Inhaltsverzeichnis

1. Grundlagen Netzwerkprogrammierung

1.1. Ziele

☑ Die Grundlagen der Netzwerprogrammierung: Protokolle, Adressen, Kommunikation

☑ Internetprotokolle verstehen und anwenden können

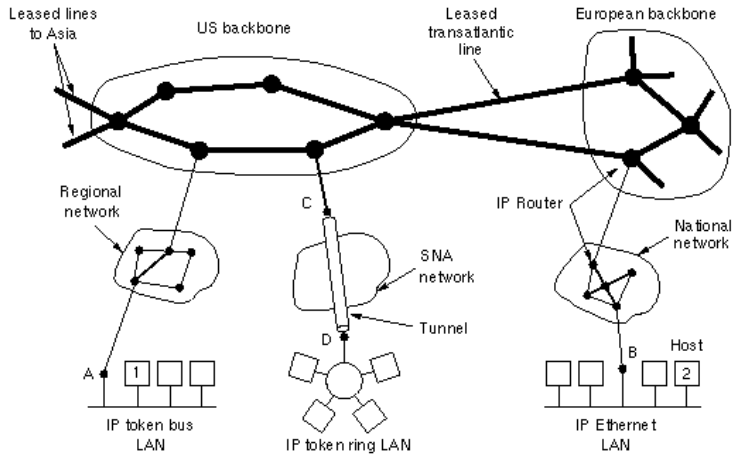
1.2. +Eine kurze Geschichte des Internet

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:

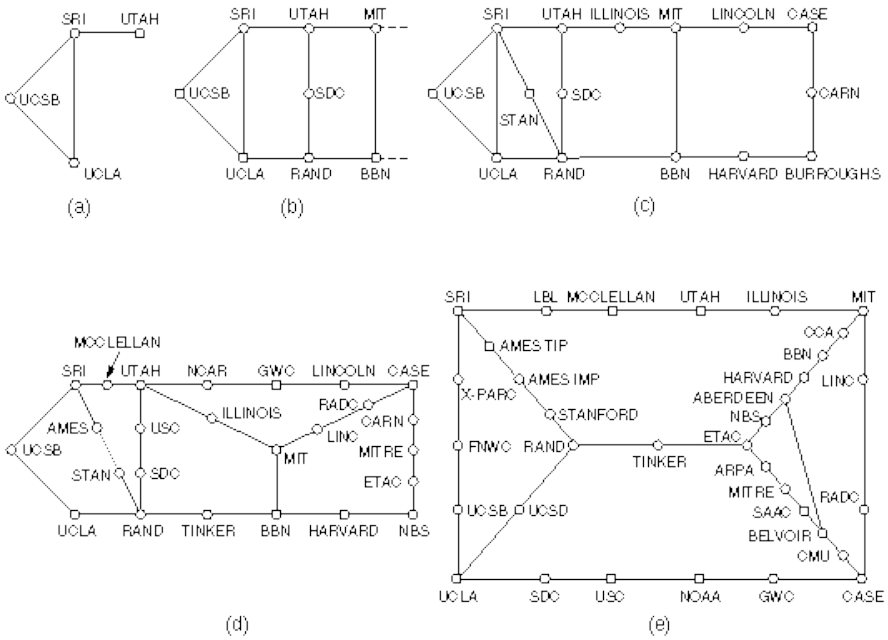
Aus diesem Grund beauftragte das US-Verteidigungsministerium die *Advanced Research Projects Agency (**ARPA**)* mit der **Entwicklung einer zuverlässigen Netztechnologie**. Die ARPA wurde 1957 als Reaktion auf den Start des Sputniks durch die UdSSR gegründet. Die ARPA hatte die Aufgabe Technologien zu 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 California, 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.

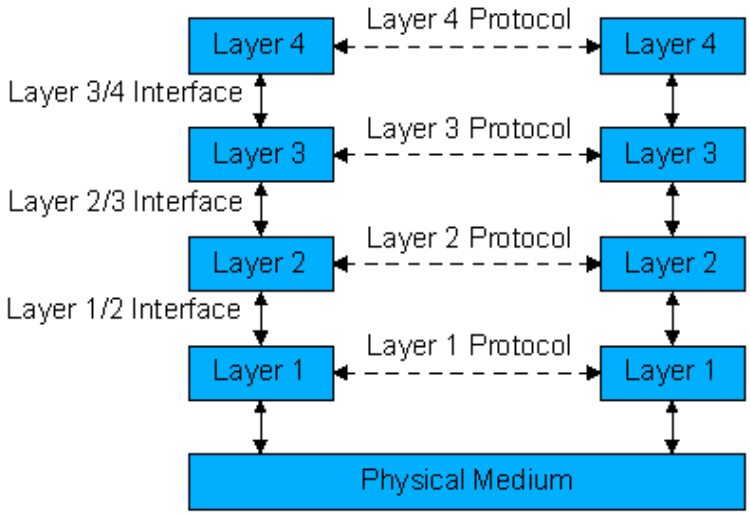
Protokolle sind Regeln, die den Nachrichtenaustausch - oder allgemeiner das Verhalten - zwischen (Kommunikations)Partnern regeln ("Protocols are formal rules of behaviour").

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 **Unverfälscht-heit** 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):

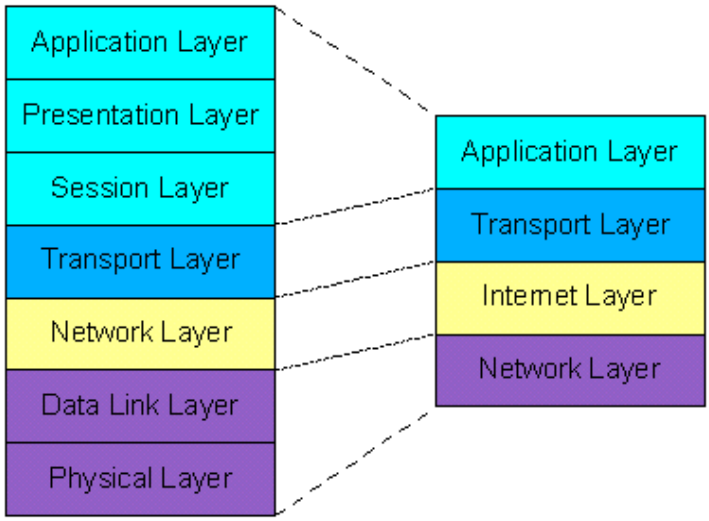
Die Aufgabe der Sicherungsschicht (Verbindungsschicht) ist die **gesicherte Übertragung von Daten**. Vom Sender werden hierzu die Daten in Rahmen (frames) aufgeteilt und sequentiell an den Empfänger gesendet. Vom Empfänger werden die empfangenen Daten durch Bestätigungsrahmen quittiert. Protokollbeispiele 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.

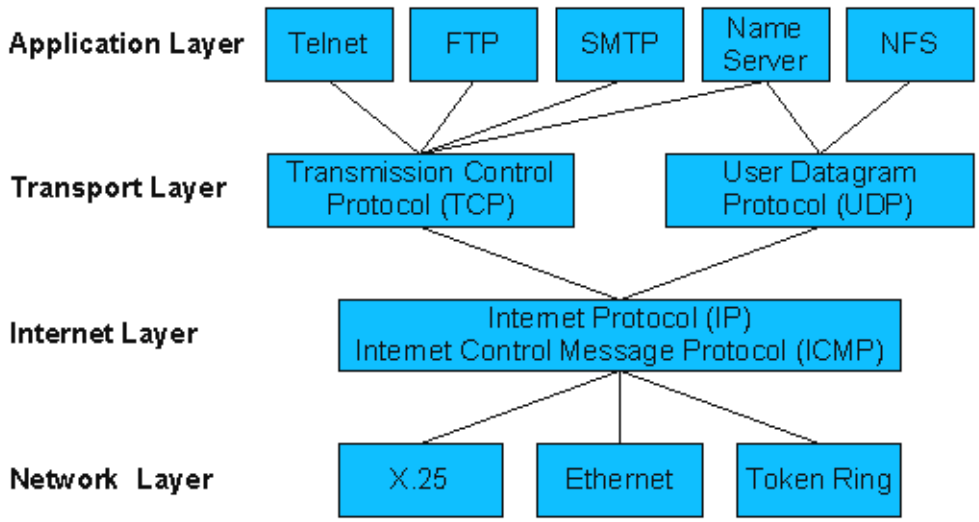
Applikationssschicht (application layer): Die Applikationsschicht (auch Verarbeitungsschicht genannt) umfaßt alle höherschichtigen Protokolle des TCP/IP-Modells. Zu den ersten Protokollen der Verarbeitungsschicht zählen **TELNET** (für virtuelle Terminals), **FTP** (Dateitransfer) und **SMTP** (zur Übertragung 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



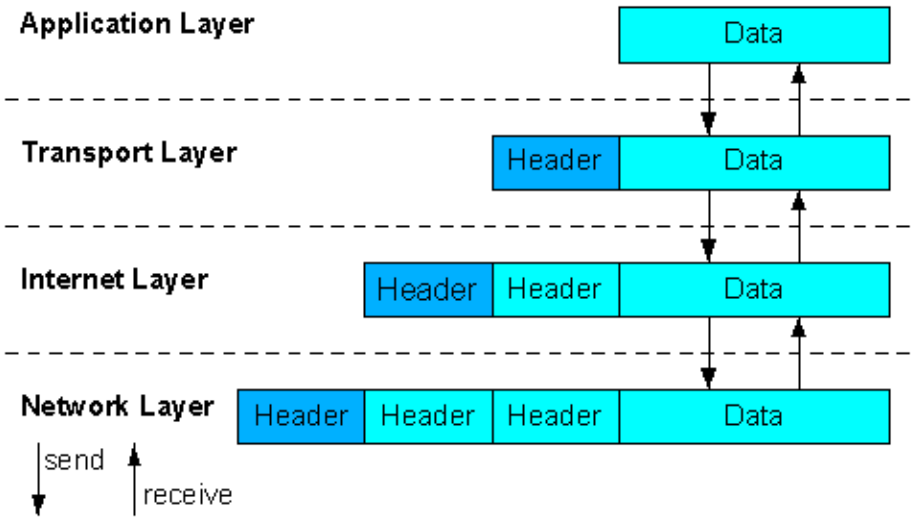
Die TCP/IP-Protokoll-Architektur.

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.

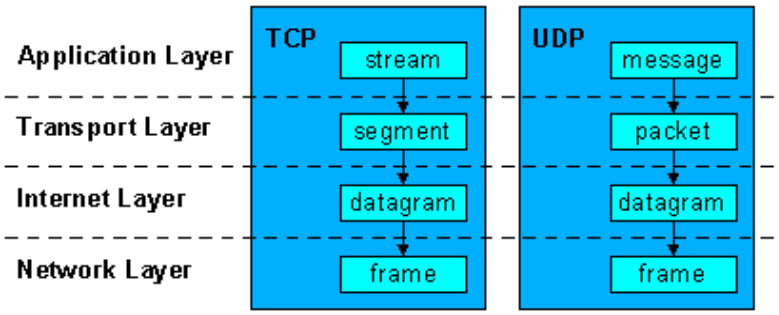
Diese Kontrollinformationen dienen der korrekten Zustellung der Daten. Das Zufügen von

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.

