

Netzwerke unter Java

Thomas Bachran
Wladimir Palant
Markus Stilkerieg

5. April 2000

Inhaltsverzeichnis

1	Grundlegende Prinzipien eines Netzwerkes	3
1.1	Aufbau eines Netzwerkes (Schichtenmodell)	3
1.1.1	Hardwareschicht	3
1.1.2	Netzwerkschicht	4
1.1.3	Transportschicht	4
1.1.4	Anwendungsschicht	4
1.2	Adressierung in IP-Netzen (IP-Adressen)	5
1.2.1	Verbindungsaufbau zwischen verschiedenen Netzwerken (IP-Routing)	5
1.2.2	Domain Name System (DNS)	6
1.3	Verbindungsaufbau (Server-Client-Schema)	6
1.4	Sicherheit im Netzwerk (Firewalls und Proxy-Server)	7
2	Die Möglichkeiten des Paketes „java.net“	9
2.1	Sichere Verbindungen (TCP)	9
2.2	Arbeiten mit einzelnen Datenpaketen (UDP)	9
2.3	Mehrfachadressierung (Multicasting)	10
2.4	Protokolle der Anwendungsschicht (HTTP)	11
2.5	Verbindungen auf hoher Abstraktionsebene (URLs)	12
2.6	Berechtigungen und deren Nachweis (Authentifizierung)	13
3	Die Umsetzung in Java	14
3.1	Der Kanal zu anderen Computern (Sockets)	14
3.1.1	Programmierung eines Clients	14
3.1.2	Programmierung eines Servers	18
3.2	Das Verschicken von Paketen (Datagrams)	20
3.3	Pakete mit mehreren Zielen (Multicasting)	23
3.4	Zugriff auf Internet-Ressourcen (URLs)	24
3.5	Authentifizierung	26
4	Anwendungsbeispiele	27
4.1	Server-Client-Kommunikation über Sockets	27
4.2	Zeitserver im Netzwerk	30

4.3	Ein kleiner Text-Webbrowser	32
4.4	Authentifizierung	33
5	Weiterführende Themen	35
5.1	Austausch von Objekten über das Netzwerk (Objektserialisierung)	35
5.2	Verteilte Systeme mit Java	36
5.2.1	Remote Methode Invocation (RMI)	37
5.2.2	Common Object Request Broker Architecture (CORBA)	37
5.3	Der dynamische Web-Server (Servlets)	38
	Literaturverzeichnis	39
	Index	40

Kapitel 1

Grundlegende Prinzipien eines Netzwerkes

Manchmal steht man vor einem Problem: man hat etwas geschrieben, das nun schön gedruckt werden sollte. Aber einen Laserdrucker hat man selber nicht, sondern nur ein Freund. Wie kriegt man die Datei nun dahin? Mit Disketten dauert es lange, man sieht den Freund ja erst am nächsten Tag. Aber es geht auch einfacher und schneller, nämlich per E-Mail. Voraussetzung ist natürlich, daß beide eine Verbindung zu einem Netzwerk haben.

Aber was heißt überhaupt „Verbindung“? Was ist ein Netzwerk eigentlich und wieso kann man damit Dateien verschicken? Nun, ein Netzwerk sind mehrere verbundene Computer, und zwar so, daß sie Daten miteinander austauschen können. Die Daten können heutzutage nicht nur E-Mails sein, das Netzwerk kann und wird für eine riesige Anzahl von Anwendungen benutzt: angefangen mit dem Uhrenabgleich mit einem Zeitserver bis hin zur Sprach- und Videoübertragung. Allerdings muß für eine problemlose Datenübertragung eine Vielzahl von Regelungen getroffen werden. Mit Hilfe des hier vorgestellten Schichtenmodells¹ werden diese in mehrere Schichten unterteilt.

1.1 Aufbau eines Netzwerkes (Schichtenmodell)

1.1.1 Hardwareschicht

Erstmals muß geregelt werden, wie die Daten in der Leitung repräsentiert werden. Man kann z.B. festlegen: wenn Strom fließt, bedeutet es eine 1, wenn kein Strom fließt, steht es für eine 0. Dann muß es auch Geräte geben, die diese Stromänderungen erzeugen, registrieren und auswerten. Normalerweise verwendet man hierfür Netzwerkkarten oder Modems, aber es gibt auch viele andere Möglichkeiten wie z.B. Funksender und -empfänger. Das alles zusammen bildet die **Hardwareschicht** eines Netzwerkes, also die Ebene der Geräte.

Beim Programmieren unter Java braucht man sich aber um die Art der verwendeten Geräte keine Sorgen zu machen, alle Unterschiede werden vollkommen ausgeglichen, so daß es keine Rolle spielt, ob man mit Hilfe eines Modems, einer Netzwerkkarte oder etwas anderem seine Verbindung zum

¹Für die Modellierung eines Netzwerkes wird oft das OSI/ISO-7-Schichten-Modell benutzt. Wir gehen hier aber nur auf ein einfacheres Modell mit größerer Unterteilung (4 Schichten) ein, da es alle für uns relevanten Information bereits enthält.

Netzwerk erhält. Dementsprechend braucht man sich unter Java mit dieser Schicht nicht zu beschäftigen.

1.1.2 Netzwerkschicht

Man muß auch dafür sorgen, daß die Daten bei dem Computer ankommen, für den sie bestimmt sind. Es muß also eine Möglichkeit geben, Zielcomputer für die Daten anzugeben, jeder Computer im Netzwerk muß also eindeutig identifizierbar sein. Das wird von der **Netzwerkschicht** übernommen, z.B. von dem **IP** (Internet Protocol).

Dieses Protokoll wurde mit dem Ziel entwickelt, die problemlose Benutzung von *mehreren* verbundenen Netzwerken zu ermöglichen. Nach diesem Prinzip wurde das Internet aufgebaut, denn es ist nichts anderes als ein Zusammenschluß von vielen Netzwerken unter Benutzung des IP (IP-Netz). Das IP ist derzeit das einzige, das von Java unterstützt wird, es wird später noch genauer darauf eingegangen.

1.1.3 Transportschicht

Wenn Daten übertragen werden, besteht immer die Möglichkeit, daß die Übertragung z.B. durch Magnetfelder gestört wird. Je schneller die Daten übertragen werden, desto schwieriger wird es, die Verbindung gegen äußere Einflüsse abzuschirmen, denn die Signale werden u.a. immer kürzer und entsprechend schwieriger zu registrieren. Ein Fehler kann grundsätzlich nie ausgeschlossen werden.

Nun sind aber für viele Anwendungen Übertragungsfehler nicht akzeptabel, z.B. ist eine ZIP-komprimierte Datei mit nur einem falschen Bit nicht mehr verwendbar. Es wird also eine **Fehlerkorrektur** benötigt. Denkbar wäre, daß die Daten in kleinere Datenpakete aufgeteilt werden und mit jedem der Pakete die Summe aller Bytes gesendet wird. Der Empfänger müßte dann diese Summe überprüfen. Sollte sie nicht mehr stimmen, ist ein Übertragungsfehler aufgetreten und das Datenpaket muß beim Sender erneut angefordert werden.

Für solcherlei Aufgaben ist die **Transportschicht** des Netzwerkes zuständig. In Java werden die Übertragungsprotokolle **TCP** (Transmission Control Protocol) und **UDP** (User Datagram Protocol) unterstützt, die im nächsten Kapitel vorgestellt werden.

1.1.4 Anwendungsschicht

Ist sichergestellt, daß die Daten an der richtigen Stelle und fehlerfrei (wenn nötig) ankommen, kann man mit Hilfe eines Programms Daten über das Netzwerk schicken (z.B. eine Videokonferenz starten). Nun tritt ein neues Problem auf. Auf dem Zielcomputer muß ein Programm gestartet sein, das diese Daten verstehen und beantworten kann (manchmal, aber nicht immer, ist es dasselbe Programm). Und unser Programm muß wiederum die Antworten verstehen können. Dafür muß genau geregelt werden, in welcher Form Informationen zu verschicken sind, also wieder ein Protokoll.

Dies ist die letzte Schicht des Schichtenmodells, nämlich die **Anwendungsschicht**. Hier haben wir es mit einer besonderen Vielfalt von Protokollen zu tun, denn fast jedes Programm definiert ein eigenes Protokoll, denn

die übertragenen Daten sind nur selten gleichartig. Ein Videokonferenzprogramm würde beispielsweise mit HTML-Seiten nichts anfangen können.

1.2 Adressierung in IP-Netzen (IP-Adressen)

Das Problem der Adressierung von Computern wird im IP so gelöst, daß jeder Computer eine eindeutige **IP-Adresse** erhält, die 32 Bit lang ist. Der besseren Lesbarkeit wegen schreibt man die einzelnen Bytes oft als Dezimalzahlen mit Punkten dazwischen, wie z.B. *131.220.4.1*

Dabei erlaubt das IP nicht nur die Kommunikation im Netzwerk, sondern auch die Kommunikation zwischen mehreren verbundenen Netzwerken. Ohne diese Möglichkeit wäre z.B. das Internet undenkbar. Um dies zu ermöglichen, setzt sich die IP-Adresse aus einer **Netzwerk-ID** und einer **Host-ID** zusammen. Die Netzwerk-ID identifiziert das Netzwerk, die Host-ID den Computer in diesem Netzwerk. Natürlich wird hier Eindeutigkeit gefordert. Für die Eindeutigkeit der Netzwerkbezeichnungen im Internet sorgt das Network Information Center. Je nach Größe des Netzwerkes werden im Internet die ersten 8, 16 oder 24 Bit der IP-Adresse für die Netzwerkbezeichnung vergeben. Dementsprechend werden die IP-Adressen in **Klassen** eingeteilt:

Klasse	Adreßbereich	Länge der Netzwerk-ID
A	1.0.0.1 - 126.255.255.254	8 Bit
B	128.1.0.0 - 191.254.255.254	16 Bit
C	192.0.1.0 - 223.255.254.255	24 Bit
D	224.0.0.3 - 239.255.255.254	– (s. Multicasting)

Zusätzlich gibt es noch einige Sonderfälle. IP-Adressen von 0.0.0.1 bis 0.255.255.254 sind für interne Verwendung in Programmen bestimmt, wenn die Netzwerk-ID nicht bekannt ist. Es können keine Daten an diese Adressen geschickt werden.

Adressen von 127.0.0.1 bis 127.255.255.254 verweisen stets auf den eigenen Computer (**Loopback**) und können für Tests verwendet werden.

Adressen, deren Host-ID Null ist (z.B. 131.220.0.0), kennzeichnen das ganze Netzwerk und können nicht in Programmen verwendet werden. Wenn dagegen in der Host-ID alle Bits gesetzt sind (z.B. 131.220.255.255), werden alle Computer des Netzwerkes angesprochen. Diese Adressen sind im Zusammenhang mit Multicasting interessant.

Auch wenn es zunächst nicht so scheint, wird die Anzahl der Netzwerke im Internet durch diese Regelungen stark eingeschränkt. So darf es zum Beispiel nur 126 Netzwerke mit mehr als 65000 Computern (Klasse A) geben. Diese Netzwerk-IDs sind bereits alle vergeben, auch bei anderen Netzwerkgrößen wird es wegen des schnellen Wachstums des Internets bald knapp. Dieses Problem soll mit der Version 6 des IP-Protokolls behoben werden (hier wird die Version 4 vorgestellt), indem 128-Bit-lange IP-Adressen verwendet werden. Allerdings ist die neue Version noch nicht sehr verbreitet und wird auch von Java momentan nicht unterstützt.

