Inhaltsverzeichnis

1. Grundlagen Netzwerkprogrammierung

1.1. Ziele

☑ Die Grundlagen der Netzwerprogrammierung: Protokolle, Adressen, Kommunikation

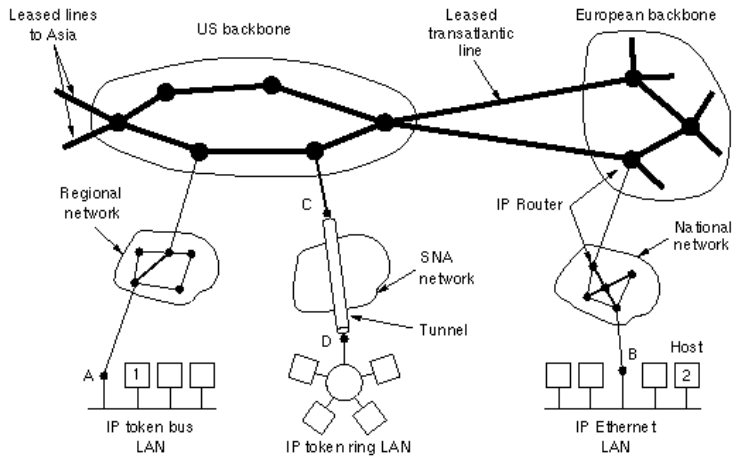
☑ Internetprotokolle verstehen und anwenden können

1.2. +Eine kurze Geschichte des Internet

Das Internet ist eine Sammlung von Teilnetzen:

Es gibt keine echte Struktur des Netzes, sondern mehrere größere ***Backbones***, die quasi das Rückgrat (wie der Name Backbone ja schon sagt) des Internet bilden.
Die Backbones werden aus Leitungen mit sehr **hoher Bandbreite** und schnellen Routern gebildet.

An die Backbones sind wiederum **größere regionale Netze** angeschlossen, die LANs von Universitäten, Behörden, Unternehmen und Service-Providern verbinden.



"Das Internet" (Quelle: A.S. Tanenbaum: Computernetworks).

1960+: DoD wollte ausfallssicheres Netz:

Gegen Ende der **sechziger Jahre**, als der "kalte Krieg" seinen Höhepunkt erlangte, wurde vom US-Verteidigungsministerium (**Department of Defence - DoD**) eine Netzwerktechnologie gefordert, die in einem hohen Maß **ausfallssicher** ist. Das Netz

sollte dazu in der Lage sein, auch im Falle eines Atomkrieges weiter zu operieren. Eine Datenübermittlung über Telefonleitungen war zu diesem Zweck nicht geeignet, da diese gegenüber Ausfällen zu verletzlich waren (sind).

ARPA (Technologien f. Militär) wurde beauftragt:

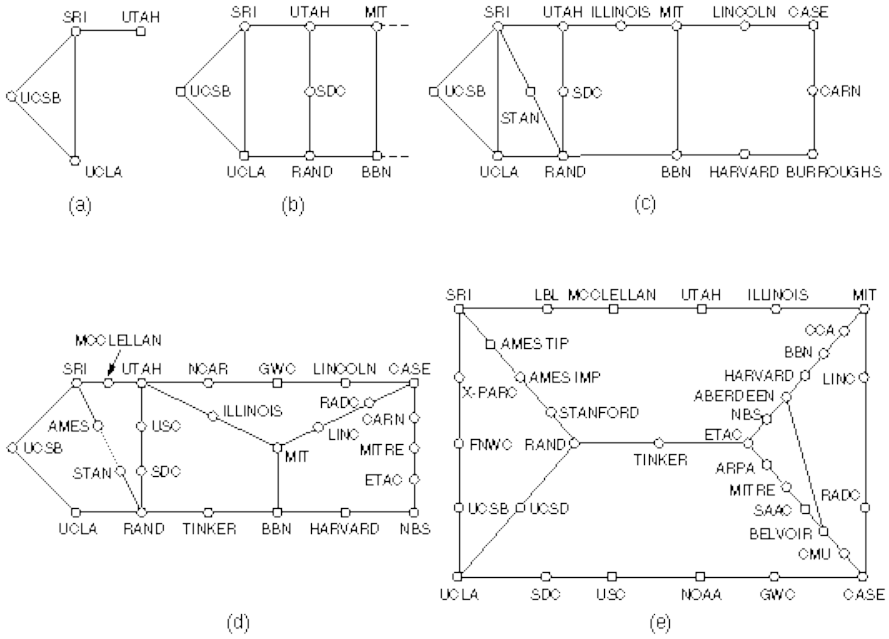
Aus diesem Grund beauftragte das US-Verteidigungsministerium die *Advanced Research Projects Agency (**ARPA**)* mit der **Entwicklung einer zuverlässigen Netztechnologie**. Die ARPA wurde 1957 als Reaktion auf den Start des Sputniks durch die UdSSR gegründet. Die ARPA hatte die Aufgabe Technologien zu entwickeln, die für das Militär von Nutzen sind. Zwischenzeitlich wurde die ARPA in *Defense Advanced Research Projects Agency (**DARPA**)* umbenannt, da ihre Interessen primär militärischen Zwecken dienten. Die ARPA war keine Organisation, die Wissenschaftler und Forscher beschäftigte, sondern **verteilte Aufträge an Universitäten und Forschungsinstitute**.

PACKET-SWITCHED Netzwork (virtuelle Verbindungen)

Um die geforderte Zuverlässigkeit des Netzes zu erreichen, fiel die Wahl darauf, das Netz als ein *paketvermitteltes Netz **(packet-switched network**)* zu gestalten. Bei der Paketvermittlung werden zwei Partner während der Kommunikation nur **virtuell** miteinander verbunden. Die zu übertragenden **Daten werden vom Absender in Stücke variabler oder fester Länge zerlegt und über die virtuelle Verbindung übertragen**; vom Empfänger werden diese Stücke nach dem Eintreffen wieder zusammengesetzt. Im Gegensatz dazu werden bei der *Leitungsvermittlung (circuit switching)* für die Dauer der Datenübertragung die Kommunikationspartner fest miteinander verbunden.

1969 ARPANET mit 4 Knoten (University of Clifornia, Stanford)

Ende 1969 wurde von der *University of California Los Angeles (**UCLA**)*, der *University of California Santa Barbara (**UCSB**)*, dem *Stanford Research Institute (**SRI**)* und der *University of Utah* ein experimentelles Netz, das ***ARPANET***, mit vier Knoten in Betrieb genommen. Diese vier Universitäten wurden von der (D)ARPA gewählt, da sie bereits eine große Anzahl von ARPA-Verträgen hatten. Das ARPA-Netz wuchs rasant (siehe Abbildung) und überspannte bald ein großes Gebiet der Vereinigten Staaten.



Wachstum des ARPANET
a)Dezember 1969 b)July 1970 c)März 1971 d)April 1971 e)September 1972.
(Quelle: A.S. Tanenbaum: Computernetworks)

1974 TCP/IP Protokolle um versch. Netze zu verbinden

Mit der Zeit und dem Wachstum des ARPANET wurde klar, daß die bis dahin gewählten Protokolle nicht mehr für den Betrieb eines größeren Netzes, das auch mehrere (Teil)Netze miteinander verband, geeignet war.
Aus diesem Grund wurden schließlich weitere Forschungsarbeiten initiiert, die **1974** zur Entwicklung der ***TCP/IP-Protokolle* bzw. des *TCP/IP-Modells*** führten. TCP/IP wurde mit mit der Zielsetzung entwickelt, mehrere verschiedenartige **Netze** zur Datenübertragung miteinander zu **verbinden**.

TCP/IP in Berkeley UNIX integriert

Um die Einbindung der TCP/IP-Protokolle in das ARPANET zu forcieren beauftragte die (D)ARPA die Firma *Bolt, Beranek & Newman (BBN)* und die *University of California at Berkeley* zur Integration von **TCP/IP in Berkeley UNIX**. Dies bildete auch den Grundstein des Erfolges von TCP/IP in der UNIX-Welt.

1983: MILNET vom ARPANET abgetrennt

Im Jahr **1983** wurde das ARPANET schließlich von der *Defence Communications Agency (DCA)*, welche die Verwaltung des ARPANET von der (D)ARPA übernahm, aufgeteilt. Der militärisch Teil des ARPANET, wurde in ein separates Teilnetz, das ***MILNET***, abgetrennt, das durch streng kontrollierte Gateways vom Rest des ARPANET - dem Forschungsteil - separiert wurde.

Das ARPANET wird zum INTERNET

Nachdem TCP/IP das einzige offizielle Protokoll des ARPANET wurde, nahm die Zahl der angeschlossenen Netze und Hosts rapide zu. Das ARPANET wurde von Entwicklungen, die es selber hervorgebracht hatte, überrannt. Das ARPANET in seiner ursprünglichen Form existiert heute nicht mehr, das MILNET ist aber noch in Betrieb. (Zum Wachstum des Internet https://de.wikipedia.org/wiki/Geschichte_des_Internets)

Die Sammlung von Netzen, die das ARPANET darstellte, wurde zunehmend als ***Netzverbund*** betrachtet. Dieser Netzverbund wird heute allgemein als *das* ***Internet*** bezeichnet. Der Leim, der das Internet zusammenhält, sind die TCP/IP-Protokolle.

RFCs - Request for Comments.

Eine wichtige Rolle bei der Entstehung und Entwicklung des Internet spielen die sogenannten RFCs. **RFCs sind Dokumente, in denen u.a. die Standards für TCP/IP bzw. das Internet veröffentlicht werden.**
Ist ein Dokument veröffentlicht, wird ihm eine RFC-Nummer zugewiesen. Das originale RFC wird nie verändert oder aktualisiert. Sind Änderungen an einem Dokument notwendig, wird es mit einer neuen RFC-Nummer publiziert.
Das 'System' der RFCs leistet einen wesentlichen Beitrag zum Erfolg von TCP/IP und dem Internet.

- rfc768 - UDP
- rfc783 - TFTP
- rfc791 - IP
- rfc792 - ICMP
- rfc793 - TCP
- rfc814 - Name, adresses, ports and routes
- rfc821/2 - Mail
- rfc825 - Specification for RFC's
- rfc826 - ARP

- rfc854 - TELNET
- rfc894 - A Standard for the Transmission of IP Datagrams over Ethernet
- rfc903 - RARP
- rfc950 - Internet Standard Subnetting Procedure (Subnets)
- rfc959 - FTP

Anm.: Wer weitere Quellen zur Geschichte des Internet sucht, wird hier fündig:

- Internet Society - ISOC: History of the Internet
- Musch J.: Die Geschichte des Netzes: ein historischer Abriß
- Hauben M.: Behind the Net: The Untold History of the ARPANET and Computer Science
- Hauben R.: The Birth and Development of the ARPANET
- w3history: Die Geschichte des World Wide Web

1.3. Referenzmodelle, Protokolle, Protokollhierarchien

Ein **Rechnernetz** (Computer Network) besteht aus miteinander verbundenen autonomen Computern.

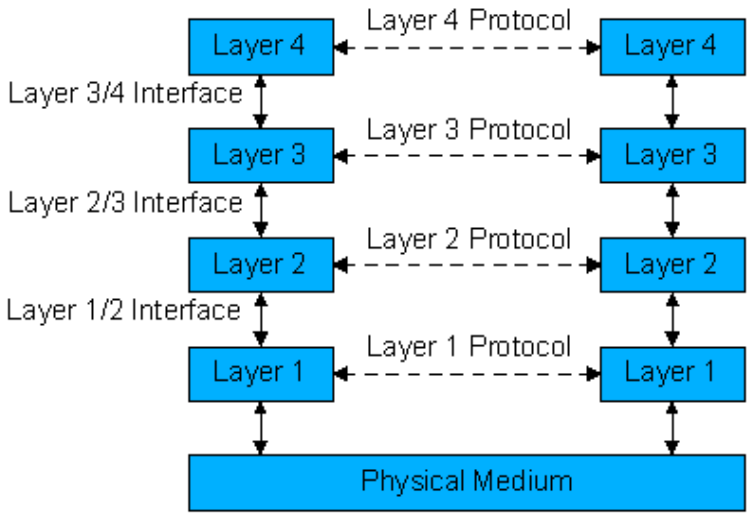
Protokolle sind Regeln, die den Nachrichtenaustausch - oder allgemeiner das Verhalten - zwischen (Kommunikations)Partnern regeln ("Protocols are formal rules of behaviour").

