf B.

Wenn A eine Methode auf dem remote Interface von B aufruft, erhält ein Proxy-Objekt, das Stub (Stummel) genannt wird diesen Aufruf. Der Stub leitet den Aufruf über das Netzwerk zu einem Skeleton auf der Server-Maschine. Der Skeleton ruft die Methode des Objekts B auf und leitet die Ergebnisse des Aufrufs an den Stub auf der Client-Maschine wieder zurück, der seinerseits die Ergebnisse an A übergibt.

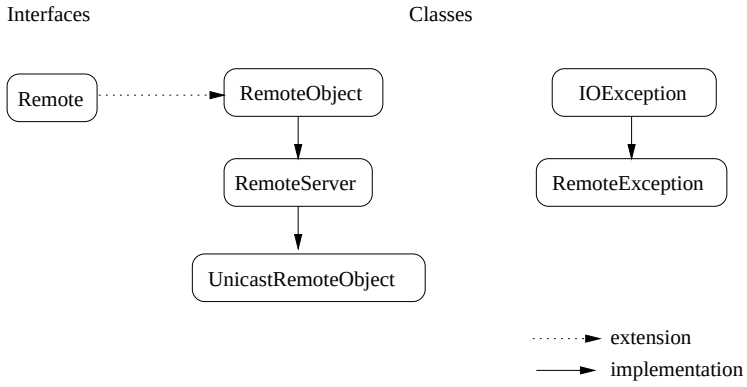


In RMI können entfernte Objekte (remote objects) als Argumente von Methoden oder als Ergebnisse von Methoden-Aufrufen zwischen Client und Server ausgetauscht werden. Während lokale Objekte (local objects) als Methoden -Argumente Wertparameter sind, werden entferte Objekte als Referenzparameter übergeben.

6.1 Eigenschaften von RMI

Java RMI stellt Programmierern viele sinnvolle Eigenschaften zur Konstruktion verteilter Anwendungen zur Verfügung. Viele dieser Eigenschaften stammen daher, dass RMI ein Java-spezifisches Modell verteilter Objekte ist. (Es können nur Java Objekte über RMI miteinander kommunizieren.) Z.B. ist die Definition eines entfernten Interface zu dem eines lokalen Interface analog. Es wird keine spezielle Sprache zur Definition eines entfernten Interface benötigt. Das folgende Bild zeigt die speziellen Interfaces und Klassen, die von RMI benötigt werden, um eine verteilte Applikation zu entwickeln.

Der Programmierer definiert zunächst ein Interface der entfernten Methoden, die an Clients exportiert werden sollen. Danach spezifiziert er die Implementation der Methoden. Alle Interfaces entfernter Objekte müssen das Interface Remote erweitern, das im Paket java.rmi definiert ist. Entfernte Methodenaufrufe werfen grundsätzlich im Fehlerfall ein Objekt der Klasse RemoteException aus. Die RemoteException ist notwendig wegen auftretender Fehler beim entfernten Methondenaufruf, z.B. Netzwerkfehler oder Server-Rechner läuft nicht.



Die Server-Klasse implementiert dann dieses Interface. Sie ist ein Nachfahre der Klasse UnicastRemoteObject.

Bei der Kommunikation zwischen Client und Server (wenn also der Client eine Methode auf einem Remote-Objekt aufruft) sind drei Arten von Datentypen zu unterscheiden:

- Werden beim Aufruf einer Methode primitive Datentypen übergeben oder zurückgegeben (int, char, boolean, usw.), werden sie wie gewöhnlich per Wert übergeben (call by value). In diesem Fall besteht also überhaupt kein Unterschied zu lokalen Java-Programmen.
- Lokale Objekte können dagegen nur dann als Parameter oder Rückgabewert verwendet werden, wenn sie serialisierbar sind. Sie werden bei der Übertragung kopiert und somit ebenfalls per Wert übergeben. Viele Klassen, wie etwa String, Calendar, die numerischen Wrapper-Klassen oder die Collection-Klassen sind bereits serialisierbar und können direkt verwendet werden. Eigene Klassen müssen das Interface Serializable implementieren.
- Verweise auf Remote-Objekte, wie sie beispielsweise vom Namens-Service zurückgegeben werden, haben dagegen Referenzcharakter und werden wie gewöhnliche Objektreferenzen behandelt.

Objekte, die weder Remote-Referenzen noch serialisierbar sind, können per RMI nicht ausgetauscht werden. Beispiele dafür sind die Klassen Thread, System oder RandomAccessFile. Derartige Objekte haben allerdings auch meist nur lokale Bedeutung. Die Übertragung an eine andere virtuelle Maschine macht keinen Sinn.