1.2.1 Verbindungsaufbau zwischen verschiedenen Netzwerken (IP-Routing)

Eigentlich sind auch im Internet alle Netzwerke strikt voneinander getrennt. Daß trotzdem Informationen zwischen den Netzwerken ausgetauscht werden

können liegt daran, daß sogenannte **Router** als Verbindungsglieder dienen.

Ein Router ist ein Computer, der in mehreren Netzwerken enthalten ist. Wenn ein Datenpaket von einem Computer zu einem anderen im anderen Netzwerk gesendet werden soll, wird dieser erst an den Router übermittelt. Der Router überprüft, ob er den Zielcomputer direkt erreichen kann. Sollte das der Fall sein, wird das Datenpaket an diesen Computer weitergeleitet.

Für die Netzwerke, die er nicht direkt erreichen kann, verwaltet der Router eine **Routing-Tabelle**. Diese enthält die Adresse desjenigen Routers, der die Daten in dieses Netzwerk zustellen kann (möglicherweise über noch einen Router), an den das Datenpaket dann auch weitergeleitet wird.

Eines der Probleme besteht darin, daß die Routing-Tabelle möglichst effizient gestaltet wird, so daß nur wenige Schritte für die Zustellung der Daten gebraucht werden. Dafür existieren mehrere mehr oder weniger komplizierte Algorithmen.

1.2.2 Domain Name System (DNS)

Da die IP-Adressen nur für Computer brauchbar sind, für Menschen jedoch kaum zu merken sind, wurden das **Domain Name System** (DNS) eingeführt. IP-Adressen werden dabei **DNS-Namen** wie *zeus.informatik.uni-bonn.de* zugeordnet². Ein DNS-Name setzt sich aus dem Computernamen (z.B. *zeus*) und der **Domäne** (engl: Domain) wie *informatik.uni-bonn.de* zusammen.

Mittlerweile gibt es im Internet so viele Computer, daß es unmöglich wäre, alle DNS-Namen zentral zu verwalten. Deswegen besitzt jede Domäne eine eigene Datenbank, in der die Liste der Computer in der Domäne mit den zugehörigen IP-Adressen gespeichert wird. Diese Datenbank wird durch den sogenannten **Name-Server** zur Verfügung gestellt.

Wenn nun ein Computer z.B. eine Verbindung mit *zeus.informatik.uni-bonn.de* aufbauen will, muß er zuerst die IP-Adresse bei seinem Name-Server anfordern. Gehören beide Computer zu derselben Domäne, kann der Name-Server die IP-Adresse aus seiner Datenbank übermitteln, ansonsten fragt er diese beim Name-Server der Domäne *informatik.uni-bonn.de* ab. Erst dann erhält man die notwendige IP-Adresse und kann eine Verbindung aufbauen.

1.3 Verbindungsaufbau (Server-Client-Schema)

Es kommt sehr oft vor, daß die Verbindung zwischen zwei Computern nach dem **Server-Client-Schema** abläuft. Der eine Computer stellt einen bestimmten Dienst zur Verfügung (**Server**) z.B. HTML-Seiten mit Java-Dokumentation. Dieser Computer wartet passiv, bis eine Verbindung zu ihm aufgebaut wird. Der andere Computer nutzt diesen Dienst (**Client**), indem er eine Verbindung zum Server aufbaut und z.B. die Java-Dokumentation zum Anzeigen im Webbrowser herunterlädt.

Es kann durchaus vorkommen, daß der Server seinerseits als Client eine Verbindung zu einem anderen Server aufbaut, dann wird eine Unterschei-

²Es ist zu beachten, daß eine IP-Adresse mehr als einen DNS-Namen haben kann. Entsprechend ist auch die Umwandlung des Namens in eine IP-Adresse nicht eindeutig. Letzteres kann z.B. den Sinn haben, mehrere Computern nach außen als einen darzustellen und die Zugriffe gleichmäßig zu verteilen (falls ein einziger Computer nicht leistungsfähig genug wäre).

dung zwischen Servern und Clients kompliziert. Auch gleichberechtigte Verbindungen sind denkbar. Das Server-Client-Schema ist nur eines der möglichen.

Nun wäre es eine Verschwendung, wenn ein Computer nur einen Dienst anbieten könnte, besonders wenn dieser Dienst nicht sehr oft genutzt wird. Zum Glück bieten einige Übertragungsprotokolle wie auch TCP und UDP eine bequeme Möglichkeit, mehrere Dienste auf einem Computer ablaufen zu lassen. Den Diensten werden sogenannte **Ports** zugeordnet, wobei Ports von 0 bis 65535 möglich sind. Eine Verbindung wird dann stets mit einem bestimmten Port des Computers, also mit dem zugehörigen Dienst, aufgebaut. Die Protokolle TCP bzw. UDP sorgen dafür, daß die Daten richtig ankommen.

Die Port-Nummer läßt sich eigentlich für jeden Dienst beliebig festlegen³. Allerdings sind für einige Dienste die Port-Nummern zu inoffiziellen Standards geworden, z.B. verwendet man für einen HTTP-Server meistens den Port 80 und für einen FTP-Server den Port 21.

1.4 Sicherheit im Netzwerk (Firewalls und Proxy-Server)

Nicht immer ist eine E-Mail etwas erfreuliches. Es gibt Anbieter im Internet, die dieses Medium für Massenwerbung mißbrauchen (Spam). Hinterläßt man an einigen Stellen im Internet seine E-Mail-Adresse, könnte man schnell mit Spam überflutet werden. Auch werden Viren gerne per Email verschickt. Öffnet man die harmlos erscheinende Datei im Anhang, wird der Computer infiziert und der Virus verschickt sich weiter an alle Adreßbucheinträge.

Möglichkeiten des Mißbrauchs im Netzwerk sind zahlreich und nicht immer kann ein Anwender seinen PC selber schützen (meistens besitzt man einfach nicht die nötigen Kenntnisse). Ideal wäre also die Möglichkeit, die eingehenden Daten zu filtern und beispielsweise Spam und Viren nicht durchzulassen. Aber wo soll man den Filter aufstellen? Auf jedem Computer? Das wäre für einen Netzwerkadministrator ein riesiger Aufwand, außerdem wäre es nicht sicher genug: Ein Anwender könnte das Filterprogramm auf seinem Computer deaktivieren, z.B. um eine schnellere Verbindung zu haben. Andererseits hat man aber das Problem, daß Verbindungen im Netzwerk auf verschiedenste Weisen laufen können, es gibt nie einen eindeutigen Verbindungsweg von einem Computer zum anderen. Es wäre wieder zu aufwendig, hunderte von möglichen Verbindungswegen zu filtern. Wie löst man also das Problem?

Man wendet hier einen Trick an: man läßt alle Verbindungen vom Netzwerk ins Internet über einen Punkt laufen. Dies wird erreicht, indem zunächst das Netzwerk komplett vom Internet getrennt wird. Es bleibt nur die Verbindung über einen Computer, auf dem ein sogenanntes **Firewall-Programm** läuft, das weder Daten aus dem Netzwerk noch Daten in das Netzwerk durchläßt.

Da könnte man doch die Verbindung zum Internet ganz aufheben, wenn sowieso keine Daten durchgelassen werden! Oder? Nicht unbedingt. Möchte

³Ports von 0 bis 1023 lassen sich unter UNIX (und UNIX-Derivaten wie Linux oder Solaris) nur mit root-Rechten betreiben. Besitzt man diese nicht, sollte man auf höhere Port-Nummern ausweichen. In Applets können diese Ports ebenfalls nicht verwendet werden.

jetzt ein Computer Daten von einem Server im Internet haben, so muß er eine Verbindung zum **Proxy-Server** (Proxy heißt wörtlich Bevollmächtigter), der ein Teil der Firewall ist, aufbauen und eine Anforderung für diese Daten schicken. Der Proxy-Server baut nun seinerseits eine Verbindung zum Internet auf, erhält die Daten und leitet sie an den Computer weiter.

Wo liegen nun die Vorteile des Systems? Jeder Computer kommt immer noch an die benötigten Daten, wenn auch etwas umständlicher. Dafür kann aber am Proxy beliebig gefiltert werden, beispielsweise kann die Kommunikation mit bestimmten Ports unterbunden werden. Will man z.B. keine Telnet-Verbindungen zulassen, weil dabei Paßwörter ausspioniert werden könnten, läßt man keine Kommunikation über den Port 23 durch. Man kann auch die einzelnen Datenpakete überprüfen. Wird z.B. ein Virus gefunden, werden die Daten nicht durchgelassen. Und letztlich kann man dem Benutzer die Daten auch verweigern und so z.B. den Zugriff auf bestimmte Webseiten verbieten (ein ernstes Problem in einigen Firmen).

Man sollte sich aber im Klaren sein, daß auch eine Firewall keinen absoluten Schutz bieten kann. Werden Telnet-Verbindungen über Port 23 gesperrt, hat der Anwender immer noch die Möglichkeit, einen anderen Port zu verwenden. Dabei geht er das Risiko ein, daß irgendwo auf dem Weg zum Zielcomputer sein Paßwort abgefangen wird. Auch kann sicherlich nicht jeder Virus von der Firewall erkannt werden. Eine Firewall ist also kein Grund für den Anwender, sich nicht mehr um die Sicherheit seiner Daten zu kümmern, es ist lediglich eine Ergänzung zu den eigenen Sicherheitsmaßnahmen.

Kapitel 2

Die Möglichkeiten des Paketes „java.net“

Das Paket „java.net“ bietet nun, wie schon in Kapitel 1 erwähnt, die Möglichkeit über ein IP-Netzwerk mit anderen Computern zu kommunizieren. Dabei benutzt es unterschiedliche Techniken. Diese Techniken und die Protokolle auf denen sie aufbauen (TCP und UDP) werden im folgenden behandelt.

2.1 Sichere Verbindungen (TCP)

Eine Technik ist, einen **Kommunikationskanal** zu einem anderen Computer aufzubauen (point-to-point). Diese Möglichkeit nutzt **TCP** (Transmission Control Protocol). Hierbei wird eine automatische Fehlerkorrektur durchgeführt. Das heißt, die Daten, die der sendende Computer verschickt, kommen dadurch genauso an, wie sie verschickt werden. Dies ermöglicht eine sichere Kommunikation. Für jeden Computer, mit dem man kommunizieren möchte, muß man einen eigenen Kanal öffnen.

Eine solche Übertragung wird bei fast allen Implementationen der Anwendungsschicht im Internet genutzt (siehe **Schichtenmodell**).

Ein Beispiel, wo eine fehlerfreie Übertragung nötig ist, ist das Herunterladen eines Java-Programms. Sollte nur ein Bit nicht korrekt ankommen, so ist das Programm im Normalfall nicht mehr fehlerfrei ausführbar. In einem solchen Fall würde man also auf eine sichere Verbindung setzen.