Die Verletzung eines vereinbarten Protokolls erschwert die Kommunikation oder macht sie sogar gänzlich unmöglich.

Ein Beispiel für ein Protokoll "aus dem täglichen Leben" ist z.B. der Funkverkehr: Die Kommunikationspartner bestätigen den Empfang einer Nachricht mit *Roger* und leiten einen Wechsel der Sprechrichtung mit *Over* ein. Beendet wird die Verbindung schließlich mit *Over and out*.

Datenaustausch durch viele Protokolle mit indiv. Teilaufgaben

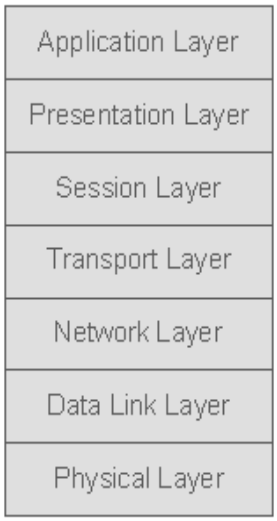
Ähnliche Protokolle werden auch beim Datenaustausch zwischen verschiedenen Computern benötigt - auch wenn hier die Komplexität der Anforderungen etwas höher ist. Aufgrund dieser höheren Komplexität werden viele Aufgaben nicht von einem einzigen Protokoll abgewickelt. In der Regel kommen eine ganze **Reihe von Protokollen, mit verschiedenen Teilaufgaben**, zum Einsatz. Diese Protokolle sind dann in Form von **Protokollschichten** mit jeweils **unterschiedlichen Funktionen** angeordnet.



Anordnung von Protokollen zu einem Protokollstapel.

1.3.1. Das OSI-Referenzmodell

Das *Open Systems Interconnection (**OSI**)-Referenzmodell* ist ein Modell, dass auf einem Vorschlag der *International Standards Organisation (ISO)* basiert.



Das OSI-Referenzmodell.

Das OSI-Modell (die offizielle Bezeichnung lautet *ISO-OSI-Referenzmodell*) besteht aus **sieben** Schichten. Die Schichtung beruht auf dem Prinzip, daß eine Schicht der jeweils übergeordneten Schicht bestimmte **Dienstleistungen anbietet**.

Das OSI-Modell beschreibt, welche Aufgaben die Schichten erledigen sollen.

Den Schichten im OSI-Modell sind die folgenden Aufgaben zugeordnet:

Anwendungsschicht (application layer):

Die Anwendungsschicht enthält eine große Zahl häufig benötigter Protokolle, die einzelne Programme zur Erbringung ihrer Dienste definiert haben. Auf der Anwendungsschicht finden sich z.B. die **Protokolle für die Dienste ftp, http, telnet, mail** etc.

Darstellungsschicht (presentation layer):

Die Darstellungsschicht regelt die Darstellung der Übertragungsdaten in einer von der darüber liegenden Ebene unabhängigen Form. Computersysteme verwenden z.B. oft verschiedene Codierungen für Zeichenketten (z.B. ASCII, Unicode), Zahlen usw. Damit diese Daten zwischen den Systemen ausgetauscht werden können, **kodiert die Darstellungsschicht die Daten auf eine standardisierte und vereinbarte Weise.**

Sitzungsschicht (session layer):

Die Sitzungsschicht (oft auch Verbindungsschicht oder Kommunikationssteuerschicht genannt) ermöglicht den **Verbindungsauf- und abbau.** Von der Sitzungsschicht wird der Austausch von Nachrichten auf der Transportverbindung geregelt. Sitzungen können z.B. ermöglichen, ob der Transfer gleichzeitig in zwei oder nur eine Richtung erfolgen kann. Kann der Transfer jeweils in nur eine Richtung stattfinden, regelt die Sitzungsschicht, welcher der Kommunikationspartner jeweils an die Reihe kommt.

Transportschicht (transport layer):

Die Transportschicht übernimmt den Transport von Nachrichten zwischen den Kommunikationspartnern. Die Transportschicht hat die grundlegende Aufgaben, den **Datenfluß zu steuern(Reihenfolge der Datensegmente)** und die **Unverfälschtheit** der Daten sicherzustellen. Beispiele für Transportprotokolle sind **TCP und UDP**.

Netzwerkschicht (network layer):

Die Netzwerkschicht (Vermittlungsschicht) hat die Hauptaufgabe eine **Verbindung zwischen Knoten im Netzwerk** herzustellen. Die Netzwerkschicht soll dabei die übergeordneten Schichten von den Details der Datenübertragung über das Netzwerk befreien. Eine der wichtigsten Aufgaben der Netzwerkschicht ist z.B. die Auswahl von **Paketrouten** bzw. das Routing vom Absender zum Empfänger. Das Internet Protokoll (**IP**) ist in der Netzwerkschicht einzuordnen.

Sicherungsschicht (data link layer):

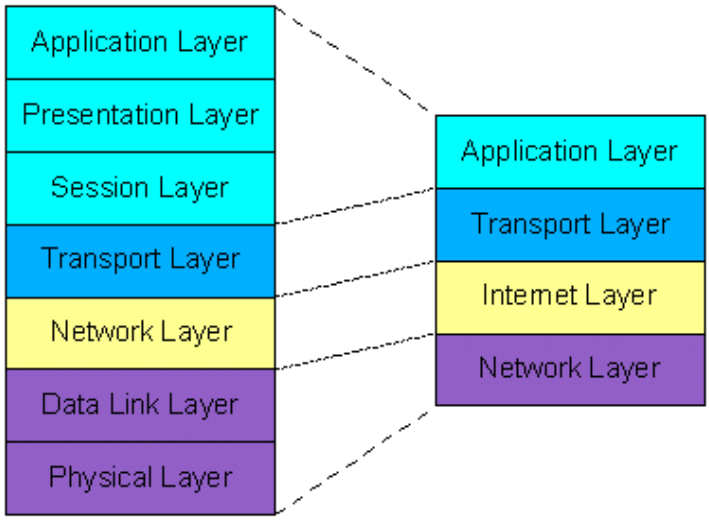
Die Aufgabe der Sicherungsschicht (Verbindungsschicht) ist die **gesicherte Übertragung von Daten**. Vom Sender werden hierzu die Daten in Rahmen (frames) aufgeteilt und sequentiell an den Empfänger gesendet. Vom Empfänger werden die empfangenen Daten durch Bestätigungsrahmen quittiert. Protokollbeispiele für die Sicherungsschicht sind HDLC (high-level data link control), **SLIP** (serial line IP) und **PPP** (point-to-point Protokoll).

Bitübertragungsschicht (physical layer):

Die Bitübertragungsschicht regelt die Übertragung von Bits über das Übertragungsmedium. Dies betrifft die Übertragungsgeschwindigkeit, die Bit-Codierung, den Anschluß (wieviele Pins hat der Netzanschluß?) etc. Die Festlegungen auf der Bitübertragungsschicht betreffen im wesentlichen die Eigenschaften des Übertragungsmedium.

1.3.2. Das TCP/IP-Referenzmodell

Im vorhergehenden Abschnitt wurde das OSI-Referenzmodell vorgestellt. In diesem Abschnitt soll nun das Referenzmodell für die TCP/IP-Architektur vorgestellt werden. Das *TCP/IP-Referenzmodell* - benannt nach den beiden primären Protokollen TCP und IP der Netzarchitektur beruht auf den Vorschlägen, die bei der Fortentwicklung des ARPANETs gemacht wurden. Das TCP/IP-Modell ist zeitlich vor dem OSI-Referenzmodell entstanden, deshalb sind auch die Erfahrungen des TCP/IP-Modells mit in die OSI-Standardisierung eingeflossen. Das TCP/IP-Referenzmodell besteht im Gegensatz zum OSI-Modell aus nur **vier** Schichten: Application Layer, Transport Layer, Internet Layer, Network Layer.



Das TCP/IP-Referenzmodell.

Applikationssschicht (application layer): Die Applikationsschicht (auch Verarbeitungsschicht genannt) umfaßt alle höherschichtigen Protokolle des TCP/IP-Modells. Zu den ersten Protokollen der Verarbeitungsschicht zählen **TELNET** (für virtuelle Terminals), **FTP** (Dateitransfer) und **SMTP** (zur Übertragung von E-Mail). Im Laufe der Zeit kamen zu den etablierten Protokollen viele weitere Protokolle wie z.B. **DNS** (Domain Name Service) und **HTTP** (Hypertext Transfer Protocol) hinzu.

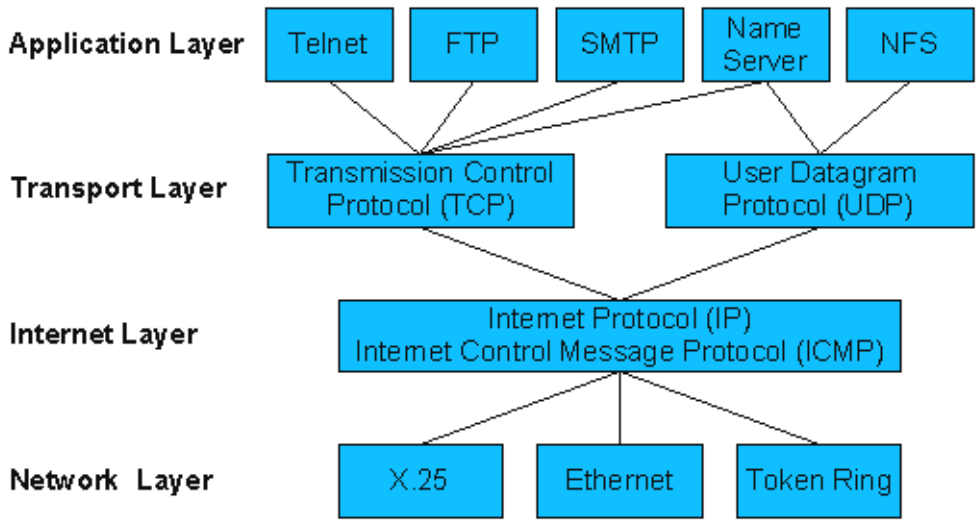
Transportschicht (transport layer): Wie im OSI-Modell ermöglicht die Transportschicht die Kommunikation zwischen den Quell- und Zielhosts. Im TCP/IP-Referenzmodell wurden auf dieser Schicht zwei Ende-zu-Ende-Protokolle definiert: das Transmission Control Protocol (**TCP**) und das User Datagram Protocol (**UDP**). TCP ist ein zuverlässiges verbindungsorientiertes Protokoll, durch das ein Bytestrom fehlerfrei einen anderen Rechner im Internet übermittelt werden kann. UDP ist ein unzuverlässiges verbindungsloses Protokoll, das vorwiegend für Abfragen und Anwendungen in Client/Server-Umgebungen verwendet wird, in denen es in erster Linie nicht

um eine sehr genaue, sondern schnelle Datenübermittlung geht (z.B. Übertragung von Sprache und Bildsignalen).

Internetschicht (internet layer): Die Internetschicht im TCP/IP-Modell definiert nur ein Protokoll namens **IP** (Internet Protocol), das alle am Netzwerk beteiligten Rechner verstehen können. Die Internetschicht hat die Aufgabe IP-Pakete richtig zuzustellen. Dabei spielt das Routing der Pakete eine wichtige Rolle. Das Internet Control Message Protocol (ICMP) ist fester Bestandteil jeder IP-Implementierung und dient zur Übertragung von Diagnose- und Fehlerinformationen für das Internet Protocol.