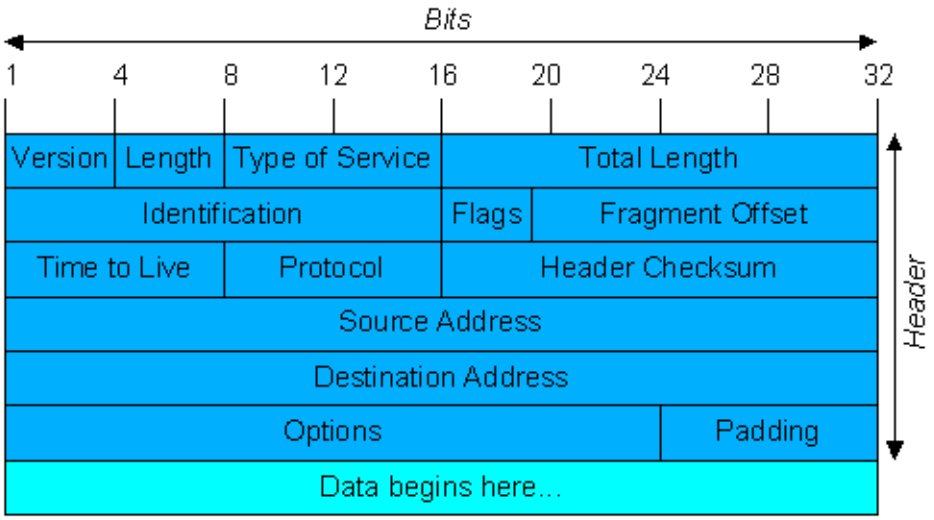
Die Funktionen von IP umfassen:

☑ 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**

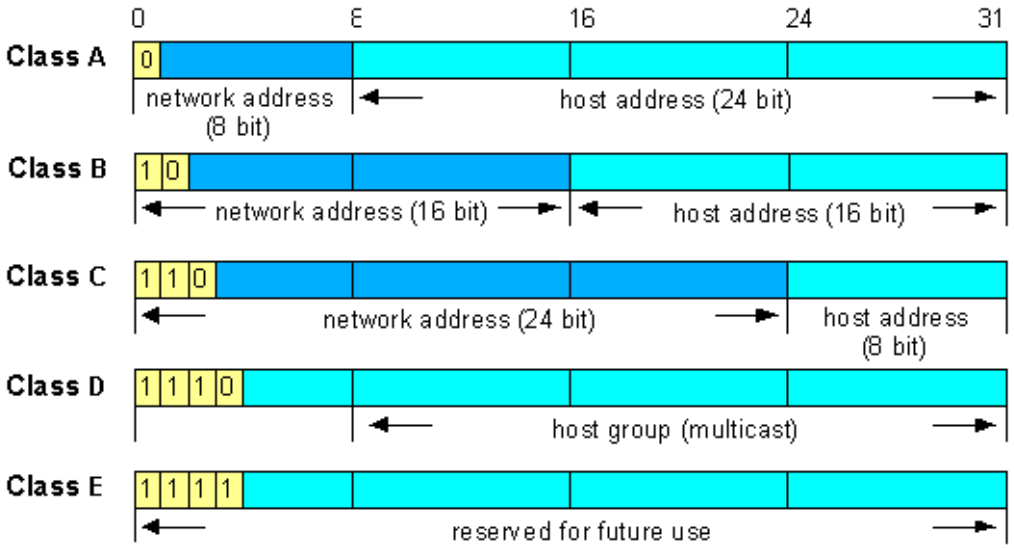
Jeder Host und Router im Internet hat eine 32-Bit lange IP-Adresse (IPv4).

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.

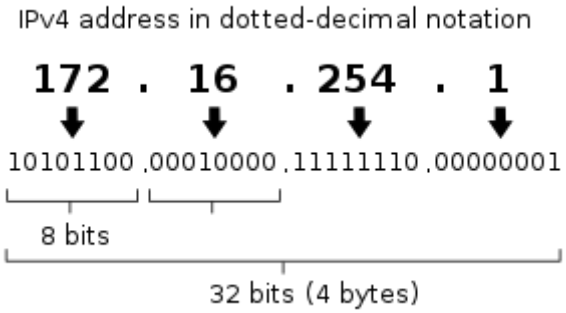
Wie die obere Abbildung zeigt, sind IP-Adressen in verschiedene Klassen, mit unterschiedlich langer Netzwerk- und Hostadresse, eingeteilt.

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, sog. *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 nach bestem Bemühen(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 (classless internet domain routing)

Siehe auch http://de.wikipedia.org/wiki/Subnetzmaske
- 1993 wurde das Class-Konzept durch CIDR (Classless internet domain routing) ersetzt.
- Ein IP-Suffix gibt die Bitanzahl für die Netzmaske an.

- **IPv4-Adresse:** 10.43.8.67/**28**
- **Netzmaske:** 11111111 . 11111111 . 11111111 . 1111**0000**
- **Netzmaske: 255.255.255.240**
- **Anzahl von Host-Adressen: (32-28= 4 Bits)**
- mit 4 Stellen im Dualsystem lassen sich 16 unterschiedliche Werte darstellen (0–15): **16 Adressen**
- aber: Broadcast- und Netzadresse können nicht für Host/Endgeräte verwendet werden, daher
- 14 IPv4-Adressen

Berechnung: CIDR für 10.43.8.67/28

Berechnung für 10.43.8.67/28	Duale Darstellung der Adressen	Dezimale Darstellung
	gegeben	
IP-Adresse	00001010.00101011.00001000.01000011	10.43.8.67/28
Netmask	11111111.11111111.11111111.1111**0000**	255.255.255.**240**
not Netmask	00000000.00000000.00000000.0000**1111**	
	gesucht	
Netz-Adresse = IP AND Netmask	00001010.00101011.00001000.01000000	10.43.8.64
Broadcast-Adresse = IP OR not Netmask	00001010.00101011.00001000.01001111	10.43.8.79
Host-Adresse = IP AND not Netmask	00000000.00000000.00000000.00000011	3
Adressen (gesamtes Netz) (inkl. Netz-Adr. , Broadcast-Adr.)	10.43.8.64 bis 10.43.8.79	
Adressen (nur Endgeräte/Hosts)	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).

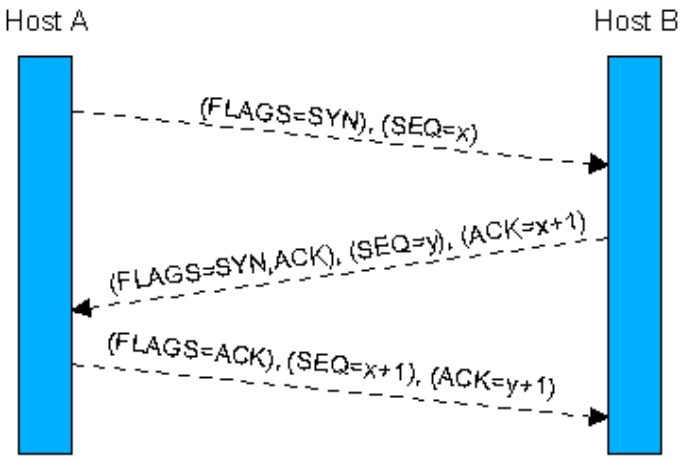
Zuverlässigkeit

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 Byteströmen 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

1.8.1. C-API (Funktionen u. Strukturen) zur Namensauflösung

Im folgenden wollen wir die Funktionen zur Namensauflösung besprechen:

```
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>

struct hostent* gethostbyname(const char *name);
struct hostent* gethostbyaddr(const char *addr, int len,int type);

unsigned long inet_addr(const char *cp);
char* inet_ntoa(const struct in_addr in);
```

Folgende Header-Dateien müssen benutzt werden:

```
UNIX:
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>     /* Internet addresses struct in_addr */
```

```
#include <arpa/inet.h>      /* inet (3N) function definitions    */
#include <netdb.h>          /* /etc/hosts data structure         */


Windows (dev-cpp):
#include <winsock2.h>
```

Folgende Strukturen sind notwendig:

```
Host entries are represented by the struct hostent structure defined in
<netdb.h>:

 struct hostent {
   char    *h_name;         // hostname
   char    **h_aliases;     // alias list
   int     h_addrtype;      // host address type (meist AF_INET)
   int     h_length;        // length of address (meist 4)
   char    **h_addr_list;   //list of addresses

#define h_addr  h_addr_list[0]  //address, for backward compatiblity
                                 //(32 Bit IP Adresse)
  };
```

wird benutzt von:

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len,int type);
```

repräsentiert die interne Datenstruktur der Datei /etc/hosts bzw. des Nameservers

Hier einige Beispiele zur Verwendung der einzelnen Funktionen:

```
Frage:
gegeben: DWST20.EDVZ.SBG.AC.AT:
gesucht: host entry
```

```
Antworten:

struct hostent *hp= gethostbyname("dwst20.edvz.sbg.ac.at");
      Internet-Adresse (32 Bit):     hp->h_addr;

      Internet-Hostname:             hp->h_name;

      Internet_Hostname-aliases:     while (*hp->h_aliases)
                                         printf("\n\t%s", *hp->h_aliases++);
```

```
Frage:
gegeben: 141.201.53.20:
gesucht: 32-bit Internet-Adresse
gesucht: host entry
```

```
u_long addr= inet_addr("141.201.53.20");  //* liefert 32-bit Internet-Adresse *


struct hostent *hp= gethostbyaddr((char*)&addr, 4, AF_INET);
```

```
Frage:
gegeben: 32-bit Internet Adresse:
gesucht: 141.201.53.20
```

```
char *inet_ntoa((struct in_addr) addr);   //* liefert "141.201.53.20" *


printf("\n%s", inet_ntoa((struct in_addr) addr);       //* liefert "141.201.53.20" *
```

Das folgende Programm soll als Zusammenfassung des bisher Gesagten dienen. Das Programm fragt die /etc/hosts ab oder, wenn /etc/resolv.conf einen nameserver eintrag enthält, diesen nameserver. Es handelt sich also um einen klassischen Nameresolver.

1.8.2. Beispiel: Name-resolver (t_host.c) für Linux

Datei: t_host.c (Linux)

```
/*
 t_host.c
 anton hofmann 6.6.97
 aufruf: t_host.exe www 141.201.53.20 info.sbg.ac.at localhost
 gcc -o t_host.exe t_host.c
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>     /* Internet addresses struct in_addr */
#include <arpa/inet.h>      /* inet (3N) function definitions    */
#include <netdb.h>          /* /etc/hosts data structure         */
```