Als ein weiteres Beispiel kann man Bilder nehmen. Sollen diese übertragen werden, so hängt das Verfahren, welches man benutzt, von der gewünschten Übertragungsart ab. Zum einen kann man die Daten in komprimierter Form versenden. In diesem Fall ist eine sichere Verbindung notwendig, damit man die ankommenden Bilder wieder dekomprimieren kann. Sollte man die Daten jedoch in unkomprimierter Form verschicken, so ist es eine Frage der geforderten Qualität. Stört man sich nicht an kleineren Darstellungsfehlern, so kann man eventuell auf eine Fehlerkorrektur verzichten. Dies kann einen Geschwindigkeitsvorteil bei der Übertragung bringen, da die zeitaufwendige Fehlerkorrektur wegfällt.

2.2 Arbeiten mit einzelnen Datenpaketen (UDP)

Möchte man auf die Fehlerkorrektur verzichten, so muß man die Verwaltung von Datenpaketen, sogenannten **Datagrams**, selber übernehmen. Dies wird durch **UDP** (User Datagram Protokoll) ermöglicht. Bei UDP fällt der Aufbau einer Verbindung weg. Man versendet nur einzelne Datenpakete. Es findet außerdem keine Überprüfung statt, ob die Pakete erfolgreich angekommen sind. Man könnte nun eine solche Überprüfung von Hand hinzufügen. Aber dann könnte man überlegen, TCP einsetzen. Man kann sich natürlich die Frage stellen, in welchen Fällen der Einsatz von UDP sinnvoll ist.

Ein Beispiel hierfür ist die Synchronisation von zwei Uhren. Nehmen wir an, daß in einem Netzwerk der Server die Uhrzeit für die Clients festlegen soll. Der Server muß also die Uhrzeit an die Clients schicken. Es macht überhaupt keinen Sinn nach der Übertragung der Uhrzeit nachzufragen, ob das Datenpaket richtig angekommen ist. Sollte es nicht angekommen sein, so würde es keinen Sinn machen das Paket neu zu schicken, da schon wieder Zeit vergangen ist. Der Server würde also sowieso schon das nächste Paket verschicken. Hier ist eine Fehlerkorrektur also unnötig.

Ein Nachteil, TCP zu nutzen, wäre die Einschränkung, nur einen Computer ansprechen zu können. TCP baut ja einen Kommunikationskanal zwischen zwei Computern auf. Würde man also mehrere Computer ansprechen wollen, so müßte man zu jedem Computer eine Verbindung aufbauen und über jede Verbindung die Zeitinformationen verschicken. Dies würde in einem großen Netzwerk eine unnötige Netzlast erzeugen. Daher sollte man in einem solchen Fall immer UDP einsetzen, da dieses das **Multicasting** unterstützt. Hierbei wird vom Server nämlich nur *ein* Paket mit mehreren Zielen verschickt, während TCP das Paket einzeln über jeden geöffneten Kanal verschicken würde.

2.3 Mehrfachadressierung (Multicasting)

UDP bietet nicht nur die Möglichkeit, Pakete an einen Zielcomputer zu schicken, sondern ermöglicht durch Angabe von mehreren Adressen das Versenden von Paketen an mehrere Computer. Dieses Verfahren wird als **Multicasting** bezeichnet. Multicasting kann in mehreren Bereichen sinnvoll eingesetzt werden. Ein Beispiel wäre die Zeitsynchronisation aus dem vorigen Abschnitt.

Beispiel: „Klonen von Computern“

Möchte man ein Netzwerk einrichten, wo mehrere Arbeitsstationen gleich konfiguriert werden, so kann man jeden Computer einzeln einrichten und müßte an jeden Computer das Betriebssystem und die Anwendungsprogramme einzeln übertragen. Eine andere Möglichkeit wäre aber einen Prototyp eines Computer einzurichten und ein kleines Programm zu schreiben, welches die Daten der Festplatte per UDP-Multicasting an die anderen Computer schickt. Auf diesen müßte nun ein Empfangsprogramm seine Arbeit verrichten, welches die versendeten Daten empfängt und anschließend auf die Festplatte der „Klon“-Rechner schreibt. Somit macht man sich nur die Arbeit, einen Computer einzurichten, und „klont“ die anderen Installationen innerhalb kürzester Zeit.

Was passiert nun aber bei einem Übertragungsfehler?

In diesem Fall besteht das Problem, daß man eine Fehlerkorrektur von Hand hinzufügen muß. Der Vorteil ist allerdings, daß die Netzwerkauslastung geringer ist. Es wird nicht jedes große Paket mehrfach verschickt, sondern nur einmal. Sollte ein Fehler auftreten, so kann man das fehlerhafte Paket erneut verschicken und dieses als doppelt versandtes Paket besonders markieren. Dadurch ist dann den Empfängern klar, daß dieses doppelte Paket nur zu berücksichtigen ist, falls das erste Paket nicht angekommen ist. Das ist besonders in der Situation wichtig, in der nur ein Teil der Computer das Datenpaket nicht korrekt erhalten hat. Alle anderen brauchen sich um das erneut versandte Paket nicht zu kümmern.

2.4 Protokolle der Anwendungsschicht (HTTP)

Wie tauschen nun eigentlich Anwendungen die Daten über das Netzwerk aus? Kann es da überhaupt ein gemeinsames Protokoll geben? Die einen Programme fordern Dateien vom Server an, die anderen übertragen Spielzüge eines Netzwerkspiels in einer gleichberechtigten Verbindung und es gibt noch unzählige andere Möglichkeiten. Bei dem einen soll es möglichst schnell gehen, bei dem anderen kommt es auf die Fehlerfreiheit an, jedes Programm hat seine eigenen Anforderungen an die Übertragung. Deswegen sind die Protokolle der Anwendungsschicht fast genauso zahlreich wie die Programme selber.

Wir wollen uns **HTTP** (Hypertext Transfer Protocol) als Beispiel für ein Protokoll der Anwendungsschicht anschauen. Die Funktion eines **Webserver**s ist eigentlich sehr einfach: er soll lediglich auf Anforderung eine Datei übertragen. Es kommen natürlich auch Zusatzfunktionen wie das Protokollieren von Zugriffen, Zugriffsrechte oder just-in-time-Generierung von HTML-Seiten dazu, aber davon merkt ein Client nichts. Der Client baut die Verbindung zum Server auf und schickt eine Zeile wie z.B. *GET /index.html*. Der Server schickt seinerseits die Datei */index.html* (die sich meistens in irgendeinem Verzeichnis des Servers befindet) zurück und beendet die Verbindung. Es wird immer nur eine Datei pro Verbindung übertragen, für eine zweite muß eine neue Verbindung aufgebaut werden.

Nun enthält das Protokoll doch noch zusätzliche Möglichkeiten. Wird an die Anforderung noch die Version des HTTP angehängt (z.B. *GET /index.html HTTP/1.0*), dann sendet der Server einige Zeilen Zusatzinformation vor der eigentlichen Datei, die durch eine Leerzeile abgeschlossen werden. Hier als Beispiel die Antwort eines Apache-Webserver unter Windows:

```
HTTP/1.1 200 OK
Date: Tue, 29 Feb 2000 20:24:44 GMT
Server: Apache/1.3.9 (Win32)
Last-Modified: Thu, 30 Dec 1999 22:22:48 GMT
ETag: "0-689-386bdb38"
Accept-Ranges: bytes
Content-Length: 1673
Connection: close
Content-Type: text/html
```

Dahinter folgen die eigentlichen Daten. Wie man sieht, hat der Server zuerst die Version von HTTP geschickt, die er unterstützt, dann folgt der Status-Code (2xx bedeutet Erfolg, 4xx oder 5xx sind Fehler). Die Statuszeile wird von Informationen wie die Bezeichnung des Webserver-Programms (Apache/1.3.9) und des Betriebssystems (Win32), Datum der letzten Änderung der Datei (30. Dezember 1999), ihrer Größe (1673 Bytes) und Dateityp (text/html) gefolgt. Die Bedeutung der einzelnen Felder ist im HTTP genau festgelegt, das soll hier aber nicht weiter ausgeführt werden.

Entsprechend hat auch der Webbrowser die Möglichkeit, nach der GET-Anforderung zusätzliche Informationen zu senden. Folgendes sendet z.B. der Internet Explorer (durch ... ist angedeutet, daß die Zeile eigentlich weitergeht, aber nicht mehr auf das Blatt paßt):

```
GET /index.html HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Language: de
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows 95)
Host: www.informatik.uni-bonn.de
Connection: Keep-Alive
```

Wie man sieht, teilt der Browser ebenfalls seine Bezeichnung und die des Betriebssystems mit, zusätzlich noch die Dateiformate, die verarbeitet werden können, sowie seine Sprachversion. Wie man hier auch sieht, kann die Richtigkeit der Angaben nicht garantiert werden: Mozilla ist die Bezeichnung für den Netscape Communicator und nicht für den Internet Explorer, die falsche Browserbezeichnung ist auf irgendeine Weise aus der Konkurrenz der beiden Browser entstanden.

Interessant ist die Anweisung „*Connection: Keep-Alive*“. Diese hat zur Folge, daß der Server (falls er die Version 1.1 des HTTP unterstützt) nach der Übertragung der Datei die Verbindung nicht schließt, sondern weitere Anforderungen zuläßt. Das ist dadurch möglich geworden, daß die Dateigröße ebenfalls übertragen wird, und der Browser so erkennt, wo die Datei endet. Der klare Vorteil: es müssen weniger Verbindungen aufgebaut werden, dabei wird Zeit sowohl auf der Seite des Clients als auch des Servers eingespart.

2.5 Verbindungen auf hoher Abstraktionsebene (URLs)

Ein **URL** (Uniform Resource Locator) ist ein universelles Hilfsmittel zur Adressierung von Internet-Ressourcen. Der Aufbau von URLs ist recht komplex und ermöglicht es, sehr unterschiedliche Typen von Adressen zu verwalten. Ein URL kann beispielsweise die Adresse einer Webseite beinhalten, oder die anderer Dateien, wie z.B. von Bildern und Applets. Wir wollen hier nur auf die URLs eingehen, die man zur Darstellung von Webseiten-Adressen verwendet.

Ein URL besteht aus einem **Protokollbezeichner** und der eigentlichen Adresse, die durch die Zeichenkette „://“ voneinander getrennt werden. Das Standardprotokoll zur Übertragung von HTML-Seiten ist *http*; weitere mögliche Protokollbezeichner in einem URL sind u.a. *ftp*, *gopher*, *file*, *news* und *mailto*, die hier nicht weiter besprochen werden sollen.

Die eigentliche Adresse besteht bei Webseiten typischerweise aus dem DNS-Namen, einer mit Doppelpunkt getrennten, optionalen Port-Nummer (bei HTTP 80), und einem durch einen Schrägstrich getrennten Dateinamen, z.B. *http://www.informatik.uni-bonn.de:80/index.html*.

Mit einer durch # getrennten Marke kann man auch auf Teilbereiche innerhalb dieser Datei zugreifen, z.B. *index.html#anfang*. Die meisten Browser und Server sind in der Lage, fehlende Bestandteile eines URLs weitgehend zu ergänzen. Fehlt beispielsweise die Port-Nummer, wird bei HTTP :80 eingefügt. Bezeichnet der URL lediglich ein Verzeichnis, hängen Server in der Regel einen Standard-Dateinamen wie z.B. *index.html* an; man hätte also auch nur *www.informatik.uni-bonn.de* angeben können.