6.2 Einfaches Beispiel für Client und Server

Die folgende Client-Server-Anwendung soll folgendes leisten: Beim Aufruf des Servers vom Client wird dem Client eine Zeichenkette, z.B. “hello world”, zu-

rückgegeben.

6.2.1 Entwicklung des Server

Der erste Schritt bei der Entwicklung der Anwendung ist das Design des Remote Interface. Durch Vererbung aus dem Interface java.rmi.Remote zeigt man im Interface an, dass die Methoden dieses Interface von jeder virtuellen Java-Maschine aufgerufen werden können. Die Methode hello_method() ist eine RMI-Methode, also muss die Fehlerbehandlung der Netzverbindung durch das Objekt RemoteException definiert werden.

```
import java.rmi.*;
interface hello_interface extends Remote {
     public String hello_method()
      throws RemoteException;
}
```

Die Implementierung des Remote Interface ist die Server-Klasse mit der Server-Methode hello_method(). Der Server muss die entfernten Objekte erzeugen und installieren. Diese Initialisierungen können in einer main-Methode durchgeführt werden. Die main-Methode kann dabei (wie im Beispiel unten) in der Server-Klasse oder auch in einer andernen Klasse definiert werden. Grundsätzlich sollte die Start-Prozedur des Servers folgende Schritte beinhalten:

- erzeugen und installieren eines Security Manager
- erzeugen einer oder mehrerer Instanzen eines entfernten Objekts
- registrieren eines oder mehrerer Objekte im RMI-Registry

Die Implementation der Server-Klasse hello_server zum Interface hello_interface ist wie folgt deklariert:

```
public class hello_server
extends UnicastRemoteObject
implements hello_interface {
...
}
```

Diese Deklaration besagt, dass die Klasse hello_server das Interface hello_interface implementiert und ein Nachfahre der Klasse UnicastRemoteObject ist. Die Superklasse UnicastRemoteObject enthält eine Reihe von Konstruktoren und statischen Methoden, um ein Objekt zu exportieren, d.h. das Server-Objekt für Client-Requests vorzubereiten. Durch diese Vererbung können dann Unicast-Verbindungen

(Punkt zu Punkt) mit dem RMI zu Grunde liegenden Socket-Mechanismus durchgeführt werden.

Als nächstes wird ein Konstruktor für die Klasse hello_server angegeben:

```
public hello_server()
 throws RemoteException {
  super();
 }
```

Die Klasse hello_server hat also nur einen Konstruktor ohne Argumente. Der Konstruktor ruft nur den Konstruktor der Superklasse UnicastRemoteObject super() auf. Dies würde auch geschehen, wenn wir den Konstruktor super() weggelassen hätten. Er wird also nur aufgeführt, um Klarheit über den Ablauf zu schaffen. Da der Konstruktor der Superklasse UnicastRemoteObject die Exception RemoteException deklariert, muss auch der Konstruktor hello_server() diese Exception deklarieren. Eine RemoteException kann z.B. auftreten, wenn Ressourcen zur Kommunikation nicht verfügbar sind oder wenn eine passende Stub-Klasse nicht gefunden werden kann.

Nun folgt die Implementation für die entfernte Methode hello_method(). Der Rückgabewert der Metheode ist ein lokales Objekt der Klasse String. Es kann direkt versendet werden, da die Klasse String bereits serialisierbar ist, d.h. das Interface Serializable implementiert.

```
public String hello_method()
 {
  return "hello world";
 }
```

Die Klasse des entfernten Object muss für jede entfernte Methode aus dem Interface eine Implementation zur Verfügung. Hier gibt es nur eine entfernte Methode hello_method().

Mit der main()-Methode der Klasse hello_server wird ein Server-Objekt angelegt und in der RMI-Registry angemeldet. Die main()-Methode ist eine lokale und keine entfernte Methode, d.h. sie kann nicht von einer anderen virtuellen Maschine aufgerufen werden. Als statische Methode ist sie an die Klasse hello_server und nicht an Instanzen dieser Klasse gebunden.

```
public static void main(String args[]) {
...
}
```

Der erste Schritt der main()-Methode ist das Anlegen und Installieren des Security Manager, der das lokale Systems vor unerlaubten Zugriffen auf System Ressourcen durch heruntergeladenen Java-Kode von einer anderen virtuellen Maschine schützt. Z.B. überwacht der Security Manager den Zugriff auf das lokale Dateisystem.

```
if (System.getSecurityManager() ==
  null) {
  System.setSecurityManager(
   new RMISecurityManager());
 }
```

Die RMI-Kommunikation zwischen Client und Server könnte auch völlig ohne SecurityManager, Web-Server und Policy-Dateien durchgeführt werden. In diesem Fall wäre es aber nicht möglich, zur Laufzeit Bytecode zwischen den beiden Maschinen zu übertragen.