```
int main(int argc, const char **argv){
  u_long addr;
  struct hostent *hp;
  char **p;
  int i;


  for (i=1; i< argc; i++){

    hp = gethostbyname(argv[i]);   //"a.b.c.d"
    if (hp == NULL){ //argument nicht von der form "a.b.c.d"

      /* inet_addr(): "1.2.3.4"  --> InternetAddress u_long */
      if ((int)(addr = inet_addr(argv[i])) == -1){
          printf("IP-address must be of the form 1.2.3.4\n"); exit (2);
      }

      hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
      if (hp == NULL) {
          printf("\nhost information for %s not found\n", argv[i]);
          exit (3);
      }
    }

    // Ausgabe
    for (p = hp->h_addr_list; *p != 0; p++) {
      struct in_addr in;   //struktur f. 32 Bit IP-Adresse
      char **q;

      memcpy(&in.s_addr, *p, sizeof (in.s_addr));

      /* inet_ntoa(): struct in_addr --> "1.2.3.4" */
      printf("%s\t%s", inet_ntoa(in), hp->h_name);

      for (q = hp->h_aliases; *q != 0; q++)
          printf(" %s", *q);

      putchar('\n');
    }
  }
}
```

1.8.3. Aufgabe: t_host.c

Bringen Sie obiges Programm zum Laufen und ermitteln Sie die Adressen

./t_host.exe www.ripe.net localhost 141.201.80.2

1.8.4. Aufgabe: sudo vi /etc/hosts

Tragen Sie in die Datei folgende Zeile ein:

192.168.2.104 BB beagleboard.lan

und

testen Sie danach mit

./t_host.exe BB

1.8.5. Aufgabe: t_minish.c mit lookup

Erweitern Sie t_minish.c mit folgender Funktionalität

Lookup hostname

d.h. wenn das Kommando lookup eingegeben wurde, soll analog zu obigen Programm t_host.c

eine Ausgabe erfolgen.

1.8.6. Beispiel: Name-resolver (t_host.c) für Windows (dev-cpp)

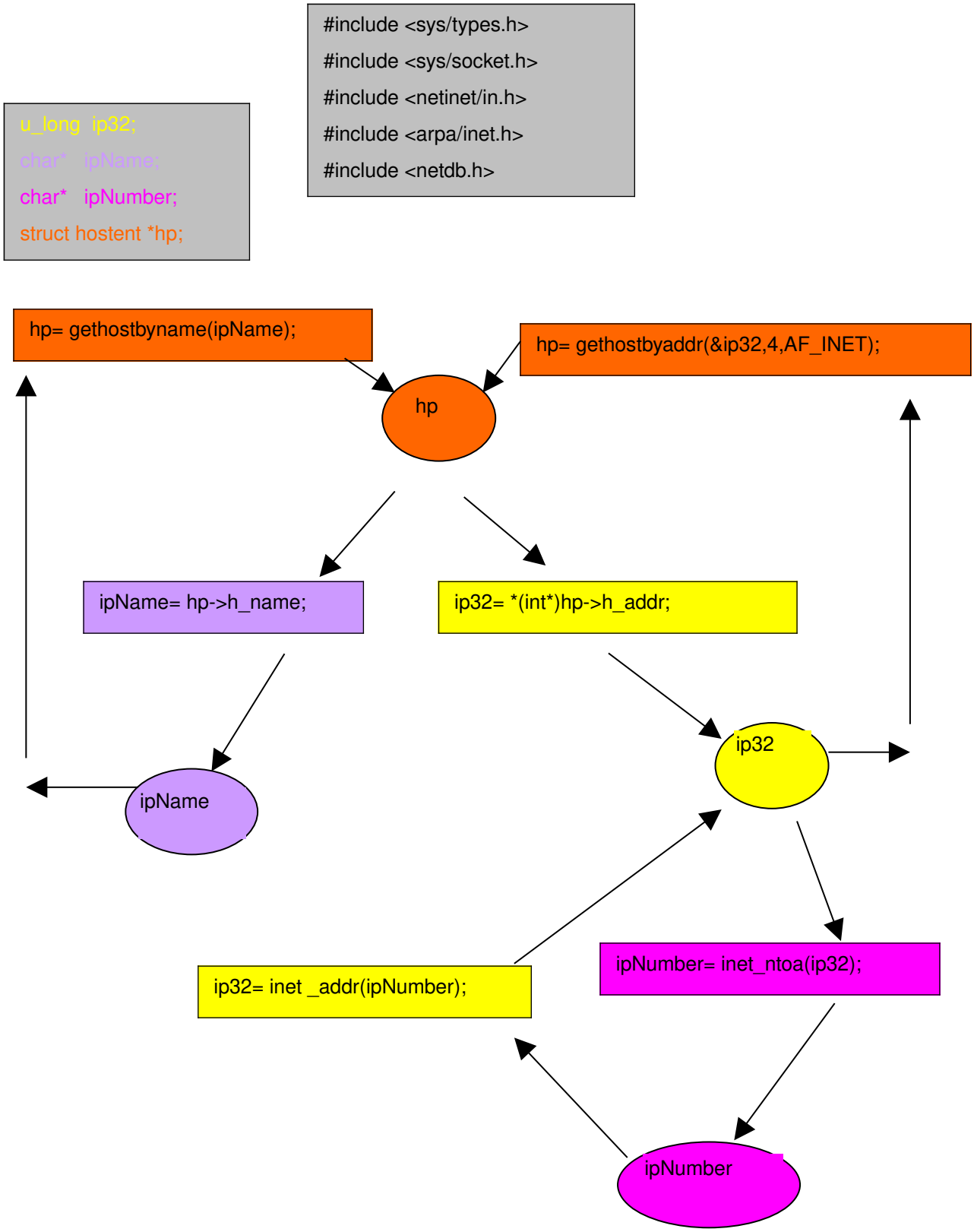
```
/*
 * t_host.c
 * a.hofmann
 * Aufruf: t_host.exe localhost www.ibm.com
 *
 * Dev-cpp:
 * Projekt-Optionen-Parameter-Linker    -lwsock32
 *
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <winsock2.h>

/* --------------------- Socket-DLL initialisieren */
void WSAInit(void)
{
int err;
     WORD wVersionReg = MAKEWORD(2,0);
     WSADATA wsaData;
```

```
     if (err = WSAStartup(wVersionReg, &wsaData) != 0)
     {
          printf("Fehler beim initialisieren der Winsock.dll!\n");
     }
}


int main(int argc, const char **argv) {
  u_long addr;
  struct hostent *hp;
  char **p;
  int i;


/* --------------------- Socket-DLL initialisieren */
  WSAInit();

  for (i=1; i< argc; i++){

    hp = gethostbyname(argv[i]);
    if (hp == NULL){

      /* inet_addr(): "a.b.c.a"  --> InternetAddress u_long */
      if ((int)(addr = inet_addr(argv[i])) == -1){
     printf("IP-address must be of the form a.b.c.d\n"); exit (2);
      }

      hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
      if (hp == NULL) {
     printf("\nhost information for %s not found\n", argv[i]); exit (3);
      }
    }


    for (p = hp->h_addr_list; *p != 0; p++) {
      struct in_addr in;
      char **q;

      memcpy(&in.s_addr, *p, sizeof (in.s_addr));
      /* inet_ntoa(): struct in_addr --> "a.b.c.d" */
      printf("%s\t%s", inet_ntoa(in), hp->h_name);

      for (q = hp->h_aliases; *q != 0; q++)
           printf(" %s", *q);

      putchar('\n');
    }
  }

  system("pause");
}
```

Im obigen Beispiel wird auch die Datenstruktur für die **IP-Adresse** benötigt. Hier nun der genaue Aufbau.

```
* Internet address defined in <netinet/in.h>
* New code should use only the s_addr field.

struct in_addr {
     union {
          struct { u_char s_b1, s_b2, s_b3, s_b4; } S_un_b;
          //ip als 4 mal 8 bit werte

          struct { u_short s_w1, s_w2; } S_un_w;
          //ip als 2 mal 16 bit werte

           u_long S_addr;
          //ip als 32 bit wert

        } S_un;
#define s_addr  S_un.S_addr          //ip als 32 bit wert
};
```

1.8.7. Zusammenfassung der Funktionen zur Namensauflösung

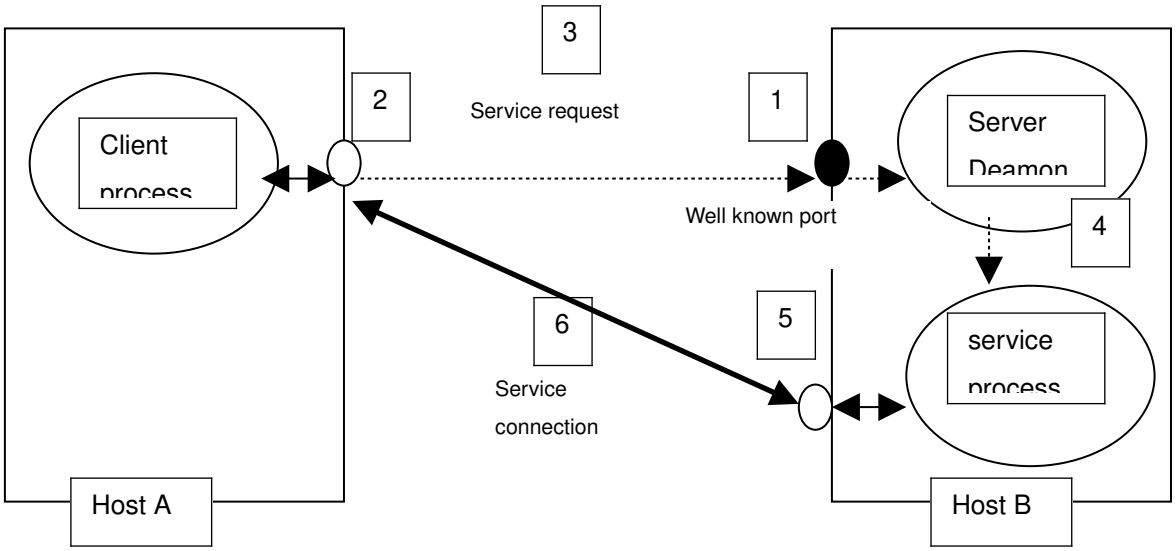


1.9. Client / Server Verbindungen

Dieses Modell ist dadurch gekennzeichnet, dass der Client einen sogenannten Request an der Server schickt. Der Server 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 mit C

Ein Socket (Steckdose) ist ein Kommunikationsendpunkt.

Mit Sockets lässt sich der Austausch von Nachrichten zwischen Prozessen

- **verbindungsorientiert** oder im
- **Datagramm-Stil** recht einfach programmieren.

Durch Angabe eines Sockettyps wird die Art der Kommunikation festgelegt:

SOCK_STREAM = verbindungsorientiert,

SOCK_DGRAM = Datagramm.

Die kommunizierenden Prozesse können auf demselben Rechner ablaufen (127.0.01) oder auf **unterschiedlichen** miteinander vernetzten Maschinen.

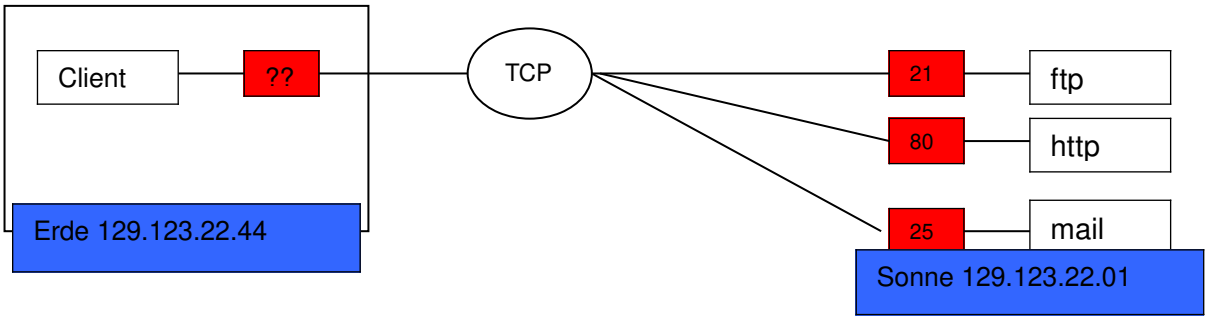
Ein Beispiel für eine Adressfamilie ist **AF_UNIX**. Sie defniert einen Adressierungsmechanismus für die rechnerinterne Kommunikation zwischen UNIX-Prozessen mit Hilfe des Socket-Mechanismus. Als Adressobjekte werden Pfade im Dateisystem verwendet, genau wie bei FIFOs.
Welche Adressfamilien unterstützt werden, hängt natürlich davon ab, welche Netzwerk-protokolle das Betriebssystem beherrscht. UNIX-Systeme werden zumindest AF_UNIX und **AF_INET** unterstützen.

Die nachfolgende Darstellung beschränkt sich auf die Adressfamilie **AF_INET** für die (rechner übergreifende) Kommunikation von Prozessen mit ARPA-Internet-Protokollen.

Die Verwendung anderer Adressfamilien erfolgt nach dem gleichen Schema und mit den gleichen Systemaufrufen, nur die Datenstrukturen für die Kommunikationsadressen sind unterschiedlich.

2.1. Internet-Sockets

Für verbindungorientierte Kommunikation wird als Basisprotokoll TCP (Transmission Control Protocol), für Datagramme UDP (User Datagram Protocol) verwendet. Die verwendeten Adressen bestehen aus der IP-Nummer des Rechners und einer Port-Nummer. Die IP-Nummer identifziert einen Rechner im Netzwerk. Die Port-Nummer ist eine innerhalb des Rechners eindeutige Zieladresse für Nachrichten.