Netzwerkschicht (network layer): Unterhalb der Internetschicht befindet sich im TCP/IP-Modell eine große Definitionslücke. Das Referenzmodell sagt auf dieser Ebene nicht viel darüber aus, was hier passieren soll. Festgelegt ist lediglich, daß zur Übermittlung von IP-Paketen ein Host über ein bestimmtes Protokoll an ein Netz angeschlossen werden muß. Dieses Protokoll ist im TCP/IP-Modell nicht weiter definiert und weicht von Netz zu Netz und Host zu Host ab. Das TCP/IP-Modell macht an dieser Stelle vielmehr Gebrauch von bereits vorhandenen Protokollen, wie z.B. Ethernet (IEEE 802.3), Serial Line IP (SLIP), etc.

1.3.3. Die TCP/IP-Protokoll-Architektur

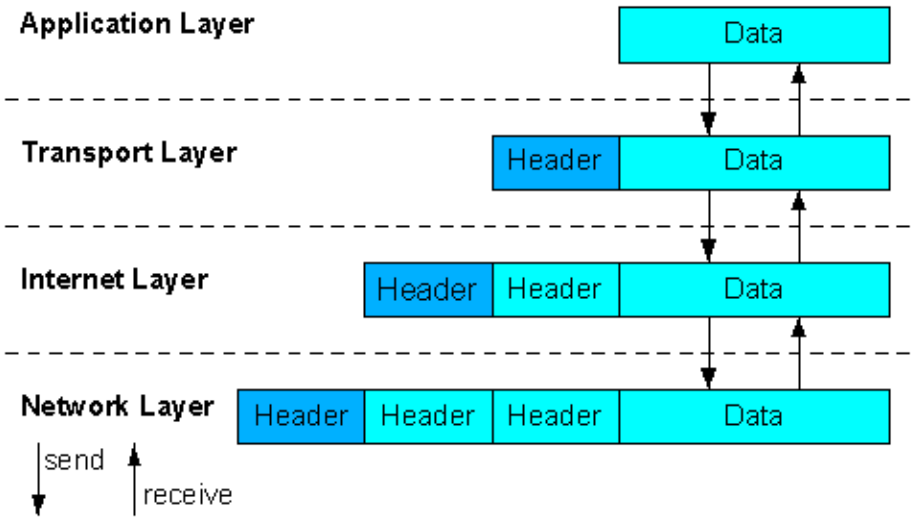


Die TCP/IP-Protokoll-Architektur.

Daten, die von einem Applikationsprogramm über ein Netzwerk versendet werden, durchlaufen den TCP/IP-Protokollstapel von der Applikationsschicht zur Netzwerkschicht.

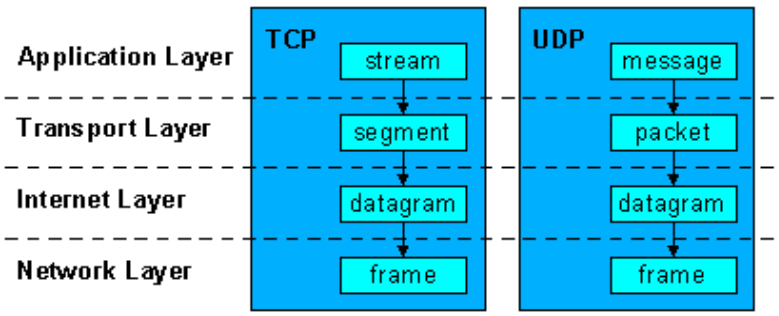
Von jeder Schicht werden dabei Kontrollinformationen in Form eines Protokollkopfes angefügt.

Diese Kontrollinformationen dienen der korrekten Zustellung der Daten. Das Zufügen von Kontrollinformationen wird als *Einkapselung (encapsulation)* bezeichnet.



Dateneinkapselung (Quelle: Hunt: TCP/IP Network Administration).

Applikationen, die das Transmission Control Protocol benutzen, bezeichnen Daten als *Strom (**stream**)*; Applikationen, die das User Datagram Protocol verwenden, bezeichnen Daten als *Nachricht (**message**)*. Auf der Transportebene bezeichnen die Protokolle TCP und UDP ihre Daten als *Segment (**segment**)* bzw. *Paket (**packet**)*. Auf der Internet Schicht werden Daten allgemein als *Datengramm (**datagram**)* benannt. Oft werden die Daten hier aber auch als Paket bezeichnet. Auf der Netzwerkebene bezeichnen die meisten Netzwerke ihre Daten als *Pakete* oder *Rahmen (**frames**)*.



Bezeichnung der Daten auf den verschiedenen Schichten des TCP/IP-Modells ([Hu95]).

1.4. Internet Protokoll (IP)

Das *Internet Protokoll (Internet Protocol - IP)* ist der Leim, der dies alles zusammenhält. IP stellt die **Basisdienste** für die Übermittlung von Daten in TCP/IP-Netzen bereit und ist im RFC **791** spezifiziert. Hauptaufgaben des Internet Protokolls sind die **Adressierung** von Hosts und das **Fragmentieren** von Paketen. Diese **Pakete** werden von IP nach bestem Bemühen ("best effort") von der Quelle zum Ziel **befördert**, unabhängig davon, ob sich die Hosts im gleichen Netz befinden oder andere Netze dazwischen liegen.

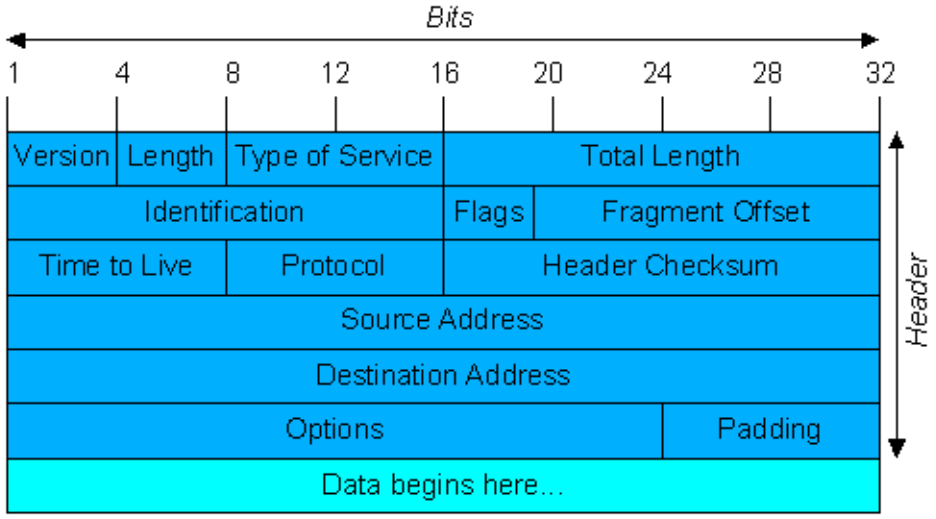
Die Funktionen von IP umfassen:

☑ Die Definition von **Datengrammen**, welche die Basiseinheiten für die Übermittlung von Daten im Internet bilden.
☑ Definition des **Adressierungsschemas**.
☑ **Übermittlung** der Daten von der Transportebene zur Netzwerkschicht.
☑ **Routing** von Datengrammen durch das Netz.
☑ **Fragmentierung und Zusammensetzen** von Datengrammen.

IP ist ein
☑ ***verbindungsloses*** Protokoll, d.h. zur Datenübertragung wird **keine Ende-zu-Ende**-Verbindung der Kommunikationspartner etabliert.
☑ Ferner ist IP ein ***unzuverlässiges*** Protokoll, da es über keine Mechanismen zur Fehlererkennung und -behebung verfügt. Diese Aufgaben übernehmen dann übergeordnete Schichten.

1.4.1. +IP-Header /IP datagram

Ein IP-Datengramm besteht aus einem Header und den zu übertragenden Daten. Der Header hat einen festen **20 Byte** großen Teil, gefolgt von einem optionalen Teil variabler Länge. Der Header umfaßt alle Informationen, die notwendig sind, um das Datengramm dem Empfänger zuzustellen. Ein Datengramm kann theoretisch maximal 64 KByte groß sein, in der Praxis liegt die Größe ungefähr bei **1500** Byte (das hängt mit der maximalen Rahmengröße des Ethernet-Protokolls zusammen).



Der IP-Header.

Die Felder des in der Abbildung dargestellten Protokollkopfes haben die folgende Bedeutung:

Version:

Das *Versions-Feld* enthält die Versionsnummer des IP-Protokolls. Durch die Einbindung der Versionsnummer besteht die Möglichkeit über eine längere Zeit mit verschiedenen Versionen des IP Protokolls zu arbeiten. Einige Hosts können mit der alten und andere mit der neuen Version arbeiten. Die derzeitige Versionsnummer ist 4, aber die Version 6 des IP Protokolls befindet sich bereits in der Erprobung.

Length:

Das Feld *Length (Internet Header Length - IHL)* enthält die Länge des Protokollkopfs, da diese nicht konstant ist. Die Länge wird in 32-Bit-Worten angegeben. Der kleinste zulässige Wert ist 5 - das entspricht also 20 Byte; in diesem Fall sind im Header keine Optionen gesetzt. Die Länge des Headers kann sich durch Anfügen von Optionen aber bis auf 60 Byte erhöhen (der Maximalwert für das 4-Bit-Feld ist 15).

Type of Servive:

Über das Feld *Type of Service* kann IP angewiesen werden Nachrichten nach bestimmten Kriterien zu behandeln. Als Dienste sind hier verschiedene Kombinationen aus Zuverlässigkeit und Geschwindigkeit möglich. In der Praxis wird dieses Feld aber ignoriert, hat also den Wert 0.

Total Length:

Enthält die gesamte *Paketlänge*, d.h. Header und Daten. Da es sich hierbei um ein 16-Bit-Feld handelt ist die Maximallänge eines Datengramms auf 65.535 Byte begrenzt. In der Spezifikation von IP (RFC 791) ist festgelegt, daß jeder Host in der Lage sein muß, Pakete bis zu einer Länge von 576 Bytes zu verarbeiten. In der Regel können von den Host aber Pakete größerer Länge verarbeitet werden.

Identification:

Über das *Identifikationsfeld* kann der Zielhost feststellen, zu welchem Datengramm ein neu angekommenes Fragment gehört. Alle Fragmente eines Datengramms enthalten die gleiche Identifikationsnummer, die vom Absender vergeben wird.

Flags:

Das Flags-Feld ist drei Bit lang. Die Flags bestehen aus zwei Bits namens *DF - Don't Fragment* und *MF - More Fragments*. Das erste Bit des Flags-Feldes ist ungenutzt bzw. reserviert. Die beiden Bits DF und MF steuern die Behandlung eines Pakets im Falle einer Fragmentierung. Mit dem DF-Bit wird signalisiert, daß das Datengramm nicht fragmentiert werden darf. Auch dann nicht, wenn das Paket dann evtl. nicht mehr weiter transportiert werden kann und verworfen werden muß. Alle Hosts müssen, wie schon gesagt Fragemente bzw. Datengramme mit einer Größe von 576 Bytes oder weniger verarbeiten können. Mit dem MF-Bit wird angezeigt, ob einem IP-Paket weitere Teilpakete nachfolgen. Diese Bit ist bei allen Fragmenten außer dem letzten gesetzt.

Fragment Offset:

Der *Fragmentabstand* bezeichnet, an welcher Stelle relativ zum Beginn des gesamten Datengramms ein Fragment gehört. Mit Hilfe dieser Angabe kann der Zielhost das Originalpaket wieder aus den Fragmenten zusammensetzen. Da dieses Feld nur 13 Bit groß ist, können maximal 8192 Fragmente pro Datengramm erstellt werden. Alle Fragmente, außer dem letzten, müssen ein Vielfaches von 8 Byte sein. Dies ist die elementare Fragmenteinheit.