Nun wird ein Objekt der entfernten Klasse durch folgende Anweisung angelegt:

```
hello_interface hello_object =
        new hello_server();
```

Der Typ der Variablen hello_object ist hello_interface und nicht hello_server. Das zeigt an, dass das Client-Interface aus den Methoden von hello_interface und nicht aus den Methoden der Klasse hello_server besteht.

Bevor ein Client entfernte Methoden aufrufen kann, muss er eine Referenz zu dem entfernten Objekt finden. Hierfür wird in Java das RMI-Registry verwendet, bei dem sich das Server-Objekt mit einem Namen anmeldet.

```
try {
   System.out.println("Server:
    Creating myself");
   String name = "hello_server";
   Naming.rebind(name, hello_object);
   System.out.println("Server: ready");
} catch (Exception e) {
   System.out.println("Server: Exeption" +
                      e.getMessage());
   e.printStackTrace();
}
```

Das Interface java.rmi.Naming wird benutzt, um entfernte Objekte zu registrieren und wiederzufinden. Nach der Anmeldung des Objekts mit der Methode Naming.rebind() kann das Objekt unter dem angegebenen Namen von Client-Objekten über die RMI-Registry gefunden werden. Da Naming.rebind() RemoteExceptions generieren kann, muss es über ein try ... catch-Klausel abgearbeitet werden. Alternativ kann die main()-Methode mit der throws-Klausel der RemoteException ausgestattet werden. Der erste Parameter der Naming.rebind()-Methode ist eine URL-formatierte Zeichenkette, der zweite Parameter ist der Objekt-Name. Nachdem das Server-Objekt angemeldet ist, kann der Server von Client-Objekten erreicht werden. Hier ist das komplette Beispielprogramm des Servers hello_server.java:

```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import hello_interface.*;

public class hello_server
extends UnicastRemoteObject
implements hello_interface {
 public hello_server()
 throws RemoteException {
  super();
 }
 public String hello_method()
 {
  return "hello world";
 }
 public static void main(String args[]) {
  if (System.getSecurityManager() ==
   null) {
   System.setSecurityManager(
    new RMISecurityManager());
  }
  try {
   System.out.println("Server:
    Creating myself");
   String name = "hello_server";
   hello_interface hello_object =
        new hello_server();
   Naming.rebind(name, hello_object);
   System.out.println("Server: ready");
```

```
  } catch (Exception e) {
   System.out.println("Server: Exeption" +
                      e.getMessage());
   e.printStackTrace();
  } /* catch */
 } /* main */
} /* class */
```

6.2.2 Entwicklung des Client

Das Client-Programm beginnt wie das Server-Programm mit der Installation des Security Manager:

```
if (System.getSecurityManager() ==
   null) {
  System.setSecurityManager(
    new RMISecurityManager());
}
```

Das ist notwendig, weil RMI auch Kode vom Server zum Client laden kann. Danach setzt der Client den Namen des aufzusuchenden Server-Objekts zusammen. Der Name des Server-Objekts wird in URL-Schreibweise angegeben, wobei durch rmi: angezeigt wird, dass es sich um ein RMI-Objekt handelt. Als nächstes kontaktiert der Client mit der Methode Naming.lookup() die RMI-Registry des Server-Rechners, um eine Referenz auf das Server-Objekt zu erhalten. Der Typ des Server-Objekts ist wieder das remote interface hello_interface, das durch einen Cast von der Naming.lookup()-Methode an das Objekt h übergeben wird. Sollte das Server-Objekt nicht gefunden werden, z.B. weil die URL falsch ist, der Server-Rechner oder Server-Prozess nicht gestartet ist oder die Netzwerkverbindung gestört ist, so wird eine Exception ausgelöst.

```
try {
 String
  name = "rmi://rom/hello_server";
 hello_interface h =
   (hello_interface)Naming.lookup(name);
 System.out.println(h.hello_method());
} catch (Exception e) {
 System.out.println(
   "Server: Exeption "+
```

```
      e.getMessage());
   e.printStackTrace();
  }
```

Wenn die Referenz auf das Server-Objekt korrekt zurückgeliefert wurde, wird zum Schluss die entfernte Methode hello_method() auf dem entfernten Objekt h aufgerufen.