```
Die Datei /etc/services enthält die Zuordnung: Port <-> Protokoll

echo       7/tcp
echo       7/udp
ftp        21/tcp
telnet     23/tcp
smtp       25/tcp          mail
…
```

2.1.1. Socket-Aufruf

Zwei Prozesse, die mittels Sockets kommunizieren wollen, müssen zunächst jeweils per socket-Aufruf eine Socket-Datenstruktur im Betriebssystem-Adressraum anlegen und dabei Sockettyp, Adressfamilie und Basisprotokoll festlegen. Der Aufruf erzeugt einen Deskriptor in der Deskriptortabelle des aufrufenden Prozesses, der für alle nachfolgenden Socket-Operationen benötigt wird.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int adressfamilie, int typ, int protokoll);
```

Durch Adressfamilie und Typ ist meist das Protokoll schon festgelegt, man kann als dritten Parameter in der Regel einfach 0 übergeben und erhält das Default-Protokoll (für Datagramme: UDP, verbindungsorientiert: TCP).
Aufrufbeispiel:

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

2.1.2. Socket Datenstruktur (AF_INET+IP+Port)

Der oben beschriebene socket-Aufruf spezifziert keine Absender- und Empfängeradressen für Nachrichten. Ein Prozess, der eine Nachricht empfangen will, muss sich mit einem bind-Aufruf vom Betriebssystem eine Kommunikationsadresse (Port) zuteilen lassen. Diese wird in der Socket-Datenstruktur abgespeichert.

Eine Socket Datenstruktur ist innerhalb von AF_INET vom Typ **struct sockaddr_in** und besteht aus folgenden Komponenten:

Adressfamilie	IP-Nummer	Port
sin family	sin addr	sin port
AF_INET	192.168.10.10	80

```
Beispiel:
struct sockaddr_in MySocket;

MySocket.sin_family = AF_INET;
MySocket.sin_addr.s_addr = inet_addr("192.168.10.10");
MySocket.sin_port= htons(80);
```

Der Typ von sin_addr ist **struct in_addr**, eine Struktur mit nur einer Komponente **s_addr**.

IP-Adressen sind genaugenommen nicht **Identifkation für den Rechner** sondern für eine Netzwerkschnittstelle eines Rechners. Manche Rechner haben mehrere Netzwerkschnittstellen. Mit INADDR_ANY kann man dem Betriebssystem mitteilen, dass über alle vorhandenen Schnittstellen empfangene Nachrichten an den Empfängerprozess weitergeleitet werden sollen.

Eine **Portnummer** dient rechnerintern als eindeutiges **Nachrichten-Postfach**. Server für **wohlbekannte** Dienste verwenden feste, für den Dienst reservierte Portnummern, z.B. 80 für einen WWW-Server (s. **/etc/services**). Die Nummern 1 bis 1023 sind dafür reserviert.

Ein Programm, das eine **beliebige** neue Portnummer benötigt, übergibt an das Betriebssystem beim bind-Aufruf eine **0 als Port**. Das Betriebssystem sucht daraufhin eine noch nicht vergebene Nummer und trägt sie in die Adressstruktur des Socket ein.

2.1.3. Datendarstellung im Netz und im Rechner

In heterogenen Netzen bestehen Probleme durch unterschiedliche Datenrepräsentation (z.B. **Byte-Reihenfolge** für Integer-Repräsentation).
Einerseits muss beim Datenaustausch zweier Rechner mit unterschiedlicher Repräsentation festgelegt werden, welches Format benutzt werden soll: das des Senders, das des Empfängers oder ein kanonisches Format.

Eine allgemeine Lösung dieses Problems sind Datendefnitionssprachen wie XDR oder IDL; hier wird darauf nicht weiter eingegangen.

Ein anderes Darstellungsproblem betrift jedoch die Adressen selbst:
Eine Nachricht durchläuft oft auf dem Weg zu ihrem Empfänger diverse Stationen (Gateway-Rechner, Router), die jeweils die Zieladresse lesen müssen, um die Nachricht weiterleiten zu können. Welches Darstellungsformat erwarten diese Stationen? Soll das erste oder das letzte Byte das höchstwertige sein ?

Dazu ist ein einheitliches Netzwerkformat defniert, das für die übertragung der Adressen verwendet wird. Mit 4 Konversionsroutinen können Portnummern (unsigned short integer) und IP-Nummern (unsigned long integer) vom Rechnerformat in das Netzformat und zurück konvertiert werden:

htons() host-to-net short

htonl() host-to-net long

ntohs() net-to-host short

ntohl() net-to-host long

Beim Binden von Adressen an Sockets ist in jedem Fall das Netzwerkformat zu verwenden.

Wenn also die Portnummer 80 eines WWW-Servers adressiert werden soll, darf in die Adressstruktur nicht einfach 80 eingesetzt werden, sondern **htons(80)** .Siehe auch Beispiel unten.

2.2. Verbindungsorientierte Kommunikation

Das Kommunikationsschema bei verbindungsorientierter Kommunikation, die innerhalb der AF_INET-Domain auf dem TCP-Protokoll basiert, ist asymetrisch: Ein Kommunikationspartner spielt den **Server**, **andere** sind dessen **Clients**. Die Nachrichten, die ein Client an den Server schickt, sind beliebig, wir bezeichnen Sie aber im folgenden als Aufträge.

2.2.1. Verbindungsaufbau bei Client/Server

Der **Client**	Der **Server**
	• **socket** Socket erzeugen
	• **bind** Server-Adr. an Socket binden
	• **listen** Auftrags-Queue initialisieren
• **socket** Socket erzeugen	
• **connect** Vbdg zum Server	• **accept** Auf Auftrag warten
• **write**	• **read**
• **read**	• **write**
• **close**	• **close**

Das Concurrent Server- Prinzip

Typischerweise wird der Server als concurrent server agieren: Ein Thread des Servers, nennen wir ihn Master-Thread, wartet auf Aufträge. Diese werden über den Auftragseingangssocket, der mit den socket- und bind-Aufrufen initialisiert wurde, übermittelt. Bei Eingang eines Auftrags erzeugt der Master-Thread einen neuen Slave-Thread, der den Auftrag bearbeitet (klassischerweise ein Childprozess). Währenddessen kümmert sich der Master-Thread schon um den nächsten Auftrag.
Da der Auftragseingangs-Socket der Auftragsannahme dient, stellt sich die Frage, wie der Slave-Thread mit dem Client kommunizieren soll.

Die Antwort liefert die **accept**-Semantik: Der accept-Aufruf erzeugt für die Bearbeitung des eingegangenen Auftrags automatisch einen **weiteren Socket** und eine **neue Portnummer**. Er bindet den neuen Socket an die neue Portnummer.
Der Slave-Thread kann über diesen neuen Socket zur Auftragsabwicklung mit dem Client

kommunizieren, ohne mit dem gleichzeitig möglichen Auftragseingang von anderen Clients zu kollidieren.

Auf der Client-Seite kehrt der connect-Aufruf zurück, sobald beim Server der neue Kommunikationsport eingerichtet wurde. Der **connect**-Aufruf gibt dem Client nicht nur diesen Kommunikationsport des Servers bekannt, sondern erzeugt gleichzeitig auch beim **Client einen Port** und bindet diesen an dessen Socket. Daher ist ein bind-Aufruf beim Client überflüssig.

2.2.2. Datenaustausch (read, write)

Für den Datenaustausch nach erfolgtem Verbindungsaufbau stehen diverse Funktionen zur Verfügung. Im einfachsten Fall verwenden Client und Server **read** und **write** in Verbindung mit den Socket-Deskriptoren. Der Verbindungsabbau erfolgt mit **close**.