Time to Live:

Das Feld *Time to Live* ist ein Zähler, mit dem die Lebensdauer von IP-Paketen begrenzt wird. Im RFC 791 ist für dieses Feld als Einheit Sekunden spezifiziert. Zulässig ist eine maximale Lebensdauer von 255 Sekunden (8 Bit). Der Zähler muß von jedem Netzknoten, der durchlaufen wird um mindestens 1 verringert werden. Bei einer längeren Zwischenspeicherung in einem Router muß der Inhalt sogar mehrmals verringert werden. Enthält das Feld den Wert 0, muß das Paket verworfen werden: damit wird verhindert, daß ein Paket endlos in einem Netz umherwandert. Der Absender wird in einem solchen Fall durch eine Warnmeldung in Form einer *ICMP-Nachricht* (siehe weiter unten) informiert.

Protocol:

Enthält die Nummer des Transportprotokolls, an das das Paket weitergeleitet werden muß. Die Numerierung von Protokollen ist im gesamten Internet einheitlich und im RFC 1700 definiert. Bei UNIX-Systemen sind die Protokollnummern in der Datei `/etc/protocols` abgelegt.

Header Checksum:

Dieses Feld enthält die Prüfsumme der Felder im IP-Header. Die Nutzdaten des IP-Datengramms werden aus Effiziengründen nicht mit geprüft. Diese Prüfung findet beim

Empfänger innerhalb des Transportprotokolls statt. Die Prüfsumme muß von jedem Netzknoten, der durchlaufen wird, neu berechnet werden, da der IP-Header durch das Feld Time-to-Live sich bei jeder Teilstrecke verändert. Aus diesem Grund ist auch eine sehr effiziente Bildung der Prüfsumme wichtig. Als Prüfsumme wird *das 1er-Komplement der Summe aller 16-Bit-Halbwörter der zu überprüfenden Daten* verwendet. Zum Zweck dieses Algorithmus wird angenommen, daß die Prüfsumme zu Beginn der Berechnung Null ist.

Source Address, Destination Address:

In diese Felder werden die 32-Bit langen Internet-Adressen zur eingetragen. Die Internet-Adressen werden im nächsten Abschnitt näher betrachtet.

Options und Padding:

Das Feld *Options* wurde im Protokollkopf aufgenommen, um die Möglichkeit zu bieten das IP-Protokoll um weitere Informationen zu ergänzen, die im ursprünglichen Design nicht berücksichtigt wurden.

Weitere Details zu den Optionen sind in RFC 791 zu finden.

1.4.2. IP-Adressen: Class A,B,C,...

Die IP-Adresse besteht aus 2 Teilen:
IP-Adresse = **Netzwerk-Adresse + Host-Adresse**

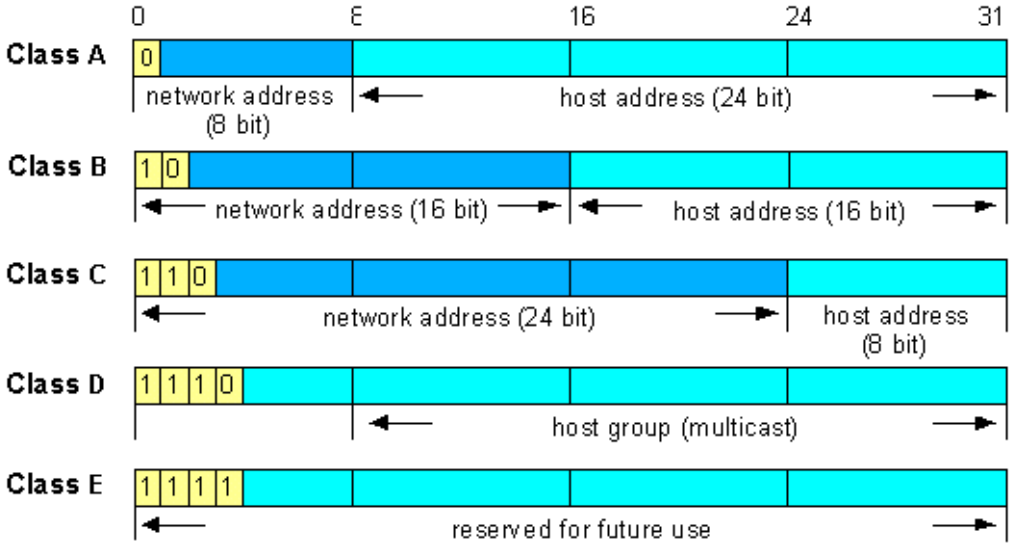
Jeder Host und Router im Internet hat eine 32-Bit lange IP-Adresse (IPv4).

Eine IP-Adresse ist **eindeutig**: kein Knoten im Internet hat die gleiche IP-Adresse wie ein anderer.

Maschinen, die an mehrere Netze angeschlossen sind, haben in jedem Netz eine eigene IP-Adresse.

Die Netzwerkadressen werden vom *Network Information Center (NIC)* [http://www.internic.net] vergeben, um Adresskonflikte zu vermeiden. Seit einiger Zeit hat diese Aufgabe die Internet Assigned Numbers Authority (IANA) [http://www.iana.org] bzw. ihre Vertreter in den verschiedenen Gebieten (Asia Pacific Network Informatrion Center (APNIC), American Registry for Internet Numbers (ARIN), Reseau IP Europeens (RIPE)) übernommen.

Die Adressen werden nicht einzeln zugeordnet, sondern nach **Netzklassen** vergeben. Beantragt man IP-Adressen für ein Netz, so erhält man nicht für jeden Rechner eine Adresse zugeteilt, sondern einen **Bereich von Adressen**, der selbst zu verwalten ist.



IP-Adressformate.

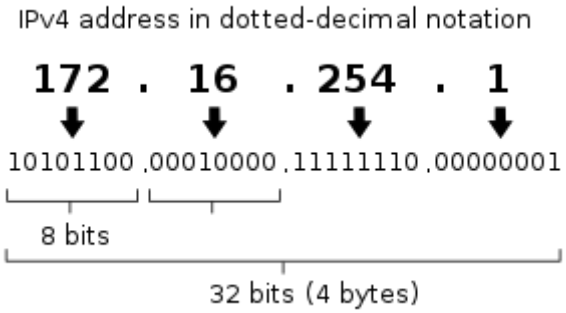
Wie die obere Abbildung zeigt, sind IP-Adressen in verschiedene Klassen, mit unterschiedlich langer Netzwerk- und Hostadresse, eingeteilt.

Die **Netzwerk-Adresse** definiert das Netzwerk, in dem sich ein Host befindet.
Alle Hosts eines Netzes haben die gleiche Netzwerkadresse.

Die **Host-Adresse** identifiziert einen bestimmten Rechner innerhalb eines Netzes.
Ist ein Host an mehrere Netze angeschlossen, so hat er **für jedes Netz eine eigene IP-Adresse**.

"*Eine IP-Adresse identifiziert keinen bestimmten Computer [Host], sondern eine Verbindung zwischen einem Computer [Host] und einem Netz. Einem Computer [Host] mit mehreren Netzanschlüssen (z.B. ein Router) muss für jeden Anschluss eine IP-Adresse zugewiesen werden.*" *(*[Co98]*)*

IP-Adressen sind 32-Bit große Zahlen, die normalerweise nicht als Binärzahl, sondern in gepunkteten Dezimalzahlen geschrieben werden. In diesem Format wird die 32-Bit große Zahl in 4 Byte getrennt, die mit Punkten voneinander getrennt sind. Die Adresse 01111111111111111111111111111111 wird so z.B. als 127.255.255.255 geschrieben. Die niedrigste IP-Adresse ist 0.0.0.0., die höchste 255.255.255.255.
Wie zuvor gesagt, sind IP-Adressen in Klassen unterteilt. Der Wert des ersten Bytes gibt die Adressklasse an:



Quelle: https://en.wikipedia.org/wiki/IP_address

Adressklasse					
Erstes Byte	**Bytes für die Netzadresse**	**Bytes für die Hostadresse**	**Adressformat***	**Anzahl Hosts**	
Klasse A	1-126	1	3	N.H.H.H	2^{24} (~16 Mio.)
Klasse B	128-191	2	2	N.N.H.H	2^{16} (~64000)
Klasse C	192-223	3	1	N.N.N.H	254
*N steht für einen Teil der Netzadresse, H für einen Teil der Hostadresse.					

Klasse A:

Das erste Byte hat einen Wert **kleiner als 128**, d.h. das erste Bit der Adresse ist 0. Das erste Byte ist Netzwerknummer, die letzten drei Bytes identifizieren einen Host im Netz. Es gibt demzufolge also 126 Klasse A Netze, die bis zu 16 Millionen Host in einem Netz.

Klasse B:

Ein Wert von **128 bis 191** für das erste Byte (das erste Bit ist gleich 1, Bit 2 gleich 0) identifiziert eine Klasse B Adresse. Die ersten beiden Bytes identifizieren das Netzwerk, die letzen beiden Bytes einen Host. Das ergibt 16382 Klasse B Netze mit bis zu 64000 Hosts in einem Netz.

Klasse C:

Klasse C Netze werden über Werte von **192 bis 223** für das erste Byte (die ersten beiden Bits sind gleich 1, Bit 3 gleich 0) identifiziert. Es gibt 2 Millionen Klasse C Netze, d.h. die ersten drei Bytes werden für die Netzwerkadresse verwendet. Ein Klasse C Netz kann bis zu 254 Host beinhalten.

Klasse D:

Klasse D Adressen, sogenannte *Multicast-Adressen*, werden dazu verwendet ein Datengramm an mehrere Hostadressen gleichzeitig zu versenden. Das erste Byte einer Multicast-Adresse hat den Wertebereich von **224 bis 239**, d.h. die ersten drei Bytes sind gesetzt und Byte 4 ist gleich 0. Sendet ein Prozeß eine Nachricht an eine Adresse der Klasse D, wird die Nachricht an alle Mitglieder der adressierten Gruppe versendet.

Die Übermittlung der Nachricht erfolgt, wie bei IP üblich, nach bestem Bemühen, d.h. ohne Garantie, daß die Daten auch tatsächlich alle Mitglieder einer Gruppe erreichen.

Der weitere Bereich der IP-Adressen von 240 bis 254 im ersten Byte ist für zukünftige Nutzungen reserviert. In der Literatur wird dieser Bereich oft auch als Klasse E bezeichnet (vgl. [Co98]).

IP-Adressen für den privaten Gebrauch

Im Internet müssen die Netzkennungen eindeutig sein. Aus diesem Grund werden die (Netz)Adressen, wie weiter oben schon gesagt, von einer zentralen Organisation vergeben. Dabei ist sichergestellt, daß die Adressen eindeutig sind und auch im Internet sichtbar sind. Dies ist aber nicht immer notwendig. Netze, die keinen Kontakt zum globalen Internet haben, benötigen keine Adresse, die auch im Internet sichtbar ist. Es ist auch nicht notwendig, dass sichergestellt ist, dass diese Adressen in keinem anderen, privaten Netz eingesetzt werden. Aus diesem Grund wurden Adressbereiche festgelegt, die nur für private Netze bestimmt sind. Diese Bereiche sind in RFC 1918 (*Address Allocation for Private Internets*) festgelegt (RFC 1597, auf das sich oft auch neuere Literatur bezieht, ist durch RFC 1918 ersetzt). Diese IP-Nummern dürfen im Internet nicht weitergeleitet werden. Dadurch ist es möglich, diese Adressen in beliebig vielen, nicht- öffentlichen Netzen, einzusetzen.
Die folgenden Adressbereiche sind für die Nutzung in privaten Netzen reserviert:

Klasse A: **10.0.0.0**

Für ein privates Klasse A-Netz ist der Adressbereich von 10.0.0.0 bis 10.255.255.254 reserviert.