2.6 Berechtigungen und deren Nachweis (Authentifizierung)

Nicht alle Server erlauben einen uneingeschränkten Zugriff auf Daten. Dies kann sinnvoll sein, wenn bestimmte Dateien oder Verzeichnisse nicht von beliebigen Benutzern gelesen oder sogar verändert werden sollen. Für diesen Zugriff benötigt man eine gültige Kombination aus **Benutzernamen** und **Paßwort**, die auf Anfrage des Servers an ihn gesendet werden. Aber auch der unbeschränkte (Lese-)Zugriff auf Daten funktioniert nicht immer ganz ohne Namen, z.B. verlangen die meisten FTP-Server als Benutzernamen „anonymous“, als Paßwort wird oft die eigene E-Mail-Adresse gesendet.

Kapitel 3

Die Umsetzung in Java

3.1 Der Kanal zu anderen Computern (Sockets)

Wir wollen die **Server-Client-Kommunikation** jetzt genauer unter die Lupe nehmen. Zwischen den beiden Computern ist eine TCP-Verbindung aufgebaut, die eine Kommunikation in beide Richtungen ermöglicht. Dabei ist jeder dieser Computer in Besitz eines sogenannten **Socket**-Objektes¹, das diese Verbindung repräsentiert. Über einen Socket können Daten sowohl empfangen als auch gesendet werden.

Nun soll zwischen der Sichtweise des Clients und des Servers unterschieden werden.

3.1.1 Programmierung eines Clients

Verbindungsaufbau

Ein Client kennt die IP-Adresse des Servers und erstellt ein *Socket*-Objekt, um die Verbindung zum Server aufzubauen. Dabei muß zusätzlich die Port-Nummer bekannt sein, an der der Server zur Verfügung steht. Das könnte z.B. so ablaufen:

```
Socket sock=new Socket("www.uni-bonn.de",80);
```

So würde man versuchen, eine Verbindung zum Webserver der Uni Bonn eine HTTP-Verbindung aufzubauen (Port 80 wird normalerweise für den HTTP-Dienst benutzt).

Nun könnte es an mehreren Stellen zu **Problemen** kommen:

- Erstmals wandelt der Konstruktor den DNS-Namen *www.uni-bonn.de* in eine IP-Adresse um. Dies könnte fehlschlagen, z.B. weil die Modem-Verbindung gerade nicht aufgebaut und kein Name Server verfügbar ist. In diesem Fall wird *UnknownHostException* ausgelöst.

Es dürfte klar sein, daß eine Umwandlung des DNS-Namens Zeit kostet. Soll mehrmals dieselbe Adresse verwendet werden, so ist es von Vorteil, diese Umwandlung nur einmal vorzunehmen. Dafür existiert

¹Mit Hilfe des Pakets *javax.net.ssl* können auch sichere SSL-Verbindungen aufgebaut werden. Das entsprechende Objekt heißt dann *SSLSocket*. Allerdings ist dieses Paket nicht standardmäßig in JDK 1.2.2 enthalten und muß von <http://java.sun.com> heruntergeladen werden.

die Klasse *InetAddress*, die eine IP-Adresse aufnimmt und ersatzweise im Konstruktor des Sockets verwendet werden kann. Das Objekt würde man dann mit der statischen Methode *getByName* erstellen:

```
InetAddress uni=InetAddress.getByName("www.uni-bonn.de");
Socket sock=new Socket(uni,80);
```

- Anschließend wird die Verbindung zum Server aufgebaut. Ist an dem vorgegebenen Port kein Server installiert, wird *ConnectException* ausgelöst.
- Ein Applet darf nur eingeschränkt Verbindungen aufbauen. Erlaubt sind nur Verbindungen zu dem Server, von dem das Applet geladen wurde, ansonsten wird *SecurityException* ausgelöst.
- Es sind auch Ausnahmen anderer Art (wie *IllegalArgumentException*), möglich, denn der Konstruktor kann eine beliebige Unterklasse von *IOException* auslösen. Deswegen sollte man am Schluß immer auch *IOException* abfangen.

Senden und Empfangen von Daten

Ein Socket besteht aus zwei **Streams**: aus einem *OutputStream* zum Senden und einem *InputStream* für den Empfang. Diese lassen sich mit den Methoden *getInputStream* und *getOutputStream* erhalten.

So würde z.B. der Client vorgehen, um alles, was er vom Server erhält, an diesen zurückzuschicken:

```
try
{
    InputStream in=sock.getInputStream();
    OutputStream out=sock.getOutputStream();

    while (true)
    {
        byte b=(byte)in.read();
        out.write(b);
    }
}
catch (SocketException e)
{
    System.err.println("Verbindung wurde getrennt");
}
catch (IOException e)
{
    System.err.println(e.toString());
}
```

Das Programm empfängt byteweise (nicht sehr effizient, aber fürs Erste reicht das) Daten und schickt sie sofort zurück.

Man sollte beachten, daß der Server (wie auch der Client) die Verbindung jederzeit trennen kann. Bei einem Versuch, über einen geschlossenen Socket

Daten zu lesen oder zu senden, wird die *SocketException* ausgelöst. Nur in diesem Fall bricht das obere Programm ab.

Zusätzlich muß auf *IOException* geprüft werden, denn diese könnte wie bei allen Streams ausgelöst werden.

Trennen von Sende- und Empfangsroutinen

Meistens ist das gleichzeitige Empfangen und Senden von Daten nicht ganz so einfach. Beispielsweise sollen die über die Tastatur eingegebenen Daten an den Server gesendet und die Antworten auf dem Bildschirm dargestellt werden.

Hier haben wir das Problem, daß sowohl das Lesen der Daten von der Tastatur als auch der Empfang die Ausführung blockieren und nicht gleichzeitig ausgeführt werden können. Die Lösung ist, das Programm in einen sendenden und einen empfangenden **Thread** aufzuteilen. Die Empfangsroutine würde dann in einen unabhängigen Thread ausgelagert:

```
class ReceiveThread extends Thread
{
    InputStream in;

    public ReceiveThread(InputStream in)
    {
        super();
        this.in=in;
    }

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

        try
        {
            while (true)
            {
                int len=in.read(b);
                if (len>0)
                    System.out.print(new String(b,0,len));
            }
        }
        catch (SocketException e)
        {
            System.err.println("Verbindung wurde getrennt");
        }
        catch (IOException e)
        {
            System.err.println(e.toString());
        }
    }
}
```

Hier gibt es eine zusätzliche Änderung: es wird nicht mehr byteweise

empfangen, sondern in 100-Byte-großen Paketen. Damit ist der Empfang etwas effizienter geworden.

Nun kann man auch die Senderoutine aus dem oberen Abschnitt ändern:

```
try
{
    InputStream in=sock.getInputStream();
    OutputStream out=sock.getOutputStream();
    BufferedReader keyboard=new BufferedReader(
        new InputStreamReader(System.in));

    (new ReceiveThread(in)).start();

    while (true)
    {
        String str=keyboard.readLine();
        out.write((str+'\n').getBytes());
    }
}
catch (SocketException e)
{
    System.err.println("Verbindung wurde getrennt");
}
catch (IOException e)
{
    System.err.println(e.toString());
}
```

Dieses Programm startet also erstmals den empfangenden Thread und liest dann zeilenweise die Benutzereingaben über einen *BufferedReader*. Das Senden geschieht jetzt auch effizienter, nämlich zeilenweise. Dabei muß am Ende jeder Zeile das Carriage-Return-Zeichen angehängt werden, das beim Einlesen gelöscht wurde. Da wir über eine Stream-Klasse senden, muß die Zeichenfolge zuerst mit der Methode *getBytes* in ein Byte-Array umgewandelt werden.

Verwendung der Klasse *BufferedReader* für den Empfang

Nun könnte man sich denken: warum empfangen wir die Daten nicht genauso zeilenweise? Folgende Empfangsroutine wäre eigentlich auch möglich:

```
class ReceiveThread extends Thread {
    BufferedReader in;

    public ReceiveThread(InputStream in)
    {
        super();
        this.in=new BufferedReader(new InputStreamReader(in));
    }

    public void run()
    {
```

```

try
{
    while (true)
    {
        String str=in.readLine();
        if (str!=null)
            System.out.println(str);
    }
}
catch (SocketException e)
{
    System.err.println("Verbindung wurde getrennt");
}
catch (IOException e)
{
    System.err.println(e.toString());
}
}
}

```

Obwohl diese Methode unter Umständen etwas einfacher ist, hat sie einen Nachteil: Wegen der Umwandlung der Bytes in Unicode sind Reader-Klassen wesentlich langsamer. Man sollte sie nur verwenden, wenn es eindeutige Vorteile bringt. Ansonsten kann man beispielsweise auch *BufferedReader* einsetzen.

Setzen einer zeitlichen Begrenzung für den Empfang

Es könnte in einem Programm nützlich sein, daß der Empfang nach einer bestimmten Zeit unterbrochen wird, wenn noch immer nichts empfangen wurde (**Timeout**).

Bei vielen Anwendungen wie z.B. einem Webbrowser bedeutet das längere Fehlen einer Antwort vom Server, daß etwas nicht in Ordnung ist. In diesem Fall wird die Verbindung unterbrochen.

Eine andere denkbare Anwendung ist, daß das Programm von Zeit zu Zeit prüfen soll, ob ein Abbruch-Signal gekommen ist. Dazu wird das Warten auf Empfang regelmäßig durch den Timeout unterbrochen und, falls das Signal tatsächlich erhalten wurde, nicht wieder aufgenommen.

Die Timeout-Zeit in Millisekunden wird mit der Methode *setSoTimeout* des Sockets eingestellt. Beispielsweise legt *sock.setSoTimeout(300)* fest, daß, wenn nach 300 Millisekunden nicht empfangen wurde, *InterruptedIOException* ausgelöst wird.

3.1.2 Programmierung eines Servers

Bereitstellen eines Dienstes

Allererstes muß ein Server-Programm deutlich machen, daß an einem Port ein Dienst zur Verfügung gestellt wird. Dafür wird ein **ServerSocket**-Objekt² erstellt, z.B.:

²Für SSL-Verbindungen wird entsprechend das Objekt *SSLServerSocket* des Pakets *javax.net.ssl* verwendet

```
ServerSocket server = new ServerSocket(5555);
```

Hier wird an Port 5555 ein Server registriert. Dabei kann passieren, daß dieser Port bereits von einem anderen Server belegt ist. Dann wird *BindException* ausgelöst. Außerdem könnte es sein, daß man nicht die Berechtigung hat, einen Server zu betreiben (z.B. in Applets mit Port-Nummer zwischen 0 und 1023). Das meldet Java mit *SecurityException*. Auch andere Unterklassen von *IOException* sind wieder möglich.

Man beachte, daß der *ServerSocket*-Objekt immer mit der Methode *close* geschlossen werden muß, wenn er nicht mehr gebraucht wird. Wenn das nicht getan wird, bleibt der Server angemeldet und blockiert den Port. So kann an diesem Port kein anderer Server angemeldet werden, außerdem können Clients nicht erfahren, daß der Port nicht belegt ist.