Für Sonderfunktionen lassen sich aber auch folgende Funktionen verwenden: **send, sendto, sendmsg, recv , recvfrom, recvmsg.**

int read (int sd, char* buf, int len);

```
int sd     socket
char* buf  Speicherbereich in den geschrieben werden soll
int len    Anzahl der zu lesenden Zeichen
Rückgabe   Anzahl der tatsächlich gelesenen Daten. -1 im Fehlerfall.
```

Achtung:

```
Wenn die übertragene Nachricht länger ist, als in len angegeben wurde,
bleiben die restlichen Daten im Nachrichtenspeicher erhalten (nur bei
SOCK_STREAM). D.h. bei erneutem Aufruf von read() werden die restlichen
Daten aus dem Nachrichtenspeicher gelesen. Dies bedeutet, dass man mit
folgender Anweisung einen Nachrichtenspeicher vollständig auslesen kann:

 do {
   anz= read(sd, buffer, 128);
   ...
}while (anz > 0);

Bei SOCK_DGRAM ist dies nicht der Fall, d.h. nach erfolgtem read()
werden evtl. verbleibende daten im Nachrichtenspeicher verworfen.
```

```
int write (int sd, char* buf, int len);

int sd     socket
char* buf  Speicherbereich der versendet weden soll
int len    Anzahl der zu verschickenden Zeichen
Rückgabe   Anzahl der tatsächlich versendeten Daten. -1 im Fehlerfall.
```

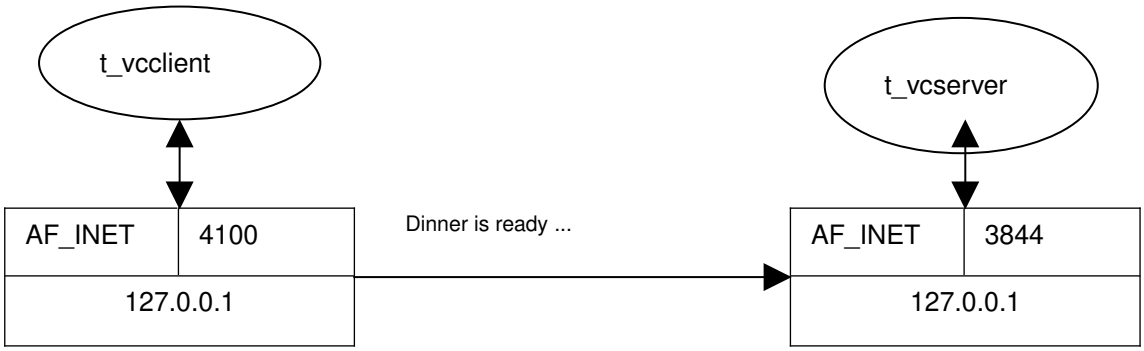
2.2.3. Beispiel: Verbindungsorientierte Kommunikation

Dieses Beispiel besteht aus dem Client-Programm (t_vcclient.c) und dem Server-Programm (t_vcserver.c). der Server startet und gibt am Bildschirm seine Portnummer aus. Diese Portnummer wird beim Aufruf des Client als Argument übergeben, sodass der Client zum Server eine Verbindung aufbauen kann.
Nach erfolgtem Verbindungsaufbau sendet der Client einen fixen String an den Server, denn dieser nach erhalt am Bildschirm ausgibt.
Beide Programme können am selben Host laufen, aber auch an verschiedenen. Die folgenden Grafiken zeigen dies:

Aufrufreihenfolge auf **einem** Host (127.0.0.1 ist localhost):

```
hofmann@localhost> ./t_vcserver &
 * prints out hostname and portname (bsp: 3844),
 * which should be used by (t_vcclient.exe)
```

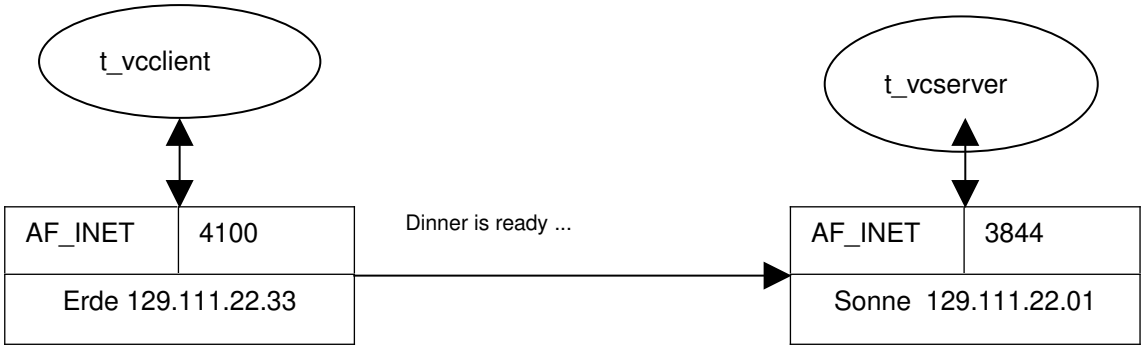
```
hofmann@localhost> ./t_vcclient.exe localhost 3844
```



Aufrufreihenfolge auf **verschiedenen** Hosts:

```
hofmann@sonne> ./t_vcserver &
 * prints out hostname and portname(bsp: 3844),
 * which should be used by (t_vcclient.exe)
```

```
hofmann@erde> ./t_vcclient.exe sonne 3844
```



2.2.4. Beispiel: t_vcclient.c (Linux)

```
/*
 * t_vcclient.c      TCP client (Virtual Circiut)
 * anton hofmann 9.6.97
 * gcc t_vcclient.c -o t_vcclient.exe
 * ./t_vcclient.exe hostname port#
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h> /* sockaddr_in socket-address internet-style  */
#include <netdb.h>      /* /etc/hosts table entries                   */



struct hostent * getHostEntry( const char*);
```

```
#define DATA      "Dinner is ready when the smoke alarm goes off.\n"

int main (int argc, char ** argv)
{

     int sd;                    /* socket descriptors */
     struct sockaddr_in    name; /* Internet socket name    */
     struct hostent* hp;        /* host entry        */

     /* 1. create a client socket */
     /* ------------------------- */
     sd = socket (AF_INET, SOCK_STREAM,0);
     if (sd < 0) {
          perror ("INET Domain Socket"); exit(1);
     }

     /* 2. init fields in an internet address structure */
     /* ----------------------------------------------- */
     name.sin_family = AF_INET;
     name.sin_port   = htons(atoi (argv[2]));   /* system binds port# */

     hp= getHostEntry(argv[1]);
     if (hp==NULL) {
          perror ("ERROR: getHostEntry()"); exit(1);
     }

     memcpy (&name.sin_addr.s_addr, hp->h_addr, hp->h_length);

     /* 3. connect to server */
     /* -------------------- */
     if ( connect (sd, (struct sockaddr*)&name, sizeof(name)) < 0) {
       perror ("INET Domain Connect"); exit(2);
     }

     write (sd, DATA , sizeof(DATA));

     close(sd);
     return 0;
}


/*
------------------------------------------------------------------------
*/
struct hostent * getHostEntry( const char* pHost) {
   struct hostent * hp;
   u_long addr;

   hp = gethostbyname(pHost);

   if (hp == NULL){
        /* inet_addr(): "a.b.c.a"  --> InternetAddress u_long */
        if ((int)(addr = inet_addr(pHost)) == -1){
```

```
            return NULL;
        }

        hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
        return hp;
    }
    return hp;
} //end getHostEntry
```

2.2.5. Beispiel: t_vcserver.c (LINUX)

```
/*
 * t_vcserver.c  TCP server (Virtual Circiut)
 * anton hofmann 9.6.97
 * ./vcserver &
 * prints out hostname and portname, which should be used by (t_vcclient.exe)
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

#include <netinet/in.h>      /* sockaddr_in socket-address internet-style   */
#include <netdb.h>           /* /etc/hosts table entries                    */
#include <arpa/inet.h>

int main (int argc, char ** argv)
{
     int rc;                 /* system call return code   */
     int sd, ld;             /* server/listen descriptors */
     int addrlen, nbytes;    /* sockaddr length; read nbytes    */

     struct sockaddr_in      name; /* Internet socket name*/
     struct sockaddr_in*     ptr;  /* pointer to get port number      */
     struct sockaddr         addr; /* generic socket name       */

     char        buf[256];   /* basic I/O buffer          */
     struct hostent*   hp;         /* host entry                */


     /* 1. create a listen socket */
     /* ------------------------- */
     ld = socket (AF_INET, SOCK_STREAM,0);
     if (ld < 0) {
          perror ("INET Domain Socket"); exit(1);}
```

```
/* 2. init fields in an internet address structure */
/* ----------------------------------------------- */
name.sin_family  = AF_INET;
name.sin_port    = htons(0);           /* system binds a free port# */
name.sin_addr.s_addr= INADDR_ANY;      /* wildcard accept*/


/* 3. bind the internet address to the internet socket    */
/* ------------------------------------------------------ */
if (bind (ld, (struct sockaddr*)&name, sizeof (name)) < 0) {
     perror ("INET Domain Bind"); exit(2);}


/* 4. find out the port number assigned to our socket     */
/* ------------------------------------------------------ */
addrlen= sizeof (addr);/* need int to store return value */
if ( (rc= getsockname (ld, &addr, &addrlen)) < 0){
     perror ("INET Domain getsockname"); exit(3);}


ptr= (struct sockaddr_in *) &addr;
printf ("\n\tSocket has port number: #%d\n", htons(ptr->sin_port));


/* 5. mark socket as passive listen socket */
/* --------------------------------------- */
if ( listen(ld, 5) < 0) {
     perror ("INET Domain Listen"); exit(4);}

while (1){
     /* 6. wait and answer */
     /* ------------------ */
     sd= accept (ld, &addr, &addrlen);
     if ( sd < 0){
          perror ("INET Domain Accept"); exit(5);}


     /* 7. find out who's calling us ..... */
     /* ---------------------------------- */
     rc = getpeername (sd, &addr, &addrlen);
     if ( rc < 0){
          perror ("INET Domain getpeername"); exit(6);}


/**/  /* announce the caller, just for our example   */
/**/  printf ("\n\tCalling socket from host %s\n",
                                  inet_ntoa (ptr->sin_addr));

/**/  printf ("\n\t              has port number #%d\n",
                                  htons(ptr->sin_port));
/**/
/**/  if ((hp= gethostbyaddr((char*)&ptr->sin_addr, 4, AF_INET))!=NULL){
/**/        printf ("\tFrom hostname: %s\n\tWith aliases: ",hp->h_name);
```

```
     /**/       while (*hp->h_aliases)
     /**/             printf("\n\t\t\t%s", *hp->h_aliases++);
     /**/       printf("\n\n");
     /**/  }
     /**/  else {
     /**/       perror ("\n\tgethostbyaddr() faild");
     /**/       printf ("\n\th_errno is %d\n\n", h_errno);
     /**/  }


           do
           {
             memset (buf, 0, sizeof(buf));
             nbytes= read (sd, buf, sizeof (buf));

             if ( nbytes < 0){
               perror ("INET Domain Read"); exit(7);}

             else if (nbytes == 0)
               printf ("\nClosing connection ....\n");
             else
               printf ("\nReceived: %s\n", buf);

           }while (nbytes != 0);

           close(sd);
     }//while
}//main
```