Klasse B: **172.16.0.0 bis 172.31.0.0**

Für die private Nutzung sind 16 Klasse B-Netze reserviert. Jedes dieser Netze kann aus bis zu 65.000 Hosts bestehen (also z.B. ein Netz mit den Adressen von 172.17.0.1 bis 172.17.255.254).

Klasse C: 192.168.0.0 bis 192.168.255.0

256 Klasse C-Netzte stehen zur **privaten** Nutzung zur Verfügung. Jedes dieser Netze kann jeweils 254 Hosts enthalten (z.B. ein Netz mit den Adressen 192.168.0.1 bis 192.168.0.254).

Jeder kann aus diesen Bereichen den Adressbereich für sein eigenes privates Netz auswählen. Auch die folgenden Netzadressen sind reserviert und haben die Bedeutung:

Die Netz-Adressen 0 und 127

- Adressen mit der **Netznummer 0** beziehen sich auf das **aktuelle Netz**.
  Mit einer solchen Adresse können sich Hosts auf ihr eigenes Netz beziehen, ohne die Netzadresse zu kennen (allerdings muß bekannt sein, um welche Netzklasse es sich handelt, damit die passende Anzahl Null-Bytes gesetzt wird).

- **127** steht für das ***Loopback*** *Device* eines Hosts.
  Pakete, die an eine Adresse der Form 127.x.y.z gesendet werden, werden nicht auf einer Leitung ausgegeben, sondern lokal verarbeitet. Dieses Merkmal wird häufig zur Fehlerbehandlung benutzt.

Die Host-Adressen 0 und 255

Bei allen Netzwerkklassen sind die Werte 0 und 255 bei den Hostadressen reserviert.

- Eine IP-Adresse, bei der alle Hostbits auf 0 gesetzt sind, identifiziert das Netz selbst.
  Die Adresse 80.0.0.0 bezieht sich so z.B. auf das Klasse A Netz 80.
  Die Adresse 128.66.0.0 bezieht sich auf das Klasse B Netz 128.66.

- Eine IP-Adresse, bei der alle Host-Bytes den Wert **255** haben, ist eine ***Broadcast-****Adresse*. Eine Broadcast-Adresse wird benutzt, um alle Hosts in einem Netzwerk zu adressieren.

1.4.3. IP-Adressen: CIDR (class internet domain routing)

Siehe auch http://de.wikipedia.org/wiki/Subnetzmaske

1993 wurde das Class-Konzept durch CIDR (Class internet domain routing) ersetzt.

Ein Präfix gibt die Bitanzahl für die Netz-Adresse an.

IPv4-Adresse = 10.43.8.67/**28**

Netzmaske: 11111111.11111111.11111111.1111**0000**
Netzmaske = 255.255.255.240

Host-Adressen: (32-28= 4 Bits)
mit 4 Stellen im Dualsystem lassen sich 16 unterschiedliche Werte darstellen, nämlich 0–15)
→ **16 Adressen** − (Broadcast- und Netzadresse)
→ 14 IPv4-Adressen zu vergeben.

Berechnung: CIDR

gegeben	**Duale Darstellung der Adressen**	**Dezimale Darstellung**
IP-Adresse	00001010.00101011.00001000.01000011	10.43.8.67
Netmask	11111111.11111111.11111111.1111**0000**	255.255.255.**240**
not Netmask	00000000.00000000.00000000.0000**1111**	
Berechnung		
Net-Adr.: IP AND Netmask	00001010.00101011.00001000.01000000	10.43.8.64
Host-Adr.: IP AND not Netmask	00000000.00000000.00000000.00000011	3
Broadcast-Adr.: IP OR not Netmask	00001010.00101011.00001000.01001111	10.43.8.79
Adress-Bereich:	10.43.8.64 bis 10.43.8.79	
Adress-Bereich (Endgerät):	10.43.8.65 bis 10.43.8.78	
da die erste und letzte Adresse in einem Adressbereich jeweils die Netz- und Broadcast-Adresse ist und somit an kein Endgerät vergeben werden kann.		

1.5. Transmission Control Protocol (TCP)

Das *Transmission Control Protocol (TCP)* ist ein
•***zuverlässiges***,
•***verbindungsorientiertes***,
•***Bytestrom*** Protokoll.

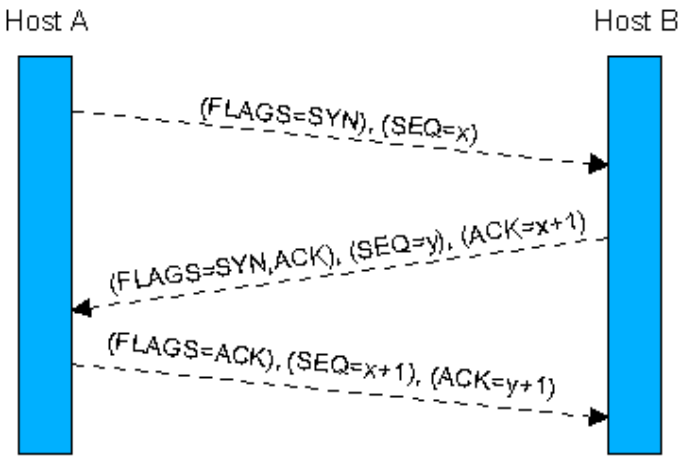
Die Hauptaufgabe von TCP besteht in der Bereitstellung eines **sicheren Transports** von Daten durch das Netzwerk.
TCP ist im RFC 793 definiert. Diese Definitionen wurden im Laufe der Zeit von Fehlern und Inkonsistenzen befreit (RFC 1122) und um einige Anforderungen ergänzt (RFC 1323).

Zuverlässigkeit

Das Transmission Control Protocol stellt die ***Zuverlässigkeit*** der Datenübertragung mit einem Mechanismus, der als ***Positive Acknowledgement with Re-Transmission*** *(PAR)* bezeichnet wird, bereit. Dies bedeutet nichts anderes als das, dass das System, welches Daten sendet, die Übertragung der Daten solange wiederholt, bis vom Empfänger der Erhalt der Daten quittiert bzw. positiv bestätigt wird.
Die Dateneinheiten, die zwischen den sendenden und empfangenden TCP-Einheiten ausgetauscht werden, heißen ***Segmente***.
Ein TCP-Segment besteht aus einem mindestens 20 Byte großen Protokollheader und den zu übertragenden Daten. In jedem dieser Segmente ist eine **Prüfsumme** enthalten, anhand derer der Empfänger prüfen kann, ob die Daten fehlerfrei sind. Im Falle einer fehlerfreien Übertragung sendet der Empfänger eine Empfangsbestätigung an den Sender. Andernfalls wird das Datengramm verworfen und keine Empfangsbestätigung verschickt.
Ist nach einer bestimmten Zeitperiode (timeout-period) beim Sender keine Empfangsbestätigung eingetroffen, verschickt der Sender das betreffende Segment erneut.

verbindungsorientiert

TCP ist ein **verbindungsorientiertes** Protokoll. Verbindungen werden über ein *Dreiwege-Handshake (**three-way handshake**)* aufgebaut. Über das Dreiwege-Handshake werden Steuerinformationen ausgetauscht, die die logische *Ende-zu-Ende-Verbindung* etablieren. Zum Aufbau einer Verbindung sendet ein Host (Host 1) einem anderen Host (Host 2), mit dem er eine Verbindung aufbauen will, ein Segment, in dem das SYN-Flag gesetzt ist. Mit diesem Segment teilt Host 1 Host 2 mit, das der Aufbau einer Verbindung gewünscht wird. Die Sequenznummer des von Host 1 gesendeten Segments gibt Host 2 außerdem an, welche Sequenznummer Host 1 zur Datenübertragung verwendet. Sequenznummern sind notwendig, um sicherzustellen, daß die Daten vom Sender in der richtigen Reihenfolge beim Empfänger ankommen. Der empfangende Host 2 kann die Verbindung nun annehmen oder ablehnen. Nimmt er die Verbindung an, wird ein Bestätigungssegment gesendet. In diesem Segment sind das SYN-Bit und das ACK-Bit gesetzt. Im Feld für die Quittungsnummer bestätigt Host 2 die Sequenznummer von Host 1, dadurch, daß die um Eins erhöhte Sequenznummer von Host 1 gesendet wird. Die Sequenznummer des Bestätigungssegments von Host 2 an Host 1 informiert Host 1 darüber, mit welcher Sequenznummer beginnend Host 2 die Daten empfängt. Schlußendlich bestätigt Host 1 den Empfang des Bestätigungssegments von Host 2 mit einem Segment, in dem das ACK-Flag gesetzt ist und die um Eins erhöhte Sequenznummer von Host 2 im Quittungsnummernfeld eingetragen ist. Mit diesem Segment können auch gleichzeitig die ersten Daten an Host 2 übertragen werden. Nach dem Austausch dieser Informationen hat Host 1 die Bestätigung, daß Host 2 bereit ist Daten zu empfangen. Die Datenübertragung kann nun stattfinden. Eine TCP-Verbindung besteht immer aus genau zwei Endpunkten (Punkt-zu-Punkt-Verbindung).



Dreiwege-Handshake (hier Verbindungsaufbau).

TCP nimmt *Datenströme* von Applikationen an und teilt diese in höchstens 64 KByte große Segmente auf (üblich sind ungefähr 1.500 Byte). Jedes dieser Segmente wird als IP-Datengramm verschickt.

Byte-Stream Semantic
Kommen IP-Datengramme mit TCP-Daten bei einer Maschine an, werden diese an TCP weitergeleitet und wieder zu den ursprünglichen Bytesströmen zusammengesetzt. Die IP-Schicht gibt allerdings keine Gewähr dafür, daß die Datengramme richtig zugestellt werden. Es ist deshalb, wie oben bereits gesagt, die Aufgabe von TCP für eine erneute Übertragung der Daten zu sorgen. Es ist aber auch möglich, daß die IP-Datengramme zwar korrekt ankommen, aber in der falschen Reihenfolge sind. In diesem Fall muß TCP dafür sorgen, daß die Daten wieder in die richtige Reihenfolge gebracht werden. Man nennte dieses Verhalten als **Byte-Stream Semantic**

Auch **TCP** verwaltet **Adressen**. Es handelt sich dabei um 16 Bit Werte, die Programme am jeweiligen Kommunikationsendpunkt spezifizieren. (s. /etc/services weiter unten). Man spricht von **Source-/Destination-Port im TCP-Header.**

1.6. User Datagram Protocol (UDP)

Das User Datagram Protocol (UDP) ist im RFC 768 definiert.

UDP ist ein
- unzuverlässiges,
- verbindungsloses Protokoll.

UDP bietet gegenüber TCP den Vorteil eines **geringen Protokoll-Overheads**. Viele Anwendungen, bei denen nur eine geringen Anzahl von Daten übertragen wird (z.B. Client/Server-Anwendungen, die auf der Grundlage einer Anfrage und einer Antwort laufen), verwenden UDP als Transportprotokoll, da unter Umständen der Aufwand zur Herstellung einer Verbindung und einer zuverlässigen Datenübermittlung größer ist als die wiederholte Übertragung der Daten.

Die Sender- und Empfänger-Portnummern erfüllen den gleichen Zweck wie beim Transmission Control Protocol. Sie identifizieren die Endpunkte der Quell- und Zielmaschine.

Das Protokoll beinhaltet keine Transportquittungen oder andere Mechanismen für die Bereitstellung einer zuverlässigen Ende-zu-Ende-Verbindung. Hierdurch wird UDP allerdings sehr **effizient** und eignet sich somit besonders für Anwendungen, bei denen es in erster Linie auf die Geschwindigkeit der Datenübertragung ankommt (z.B. verteilte Dateisysteme wie NFS).