Warten auf Verbindung

Nachdem der Server registriert wurde, muß er auf eine Verbindung warten. Dazu wird die Methode *accept* des *ServerSocket*-Objektes aufgerufen. Wurde die Verbindung mit einem Client erfolgreich, liefert diese Methode das *Socket*-Objekt zurück, das genauso wie bei einem Client verwendet werden kann:

```
Socket sock = server.accept();

InputStream in = sock.getInputStream();
OutputStream out = sock.getOutputStream();

(new ReceiveThread(in, "Client: ")).start();
```

Hier wird genauso wie in unserem Beispiel für Clients die Klasse *ReceiveThread* dazu verwenden, die empfangenen Daten anzuzeigen. Das Senden kann ebenfalls absolut analog zum Client erledigt werden.

Nun kann es sein, daß auch ein Server nicht unbegrenzt viel Zeit hat und irgendwann das Warten auf eine Verbindung unterbrechen wird. In diesem Fall kann man genauso wie beim *Socket*-Objekt einen Timeout setzen:

```
ServerSocket server = new ServerSocket(5555);
server.setSoTimeout(10000);
Socket sock = server.accept();
```

In diesem Beispiel wird festgelegt, daß der Server höchstens 10 Sekunden (10000 Millisekunden) auf den Verbindungsaufbau warten soll. Anschließend wird *InterruptedIOException* ausgelöst.

Verbindungen zu mehreren Clients

Meistens soll ein Server mehrere Verbindungen gleichzeitig aufbauen können, damit mehrere Benutzer den Dienst nutzen können. Für einen Webserver z.B. wäre absolut inakzeptabel, daß solange ein Benutzer sich die Webseite herunterlädt, alle anderen warten müssen. Man muß also eine Möglichkeit finden, mehrere Verbindungen gleichzeitig zu öffnen und zu bearbeiten.

Die Lösung ist, für jede Verbindung (also jedes *Socket*-Objekt) einen eigenen **Thread** zu erstellen, der mit dem Client kommuniziert. Das Hauptprogramm ruft währenddessen wieder die Methode *accept* auf:

```

server = new ServerSocket(5555);
while (true)
{
    Socket sock = server.accept();
    (new SocketThread(sock)).start();
}

```

SocketThread müßte dann ein Thread sein, der die Kommunikation mit dem Client übernimmt.

3.2 Das Verschicken von Paketen (Datagrams)

Wie in Kapitel 2 schon vorgestellt gibt es unter Java nicht nur die Möglichkeit, Daten über einen sicheren Kanal zu übertragen (TCP), sondern auch Daten als einfache **Pakete** ohne zusätzliche Verbindung und Fehlerüberprüfung zu verschicken (UDP). Für das Versenden und Empfangen von Paketen benutzt man die Klasse *DatagramSocket*. Die Klasse *DatagramSocket* versendet Objekte vom Typ *DatagramPacket*. Im folgenden wird das Prinzip von einem einfachen Sende- und einem einfachen Empfangsprogramm erläutert.

Das Arbeiten mit Paketen

Ein **Paket** enthält 2 unterschiedliche Informationen. Zum einen enthält es die IP-Adresse und die Port-Nummer, an die es geschickt werden soll oder von der es empfangen wurde. Diese Informationen sind z.B. interessant, wenn man auf ein Paket antworten möchte.

Zum anderen enthält das Paket natürlich die Daten, die verschickt werden. Dafür enthält es ein Byte-Array. Die Länge des gefüllten Feldes ist ebenfalls als Information enthalten. Es enthält auch die Informationen, ab welcher Stelle (Offset) in diesem Feld Informationen verschickt werden sollen.

Der Offset ist sicherlich seltener zu setzen und wird daher, falls er nicht im Konstruktor angegeben wird, als Standard auf 0 gesetzt.

Die restlichen Informationen hängen nun davon ab, ob man ein Paket empfangen oder senden möchte. Das Sendeprogramm wird alle anderen Informationen im Konstruktor angeben:

```

byte[] buffer = (new Date()).toString().getBytes();
DatagramPacket timeSend = new DatagramPacket(buffer,
buffer.length, InetAddress.getByAddress("192.168.0.1"), 5556);

```

Es wird zunächst das Feld erzeugt. In diesem Beispiel enthält es eine Datumsangabe. Bei der Erzeugung des Paketes werden nun zunächst das Feld angegeben gefolgt von der Länge des Feldes. Anschließend wird die IP-Adresse des Zielcomputers (hier: *192.168.0.1*) angegeben gefolgt von dem Port auf dem der Zielcomputer empfängt. Dies ist in unserem Beispiel das Port *5556*. Im Unterschied zum *Socket*-Objekt muß hier auf jeden Fall die Klasse *InetAddress* verwendet werden.

Das Empfangsprogramm wird hingegen nicht alle Informationen angeben müssen. So ist zum Beispiel die Information der IP-Adresse und des Ports erst interessant, wenn das Programm ein Paket empfangen hat. Daher muß

man hier nur die Informationen für das Byte-Array angeben. Es muß also ein Puffer eingerichtet sein, der die Daten des empfangenen Paketes aufnehmen kann. **Sollte der Puffer zu klein sein, so gehen überschüssige Informationen verloren!** Daher wählen wir hier die Größe 100. Das dürfte für die Datumsangabe mehr als ausreichend sein.

```
byte[] buffer = new byte[100];  
DatagramPacket timeReceive = new DatagramPacket(buffer, 100);
```

Die einzelnen Informationen des Paketes lassen sich mit einigen Methoden noch nachträglich setzen (*setAddress*, *setPort*, *setData*, *setLength*) oder auslesen (*getAddress*, *getPort*, *getData*, *getLength*, *getOffset*).

Das Erzeugen eines *DatagramSocket*-Objektes

Ein Objekt dieser Klasse wird z.B. folgendermaßen erzeugt:

```
DatagramSocket udpSend = new DatagramSocket(5555);  
DatagramSocket udpReceive = new DatagramSocket(5556);
```

Der Konstruktormparameter *5555* (*5556*) ist hierbei die Port-Nummer, an die der Socket gebunden werden soll. Wir haben hier die zwei Konstruktoren aus den Beispielprogrammen.

Pakete verschicken und empfangen

Zum Versenden und Empfangen von Paketen übergibt man der entsprechenden Methode einfach das zu verschickende bzw. empfangende Paket als Parameter. Zum Senden benutzen wir die folgende Methode:

```
udpSend.send(timeSend);
```

und zum Empfangen:

```
udpReceive.receive(timeReceive);
```

***DatagramSocket*-Objekt schließen**

In Java gibt es den Garbage-Kollektor, der im Hintergrund seine Arbeit verrichtet. Dieser entfernt Objekte, auf die keine Referenz mehr vorliegt, aus dem Speicher. So sieht es auch mit dem Objekt *udpSend* aus. Man möchte allerdings nicht immer auf den Garbage-Kollektor warten. Daher ruft man die Methode *close*, sobald ein Socket nicht mehr gebraucht wird.

```
udpSend.close();  
udpReceive.close();
```

Das Programm *SendPacket*

Das Programm *SendPacket* nutzt einen *DatagramSocket* um ein Paket mit dem aktuellen Datum an das Programm *ReceivePacket* zu schicken.

SendPacket.java

```
// Dieses Programm verschickt ein Datenpaket
// an das Programm ReceivePacket
import java.net.*;
import java.io.*;
import java.util.*;

public class SendPacket {

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

        // Erzeugen eines Datagram-Sockets auf Port 5555
        DatagramSocket udpSend = new DatagramSocket(5555);

        // Erzeugung eines Byte-Puffer und Schreiben des
        // aktuellen Datums in den Puffer
        byte[] buffer = (new Date()).toString().getBytes();
        // Erzeugung eines Packetes, welches das Datum enthält
        // und an den Computer 192.168.0.1 Port 5556 gerichtet ist
        DatagramPacket timeSend = new DatagramPacket(
            buffer, buffer.length,
            InetAddress.getByName("192.168.0.1"), 5556);

        // Ausgabe der Paketinformationen auf den Bildschirm
        System.out.println("Senden der folgenden " +
            "Informationen an");
        System.out.println("IP-Adresse: " +
            timeSend.getAddress());
        System.out.println("Port: " + timeSend.getPort());
        System.out.println("Text: " +
            (new String(timeSend.getData())));

        // Jetzt wir das Paket versendet
        udpSend.send(timeSend);

        // Der Socket wird nicht mehr gebraucht. Also wird
        // er geschlossen
        udpSend.close();
    }
}
```

Das Programm *ReceivePacket*

Dieses Programm ist das Empfangsprogramm zu *SendPacket.java*. Es empfängt ein Datumspaket und gibt dieses auf dem Bildschirm aus.

ReceivePacket.java

```
// Dieses Programm empfängt ein Datenpaket
// von dem Programm SendPacket
import java.net.*;
```

```

import java.io.*;

public class ReceivePacket {

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

        // Erstellen des Sockets
        DatagramSocket udpReceive = new DatagramSocket(5556);
        // Einen kleinen Puffer für das Datum erzeugen
        byte[] buffer = new byte[100];
        // Ein Paket für den Empfang erzeugen
        DatagramPacket timeReceive =
            new DatagramPacket(buffer, 100);

        System.out.println("Warte auf ein Paket...");

        // Auf ein Paket warten...
        udpReceive.receive(timeReceive);

        // Die Paket-Informationen ausgeben
        System.out.println("Habe ein Paket erhalten von");
        System.out.println("IP-Adresse: " +
            timeReceive.getAddress());
        System.out.println("Port: " + timeReceive.getPort());
        System.out.println("Text: " +
            (new String(timeReceive.getData())));

        // Der Socket wird nicht mehr gebraucht. Also wird
        // er geschlossen
        udpReceive.close();
    }
}

```

3.3 Pakete mit mehreren Zielen (Multicasting)

Eine weitere Möglichkeit, die „java.net“ bietet ist das Versenden von Paketen an mehrere Empfänger. Dieses Verfahren wird als **Multicasting** bezeichnet³.

Im folgenden werden zwei Beispielprogramme vorgestellt, die auf den Programmen des vorhergehenden Teils beruhen.

Das Sendeprogramm wurde zu einem kleinen **Zeitserver** ausgebaut. Dieser versendet die aktuelle Zeit und das aktuelle Datum als *String* kodiert. Das Empfangsprogramm wurde auch ein wenig erweitert und kann diesen vom Server verschickten *String* empfangen und ausgeben.

Als erstes Problem, welches zu lösen ist, stellt sich die Frage, wie man überhaupt eine solche **Multicast-Gruppe** bildet. Oder auch: Wie bildet man eine Gruppe von IP-Adressen? Die Lösung besteht darin, daß es einen

³Multicasting läßt sich nicht in Applets einsetzen!

bestimmten Bereich von IP-Adressen gibt, die für eine Mehrfachadressierung vorgesehen sind. Diesen Bereich bilden die Adressen von *224.0.0.3* bis *239.255.255.255*. Die Adressen *224.0.0.0* bis *224.0.0.2* gehören auch zu diesem Bereich. Diesen sind aber Sonderfunktionen zugewiesen, weshalb man sie nicht einsetzen sollte.