2.2.6. Beispiel: t_vcclient.c (WINDOWS)

```
/*
 * t_vcclient.c      TCP client (Virtual Circiut)
 * anton hofmann 9.6.97
 * ./t_vcclient.exe hostname port#
 * Dev-cpp:
 * Projekt-Optionen-Parameter-Linker    -lwsock32
 *
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <winsock2.h>

/* --------------------- Socket-DLL initialisieren */
void WSAInit(void)
{
int err;
     WORD wVersionReg = MAKEWORD(2,0);
     WSADATA wsaData;
     if (err = WSAStartup(wVersionReg, &wsaData) != 0)
     {
          printf("Fehler beim initialisieren der Winsock.dll!\n");
     }
}

struct hostent * getHostEntry( const char*);

#define DATA      "Dinner is ready when the smoke alarm goes off.\r\n"

int main (int argc, char ** argv)
{
     int sd;                    /* socket descriptors */
     struct sockaddr_in    name; /* Internet socket name    */
     struct hostent* hp;         /* host entry         */

/* --------------------- Socket-DLL initialisieren */
    WSAInit();
     /* 1. create a client socket */
     /* ------------------------- */
     sd = socket (AF_INET, SOCK_STREAM,0);
     if (sd < 0) {
          perror ("INET Domain Socket"); exit(1);
     }

     /* 2. init fields in an internet address structure */
     /* ----------------------------------------------- */
     name.sin_family = AF_INET;
```

```
     name.sin_port   = htons(atoi (argv[2]));   /* system binds port# */

     hp= getHostEntry(argv[1]);
     if (hp==NULL) {
          perror ("ERROR: getHostEntry()"); exit(1);
     }

     memcpy (&name.sin_addr.s_addr, hp->h_addr, hp->h_length);

     /* 3. connect to server */
     /* -------------------- */
     if ( connect (sd, (struct sockaddr*)&name, sizeof(name)) < 0) {
       perror ("INET Domain Connect"); exit(2);
     }

     send(sd, DATA , sizeof(DATA),0);

     closesocket(sd);

     WSACleanup();

}


/*
-------------------------------------------------------------------------
*/
struct hostent * getHostEntry( const char* pHost) {
   struct hostent * hp;
   u_long addr;

   hp = gethostbyname(pHost);

   if (hp == NULL){
        /* inet_addr(): "a.b.c.a"  --> InternetAddress u_long */
        if ((int)(addr = inet_addr(pHost)) == -1){
           return NULL;
        }

        hp = gethostbyaddr((char *)&addr, sizeof (addr), AF_INET);
        return hp;
   }
   return hp;
} //end getHostEntry
```

2.2.7. Beispiel: t_vcserver.c (WINDOWS)

```
/*
 * t_vcserver.c      TCP server (Virtual Circiut)
 * anton hofmann 9.6.97
 * ./t_vcserver.exe
```

```
 * prints out hostname and portname, which should be used by
(t_vcclient.exe)
 * Dev-cpp:
 * Projekt-Optionen-Parameter-Linker    -lwsock32
*/

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <winsock2.h>

/* --------------------- Socket-DLL initialisieren */
void WSAInit(void)
{
int err;
     WORD wVersionReg = MAKEWORD(2,0);
     WSADATA wsaData;
     if (err = WSAStartup(wVersionReg, &wsaData) != 0)
     {
          printf("Fehler beim initialisieren der Winsock.dll!\n");
     }
}

int main (int argc, char ** argv)
{
     int rc;              /* system call return code */
     int sd, ld;          /* server/listen descriptors    */
     int addrlen, nbytes; /* sockaddr length; read nbytes */

     struct sockaddr_in   name;/* Internet socket name    */
     struct sockaddr_in*  ptr; /* pointer to get port number   */
     struct sockaddr      addr;/* generic socket name          */

     char       buf[256]; /* basic I/O buffer        */
     struct hostent* hp;       /* host entry              */


/* --------------------- Socket-DLL initialisieren */
    WSAInit();
     /* 1. create a listen socket */
     /* ------------------------- */
     ld = socket (AF_INET, SOCK_STREAM,0);
     if (ld < 0) {
          perror ("INET Domain Socket"); exit(1);}


     /* 2. init fields in an internet address structure */
     /* ----------------------------------------------- */
     name.sin_family = AF_INET;
     name.sin_port   = htons(0);               /* system binds a free
port#*/
     name.sin_addr.s_addr= INADDR_ANY;     /* wildcard accept        */
```

```
     /* 3. bind the internet address to the internet socket     */
     /* ------------------------------------------------------ */
     if (bind (ld, (struct sockaddr*)&name, sizeof (name)) < 0) {
          perror ("INET Domain Bind"); exit(2);}


     /* 4. find out the port number assigned to our socket */
     /* -------------------------------------------------- */
     addrlen= sizeof (addr);    /* need int to store return value */
     if ( (rc= getsockname (ld, &addr, &addrlen)) < 0){
          perror ("INET Domain getsockname"); exit(3);}


     ptr= (struct sockaddr_in *) &addr;
     printf ("\n\tSocket has port number: #%d\n", htons(ptr->sin_port));


     /* 5. mark socket as passive listen socket */
     /* --------------------------------------- */
     if ( listen(ld, 5) < 0) {
          perror ("INET Domain Listen"); exit(4);}

     while (1){
          /* 6. wait and answer */
          /* ------------------ */
          sd= accept (ld, &addr, &addrlen);
          if ( sd < 0){
               perror ("INET Domain Accept"); exit(5);}


          /* 7. find out who's calling us ..... */
          /* ---------------------------------- */
          rc = getpeername (sd, &addr, &addrlen);
          if ( rc < 0){
               perror ("INET Domain getpeername"); exit(6);}


     /**/ /* announce the caller, just for our example    */
     /**/ printf ("\n\tCalling socket from host %s\n",
                                        inet_ntoa (ptr->sin_addr));
     /**/ printf ("\n\t               has port number #%d\n",
                                        htons(ptr->sin_port));
     /**/
     /**/ if((hp= gethostbyaddr((char*)&ptr->sin_addr,4,AF_INET))!=NULL)
          {
     /**/      printf ("\tFrom hostname: %s\n\tWith aliases: ",
                                        hp->h_name);
     /**/      while (*hp->h_aliases)
     /**/           printf("\n\t\t\t%s", *hp->h_aliases++);
     /**/      printf("\n\n");
     /**/ }
     /**/ else {
     /**/      perror ("\n\tgethostbyaddr() faild");
     /**/      printf ("\n\th_errno is %d\n\n", h_errno);
```

```
     /**/ }


          do
          {
            memset (buf, 0, sizeof(buf));

            nbytes= recv(sd, buf, sizeof(buf), 0);

            if ( nbytes < 0){
              perror ("INET Domain Read"); exit(7);}

            else if (nbytes == 0)
              printf ("\nClosing connection ....\n");
            else
              printf ("\nReceived: %s\n", buf);

          }while (nbytes != 0);

     closesocket(sd);

     }//while accept

    WSACleanup();

}// end main
```

2.2.8. Aufgabe: t_vcclient.c, t_vcserver.c

Bringen Sie obige Programme zum Laufen

2.3. Aufgaben: Socket Programmierung in C

2.3.1. Aufgabe: t_ping_client.c, t_pong_server.c

Studieren Sie die Programme t_vcclient.c/t_vcserver.c und erstellen Sie ein Verzeichnis socket/ping_pong, dass aus t_ping_client.c und t_pong_server.c besteht.

t_pong_server.c:
Wird zuerst gestartet mit ./t_pong_server.exe& .Gibt den Port am Bildschirm aus. Diesen muss dann der Client als Kommunikationsport (s.u. PortNr) verwenden.
Nach erfolgreicher Initialisierung liest der Server Daten vom Client und gibt diese dann sofort wieder an den

Client zurück.

t_ping_client.c:
Wird folgend aufgerufen: ./t_ping_client.exe localhost PortNr
Liest zeilenweise von der Tastatur ein, gibt diese Daten an den Server und liest vom Server dann die Daten und gibt diese am Bildschirm wieder aus.

2.3.2. Aufgabe: t_fileclient.c, t_fileserver.c

Erstellen Sie auf der Grundlage der vorhergehenden Aufgabe zwei Programme:

t_fileserver.c:
Wird zuerst gestartet mit ./t_fileserver.exe& .Gibt den Port am Bildschirm aus. 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.

t_fileclient.c:
Wird folgend aufgerufen: ./t_fileclient.exe localhost PortNr
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.

2.3.3. Aufgabe: popen.c

Ändern Sie das Programm t_fileserver so um, dass es statt des Ausliefern von Files Prozesse startet und den output dieser Prozesse an den Client schickt.
Der Benutzer des Client-Programmes soll Unix-Kommandos eingeben. (ps , ps aux, cat /etc/passwd, ...)

Hinweis:
popen()

2.3.4. Aufgabe: t_vcserver_port80.c

Schreiben Sie das Programm t_vcserver.c so um, dass es am Port 80 horcht. Als Client soll ein herkömmlicher Web-Browser verwendet werden: (http://localhost bzw.: http://localhost/einTest)

Hinweis: Beachte das http-Protokoll, den der WebBrowser verwendet dieses.
http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol

2.3.5. Aufgabe: tictactoed.c, vc_tictactoe.c

Schreiben Sie ein Client- und ein Serverprogramm zum Spiel TicTacToe. Verwendet werden soll der Port 71234.

Der Client baut eine Verbindung auf und spielt gegen den Server.

2.3.6. +Aufgabe: vc_minishd.c, vc_minish.c

Schreiben Sie das Programm t_minish.c in ein Client-Server Programm um. Als Standard Port soll 61234 verwendet werden.

t_minish_client.c:
Baut eine Verbindung zum Server auf. Benutzereingaben, die Adressbuch-Informationen abfragen werden an den Server geschickt und die erfolgte Antwort wird am Bildschirm ausgegeben. Die übrigen Befehle (Betriebssystembefehle, load und save) werden allerdings lokal beim Client ausgeführt.

t_minih_server.c:
Wird zuerst gestartet mit ./t_minish_server.exe& . Liest Daten vom jeweiligen Client , ermittelt die Antwort und gibt diese an den Client zurück.

Hinweis:
1.adb_open filename
2.adb_close

3.adb_list
4.adb_get_email nickname
5.adb_get_comment nickname
6.adb_get_record nickname

2.3.7. Aufgabe: minish_mit_nslookup.c

Erweitern Sie t_minish.c mit folgender Funktionalität
Lookup hostname
d.h. wenn das Kommando lookup eingegeben wurde, soll analog zu obigen Programm t_host.c
eine Ausgabe erfolgen.

2.3.8. Aufgabe: rated.c, rate.c

Schreiben Sie ein Programm zum Zahlenraten

Der Server:
nach dem Verbindungsaufbau durch den Client denkt sich der Server
eine Zahl zwischen 0 und 99 aus.