6.2.3 Programmübersetzung

Die Eingabedateien zur Übersetzung des Server sind das remote interface hello_interface und die Server-Klasse hello_server.java. Durch die Übersetzung

```
javac hello_server.java
```

entstehen die class-Dateien hello_server.class und hello_interface.class. Im nächsten Schritt wird aus der class-Datei hello_server.class mit Hilfe des RMI-Compilers rmic der Stub und der Skeleton des Servers generiert.

```
rmic hello_server
```

Es entstehen die Dateien hello_server_Stub.class und hello_server_Skel.class.

Schließlich wird noch das Client-Programm hello_client.java übersetzt:

```
javac hello_client.java
```

6.2.4 Programmausführung

Für den Programmstart des Servers wird zunächst die RMI-Registry gestartet:

```
rmiregistry &
```

Als nächstes muss die Datei java.policy angegeben werden. Bei der Installation des Security Manager dient diese Datei zur Definition der Zugriffsrechte.

```
grant {
 permission java.net.SocketPermission
 "*:1024-65535", "connect,accept";
 permission java.net.SocketPermission
 "*:80", "connect";
};
```

Es wird ein Zugriff auf nicht privilegierte Portnummern und auf den Webserver (Port 80) erlaubt. Näheres zum Aufbau dieser Datei kann der Dokumentiation des JDK entnommen werden.

Nun kann der Server unter Angabe der Policy-Datei gestartet werden:

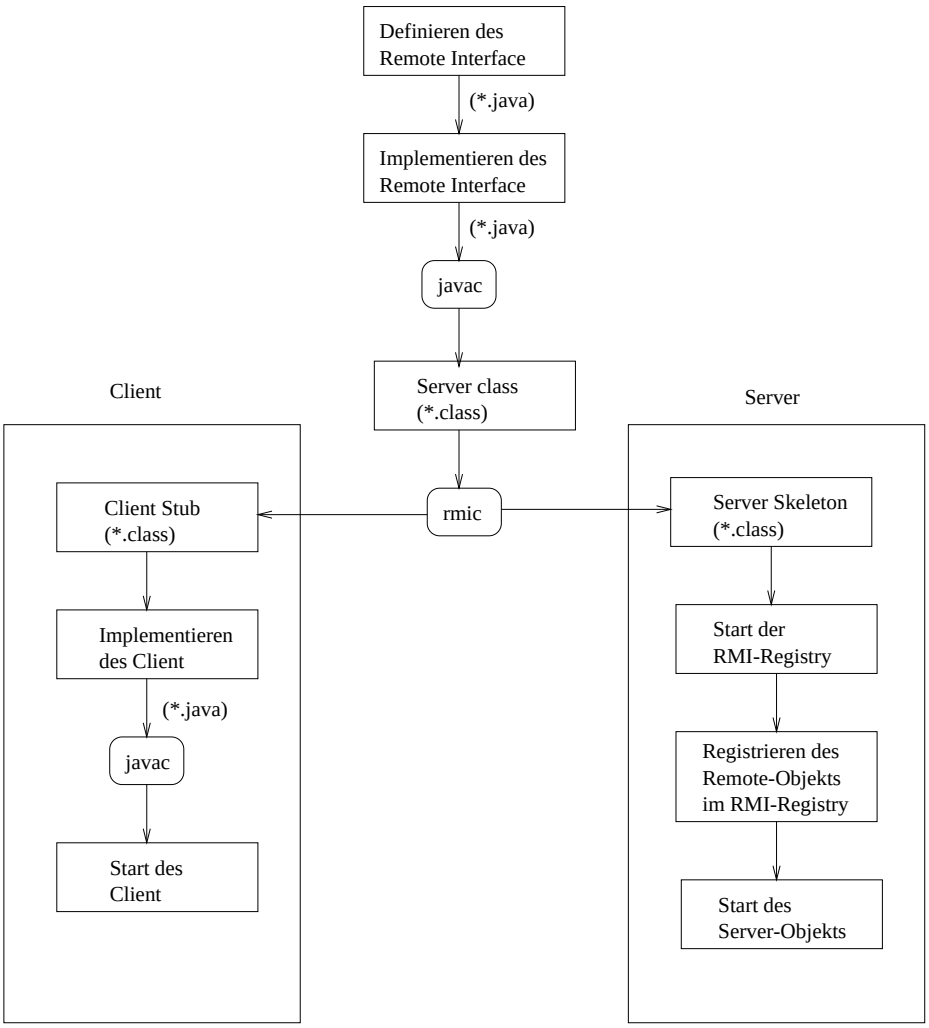
```
java -Djava.security.policy=
java.policy hello_server &
```