Die Mitglieder einer Multicast-Gruppe

Auf der Client-Seite müssen sich nun alle Empfangsprogramme in eine solche Multicast-Gruppe eintragen.

Als erstes erstellt man wie gewohnt einen Socket. Nur in diesem Fall verwendet man keinen einfachen *DatagramSocket*, sondern den um einige Funktionen erweiterten ***MulticastSocket***.

```
MulticastSocket udpReceive = new MulticastSocket(5600);
```

Die Port-Angabe bei der Erzeugung des Sockets muß dabei für alle späteren Gruppenmitglieder identisch sein!

Nun muß man der Multicastgruppe noch beitreten. Dies geschieht durch die Methode *joinGroup*. Dieser Methode übergibt man eine IP-Adresse für Mehrfachadressierung. Die IP muß also aus dem obigen Adreßbereich sein!

```
InetAddress gruppe = InetAddress.getByName("228.2.3.4");  
udpReceive.joinGroup(gruppe);
```

Im weiteren Verlauf kann man mit diesem Socket wie gewohnt arbeiten.

Möchte man keine Pakete mehr von der Gruppe erhalten, so kann man sich von dieser Gruppe wieder abmelden. Dies geschieht über die Methode *leaveGroup*.

```
udpReceive.leaveGroup(gruppe);
```

Das Abmelden von einer Gruppe sollte immer vor dem *close*-Aufruf des Sockets stehen!

Senden an eine Gruppe

Das Senden an eine Multicast-Gruppe gestaltet sich nun ganz einfach. Man versendet die gewünschten Pakete an die Gruppen-IP. In dem Beispiel von oben war dies die Adresse *228.2.3.4*. Der Sender muß dabei aber keinesfalls ein Mitglied dieser Gruppe sein! Dies wäre bei unserem Beispiel des Zeitservers auch vollkommen überflüssig.

Wir verändern am Sendeprogramm nur das Ziel des Paketes.

```
DatagramPacket timeSend = new DatagramPacket(buffer,  
buffer.length, InetAddress.getByName("228.2.3.4"), 5600);
```

Hierbei ist zum einen die Gruppen-IP zu verwenden und zum anderen der gemeinsame Port der Gruppe.

3.4 Zugriff auf Internet-Ressourcen (URLs)

Wie für alles andere auch gibt es in Java eigene Klassen für URLs; sie heißen *URL* und *URLConnection*. Ein *URL*-Objekt erlaubt das Lesen einer Ressource zu einer gegebenen URL.

Erzeugen eines neuen *URL*-Objektes

Es gibt vier Möglichkeiten (**Konstruktoren**), um ein neues *URL*-Objekt zu erzeugen. Die einfachste Möglichkeit ist, einen *String*, der den URL enthält, als Parameter zu übergeben, z.B.:

```
URL cs = new URL("http://www.cs.uni-bonn.de/");
```

Es ist auch möglich, neue *URL*-Objekte mit der Angabe einer relativen Adresse aus schon bestehenden zu erzeugen:

```
URL index = new URL(cs,"index.html");
```

Dieses Objekt würde den URL *http://www.cs.uni-bonn.de/index.html* enthalten.

Man kann den URL auch aus den einzelnen Bestandteilen zusammenbauen, dazu gibt es zwei Möglichkeiten:

```
URL url = new URL(String protocol, String host, String file);
URL url = new URL(String protocol, String host, int port,
String file);
```

Da man die Adresse eines *URL*-Objektes nicht mehr verändern kann, muß man für jede Adresse ein neues *URL*-Objekt erzeugen. Dazu können ein paar Methoden nützlich sein, die Bestandteile eines existierenden *URL*-Objektes liefern:

- *int getPort()* liefert die Port-Nummer, *-1* falls unbekannt
- *String getProtocol()* liefert die Protokollbezeichnung
- *String getHost()* liefert den Namen des Hostrechners
- *String getFile()* liefert den Dateinamen
- *String getRef()* liefert die Dateireferenz bzw. das Sprungziel in der Datei

Direktes Lesen von einem URL

Die Methode *openStream* liefert ein Objekt der Klasse *InputStream*, auf das dann Methoden wie *read* angewandt werden können, z.B.:

```
InputStream in = url.openStream();
```

Möchte man zeilenweise Daten auslesen, muß man sich zunächst ein *BufferedReader*-Objekt beschaffen:

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(url.openStream()));
String inputline = in.readLine();
```

Anschließend sollte die Verbindung mit *close()* wieder geschlossen werden:

```
in.close();
```

Lesen und Schreiben von Daten

Die Methode *URL.openConnection()* stellt eine Verbindung her und liefert ein Objekt der Klasse **URLConnection**, das neben dem Empfangen auch das Senden von Daten an einen URL erlaubt, z.B. ein Formular an ein auf einem Server laufendes CGI-Skript. Die Methoden *getOutputStream* und *getInputStream* beschaffen die nötigen E/A-Ströme.

3.5 Authentifizierung

Ein Objekt der Klasse **Authenticator** ist für das eigentliche Nachweisen einer Authentifizierung zuständig; üblicherweise wird der Benutzer dann aufgefordert, die entsprechenden Daten einzugeben. Doch wie wird dies im einzelnen implementiert?

Zunächst muß eine Klasse von *Authenticator* abgeleitet werden; von dieser Unterklasse muß dann ein Objekt beim System registriert werden. Dies geschieht durch die statische Methode *Authenticator.setDefault()*. Eine Anfrage durch den Server an den registrierten *Authenticator* wird durch Aufruf der Methode *requestPasswordAuthentication* realisiert. Die eigentliche Anfrage an den Benutzer geschieht jedoch mit *getPasswordAuthentication*, das vom System aufgerufen wird. Diese Methode muß überschrieben werden, weil die Standardimplementierung statt eines Objektes der Klasse *PasswordAuthentication* nur *null* zurückliefert. Dieses sollte aber den Benutzernamen und das Paßwort enthalten. Um eine geeignete Eingabeaufforderung zu erstellen, können die Attribute des Servers abgefragt werden; die Methoden dafür heißen *getRequestingSite*, *getRequestingPort*, *getRequestingProtocol*, *getRequestingPrompt* und *getRequestingScheme*.

Kapitel 4

Anwendungsbeispiele

4.1 Server-Client-Kommunikation über Sockets

Das Programm *UniversalClient*

Das Programm *UniversalClient* baut die Verbindung zu einem Server auf und leitet an diesen die Daten weiter, die der Benutzer über die Tastatur eingibt. Gleichzeitig werden die Antworten des Servers auf dem Bildschirm dargestellt. Über die Eingabe von „quit“ kann das Programm beendet werden.

UniversalClient.java

```
// Dieses Programm sendet Benutzereingaben an
// einen Server und zeigt die Antworten an
import java.net.*;
import java.io.*;

class UniversalClient
{
    static void error(String message)
    {
        System.err.println(message);
        System.exit(1);
    }

    public static void main(String[] args) throws IOException
    {
        if (args.length != 2)
            error("Verwendung: java UniversalClient " +
                "<server> <port>");

        Socket sock = new Socket(args[0],
            Integer.parseInt(args[1]));
        System.out.println("Verbindung hergestellt, " +
            "geben Sie ihre Daten ein");

        InputStream in = sock.getInputStream();
        OutputStream out = sock.getOutputStream();
        BufferedReader keyboard = new BufferedReader(
```

```

        new InputStreamReader(System.in));

// Empfangsthread starten
(new ReceiveThread(in,"Server: ").start());

while (true)
{
    // eine Zeile einlesen
    String str = keyboard.readLine();
    if (str.equalsIgnoreCase("quit"))
    {
        sock.close();
        System.exit(1);
    }

    // die Zeile an den Server senden
    out.write((str+'\n').getBytes());
}
}
}

class ReceiveThread extends Thread
{
    BufferedReader in;
    String message;

    static void error(String message)
    {
        System.err.println(message);
        System.exit(1);
    }

    public ReceiveThread(InputStream in,String message)
    {
        super();
        this.in = new BufferedReader(new InputStreamReader(in));
        this.message = message;
    }

    public void run()
    {
        try
        {
            while (true)
            {
                // eine Zeile empfangen
                String str = in.readLine();

                // die Zeile ausgeben
                if (str != null)
                    System.out.println(message+str);
            }
        }
    }
}

```

```

    }
}
catch (SocketException e)
{
    error("Verbindung wurde getrennt");
}
catch (IOException e)
{
    error(e.toString());
}
}
}

```

Das Programm *UniversalServer*

UniversalServer funktioniert analog zu *UniversalClient*. Der Server wartet, bis ein Client eine Verbindung zu ihm aufbaut, anschließend werden die Anfragen des Clients auf dem Bildschirm angezeigt und die Tastatureingaben an den Client geschickt. Die Klasse *ReceiveThread* wird in unveränderter Form von *UniversalClient* übernommen.

UniversalServer.java

```

// Dieses Programm arbeitet als Server und sendet
// die Benutzereingaben an die Clients
import java.net.*;
import java.io.*;

class UniversalServer
{
    static ServerSocket server = null;

    public static void main(String[] args) throws IOException
    {
        if (args.length != 1)
        {
            System.err.println("Verwendung: java UniversalServer " +
                "<port>");
            System.exit(1);
        }

        server = new ServerSocket(Integer.parseInt(args[0]));
        System.out.println("Warte auf Verbindung auf Port " +
            args[0]);
        Socket sock = server.accept();
        System.out.println("Verbindung hergestellt, " +
            "geben Sie ihre Daten ein");

        InputStream in = sock.getInputStream();
        OutputStream out = sock.getOutputStream();
        BufferedReader keyboard = new BufferedReader(

```

```

        new InputStreamReader(System.in));

(new ReceiveThread(in, "Client: ")).start();

while (true)
{
    // eine Zeile einlesen
    String str = keyboard.readLine();
    if (str.equalsIgnoreCase("quit"))
    {
        sock.close();
        System.exit(1);
    }

    // die Zeile an den Client senden
    out.write((str+'\n').getBytes());
}
}
}
}

```

4.2 Zeitserver im Netzwerk

Das Programm *TimeServer*

Das Programm *TimeServer* sendet ununterbrochen Datumsinformationen an eine Gruppen-IP.

TimeServer.java

```

// Dieses Programm verschickt Zeitinformationen an TimeClients
import java.net.*;
import java.io.*;
import java.util.*;

public class TimeServer {

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

        // Erzeugen eines Datagram-Sockets auf Port 5555
        DatagramSocket udpSend = new DatagramSocket(5555);
        // Erzeugen eines leeren Puffers
        byte[] buffer = new byte[100];
        // Erzeugung eines Packetes, welches an den Computer
        // 228.168.0.1 Port 5600 gerichtet ist
        DatagramPacket timeSend = new DatagramPacket(
            buffer, buffer.length,
            InetAddress.getByName("228.2.3.4"), 5600);

        // Eine kurze Nachricht wie man den Server stoppt ausgeben
        System.out.println("Druecken Sie ENTER, " +
            "um den Server zu beenden");
    }
}

```

```

// Die Versende-Schleife wird nach einem ENTER abgebrochen
while (System.in.available() == 0) {