Unterschiede TCP und UDP	
TCP (vgl. Telefonieren)	**UDP (vgl. Brief versenden)**
•Verbindungsaufbau	•Individuell adressiert
•Ankunft durch Empfänger quittiert	•Auslieferung nicht garantiert
•Reihenfolge garantiert	•Keine garantierte Reihenfolge
•Doppelte Pakete werden gelöscht	•Doppelte Pakete möglich
•Zeichenbyte Ströme	•Atomare messages

1.7. Einige Kommandos

netstat –i
netstat -a | more

ifconfig –a
ifconfig wdn0

ping hostname
ipconfig
ruptime
nmap

1.8. Das Konzept der Nameserver

Nameserver sind eine hierarchische, verteilte Datenbank, die zur Namensauflösung dient.

Die Administration dieser Datenbank ist ebenso verteilt, d.h. die Verwaltung bestimmter Domains ist bestimmten Administratoren zugeordnet. Diese richten sogenannte Nameserver ein, die Verbindungen zu übergeordneten Servern (Root Servern) haben. Bei der Namensauflösung, werden diese übergeordneten Nameserver kontaktiert, sofern nicht bereits lokal die Auflösung durchgeführt werden konnte. Ein extern aufgelöster Name wird lokal gespeichert, sodass bei einer erneuten Anfrage nicht wiederum der übergeordnete Server abgefragt werden muss.

Top-Level Domain

.com	.net	.org	.edu	....	.at	.de	.uk	.....

Second-Level Domain

.ibm.com .ac.at

...

Namensauflösung

Die IP Adresse erscheint in 3 Formen

- 32 Bit Wert
- Namensnotation (www.ripe.net)
- Nummernnotation (127.0.0.1)

Sogenannte Resolverprogramme (Z.B: nslookup) verwenden spezielle Funktionen (zB: **gethostbyname(), gethostbyaddr().** Beschreibung s. weiter unten), um eine Überführung von der einen Form in die andere zu bewerkstelligen. Diese Funktionen (gethostbyname(), ...) nutzten die jeweiligen Nameserver. D.h. die Resolverprogramme stellen Clientprogramme dar, die Nameserver zur Namensauflösung verwenden.

Wenn in der Datei /etc/resolv.conf ein Nameserver eingetragen ist, wird dieser verwendet, ansonsten wird als Datenbasis die /etc/hosts verwendet.

```
/etc/resolv.conf
nameserver 129.138.192.128
nameserver 129.138.192.129
…
```

```
/etc/hosts
127.0.0.1   localhost
192.138.192.128    nameserver1.domain.net nameserver1 ns1 name1
…
```

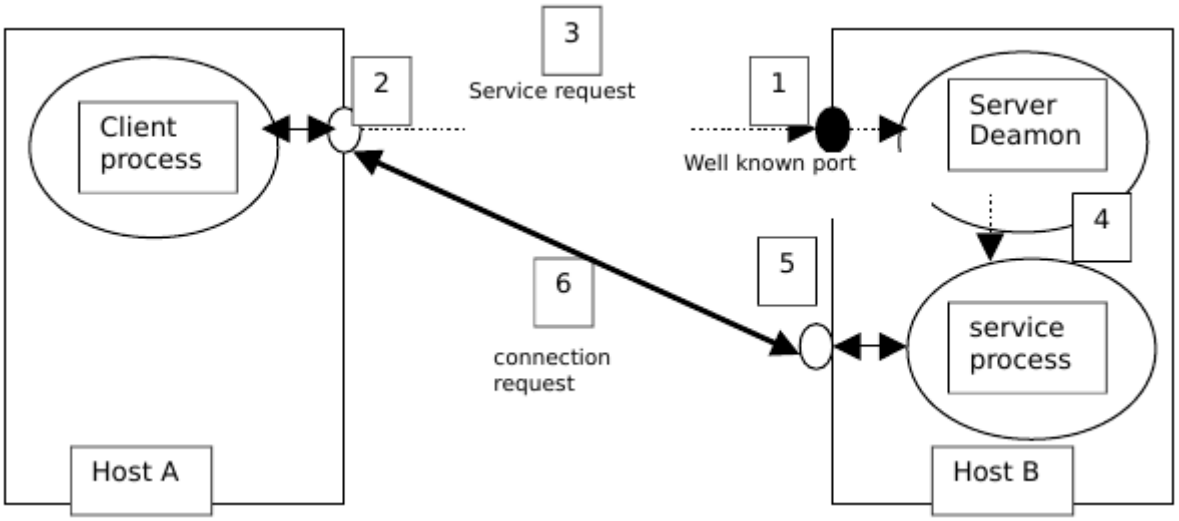
Weitere Informationen findet man u.a. auf http://wiki.ubuntuusers.de/Internet_und_Netzwerk

1.9. Client / Server Verbindungen

Dieses Modell ist dadurch gekennzeichnet, dass der Client einen sogenannten Request an der Server schickt. Dieser wird darauf hin aktiv und versucht die Anfrage zu beantworten. Dabei spielt es keine Rolle, ob Client und Server auf einem Computer oder auf verschiedenen Computern arbeiten.

Folgende Ereignisse finden in folgender Reihenfolge statt:

1. Server erzeugt und horcht an einem well-known socket
2. Client erzeugt einen Connection socket
3. Client sendet einen Request an den well-known socket
4. server akzeptiert den request und erzeugt einen service process
5. ein connection socket wird für den service process erzeugt
6. client und server kommunizieren mittels des connection sockets



2. Socket Programmierung

2.1. Die Klassen: Socket und ServerSocket

https://www.javatpoint.com/socket-programming

https://www.tutorialspoint.com/javaexamples/java_networking.htm

Als *Socket* bezeichnet man eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz. Sockets wurden Anfang der achtziger Jahre für die Programmiersprache C entwickelt und mit Berkeley UNIX 4.1/4.2 allgemein eingeführt. Das Übertragen von Daten über eine 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.

Während die Socket-Programmierung in C eine etwas mühsame Angelegenheit war, ist es in Java recht einfach geworden. Im wesentlichen sind dazu die beiden Klassen

☑ **Socket** und
☑ **ServerSocket**

erforderlich. Sie repräsentieren Sockets aus der Sicht einer Client- bzw. Server-Anwendung.

Ein Socket ist ein Kommunikationsendpunkt, der u.a. eine IP-Adresse und eine Port-Adresse besitzt.

2.1.1. Beispiel: Demo_InetAddress.java

Die **java.net.InetAddress** Klasse representiert eine IP-Adresse. Sie wird z.B. verwendet, um die IP-Adresse eines beliebigen Hosts zu erhalten. Dabei kann der Host in der Namensnotation oder der Nummernnotation angegeben werden.

```
/*
 * Demo_InetAddress.java
 */

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

public class Demo_InetAddress{
```

```
    public static void main(String[] args){
        try{
            InetAddress ip=InetAddress.getByName("www.opensource.org");
            System.out.println("Host Name: "+ip.getHostName());
            System.out.println("IP Address: "+ip.getHostAddress());

            System.out.println("-------------");

            InetAddress ip2= InetAddress.getByName("141.201.80.15");
            System.out.println("Host Name: "+ip2.getHostName());
            System.out.println("IP Address: "+ip2.getHostAddress());

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

Ausgabe:

```
$> java Demo_InetAddress.java
$> java Demo_InetAddress
Host Name: www.opensource.org
IP Address: 159.65.34.8
-------------
Host Name: uni-salzburg.at
IP Address: 141.201.80.15
```

Nachfolgend soll ein einfaches Ping-Pong-Programm die Verwendung der beiden Klassen demonstrieren. Der Client sendet PING und der Server antwortet mit PONG. Der Server horcht am Port 51234.

2.1.2. Beispiel: Demo_Client.java

```
/*
 * Demo_Client.java
 * ping-pong
 */
 import java.net.*;
 import java.io.*;

public class Demo_Client{
    public static void main(String[] args){
       try{
           // Verbindungsaufbau
           Socket client= new Socket("localhost", 51234);

           // Datentransfer
           InputStream in= client.getInputStream();
           OutputStream out= client.getOutputStream();

           PrintWriter sout= new PrintWriter(new OutputStreamWriter(out));
           BufferedReader sin= new BufferedReader(new InputStreamReader(in));
```

```
            sout.println("PING");
            sout.flush();

            String answer= sin.readLine();
            System.out.println("client received: " + answer);

            // Verbindungsabbau
            sin.close();
            sout.close();
            client.close();

        }catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

2.1.3. Beispiel: Demo_Server.java

```
/*
 * Demo_Server.java
 * ping-pong
 */
 import java.net.*;
 import java.io.*;

public class Demo_Server{
    public static void main(String[] args){
       try{
            ServerSocket server= new ServerSocket(51234);

            // Server läuft immer
            while(true){

                // wait for connection
                Socket connection= server.accept();
                InetAddress ip= connection.getInetAddress();
                System.out.println("server: connected to " + ip.getHostName());

                // Datentransfer
                InputStream in= connection.getInputStream();
                OutputStream out= connection.getOutputStream();

                PrintWriter sout= new PrintWriter(new OutputStreamWriter(out));
                BufferedReader sin= new BufferedReader(new InputStreamReader(in));

                String input= sin.readLine();
                System.out.println("server: received: " + input);

                sout.println("PONG");
                sout.flush();
```

```
                // Verbindungsabbau
                sin.close();
                sout.close();
                connection.close();
            }
        }catch (Exception ex){
            ex.printStackTrace();
        }
    }
}
```

Starte zuerst den Server in einem eigenen Terminal.
$> javac Demo_Server.java
$> java Demo_Server

Starte danach den Client in einem eigenen Terminal.
$> javac Demo_Client.java
$> java Demo_Client

Der Konstruktor erzeugt einen **ServerSocket für einen bestimmten Port.** 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.

2.2. Der Server im multithreading Betrieb

Bisher kann der Server zu einer Zeit immer nur einen Client bedienen. Wir wollen das auf folgende Weise besser machen:

Eigener Thread
Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung nicht mehr im Hauptprogramm.

1. Es wird ein **neuer Thread mit dem Connection-Socket als Argument** erzeugt.

2. Dann wird der Thread gestartet und er **erledigt die gesamte Kommunikation mit dem Client**.
3. Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet.

Das Hauptprogramm braucht sich also nur noch um den Verbindungsaufbau zu kümmern und ist von der eigentlichen Client-Kommunikation vollständig befreit.

Vorteile:

1. lange serverseitige Operationen werden im eigenen Thread abgearbeitet
2. der Server kann weitere Clients sofort bedienen.

Der Server wird dann folg. Aussehen haben:

```
/*
 * Demo_Server.java
 * ping-pong
 */
 import java.net.*;
 import java.io.*;

public class Demo_Server{
    public static void main(String[] args){
       try{
            ServerSocket server= new ServerSocket(51234);

            // Server läuft immer
            while(true){

                // wait for connection
                Socket connection= server.accept();

                ServiceThread service = new ServiceThread(connection);
                service.start();
            }

       }catch (Exception ex){
            ex.printStackTrace();
       }
    }
}
```

Es muss noch die Klasse ServiceThread erstellt werden.

```
/**
 * ServiceThread.java
 */
class ServiceThread extends Thread {
      private Socket socket;
      private BufferedReader bin;
      private PrintWriter bout;
```

```
      // Konstruktor
      public ServerThread(Socket socket){
            super();
            this.socket = socket;
            try {
              bin= new BufferedReader(
                  new InputStreamReader(socket.getInputStream()));
              bout= new PrintWriter(
                  new OutputStreamWriter(socket.getOutputStream()));
            } catch (IOException e) {e.printStackTrace();}
      }

      // run: Service
      public void run(){
           try {
               System.out.println(name + " Verbindung hergestellt...");
               InetAddress ip= socket.getInetAddress();
               System.out.println("server: connected to " + ip.getHostName());

               // Datentransfer
               String input= bin.readLine();
               System.out.println("server: received: " + input);

               bout.println("PONG");
               bout.flush();

               // Verbindungsabbau
               bin.close();
               bout.close();
               socket.close();
          } catch (IOException e) {System.err.println(e.toString());}
  } //run
}
```

2.2.1. Aufgabe: FileServer im multihreaded Betrieb

Erstellen Sie auf der Grundlage der vorhergehenden Aufgabe zwei Programme:

FileServer.java:

Wird zuerst gestartet.