Dann wiederholt, bis zum Treffer...
1.Client
* liest vom User eine Zahl (0-99) 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

2.3.9. +Aufgabe: minish_mit_pop.c

Integrieren Sie folg. Programm t_pop3.c in t_minish.c, sodass zusätzlich ein POP3-Server abgefragt werden kann.

```
/* t_pop3.c    Anton Hofmann
   Demo: pop3-Klient
    gcc t_pop3.c -o t_pop3.exe -lsocket -lnsl

          t_pop3.exe pop3-server
          t_pop3.exe mx0.int.fh-sbg.ac.at
*/

#include <string.h>                    /* Required for strlen()*/
#include <sys/socket.h>
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>
#include <errno.h>
#include <stdio.h>

#define PROG_NAME "Quick POP3 Demo"

#define DEFAULT_PROTOCOL 0       /* No protocol specified, use default*/
#define DEFAULT_POP3_PORT 110    /* Well-know port assignment*/

char sScratchBuffer[256];        /* General purpose buffer for messages*/

char prompt[]= "\nPOP3 Demo:-> ";
char help[]= "\n\
          user your_mailbox_id\n \
          pass your_password\n \
          stat\n\
          list\n\
          retr 1\n\
```

```
        dele 1\n\
        quit \n \
        help ..... get this help-message\n";

/* ---------------------------- MAIL handling ----------------------------*/
void GetMail(int hSocket)
{
        char    sReceiveBuffer[4096];  /* Buffer for incoming mail messages*/
        int         iLength;          /* Length of data received*/

        printf (help);
        printf(prompt);

        printf ("Now I am reading the welcome message .....\n");
        /* 1. read welcome message */
        iLength = recv(hSocket, sReceiveBuffer, sizeof(sReceiveBuffer), 0);
        if (iLength == 0 || iLength == -1){
            perror("The recv() function returned a socket error.");
            exit(2);
          }
        printf("\nRECEIVE= %s",sReceiveBuffer);

   do{
        printf(prompt);
        fgets (sScratchBuffer, 256,stdin);

        if (strncmp(sScratchBuffer,"help", 4)==0) {
                printf(help);
        }
        else {
                if (send(hSocket, sScratchBuffer,
                    strlen(sScratchBuffer), 0) == -1){
                     perror("The send() function returned a socket error.");
                     exit(1);
                   }

                iLength = recv(hSocket, sReceiveBuffer,
```

```
                                sizeof(sReceiveBuffer), 0);
                    if (iLength == 0 || iLength == -1) {
                        perror("The recv() function returned a socket error.");
                        exit(2);
                      }

                    sReceiveBuffer[iLength] = '\0';

                    printf("\nSEND= %s\n",sScratchBuffer);
                    printf("\nRECEIVE= %s\n",sReceiveBuffer);
            }/*if else*/

            }while (strncmp (sScratchBuffer, "quit",4) != 0 );

return;
}

/* ------------------------- CONNECT ---------------------------------------*/
int ConnectPOPServerSocket(char* host)
{
struct hostent *lpHostEnt;        /* Internet host information structure */
struct sockaddr_in sockAddr;      /* Socket address structure*/
struct servent *lpServEnt;        /* Service information structure*/
int nConnect;                     /* Socket connection results*/
int hServerSocket;

/* Resolve the host name */
lpHostEnt = gethostbyname(host);
if (!lpHostEnt)   {
    perror("Could not get IP address!");
    exit(3);
   }
else    /* Create the socket   */
{
  hServerSocket = socket(AF_INET, SOCK_STREAM, 0);
  if (hServerSocket == -1)
    {perror("Invalid socket!!"); exit(4);}
```

```
  else    /* Configure the socket */
    {
     /* Define the socket address    */
     sockAddr.sin_family = AF_INET;
     sockAddr.sin_port = htons(DEFAULT_POP3_PORT);
     memcpy (&sockAddr.sin_addr.s_addr, lpHostEnt->h_addr, lpHostEnt->h_length);

     /* Connect the socket */
    nConnect=connect(hServerSocket,(struct sockaddr *)&sockAddr, sizeof(sockAddr));

     if( nConnect==-1)    {
            perror("Error connecting socket!!");
             hServerSocket = -1;
            }
     return(hServerSocket);
     }/*else*/
}/*else*/
}
/*===== MAIN ======================================*/
int main(int argc, char *argv[]){
int hSocket;         /* Socket handle for POP server connection */

if (argc != 2) {
            perror("\nUsage: t_pop3.exe pop3-server\n");
            exit(1);
}
hSocket = ConnectPOPServerSocket(argv[1]);  /* hostname*/

if (hSocket != -1){
    GetMail(hSocket);
    close(hSocket);
}
}
```

3. +Die Zukunft: Internet Protocol Version 6

Es wird wohl noch etwas länger dauern, bis dieser Abschnitt der Arbeit endlich fertig ist, da ich im Moment ziemlich viele andere Dinge zu tun habe. Für alle, die aber jetzt schon mehr wissen wollen wenigstens ein Link auf eine englischsprachige Beschreibung von IPv6: Hinden R.: IP Next Generation (IPng)..

3.1. Die Zukunft

Das rasche (exponentielle Wachstum) des Internet zwingt dazu, das Internet Protokoll in der Version 4 (IPv4) durch ein Nachfolgeprotokoll zu ersetzen.
Bis vor einiger Zeit wurde das Internet größtenteils nur von Universitäten, Regierungsbehörden (dies aber auch fast nur in den USA und hier vor allem vom Verteidigungsministerium) und einigen Firmen aus der Industrie genutzt. Seit der Einführung des *World Wide Web (WWW)* ist das Internet aber auch zunehmend für Privatpersonen, kleinere Firmen etc. interessant. Das Internet wandelt sich von einem "Spielplatz für Akademiker" zu einem weltweiten Informations- und Unterhaltungssystem. Mit der ständig steigenden Anzahl von Benutzern des Internet werden sich auch die Anforderungen an das Netz ändern bzw. haben sich bereits geändert. Genannt sei hier nur als Beispiel das angestrebte Zusammenwachsen der Computer-, Unterhaltungs- und Telekommunikationsbranchen. Den Anforderungen, die z.B. *Video-on-demand* stellt, ist das Internet bzw. das Internet Protokoll in der Version 4 nicht gewachsen.
Vinton Cerf (der 'Vater' des Internet) bezeichnet in einem Interview mit der Zeitschrift c't [Kr98] das Internet *"(...) als die wichtigste Infrastruktur für alle Arten von Kommunikation."*. Auf die Frage, wie man sich die neuen Kommunikationsdienste des Internet vorstellen könne, antwortete Cerf:

> *"Am spannendsten finde ich es, die ganzen Haushaltsgeräte ans Netz anzuschließen. Ich denke dabei nicht nur daran, daß der Kühlschrank sich in Zukunft mit der Heizung austauscht, ob es in der Küche zu warm ist. Stromgesellschaften könnten beispielsweise Geräte wie Geschirrspülmaschinen kontrollieren und ihnen Strom genau dann zur Verfügung stellen, wenn gerade keine Spitzennachfrage herrscht. Derartige Anwendungen hängen allerdings davon ab, daß sie zu einem erschwinglichen Preis angeboten werden. Das ist nicht unbedingt ferne Zukunftsmusik; die Programmierer müßten eigentlich nur damit anfangen, endlich Software für intelligente Netzwerkanwendungen zu schreiben. Und natürlich muß die Sicherheit derartiger Systeme garantiert sein. Schließlich möchte ich nicht, daß die Nachbarkinder mein Haus programmieren!"*

Auf die Internet Protokolle kommen in der nächsten Zeit also völlig neue Anforderungen zu. Wie versucht wird, diese Anforderungen zu erfüllen, wird in den nächsten Abschnitten beschrieben.

3.2. Classless InterDomain Routing - CIDR

Der Verknappung der Internet-Adressen durch die ständig steigende Benutzerzahl wird zunächst versucht, mit dem *Classless InterDomain Routing (CIDR)* entgegen zu wirken.
Durch die Vergabe von Internet-Adressen in Klassen (Netze der Klassen A,B,C,...) wird eine große Anzahl von Adressen verschwendet. Hierbei stellt sich vor allem die Klasse B als Problem dar. Viele Firmen nehmen ein Netz der Klasse B für sich in Anspruch, da ein Klasse A Netz mit bis zu 16 Mio. Hosts selbst für eine sehr große Firma überdimensioniert scheint. Tatsächlich ist aber oft auch ein Klasse B Netz zu groß. Für viele Firmen würde ein Netz der Klasse C ausreichen. Dies wurde aber nicht verlangt, da viele Unternehmen die Befürchtung hatten, daß ein Klasse C Netz mit seinen bis zu 254 möglichen Hosts nicht ausreichen würde. Ein größeres Hostfeld für Netze der Klasse C (z.B. 10 Bit, das entspricht 1022 Host pro Netz) hätte das Problem der knapper werdenden IP-Adressen vermutlich gemildert. Ein anderes Problem wäre dadurch allerdings entstanden: die Einträge der Routing-Tabellen wären explodiert.
Ein anderes Konzept ist das Classless InterDomain Routing (RFC 1519): die verbleibenden Netze der Klasse C werden in Blöcken variabler Größe zugewiesen. Werden beispielsweise 2000 Adressen benötigt, so können einfach acht aufeinanderfolgende Netze der Klasse C vergeben werden; das entspricht einem Block von 2048 Adressen. Zusätzlich werden die verbliebenen Klasse C Adressen restriktiver und strukturierter vergeben (RFC 1519). Die Welt ist dabei in vier Zonen, von denen jede einen Teil des verbliebenen Klasse C Adreßraums erhält, aufgeteilt:

194.0.0.0 - 195.255.255.255	Europa
198.0.0.0 - 199.255.255.255	Nordamerika
200.0.0.0 - 201.255.255.255	Mittel- und Südamerika
202.0.0.0 - 203.255.255.255	Asien und pazifischer Raum
204.0.0.0 - 223.255.255.255	Reserviert für zukünftige Nutzung

Jede der Zonen erhält dadurch in etwa 32 Millionen Adressen zugewiesen. Vorteil bei diesem Vorgehen ist, daß die 32 Millionen Adressen einer Region im Prinzip zu einem Eintrag in den Routing-Tabellen komprimiert worden sind. Der Vorteil der dadurch entsteht ist, daß z.B. jeder Router, der eine Adresse außerhalb seiner Region zugesandt bekommt...