    // Erzeugung eines Byte-Puffer und Schreiben
    // des aktuellen Datums in den Puffer
    buffer = (new Date()).toString().getBytes();

    // Das Paket bekommt jetzt den Datumpuffer zugewiesen
    timeSend.setData(buffer);

    // Jetzt wir das Paket versendet
    udpSend.send(timeSend);

}

// Der Socket wird nicht mehr gebraucht. Also wird
// er geschlossen
udpSend.close();
}
}

```

Das Programm *TimeClient*

Das Programm *TimeClient* erzeugt einen *MulticastSocket* und empfängt die von *TimeServer* versendeten Datumsinformationen. Hier wird nur eine einfache Version dargestellt, die nur eine Information empfängt und dann beendet. Für eine Zeitsynchronisation wäre es interessant, daß die Zeitinformationen periodisch abgefragt werden. Dies leistet dieses Programm jedoch nicht! Es soll nur das Arbeiten mit Multicast-Sockets demonstrieren.

TimeClient.java

```

// Dieses Programm empfängt Zeitinformationen
// von dem TimeServer
import java.net.*;
import java.io.*;

public class TimeClient {

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

        // Erstellen des Sockets
        MulticastSocket udpReceive = new MulticastSocket(5600);
        // Mitglied in der Multicast-Gruppe werden...
        InetAddress gruppe = InetAddress.getByName("228.2.3.4");
        udpReceive.joinGroup(gruppe);

        // Einen kleinen Puffer für das Datum erzeugen
        byte[] buffer = new byte[100];
    }
}

```

```

// Ein Paket für den Empfang erzeugen
DatagramPacket timeReceive = new DatagramPacket(
    buffer, 100);

System.out.println("Warte auf ein Paket...");

// Auf ein Paket warten...
udpReceive.receive(timeReceive);

// Die Paket-Informationen ausgeben
System.out.println("Habe ein Paket erhalten von");
System.out.println("IP-Adresse: " +
    timeReceive.getAddress());
System.out.println("Port: " + timeReceive.getPort());
System.out.println("Text: " +
    (new String(timeReceive.getData())));

// und die Multicast-Gruppe wieder verlassen...
udpReceive.leaveGroup(gruppe);

// Der Socket wird nicht mehr gebraucht. Also wird
// er geschlossen
udpReceive.close();
}
}

```

4.3 Ein kleiner Text-Webbrowser

Das folgende Beispielprogramm liest Daten von einem URL und gibt diese zeilenweise auf dem Bildschirm aus. Der Port, auf den zugegriffen wird, wird aus dem angegebenen Protokoll ermittelt, z.B. wird bei HTTP der Port 80 benutzt.

URLReader.java

```

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

public class URLReader {
    public static void main(String[] args) throws Exception {
        URL url = new URL("ftp://localhost/index.html");
        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                url.openStream()));

        String inputLine;

        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
    }
}

```

```

        in.close();
    }
}

```

Das nächste Programm demonstriert den Einsatz von *URLConnection*. Es werden nicht nur Daten gelesen, sondern auch an die URL-Adresse gesendet. Die zu übertragenden Daten müssen zunächst mit *URLEncoder.encode(String s)* in ein übertragbares Format konvertiert werden. Außerdem muß ein Flag gesetzt werden, das anzeigt, daß wir auch Daten versenden wollen. Dies geschieht mit der Methode *setDoOutput(boolean dooutput)* der Klasse *URLConnection*.

Die Zeichenkette wird an ein CGI-Skript gesendet, daß die Zeichenkette umkehrt und dann beide Versionen zurückschickt. Die empfangenen Daten werden dann ausgegeben.

Reverse.java

```

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

public class Reverse2
{
    public static void main(String[] args) throws Exception
    {
        String eingabe = URLEncoder.encode(
            "'Dies wird gesendet'");

        URL url = new URL(
            "http://java.sun.com/cgi-bin/backwards");
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);

        OutputStream out = connection.getOutputStream();
        out.write(("string=" + eingabe).getBytes());
        out.close();

        InputStream in = connection.getInputStream();
        int b;
        while ((b = in.read()) != -1)
            System.out.write((char)b);

        in.close();
    }
}

```

4.4 Authentifizierung

Dieses Programm erstellt ein *Authenticator*-Objekt, das bei Zugriffen auf paßwortgeschützte Seiten die Eingabe von Benutzernamen und Paßwörtern verlangt. Das Programm versucht anschließend zur Demonstration, eine Verbindung zu einer solchen Seite aufzubauen.

SampleAuthenticator.java

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

class SampleAuthenticator extends Authenticator
{
    protected PasswordAuthentication getPasswordAuthentication()
    {
        System.out.println("Anfrage vom Server: " +
            getRequestingSite());
        System.out.println("Port-Nummer des Servers: " +
            getRequestingPort());
        System.out.println("Protokoll: " +
            getRequestingProtocol());
        System.out.println("Beschreibung: " +
            getRequestingPrompt());
        System.out.println("Schema: " + getRequestingScheme());

        BufferedReader in=new BufferedReader(
            new InputStreamReader(System.in));

        try
        {
            System.out.print("Geben Sie Ihren Benutzernamen ein: ");
            String UserName=in.readLine();
            System.out.print("Geben Sie Ihr Passwort ein: ");
            String Password=in.readLine();
            return new PasswordAuthentication(
                UserName, Password.toCharArray());
        }
        catch (IOException e)
        {
            System.err.println(e.toString());
            return null;
        }
    }

    public static void main(String[] args) throws IOException
    {
        // Install authenticator
        Authenticator.setDefault(new SampleAuthenticator());

        URL url=new URL("http://web.informatik.uni-bonn.de/IV/"+
            "martini/Lehre/Veranstaltungen/WS9900/InformatikI/"+
            "Folienkopien/Kapitel0.pdf");
        InputStream in=url.openStream();
        in.close();
    }
}
```

Kapitel 5

Weiterführende Themen

5.1 Austausch von Objekten über das Netzwerk (Objektserialisierung)

In „java.io“ gibt es zwei Klassen (*ObjectOutputStream* und *ObjectInputStream*), die eine weitere Möglichkeit für Netzwerkprogrammierung eröffnen. Mit Hilfe dieser beiden Klassen ist es möglich, Objekte über ein Netzwerk zu verschicken. Dabei wird die Technik der **Objektserialisierung** genutzt. Diese beiden Klassen ermöglichen aber nicht nur das Versenden von Objekten, sondern u.a. auch das Zwischenspeichern von Objekten zur späteren Nutzung.

Was ist Objektserialisierung?

Bei der Objektserialisierung werden Objekte über einen **Stream** verschickt. Dabei müssen die Objekte zunächst in einen Datenstrom umgewandelt werden. Darüber muß sich der Java-Programmierer aber meist keine Gedanken machen. Dies übernehmen die beiden oben genannten Klassen. *ObjectOutputStream* wandelt ein Objekt in einen Datenstrom um, während *ObjectInputStream* diesen wieder in ein Objekt umwandelt. Eine Instanz dieser beiden Klassen kann beliebige Input- und Output-Streams erweitern. Somit kann überall, wo mit Streams gearbeitet wird, die Objektserialisierung eingesetzt werden, was einen vielfältigen Einsatzbereich ermöglicht.

Läßt sich jedes Objekt serialisieren?

Es ist nicht immer wünschenswert, daß sich jedes Objekt serialisieren läßt. Daher gibt es die Möglichkeit zu bestimmen, welche Klassen serialisierbar sind. Man entscheidet also, ob alle Instanzen einer Klasse serialisierbar sind, und nicht, ob spezielle Instanzen serialisierbar sind. Soll eine Klasse serialisierbar sein, so muß sie das leere Interface *Serializable* oder dessen Erweiterung *Externalizable* implementieren:

```
public class MySerializableClass implements Serializable {...}
```

Man muß also keine zusätzlichen Methoden schreiben, um eine Klasse serialisierbar zu machen, da das Interface leer ist. Implementiert man jedoch die Erweiterung *Externalizable*, so muß man zwei Methoden implementieren, die den Serialisierungsprozeß komplett festlegen. Möchte man den Prozeß nicht

komplett beeinflussen, so reicht meist die Nutzung des Interfaces *Serializable* und die Deklaration der beiden Methoden *writeObject* und *readObject*, worauf hier nicht genauer eingegangen werden soll.

Schützen von Informationen

Nicht alle Informationen eines Objektes können serialisiert werden. Wenn eine Member-Variable der Klasse als *static* deklariert ist, so muß sie nicht übertragen werden, da sie keine Objektvariable ist, sondern eine Klassenvariable. Es kann aber auch durchaus sinnvoll sein, daß nicht alle Informationen des Objektes übertragen werden. Man kann diese Daten als *transient* markieren. Transiente Member-Variablen werden nicht in den Serialisierungsprozeß mit einbezogen. Dies kann zum Beispiel sinnvoll sein, wenn ein Objekt ein Paßwort enthält und dieses Paßwort nicht mitübertragen werden soll. Die Member-Variable, die das Paßwort enthält würde z.B. folgendermaßen deklariert:

```
transient String passwd;
```

Schreiben und Lesen von Objekten

Das Schreiben von Objekten auf einen Objekt-Stream läßt sich genauso ausführen wie das Schreiben eines Strings auf einen normalen Stream. Man muß zunächst einen beliebigen Input- bzw. Output-Stream erweitern:

```
FileOutputStream out = new FileOutputStream("Zeit");
ObjectOutputStream objectOut = new ObjectOutputStream(out);
```

```
FileInputStream in = new FileInputStream("Zeit");
ObjectInputStream objectIn = new ObjectInputStream(in);
```

Auf einen solchen Stream läßt sich nun ein Objekt mit den Methoden *writeObject* bzw. *readObject* schreiben oder lesen.

```
objectOut.writeObject("Heute");
objectOut.writeObject(new Date());
```

und der dazu passende Lese-Abschnitt:

```
String today = (String)objectIn.readObject();
Date date = (Date)objectIn.readObject();
```

Wie man sieht, kann man über einen Objekt-Stream nicht nur Objekte eines Typs verschicken. In unserem Beispiel werden zwei unterschiedliche Typen verschickt, zum einen ein *String*-Objekt und zum anderen ein *Date*-Objekt.

5.2 Verteilte Systeme mit Java

Wir haben bereits die Kommunikation zwischen Computern mit Hilfe von Sockets kennengelernt. Diese Methode ist recht einfach und bietet die bestmögliche Geschwindigkeit. Dennoch sind die Sockets für die Übertragung von Bytes gedacht. Sollen komplexe Daten übertragen werden, kann der Programmieraufwand enorm werden.

Ein wichtiger Bereich, wo dies der Fall ist, sind verteilte Systeme. Soll eine Anwendung auf mehrere Computer aufgeteilt werden, um die zur Verfügung stehende Rechenleistung besser auszunutzen, läßt es sich meistens nicht vermeiden, daß die Anwendungsteile viele Daten von unterschiedlichster Art miteinander austauschen müssen. Auch die Objektserialisierung ist hier lediglich eine Notlösung, die das Programm unübersichtlich und schwer nachvollziehbar macht.