Port: 51234. Diesen muss dann der Client als Kommunikationsport (s.u. PortNr) verwenden.

Nach erfolgreicher Initialisierung liest der Server Daten vom Client und öffnet die verlangte Datei. Liest die Datei zeilenweise aus und sendet die Daten zeilenweise an den Client.

FileClient.java:

Wird folgend aufgerufen: java Fileclient localhost 51234

Liest von der Tastatur einen Dateinamen ein und sendet an den Server folgenden String

GET dateiname

Dann liest der Client vom Socket die Datei ein und gibt diese am Bildschirm aus.

2.2.2. Aufgabe: LottoServer im multithreaded Betrieb (m)

LottoServer.java:
Bei Client-Anfrage ermittelt er 6 garantiert verschiedene Zufallszahlen zw. 1 und 45 und sendet diese an den Client.

LottoClient.java:
startet und fragt den Server nach den nächsten Lottozahlen. ;)
und gibt diese auf die Konsole aus.

Hinweis: Zufallszahlen

```
java.util.Random zufall= new  java.util.Random();
int zahl= zufall.nextInt(45) + 1;  // 1 .. 45
```

2.2.3. Aufgabe: SocketRate im multithreaded Betrieb (m)

Schreiben Sie ein Programm zum Zahlenraten
RateServer.java:
nach dem Verbindungsaufbau durch den Client denkt sich der Server eine Zahl zwischen 1 und 45 aus.

Dann wiederholt, bis zum Treffer...
1.Client
* liest vom User eine Zahl (1-45) ein und
* schickt diese an den Server

2.Server
* liest die geratene Zahl vom Client
* bewertet die geratene Zahl mit
  "zu tief", "getroffen" oder "zu hoch"
* sendet die Bewertung an den Client

3.Client
* liest die Bewertung und

* leitet daraus einen neuen Rateversuch ab, oder (weil getroffen) endet

2.2.4. Beispiel: Demo_EmailSend.java

https://eclipse-ee4j.github.io/javamail/

https://www.tutorialspoint.com/javamail_api/javamail_api_quick_guide.htm

```
/*
 * Demo_EmailSend.java
 *
 * for linux:
 * javac -cp jakarta.mail.jar:. Demo_EmailSend.java
 * java -cp jakarta.mail.jar:. Demo_EmailSend
 *
 * for windows:
 * javac -cp jakarta.mail.jar;. Demo_EmailSend.java
 * java -cp jakarta.mail.jar;. Demo_EmailSend
 *
 * download: JavaMail-API
 *
https://github.com/eclipse-ee4j/javamail/releases/download/1.6.3/jakarta.mail.j
ar
 *
 * see:
 * https://www.tutorialspoint.com/javamail_api/javamail_api_quick_guide.htm
 */

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.swing.JOptionPane;
import javax.swing.JPasswordField;

public class Demo_EmailSend{

    public static void main(String[] args) {

        String to = "TO@gmail.com";
        String from = "FROM@gmail.com";
        String host = "smtp.googlemail.com";

        String username = "USERNAME@gmail.com";


        JPasswordField passwordField = new JPasswordField(10);
        passwordField.setEchoChar('*');
        JOptionPane.showMessageDialog ( null, passwordField,
                    "EMAIL: enter password", JOptionPane.OK_OPTION );
        char[] chars = passwordField.getPassword();
```

```
        String password = new String(chars);

        Properties props = new Properties();
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.starttls.enable", "true");
        props.put("mail.smtp.host", host);
        props.put("mail.smtp.port", "25");

        // Get the Session object.
        Session session = Session.getInstance(props,
               new javax.mail.Authenticator() {
                 protected PasswordAuthentication getPasswordAuthentication() {
                   return new PasswordAuthentication(username, password);
               }
        });

        try {
            // Create a default MimeMessage object.
            Message message = new MimeMessage(session);

            // Set From: header field of the header.
            message.setFrom(new InternetAddress(from));

            // Set To: header field of the header.
            message.setRecipients(Message.RecipientType.TO,
                                  InternetAddress.parse(to));

            // Set Subject: header field
            message.setSubject("Testing Subject");

            // Now set the actual message
            message.setText("Hello, this is sample for to check send " +
                            "email using JavaMailAPI ");

            // Send message
            Transport.send(message);

            System.out.println("Sent message successfully....");

        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

2.2.5. Aufgabe: Java Web-Server (m)

1. Studieren Sie das HTTP-Protokoll auf Wikipedia.
2. Suchen Sie im Internet nach einer Implementation eines einfachen Web-Servers.

2.3. Http,URL, ...

Viele Dienste werden als sog. Web_Services, die mittels Http-Anfrage benutzt werden

eingesetzt. Wir wollen z.B. die Wetterdaten einer Stadt abfragen und in einem Diagramm im Internet den Temperaturverlauf anzeigen.

Hier aber einige nützliche Klassen.

2.3.1. Beispiel: Demo_URL.java

URL steht für Uniform Resource Locator, einer Position im Internet.

https://www.javatpoint.com/URL-class

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

public class Demo_URL{
    public static void main(String[] args){
        try{
            URL url=new URL("http://www.javatpoint.com:8888/java-tutorial");

            System.out.println("Protocol: "+url.getProtocol());
            System.out.println("Host Name: "+url.getHost());
            System.out.println("Port Number: "+url.getPort());
            System.out.println("File Name: "+url.getFile());

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

2.3.2. Beispiel: Demo_URLConnection.java

Eine Verbindung/Link von URL zu einer Applikation, um z.B. den Inhalt einer Webpage zu erhalten.

https://www.javatpoint.com/URLConnection-class

```
/*
 *Demo_URLConnection.java
 */

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

public class Demo_URLConnection{
    public static void main(String[] args){
        try{
            URL url;
            url= new URL("https://www.metaweather.com/api/location/551801/");
```

```
            URLConnection urlcon=url.openConnection();

            InputStream in=urlcon.getInputStream();
            BufferedReader bin= new BufferedReader(new InputStreamReader(in));

            String input;
            while((input=bin.readLine())!=null){
                System.out.println(input);
            }

            bin.close();

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

2.3.3. Beispiel: Demo_HttpURLConnection.java

HttpURLConnection ist eine Unterklasse von URLConnection und liefert spezielle Informationen (status, response code, ...) entsprechend dem HTTP-Protokoll.
https://www.javatpoint.com/java-http-url-connection

2.4. Projekt: www.metaweather.com und JSON

1. Lesen Sie von https://www.metaweather.com/api/location/551801/ die Wetterdaten (the_temp) von Wien und
2. geben Sie die Daten in lesbarer Form auf die Console aus:

Hinweis: json
Die erhaltenen Daten sind im sogenannten JSON-Format. Finden Sie einen Weg, wie Sie in Java auf diese Daten zugreifen können.
Hinweis:
print("Vienna-temperature: ... " response["consolidated_weather"][0]["the_temp"], " °C")

2.5. Projekt: www.zamg.ac.at und volkszaehler.org

Offene Daten Österreichs (**http://www.data.gv.at/**) , zB. die Wetterdaten der Zentralanstalt für Meteorologie und Geodynamik (http://www.zamg.ac.at/) , sollen regelmäßig online visuell als Diagramm angezeigt werden.
Die Daten werden stündlich vom Programm per Socket-API aus dem Internet geladen und in ein

Visualisierungsmodul per REST-API eingespielt.

Aufgabe:

1. Lesen Sie die Daten von www.zamg.ac.at/ogd und
2. spielen Sie die aktuelle Temperatur auf volkszaehler.org

Hinweis: www.zamg.ac.at

Sehen Sie hier eine telnet Session

```
▸ telnet www.zamg.ac.at 80
Trying 138.22.100.21...
Connected to www.zamg.ac.at.
Escape character is '^]'.
GET /ogd/ HTTP/1.1
Host: www.zamg.ac.at

HTTP/1.1 200 OK
Date: Sun, 05 May 2019 19:25:51 GMT
Server: Apache
Access-Control-Allow-Methods: GET
Access-Control-Allow-Origin: *
Content-Disposition: attachment; filename="tawes1h.csv"
Content-Length: 2035
Expires: Sat, 26 Jul 1997 05:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
X-GenHost: www13
Content-Type: text/csv

"Station";"Name";"Höhe m";"Datum";"Zeit";"T °C";"TP °C";"RF %";"WR °";"WG
km/h";"WSR °";"WSG km/h";"N l/m²";"LDred hPa";"LDstat hPa";"SO %"
11010;"Linz/Hörsching";298;"05-05-
2019";"21:00";5,2;2,7;84;240;13;;25,9;0;1017,2;979,1;0
11012;"Kremsmünster";383;"05-05-
2019";"21:00";4,1;1,6;85;238;16,2;240;24,8;0;1017,6;970,2;0
11022;"Retz";320;"05-05-
2019";"21:00";4,8;1,7;80;358;13,3;348;31;0;1015,5;976,6;0
11035;"Wien/Hohe Warte";203;"05-05-
2019";"21:00";4,8;2,5;86;338;18;338;51,1;0,8;1014,6;989,1;0
11036;"Wien/Schwechat";183;"05-05-
2019";"21:00";4,3;2,6;89;320;31,7;;51,8;0,9;1014,1;991,6;0
11101;"Bregenz";424;"05-05-
2019";"21:00";2,8;1,8;94;71;4,7;357;24,5;1,8;1019,7;966,4;0
11121;"Innsbruck";579;"05-05-
2019";"21:00";2,2;1,2;93;258;9;259;15,1;0,3;1019,1;948,4;0
11126;"Patscherkofel";2247;"05-05-2019";"21:00";-8,7;-
10,1;91;32;23,8;26;46,4;;;767,7;0
11130;"Kufstein";495;"05-05-
2019";"21:00";3,1;1,8;93;91;3,2;1;8,6;0,1;1018,9;959,2;0
11150;"Salzburg";430;"05-05-
2019";"21:00";2,9;1,5;91;200;5,4;;11,2;0,3;1018,9;964,7;0
```

```
11155;"Feuerkogel";1618;"05-05-2019";"21:00";-4,5;-
4,8;100;344;34,6;326;49;0,5;;832,1;0
11157;"Aigen im Ennstal";640;"05-05-
2019";"21:00";1,5;0,7;96;272;2,9;203;9,7;0,6;1018,1;939,6;0
11171;"Mariazell";866;"05-05-2019";"21:00";0,1;-
0,6;96;346;11,9;313;23,8;0,8;1016,5;913,5;0
11190;"Eisenstadt";184;"05-05-
2019";"21:00";4,5;2,8;88;315;24,1;336;51,5;2,1;1012,8;990,5;0
11204;"Lienz";659;"05-05-2019";"21:00";5,8;-
3,6;52;275;9,4;256;20,2;0;1011,9;933,1;0
11240;"Graz/Flughafen";340;"05-05-2019";"21:00";6,2;-
0,4;63;330;20,5;;40,7;0;1010,7;967,1;0
11244;"Bad Gleichenberg";280;"05-05-
2019";"21:00";6,4;1,3;70;54;11,9;41;24,5;0;1010,8;978,1;0
11265;"Villacher Alpe";2140;"05-05-2019";"21:00";-7,9;-
8,3;97;4;59,8;359;77,4;2,2;;773,1;0
11331;"Klagenfurt/Flughafen";447;"05-05-
2019";"21:00";3,5;1,9;90;270;1,8;208;14;0;1013,2;958,6;0
11343;"Sonnblick";3105;"05-05-2019";"21:00";-9,7;-
10,8;92;37;54,7;33;70,6;;;684,1;0
11389;"St. Pölten";270;"05-05-
2019";"21:00";4,4;3,5;95;49;2,9;258;13,3;0,5;1016,4;983,5;0
Connection closed by foreign host.
```