Der Client kann nun auf einer anderen virtuellen Maschine gestartet werden. Er benötigt ebenfalls eine Policy-Datei:

```
java -Djava.security.policy=
java.policy hello_client
```

6.2.5 Zusammenfassung

Die folgende Grafik zeigt noch einmal die wichtigsten Schritte bei der Entwicklung der RMI Applikation:



6.3 Dynamisches Laden und Ausführen von entferntem Byte-Kode

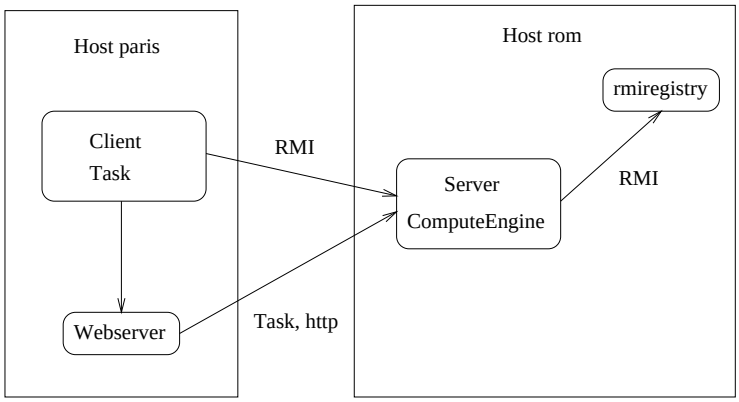
Eine der zentralen Eigenschaften von RMI ist die Fähigkeit, Byte-Kode auf eine entfernte virtuelle Maschine zu laden und auszuführen. Ein Objekt, das vorher nur auf einer virtuellen Maschine verfügbar war, kann auf eine andere virtuelle Maschine übertragen werden. RMI verändert den Typ des Objekts bei der Übertragung nicht, d.h. das Verhalten des Objekts bleibt gleich. Mit diesem Mechanismus können Objekte mit neuen Klassen auf einer entfernten virtuellen Maschine eingeführt werden. Das Verhalten einer Anwendung kann so dynamisch erweitert werden.

Als Beispiel für das dynamische Laden und Ausführen von entferntem Byte-Kode dient ein Objekt der Klasse ComputeEngine, das von einem Client-Objekt dynamisch eine Task in Form eines Byte-Kode entgegen nimmt, mit dem es eine Berechnung durchführt. Dieses Beispiel stammt aus dem Java Tutorial von Sun.

Als Verallgemeinerung dieses Beispiels können Clients mit Byte-Kodes für unterschiedliche Tasks die Eigenschaften eines Server-Rechners mit spezialisierter Hardware und Software (z.B. Rechen- oder Datenbankserver) nutzen.

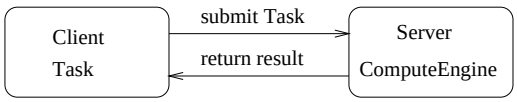
Die von dem ComputeEngine-Objekt bearbeitete Task muss also nicht mit der ComputeEngine-Klasse definiert werden. Sie kann also z.B. definiert werden, wenn das Objekt der Klasse ComputeEngine läuft. Solche Anwendungen werden häufig als behavior-based applications bezeichnet.

Damit Byte-Kode vom dynamisch geladen werden kann, wird allerdings kein eigenes, sondern das aus dem World Wide Web bekannte HTTP-Protokoll verwendet. Wie ein Web-Browser fragt also einer der beiden Teilnehmer per HTTP-GET-Transaktion bei seinem Partner nach der benötigten Klassendatei. Bei unserem Beispiel muss auf dem Client-Rechner ein Webserver laufen, um die Task vom Client- zum Server-Prozess zu übertragen:



6.3.1 Design des Remote Interface

Die Schnittstelle zwischen dem Client und der ComputeEngine erlaubt Tasks auf die ComputeEngine zu schicken und Ergebnisse vom Server an den Client zurückzusenden:



Zunächst wrid die entfernte Methode executeTask im remote interface Compute definiert. Als Parameter erhält diese Methode vom Client ein Objekt der Klasse Task; sie liefert ein Objekt der allgemeinen Klasse Object an den Client zurück. Die Datei compute.java enthält folgenden Kode:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute
extends Remote {
  Object executeTask(Task t)
  throws RemoteException;
}
```

Das zweite Interface definiert den Typ Task. Es besteht aus einer Methode execute() ohne Parameter mit Rückgabetyp Object. Damit Task übertragbar ist, ist es eine Erweiterung des interface Serializable. Die Datei Task.java enthält folgenden Kode:

```
package compute;
import java.io.Serializable;

public interface Task
extends Serializable {
  Object execute();
}
```

6.3.2 Entwicklung des Server

Die Implementation der ComputeEngine ist wie folgt deklariert:

```
public class ComputeEngine
extends UnicastRemoteObject
implements Compute
```

Zu den lokalen Methoden der Klasse ComputeEngine gehört der (einzige) Konstruktor der Klasse:

```
public ComputeEngine()
throws RemoteException {
  super();
}
```

Obgleich der Konstruktor der Superklasse UnicastRemoteObject in jedem Fall aufgerufen wird, wird er der Übersicht halber in den Rumpf des Konstruktors aufgenommen. Da der Konstruktor der Oberklasse UnicastRemoteObject eine RemoteException bewirken kann (z.B. wenn Kommunikations-Betriebsmittel nicht verfügbar sind), muss auch der Konstruktor ComputeEngine() eine entsprechende throws-Klausel besitzen.

Als nächstes wird die entfernte Methode executeTask() wie folgt implementiert:

```
public Object executeTask(Task t) {
  return t.execute();
}
```

Diese Methode implementiert das Protokoll zwischen ComputeEngine und einem Client. Ein Client stellt der ComputeEngine ein Task-Objekt t bereit. Dieses Task-Objekt t enthält eine Implementation der execute()-Methode, die innerhalb der entfernten Methode executeTask() aufgerufen wird:

```
t.execute()
```

Die main()-Methode der ComputeEngine-Klasse ist statisch (d.h. sie ist mit der Klasse und nicht mit Objekten assoziiert) und lokal (d.h. sie kann von keiner entfernten virtuellen Maschine aufgerufen werden).

```
public static void main(String[] args)
```

Als nächstes wird der Security Manager installiert:

```
if (System.getSecurityManager() ==
   null) {
   System.setSecurityManager(
    new RMISecurityManager());
  }
```

Nun wird ein Server-Objekt engine erzeugt und unter dem Namen ComputeEngine bei der RMI-Registry angemeldet:

```
String name = "ComputeEngine";
System.out.println(name);
try {
 Compute engine = new ComputeEngine();
 Naming.rebind(name, engine);
```

```
 System.out.println("ComputeEngine bound");
} catch (Exception e) {
 System.err.println("ComputeEngine
    exception: "+ e.getMessage());
 e.printStackTrace();
}
```

Wenn beim Erzeugen des Server-Objekts engine oder beim Anmelden des Objekts ein Fehler passiert, wird das Programm beendet. (Man könnte sich auch vorstellen, dass bei einem Fehler versucht wird, den Server automatisch auf einem anderen Rechner zu starten.)

Noch eine Bemerkung: Damit der Server-Prozess weiterläuft ist, es nicht erforderlich ein thread wait in main() einzubauen. Solange eine Referenz des Server-Objekts in einer lokalen oder entfernten virtuellen Maschine verzeichnet ist, wird das Server-Objekt nicht heruntergefahren. Auch die Anmeldung des Server-Objekts bei der RMI-Registry gilt als eine Referenz, so dass das Objekt erst nach dem Ende des rmiregistry-Prozesses oder nach einem unbind aus der Registry heruntergefahren wird.

Hier folgt noch einmal die gesamte Implementierung von ComputeEngine:

```
package engine;

import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class ComputeEngine
extends UnicastRemoteObject
implements Compute {
 public ComputeEngine()
 throws RemoteException {
  super();
 }
 public Object
 executeTask(Task t) {
  return t.execute();
 }
 public static void main(String[] args) {
  if (System.getSecurityManager() == null) {
   System.setSecurityManager(
    new RMISecurityManager());
```

```
  }
  String name = "ComputeEngine";
  System.out.println(name);
  try {
   Compute engine = new ComputeEngine();
   Naming.rebind(name, engine);
   System.out.println("ComputeEngine bound");
  } catch (Exception e) {
   System.err.println("ComputeEngine
      exception: " + e.getMessage());
   e.printStackTrace();
  }
 }
}
```

6.3.3 Entwicklung des Client

Das Client-Programm besteht aus zwei Teilen einem Teil zum Aufruf des ComputeEngine-Objekts und einem anderen Teil der die Task definiert, die das ComputeEngine-Objekt ausführen soll. Als Implementierung von Task soll eine Klasse Pi dienen, die wie der Name schon sagt die Zahl Pi auf eine angegebene Zahl von Nachkommastellen genau berechnet.

Die Implementierung Pi von Task ist für den Ablauf in RMI nicht weiter interessant, folgt aber nun der Vollständigkeit halber ausschnittsweise:

```
package client;

import compute.*;
import java.math.*;

public class Pi
implements Task {

/** constants used in pi computation */

private static final BigDecimal
 ZERO = BigDecimal.valueOf(0);
private static final BigDecimal
 ONE = BigDecimal.valueOf(1);
private static final BigDecimal
 FOUR = BigDecimal.valueOf(4);
```

```
/** rounding mode to use during pi computation */

private static final int
 roundingMode = BigDecimal.ROUND_HALF_EVEN;

/** digits of precision after the decimal point */
private int digits;
/**
 * Construct a task to
 * calculate pi to the specified
 * precision.
*/

public Pi(int digits) {
 this.digits = digits;
}

/**
 * Calculate pi.
 */
public Object execute() {
 return computePi(digits);
}

/**
 * Compute the value of pi
 * to the specified number of
 * digits after the decimal point.
 * The value is computed using
 * Machin's formula:
 * pi/4 = 4*arctan(1/5)
 *    - arctan(1/239)
 * and a power series expansion
 * of arctan(x) to sufficient precision.
 */

public static BigDecimal
 computePi(int digits) {
  int scale = digits + 5;
  BigDecimal arctan1_5 =
```

```
      arctan(5, scale);
    BigDecimal arctan1_239 =
     arctan(239, scale);
    BigDecimal pi =
     arctan1_5.multiply(FOUR).
      subtract( arctan1_239).multiply(FOUR);
    return pi.setScale(digits,
     BigDecimal.ROUND_HALF_UP);
  }
  public static BigDecimal
   arctan(int inverseX, int scale) {
  ...
  }
```

Die zweite Klasse ComputePi installiert zunächst den Security Manager, da ansonsten ein Laden der Task Pi vom Webserver der Client-Maschine auf das Server-Objekt nicht möglich ist:

```
if (System.getSecurityManager()
   == null) {
 System.setSecurityManager(
   new RMISecurityManager());
}
```

Dann sucht ComputePi das das Server-Objekt auf:

```
String name = "rmi://" + args[0]
   + "/ComputeEngine";
System.out.println(name);
Compute comp = (Compute) Naming.lookup(name);
```

Anschließend erzeugt ComputePi ein Objekt der Klasse Pi mit entsprechender Stellengenauigkeit.

```
Pi task = new Pi(Integer.parseInt(args[1]));
```

Zuletzt wird das Server-Objekt comp mit der Methode executeTask() aufgerufen, an die das Objekt task der Klasse Pi übergeben wird. Das Server-Objekt comp ruft dann von der übertragenen task die execute()-Methode auf, d.h. berechnet Pi und gibt das Ergebnis (als Objekt) an das Client-Objekt der Klasse ComputePI wieder zurück.

```
BigDecimal pi = (BigDecimal) (
  comp.executeTask(task));
System.out.println(pi);
```

Hier folgt die gesamte Implementierung von ComputePi:

```
package client;

import java.rmi.*;
import java.math.*;
import compute.*;

public class ComputePi {
 public static void main(String args[]) {
  if (System.getSecurityManager() == null) {
   System.setSecurityManager(
    new RMISecurityManager());
  }
  try {
   String name = "rmi://" +
    args[0] + "/ComputeEngine";
   System.out.println(name);
   Compute comp=(Compute)Naming.lookup(name);
   Pi task=new Pi(Integer.parseInt(args[1]));
   BigDecimal pi =
    (BigDecimal) (comp.executeTask(task));
   System.out.println(pi);
  } catch (Exception e) {
    System.err.println("ComputePi
      exception: " + e.getMessage());
    e.printStacktrace();
    }
  }
}
```

6.3.4 Programmübersetzung

Auf dem Server-Rechner werden die Interfaces Compute.java und Task.java und die Server-Klasse ComputeEngine.java übersetzt. Danach werden Stub- und Skeleton-Objekt für den Server mit rmic erzeugt.

Auf dem Client-Rechner muss das Stub-Objekt ComputeEngine_Stub.class vor der Übersetzung der Client-Klasse ComputePi.java zur Verfügung gestellt

werden. Danach werden die Interfaces Compute.java und Task.java und die Programme ComputePi.java und Pi.java übersetzt.

6.4 Programmausführung

Mindestens auf dem Client-Rechner muss ein Webserver gestartet sein, damit der Byte-Kode von Pi auf den Server geladen werden kann. Außerdem muss auf dem Server-Rechner die RMI-Registry gestartet sein.

Das Server-Programm wird mit folgender Anweisung gestartet:

```
java -Djava.security.policy=java.policy
      engine.ComputeEngine
```

Das Client-Programm wird wie folgt gestartet:

```
java -Djava.rmi.server.codebase=
       http://london/rmi_compute_client/
     -Djava.security.policy=java.policy
     client.ComputePi rom 65
```

Dabei gibt die Option -Djava.rmi.server.codebase die URL des Verzeichnisses auf dem Client Rechner an, in dem die Task Pi.class beim Zugriff auf den Webserver zu finden ist.

6.5 Zusammenfassung

In diesem Abschnitt wurden folgende Themen behandelt:

- Die prinzipielle Arbeitsweise von RMI
- Die Bedeutung der Begriffe Remote-Interface, Remote-Objekt, Remote-Referenz und RMI-Registry
- Die Verwendung von Stubs und Skeletons zur Kommunikation zwischen Server und Client
- Die drei unterschiedlichen Parameterarten bei der Kommunikation zwischen Client und Server
- Entwurf eines Remote-Interfaces und Bedeutung des Interfaces Remote

- Erstellen einer Implementierungsklasse durch Ableiten aus UnicastRemoteObject
- Erzeugen von Stub und Skeleton mit rmic
- Starten der RMI-Registry durch Aufruf von rmiregistry
- Erzeugen und Registrieren von Remote-Objekten und das Format der RMI-URLs
- Policy-Dateien zum dynamischen Laden von Bytecode
- Entwurf eines Clients unter Verwendung eines Web-Servers zum dynamischen Laden von Bytecode

Kapitel 7

Serververwaltung

Da Server Hauptspeicher oder zumindest Auslagerungsplatz auf dem Hintergrundspeicher belegen und Platz in den Verwaltungstabellen des Betriebssystemkerns beanspruchen, sollten nur häufig verwendete Server beim Systemstart gestartet werden. Weniger oft verwendete Server sollten bei Bedarf eines Clients gestartet werden. Diese Aufgabe erfüllt der Internet- Dämon inetd.

Der inetd-Server wartet stellvertretend für andere Server auf Verbindungsanfragen von Clients. Wird eine Verbindungsanfrage empfangen, so erzeugt inetd einen neuen Prozess und dupliziert die Netzverbindung auf die Standardein- und -ausgabe des Prozesses. Durch den Betrieb des inetd-Servers können die Anwendungs-Server also völlig netzwerk-naiv sein. Weiterhin kümmert sich inetd um einfache Dienste, wie echo, chargen und timeserver. Via inetd können auch RPC-basierte Server gestartet werden.

Um herauszufinden welche Server an welchen Ports auf Anfragen hin gestartet werden sollen, benötigt inetd zwei Konfigurationsdateien. Die Datei inetd.conf bestimmt den Server-Aufruf und das Kommunikationsprotokoll. Die Datei services enthält die Portnummer und das Kommunikationsprotokoll jedes Servers. Die Existenz von zwei getrennten Dateien hat historische Gründe. services war zuerst da und wird auch unabhängig von inetd verwendet, z.B. um die Portnummer zu einem Dienst mit getservbyname() herauszufinden.

Ein typischer Eintrag in inetd.conf lautet:

```
#1     2    3    4     5     6          7
ftp stream tcp nowait root /.../tcpd in.ftpd
```

1. Name des Dienst wie in services festgelegt,
2. Typ des Socket,
3. Protokoll,

4. nur für UDP-Sockets und RPC's von Bedeutung:
   (a) wait heißt ein Server pro Socket,
   (b) nowait heißt mehrere Server auf einem Socket.
5. Benutzer unter dem der Server läuft. Dies bestimmt die Zugriffsrechte des Client.
6. Pfad des Servers oder internal, falls inetd den Server beinhaltet, z.B. bei echo,
7. Kommandozeile des Servers.

tcpd ist ein TCP-Wrapper, der anhand von Einträgen in host.allow und host.deny überprüft, ob ein Rechner oder Netzwerk Zugriff auf den eigentlichen Server haben darf.

Mit inetd ist es wirklich einfach ein netzwerk-naives Programm als Netzwerk-Server zu installieren. Z.B. kann das UNIX-Kommando who verwendet werden, um als Server zu fungieren. Dazu braucht man folgende Dateieinträge:

```
# services:
wserve 2100/tcp

# inetd.conf
wserve stream tcp nowait nobody /.../who who
```

Anschliesend muss inetd seine Konfigurationsdateien neu einlesen. Dies geschieht, wenn das Hangup-Signal SIGHUP an den inetd-Prozess gesendet wird:

```
ps -ax | grep inetd
kill -SIGHUP <inetd-pid>
```

Beim Aufruf telnet <host> 2100 erhält man nun als Antwort das Resultat des who-Kommandos. netstat -a | less zeigt den neu installierten Server an.