Java bietet zwei Techniken, **RMI** und **CORBA**, mit denen der Programmieraufwand für verteilte Systeme wesentlich gesenkt werden kann. Diese werden im folgenden vorgestellt.

5.2.1 Remote Methode Invocation (RMI)

RMI ist die einfachere der beiden Techniken, denn hier unterscheidet sich ein verteiltes Programm kaum von einem normalen. Die notwendigen Definitionen befinden sich im Paket „java.rmi“. Die Kommunikation läuft folgendermaßen ab:

Der eine Computer stellt ein Objekt zur Verfügung, indem er ihn unter einem bestimmten Namen registriert. Dann kann der andere Computer über diesen Namen das Objekt erhalten. Das erhaltene Objekt kann anschließend ganz normal verwendet werden.

Es gibt jedoch einen entscheidenden Unterschied zu Objektserialisierung: physikalisch bleibt das Objekt weiterhin auf dem Computer, der ihn zur Verfügung gestellt hat. Auch alle Methodenaufrufe werden auf diesem Computer ausgeführt, d.h. es wird dabei kaum Rechenzeit auf dem anderen Computer verbraucht. Die Klasse, zu der das Objekt gehört, braucht nicht lokal installiert zu sein, ein gemeinsames Interface ohne irgendwelche Implementierung genügt. Der Datentransfer, der hierzu notwendig ist, wird komplett von Java übernommen und läuft unsichtbar für den Programmierer ab.

Leider hat RMI auch einen Nachteil: diese Technik wird ausschließlich von Java verwendet. Das bedeutet, daß die Programme auf allen kommunizierenden Computern in Java geschrieben sein müssen.

5.2.2 Common Object Request Broker Architecture (CORBA)

CORBA ist ein Standard, der bereits von vielen Programmiersprachen unterstützt wird. Dementsprechend mußten bei CORBA auch Kompromisse eingegangen werden, und die Verwendung ist nicht so komfortabel wie bei RMI. Dafür kann CORBA auch von nicht-objektorientierten Sprachen wie C oder Cobol verwendet werden.

Ähnlich wie bei RMI erfolgt die Kommunikation über Interfaces, die Objektmethoden zur Verfügung stellen. Allerdings können keine Java-Interfaces verwendet werden. Ein CORBA-Interface wird mit **IDL** (Interface Definition Language) definiert, indem für alle Methoden die Parameter und Rück-

gabewerte angegeben werden. Das Interface braucht nur auf dem Computer definiert zu sein, der es zur Verfügung stellt, die anderen erhalten die Methodenbeschreibung und können diese zur Laufzeit auswerten.

Nachdem das Interface definiert wurde, gibt es der Computer zur Verwendung frei. Anschließend können andere Computer mit Hilfe des **ORB** (Object Request Broker) eine Referenz darauf erhalten und damit die Methoden aufrufen.

5.3 Der dynamische Web-Server (Servlets)

Servlets sind Java-Klassen, deren grundsätzliche Funktionsweise mit der von CGI-Programmen vergleichbar ist. Sie laufen auf dem Server und erstellen dort dynamische HTML-Seiten, deren Inhalt zum Beispiel aus Datenbankabfragen resultieren kann. Servlets sind für Server das, was Applets für Browser sind, haben jedoch keine grafische Benutzeroberfläche. Sie haben gegenüber Applets den Vorteil der besseren Kompatibilität, die aus zwei Dingen resultiert: da die Servlets auf dem Server laufen, und nicht wie Applets in einem Browser, sind sie nicht von Kompatibilitätsproblemen, die durch unterschiedliche Implementierungen der JVM in Browsern bedingt sind, betroffen; zum anderen entfällt die grafische Benutzeroberfläche, wodurch Java hier seine Plattformunabhängigkeit voll ausspielen kann.

Vorteile von Servlets gegenüber CGI- und Perl-Skripten sind zum einen die schnellere Ausführung, zum anderen können Servlets mittels Threads mehrere Requests parallel bearbeiten, so daß das wiederholte Starten externer Prozesse entfällt.

Auch die Sicherheit spielt auf Webservern in zweierlei Hinsicht eine Rolle: einerseits verhindern Spracheigenschaften wie die starke Typsicherheit und das Fehlen von Adreßzeigern einen Großteil der Programmierfehler, die auf Webservern für Abstürze und Sicherheitslücken verantwortlich sind, andererseits profitieren Servlets vom Sicherheitskonzept der JVM, das eine sehr flexible Vergabe bzw. Einschränkung der Zugriffsrechte eines Programms ermöglichen.

Servlets können mit dem **Java Servlet API** entwickelt werden, einer Standarderweiterung von Java; außerdem enthält das **Java Servlet Development Kit** (JSDK) Klassen, die viele Standardaufgaben bereits abdecken, und so dem Entwickler Zeit geben, sich auf die Programmierung der eigentlichen Aufgabe zu konzentrieren.

Literaturverzeichnis

- [1] *The Java Tutorial*
Sun Microsystems, 1999
<http://www.java.sun.com/docs/books/tutorial/>
- [2] *Java2 Platform, API Specification*
Sun Microsystems, 1999, Standard Edition, v1.2.2
<http://www.java.sun.com/products/jdk/1.2/docs/api/>
- [3] James Gosling, Bill Joy, Guy Steele: *The Java Language Specification*
Sun Microsystems, 1996, Edition 1.0
<http://java.sun.com/docs/books/jls/>
- [4] *Request For Comments: RFC791, RFC1060, RFC1945, RFC2068*
The Internet Activity Board (IAB)
<http://rfc.fh-koeln.de/>
- [5] Guido Krüger: *Go To Java 2*
Addison-Wesley, 1999
<http://www.gkrueger.com/books/k99a.html>
- [6] *CORBA in 14 Tagen*
Markt&Technik Buch- und Software-Verlag GmbH
<http://www.datanetworks.ch/~steger/pages/dom/corba/inhalt.htm>

Index

- Anwendungsschicht, **4–5**, 11–12
- Authenticator, 26
 - .getPasswordAuthentication, 26
 - .getRequestingPort, 26
 - .getRequestingPrompt, 26
 - .getRequestingProtocol, 26
 - .getRequestingScheme, 26
 - .getRequestingSite, 26
 - .requestPasswordAuthentication, 26
 - .setDefault, 26
- Authentifizierung, **13, 26**
- Berechtigung, **13**, 19
- BindException, 19
- BufferedInputStream, 18
- BufferedReader, 17–18, 25
- Client, **6–7**, 14–18
- Common Object Request Broker Architecture, **38–39**
- ConnectException, 15
- CORBA, *siehe* Common Object Request Broker Architecture
- Datagram, 10, **20–23**
- DatagramPacket, 20–21
 - .getAddress, 21
 - .getData, 21
 - .getLength, 21
 - .getOffset, 21
 - .getPort, 21
 - .setAddress, 21
 - .setData, 21
 - .setLength, 21
 - .setPort, 21
- DatagramSocket, **20–23**
 - .close, 21
 - .receive, 21
 - .send, 21
- DNS, *siehe* Domain Name System
 - Name, **6**, 13, 14
- Domäne, 6
- Domain Name System, **6**
- Externalizable, 36–37
- Fehlerkorrektur, 4, 9–11
- File Transfer Protocol, 13
- Firewall, **7–8**
- FTP, *siehe* File Transfer Protocol
 - Server, 7, 13
- Hardwareschicht, **3–4**
- Host-ID, 5
- HTTP, *siehe* Hypertext Transfer Protocol
 - Server, 7
- Hypertext Transfer Protocol, **11–12**, 13, 32
- IDL, *siehe* Interface Definition Language
- IllegalArgumentException, 15
- InetAddress, 14, 20
 - .getByName, 14
- InputStream, 15, 25
- Interface Definition Language, 38
- Internet, 5, 7
 - Ressourcen, 12
- Internet Protocol, 4, **5–6**
- InterruptedIOException, 18, 19
- IOException, 15, 19
- IP, *siehe* Internet Protocol
 - Adresse, **5–6**, 14, 20, 24
 - Netz, 4–6
 - Routing, **5–6**
- Java Servlet Development Kit, 39
- JSDK, *siehe* Java Servlet Development Kit
- Kommunikationskanal, 9, 10
- Loopback, 5
- Modem, 3, 4
- Multicast-Gruppe, 23–24
- Multicasting, 5, **10–11**, 23–24
- MulticastSocket, 24
 - .close, 24

- .joinGroup, 24
- .leaveGroup, 24
- Name-Server, 6
- Network Information Center, 5
- Netzwerk, 3
- Netzwerk-ID, 5
- Netzwerkkarte, 3, 4
- Netzwerkschicht, 4, 5–6
- Object Request Broker, 39
- ObjectInputStream, 36
 - .readObject, 37
- ObjectOutputStream, 36
 - .writeObject, 37
- Objektserialisierung, **36–37**
- ORB, *siehe* Object Request Broker
- OutputStream, 15
- PasswordAuthentication, 26
- Port, 6–7, 18
 - Nummer, 6–7, 13, 14, 20, 21
- Proxy-Server, 7–8
- Remote Methode Invocation, **38**
- Reverse, Programmbeispiel, **33–34**
- RMI, *siehe* Remote Methode Invocation
- Router, 5–6
- Routing-Tabelle, 6
- SampleAuthenticator, Programmbeispiel, **34–35**
- Schichtenmodell, 3–5
- SecurityException, 15, 19
- Serialisierung, **36–37**
- Serializable, 36–37
- Server, 6–7, 18–20
- Server-Client-Schema, 6–7, 14–20, 27–30
- ServerSocket, 18–20
 - .accept, 19
 - .close, 19
 - .setSoTimeout, 19
- Servlet, **39**
- Sicherheit, 7–8, 39
- Socket, 14–20
 - .getInputStream, 15
 - .getOutputStream, 15
 - .setSoTimeout, 18
- SocketException, 15
- Stream, 15, 36
- TCP, *siehe* Transmission Control Protocol
- Telnet, 8
- Thread, 16, 19
- TimeClient, Programmbeispiel, **31–32**
- Timeout, 18–19
- TimeServer, Programmbeispiel, **30–31**
- transient, 37
- Transmission Control Protocol, 4, 7, 9, 14–20
- Transportschicht, 4, 9–11
- UDP, *siehe* User Datagram Protocol
- Uniform Resource Locator, 12–13, 25–26
- UniversalClient, Programmbeispiel, **27–29**
- UniversalServer, Programmbeispiel, **29–30**
- UnknownHostException, 14
- URL, *siehe* Uniform Resource Locator
 - .getFile, 25
 - .getHost, 25
 - .getPort, 25
 - .getProtocol, 25
 - .getRef, 25
 - .openConnection, 26
 - .openStream, 25
- URLConnection, 25–26, 33
 - .getInputStream, 26
 - .getOutputStream, 26
 - .setDoOutput, 33
- URLEncoder, 33
 - .encode, 33
- URLReader, Programmbeispiel, **32–33**
- User Datagram Protocol, 4, 7, 10, 20–23
- Verbindung, 6, 14–15, 19
 - sichere, 9
- Webbrowser, 12, 13, **32–34**
- Webseiten-Adressen, 13
- Webserver, 11, 13, 39
- Zeitserver, 23, 30–32
- Zeitsynchronisation, 10, 23, 30–32