Hinweis: volkszaehler.org

Zum Einspielen auf volkszaehler.org verwenden Sie die folg. Informationen:

- http://demo.volkszaehler.org/frontend/

- Beispiel fü permalink:

http://demo.volkszaehler.org/frontend/?from=1396381607705&to=1396468007705&uuid%5B%5D=c1b717a0-b5b6-11e3-a58c-ef8aeeeb51dc&uuid%5B%5D=147578f0-b5b7-11e3-9c82-85cc4f734eef&uuid%5B%5D=2cc3ce40-b5b8-11e3-ae7a-c5e4bb8633c9&uuid%5B%5D=245dd830-b5b7-11e3-904a-73dde2ae6d47&uuid%5B%5D=98fd2600-b4bb-11e3-afa6-fd34350f1281

- Die einzelnen Kanäle

```
Salzburg:              98fd2600-b4bb-11e3-afa6-fd34350f1281
Linz/Hörsching:        2cc3ce40-b5b8-11e3-ae7a-c5e4bb8633c9
Retz:                  c803b980-b5b6-11e3-9f74-152449a53762
Kremsmünster:          0684e4a0-b5b8-11e3-a70f-edd61321fe02
Wien/Hohe Warte:       c34283a0-b5b6-11e3-a4ac-0b13b97cbac0
Wien/Schwechat:       d54447a0-b5b6-11e3-846a-b7388ef567a1
Bregenz:               c1b717a0-b5b6-11e3-a58c-ef8aeeeb51dc
Kufstein:              147578f0-b5b7-11e3-9c82-85cc4f734eef
Feuerkogel:            1293ac50-b5b8-11e3-8d38-ad9f6f0d5630
Innsbruck:             c256ebd0-b5b6-11e3-907e-c78475285974
Patscherkofel:         245dd830-b5b7-11e3-904a-73dde2ae6d47
```

- REST-API – Allgemein: http://wiki.volkszaehler.org/development/api/start?s[]=rest

- REST-API – Volkszaehler: http://wiki.volkszaehler.org/development/api/reference

- REST-API - Volkszaehler – Beispiele:

    - Linz/Hörsching habe 25 grad:
    http://demo.volkszaehler.org/middleware.php/data/2cc3ce40-b5b8-11e3-ae7a-c5e4bb8633c9.json?operation=add&value=25

    - Salzburg habe 25 Grad:
    http://demo.volkszaehler.org/middleware.php/data/98fd2600-b4bb-11e3-afa6-fd34350f1281.json?operation=add&value=25

- Nachsehen: http://demo.volkszaehler.org/frontend/?uuid=98fd2600-b4bb-11e3-afa6-fd34350f1281

REST-API - Volkszaehler - Beispiel mit Zugriff auf die aktuellen ONLINE Daten per shell-script:

Salzburg:

```
curl -s -L http://www.zamg.ac.at/ogd | grep Salzburg | awk -F";" '{print "curl -s -L \"http://demo.volkszaehler.org/middleware.php/data/98fd2600-b4bb-11e3-afa6-fd34350f1281.json?operation=add&value=" $6 "\""}'
```

2.6. Projekt: Ein einfaches Chat System

Wir wollen in der Folge ein Einfaches Chat System bauen.

Quellle: http://java.seite.net/chat/sockets.html

☑ Das Chat-System
Insgesamt besteht das komplette Chat-System aus 3 Klassen:
☐ Dem Chat-Server,
der auf eingehende Verbindungen von etwaigen Clients wartet,

☐ der Connection-Klasse,
die diese Verbindungen dann übernimmt und den Datenaustausch mit den Clients regelt
und schliesslich

- dem Client

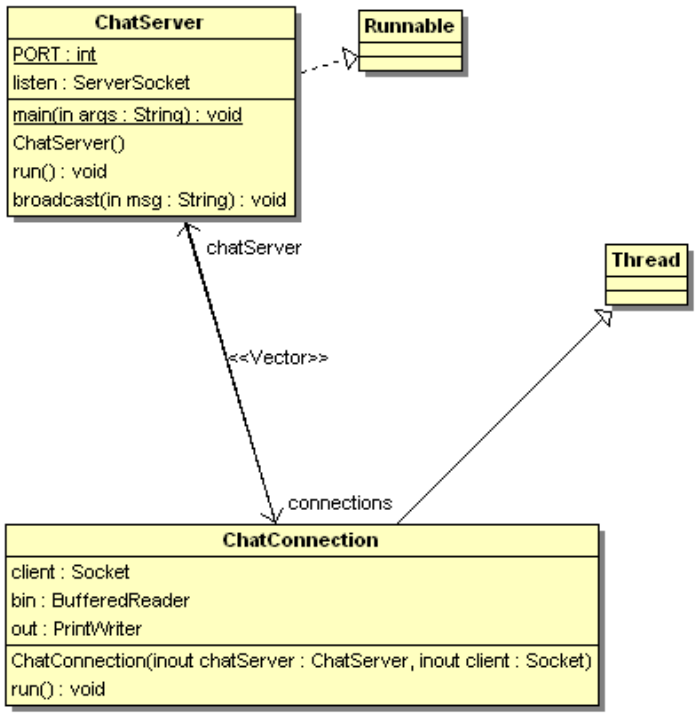
  selbst. Wir verwenden dazu 2 einfache Telnet-Programm.

  Cmd.exe

  telnet localhost 51234

  cmd.exe

  telnet localhost 51234



Die Klasse ChatServer hat u.a. folgenden Member:

```
Vector<ChatConnection> connections;
```

2.6.1. Der Chat-Server

Unsere Chat-Server-Klasse soll als eigenständiger Thread laufen und implementiert deshalb die Runnable-Schnittstelle.

```
public class ChatServer implements Runnable
{
        ...
}
```

Die Einstiegsmethode main() erzeugt einfach eine Instanz unserer Klasse, deren Konstruktor ausgeführt wird.

```
public static void main(String[] args) {
      // Serverobjekt erzeugen
      ChatServer chatServer= new ChatServer();

      // Thread starten
      Thread mainThread= new Thread(chatServer);
      mainThread.start();
}
```

Der Chat-Server ist wie in den obigen Beispielen ein einfacher Server, der einen ServerSocket zur Annahme von Requests der Clients einsetzt.

```
public static final int PORT = 51234;
private ServerSocket listen;
```

Zudem hält der Chat-Server einen Vector, der alle Verbindungen der Clients speichert. Hier sehen wir die Verwendung der Klasse ChatConnection (s.u.).

```
private Vector<ChatConnection> connections;
```

Die Klasse ChatConnection

Für jeden Verbindungswunsch eines Clients wird ein ChatConnection-Objekt erzeugt und in den Vector gelegt.

Das jeweilige ChatConnection-Objekt liest Daten von seinem Chat-Client und verwendet vom ChatServer die Methode broadcast(), um an alle anderen Clients die Daten zu senden.

Die Methode ChatServer.broadcast()

Die broadcast-Methode durchläuft alle offenen Verbindungen und übermittelt diesen die übergebene Nachricht msg. Da nur die chatserver-Klasse über alle Verbindungen auf dem laufenden ist, muss sie auch als Bestandteil der chatserver-Klasse implementiert sein - obwohl sie von dieser gar nicht benötigt wird, sondern von der nachfolgenden ChatConnection-Klasse.

```
public void broadcast(String msg)
{
        int i;
        ChatConnection you;
         System.out.println("[ChatServer] Broadcast: " + msg);
```

```
            for (i=0; i<connections.size(); i++)
            {
                    you = (ChatConnection) connections.elementAt(i);
                    you.out.println(msg);
                    you.out.flush();
            }
      }
```

☑ Zusammenfassung: Die Klasse ChatServer

```
/**
 * Übersetzen:
 * javac ChatServer.java ChatConnection.java
 *
 * Server starten:
 * java -cp . ChatServer
 *
 * ersten Client starten:
 * cmd.exe
 * telnet localhost 51234
 *
 * zweiten Client starten:
 * cmd.exe
 * telnet localhost 51234
 *
 */

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


public class ChatServer implements Runnable{
      public static final int PORT = 51234;
      private ServerSocket listen;
      private Vector<ChatConnection> connections;

      /**
       * Programmstart
       * erzeugt Serverobjekt und startet den ChatServer - Thread
       * @param args
       */
      public static void main(String[] args) {
            // Serverobjekt erzeugen
            ChatServer chatServer= new ChatServer();

            // Thread starten
            Thread mainThread= new Thread(chatServer);
            mainThread.start();
      }


      /**
       * Konstruktor, erzeugt den Listen-Socket
```

```
	 * und den Vector
	 * ------------------------------------------------------
	 */
	public ChatServer() {
		try {
			listen= new ServerSocket(PORT);
			System.out.println("ChatServer:51234 gestartet...");

			connections= new Vector<ChatConnection>();

		} catch (IOException e) {
			e.printStackTrace();
		}
	}


	/**
	 * run des ChatServers
	 * wartet auf einen Verbindungswunsch
	 * erzeugt und startet ein ChatConnection - Thread
	 * trgt in den Vector aller ChatConnection ein
	 * ------------------------------------------------------
	 */
	public void run(){
		try {
			while(true){
				Socket client= listen.accept();
				System.out.println("[ChatServer] neue Verbindung");


				// Connectionobjekt erzeugen und starten
				ChatConnection c = new ChatConnection(this,client);
				c.start();

				// in den Vector eintragen
				connections.addElement(c);

			}//while
		} catch (IOException e) {
			e.printStackTrace();
		}
	}//run



	/**
	 * schickt die msg an alle offenen ChatConnection
	 * @param msg
	 * ------------------------------------------------------
	 */
	public void broadcast(String msg)
	{
	        int i;
	        ChatConnection you;
			System.out.println("[ChatServer] Broadcast: " + msg);
```

```
                for (i=0; i<connections.size(); i++)
                {
                        you = (ChatConnection) connections.elementAt(i);
                        you.out.println(msg);
                        you.out.flush();
                }
        }

} //ChatServer
```

2.6.2. Die ChatConnection Klasse

Die Connection-Klasse wird vom Chat-Server erzeugt und behandelt die eigentlichen Verbindungen zu den Clients.

Für jede eingegangene Verbindung wird ein eigenes connection-Objekt erzeugt, das diese übernimmt. Es soll als eigenständiger Thread laufen und wird deshalb von der Thread-Klasse abgeleitet.

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

class ChatConnection extends Thread
{
        …
}
```

client übernimmt den Socket vom Chat-Server, mit dem dann die beiden Ein- und Ausgabestreams bin und bout verbunden werden. chatServer speichert den Zugriff auf das Chat-Server-Objekt.

```
        private ChatServer chatServer;
        private Socket client;
        private BufferedReader bin;
        public PrintWriter out;
```

☑ Zusammenfassung: Die Klasse ChatConnection

```
/**
 * javac ChatServer.java ChatConnection.java
 */
import java.net.*;
import java.io.*;

public class ChatConnection extends Thread{
```

```
	private ChatServer chatServer;
	private Socket client;
	private BufferedReader bin;
	public PrintWriter out;

	// Konstruktor
	public ChatConnection(ChatServer chatServer, Socket client) {
		try {
			this.client= client;
			this.chatServer= chatServer;
			this.bin= new BufferedReader(
				new InputStreamReader(client.getInputStream()));
			this.out= new PrintWriter(
				new OutputStreamWriter(client.getOutputStream()));
		} catch (IOException e) {
			e.printStackTrace();
		}
	}



	public void run(){
	 String line;

	 try{
		while(true){
			//lies
			line=bin.readLine();

			//schick an die anderen
			if(line!=null)
				chatServer.broadcast(line);
		}
	 }catch(IOException e){
		e.printStackTrace();
	 }
	}//run

}
```