3.3. Internet Protokoll Version 6 - IPv6 (IP Next Generation)

Der vorrangige Grund für eine Änderung des IP-Protokolls ist auf den begrenzten Adreßraum zurückzuführen. CIDR schafft hier zwar wieder etwas Luft, dennoch ist klar absehbar, daß auch diese Maßnahmen nicht ausreichen, um die Verknappung der Adressen für eine längere Zeit in den Griff zu bekommen.
Weitere Gründe für eine Änderung des IP-Protokolls sind die oben schon erwähnten neuen Anforderungen an das Internet. Diesen Anforderungen ist IP in der Version 4 nicht gewachsen. Die *IETF (Internet*

Engineering Task Force) begann deshalb 1990 mit der Arbeit an einer neuen Version von IP. Die wesentlichen Ziele des Projekts sind [Ta96]:

- Unterstützung von Milliarden von Hosts, auch bei ineffizienter Nutzung des Adreßraums
- Reduzierung des Umfangs der Routing-Tabellen
- Vereinfachung des Protokolls, damit Router Pakete schneller abwickeln können
- Höhere Sicherheit (Authentifikation und Datenschutz) als das heutige IP
- Mehr Gewicht auf Dienstarten, insbesondere für Echtzeitanwendungen
- Unterstützung von Multicasting durch die Möglichkeit, den Umfang zu definieren
- Möglichkeit für Hosts, ohne Adreßänderung auf Reise zu gehen
- Möglichkeit für das Protokoll, sich zukünftig weiterzuentwickeln
- Unterstützung der alten und neuen Protokolle in Koexistenz für Jahre

Im Dezember 1993 forderte die IETF mit RFC 1550 [IP: Next Generation (IPnG) White Paper Solicitation, Dec. 1993] die Internet-Gemeinde dazu auf, Vorschläge für ein neues Internet Protokoll zu machen. Auf die Anfrage wurde eine Vielzahl von Vorschlägen eingereicht. Diese reichten von nur geringfügigen Änderungen am bestehenden IPv4 bis zur vollständigen Ablösung durch ein neues Protokoll. Die drei besten Vorschläge wurden im *IEEE Network Magazine* veröffentlicht ([De93], [Fr93], [KF93]). Aus diesen Vorschlägen wurde von der IETF das *Simple Internet Protocol Plus (SIPP)* als Grundlage für die neue IP-Version ausgewählt. SIPP ist eine Kombination aus den Vorschlägen von Deering [De93] und Francis [Fr93].
Als die Entwickler mit den Arbeiten an der neuen Version des Internet Protokolls begannen, wurde natürlich auch ein Name für das Projekt bzw. das neue Protokoll benötigt. Wohl angeregt durch eine gleichnamige Fernsehsendung, wurde als Arbeitsname *IP - Next Generation (IPnG)* gewählt. Schließlich bekam das neue IP eine offizielle Versionsnummer zugewiesen: IP Version 6 oder kurz IPv6. Die Protokollnummer 5 (IPv5) wurde bereits für ein experimentelles Protokoll verwendet.
Die folgende Beschreibung von IPv6 orientiert sich an RFC 2460 [Internet Protocol, Version 6 (IPv6) Specification, Dec. 1998]. Dieses Dokument gibt den neuesten Stand der Spezifikation des Internet Protokolls in der Version 6 wieder. RFC 2460 enthält einige wesentliche Änderungen der Spezifikation gegenüber RFC 1883 [Internet Protocol, Version 6 (IPv6) Specification, Dec. 1995].

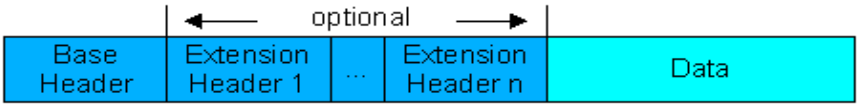
3.3.1. Die Merkmale von IPv6

Viele der als erfolgreich betrachteten Merkmale von IPv4 bleiben in IPv6 voll erhalten. Trotzdem ist IPv6 im allgemeinen nicht mit IPv4 kompatibel, wohl aber zu den weiteren Internet-Protokollen, insbesondere den Protokollen der Transportschicht (TCP, UDP); eventuell nach geringfügigen Modifikationen. Die Modifikationen betreffen im wesentlichen die erweiterte Adreßgröße (bisher 32 Bit auf nun 128 Bit).
Die wesentlichen Merkmale von IPv6 sind:

- **Adreßgröße:** Als wichtigstes Merkmal hat IPv6 gegenüber IPv4 größere Adressen. Statt bisher 32 Bit stehen nun 128 Bit für die Adressen bereit. Theoretisch lassen sich damit $2^{128} = 3.4*10^{38}$ Adressen vergeben.
- **Header-Format:** Der IPv6 (Basis)Header wurde vollständig geändert. Der Header enthält nur 7 statt bisher 13 Felder. Diese Änderung ermöglicht Routern, Pakete schneller zu verarbeiten. Im Gegensatz zu IPv4 gibt es bei IPv6 nicht mehr nur einen Header, sondern mehrere Header. Ein Datengramm besteht aus einem Basis-Header, sowie einem oder mehreren Zusatz-Headern, gefolgt von den Nutzdaten.
- **Erweiterte Unterstützung von Optionen und Erweiterungen:** Die Erweiterung der Optionen ist notwendig geworden, da einige, bei IPv4 notwendige Felder nun optional sind. Darüber hinaus unterscheidet sich auch die Art, wie die Optionen dargestellt werden. Für Router wird es damit einfacher, Optionen, die nicht für sie bestimmt sind, zu überspringen. Dies ermöglicht ebenfalls eine schnellere Verarbeitung von Paketen.
- **Dienstarten:** IPv6 legt mehr Gewicht auf die Unterstützung von Dienstarten. Damit kommt IPv6 den Forderungen nach einer verbesserten Unterstützung der Übertragung von Video- und Audiodaten entgegen. IPv6 bietet hierzu eine Option zur Echtzeitübertragung.
- **Sicherheit:** IPv6 beinhaltet nun im Protokoll selbst Mechanismen zur sicheren Datenübertragung. Wichtige neue Merkmale von IPv6 sind hier Authentifikation (authentication), Datenintegrität (data integrity) und Datenverlässlichkeit (data confidentiality).
- **Erweiterbarkeit:** IPv6 ist ein erweiterbares Protokoll. Bei der Spezifikation des Protokolls wurde nicht versucht alle potentiell möglichen Einsatzfelder für das Protokoll in die Spezifikation zu integrieren. Vielmehr bietet IPv6 die Möglichkeit über Erweiterungs-Header (s.u.) das Protokoll zu erweitern. Damit ist das Protokoll offen für zukünftige Verbesserungen.

3.3.2. Das IPv6 Datengrammformat

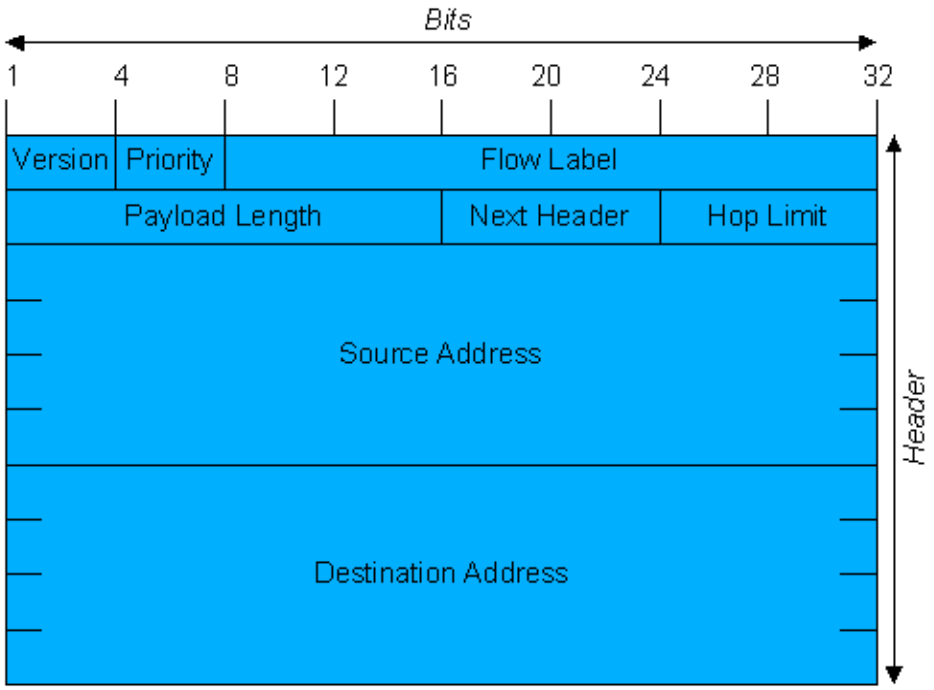
Ein IPv6-Datengramm besteht aus dem *Basis-Header* (s.u. Der IPv6-Basis-Header), gefolgt von den optionalen *Zusatz-Headern* (s.u. Erweiterungs-Header) und den Nutzdaten.



Allgemeine Form eines IPv6-Datengramms.

3.3.3. Der IPv6-Basis-Header

Der IPv6-Basis-Header ist doppelt so groß wie der IPv4-Header. Der IPv6-Basis-Header enthält weniger Felder als der IPv4-Header, dafür ist aber die Adreßgröße für die Quell- und Zieladresse von bisher 32-Bit auf nunmehr 128-Bit erweitert worden.



IPv6 Basis-Header.

Version:

Mit dem Feld *Version* können Router überprüfen, um welche Version des Protokolls es sich handelt. Für ein IPv6-Datengramm ist dieses Feld immer 6 und für ein IPv4-Datengramm dementsprechend immer 4. Mit diesem Feld ist es möglich für eine lange Zeit die unterschiedlichen Protokollversionen IPv4 und IPv6 nebeneinander zu verwenden. Über die Prüfung des Feldes Version können die Daten an das jeweils richtige "Verarbeitungsprogramm" weitergeleitet werden.

Priority:

Das Feld *Priority* (oder *Traffic Class*) ...

Flow Label

Das Feld *Flow Label*...

Payload Length

Das Feld *Payload Length (Nutzdatenlänge)* gibt an, wie viele Bytes dem IPv6-Basis-Header folgen, der IPv6-Basis-Header ist ausgeschlossen. Die Erweiterungs-Header werden bei der Berechnung der Nutzdatenlänge mit einbezogen. Das entsprechende Feld wird in der Protokollversion 4 mit *Total Length* bezeichnet. Allerdings bezieht IPv4 den 20 Byte großen Header auch mit in die Berechnung ein, wodurch die Bezeichnung "total length" gerechtfertigt ist.

Next Header

Das Feld *Next Header* gibt an, welcher Erweiterungs-Header dem IPv6-Basis-Header folgt. Jeder folgende Erweiterungs-Header beinhaltet ebenfalls ein Feld Next Header, das auf den nachfolgenden Header verweist. Ist dies der letzte zu IPv6 zugehörige Header, so gibt das Feld an, welches

Transportprotokoll (z.B. TCP oder UDP) folgt. Eine genauere Beschreibung des Konzepts mehrerer Header folgt im Abschnitt Erweiterungs-Header.

Hop Limit

Mit dem Feld *Hop Limit* wird festgelegt, wie lange ein Paket überleben darf. Der Wert des Feldes wird nach jeder Teilstrecke gesenkt. Ein Datengramm wird dann verworfen, wenn das Feld Hop Limit auf Null herunter gezählt ist, bevor das Datengramm sein Ziel erreicht hat. IPv4 verwendet hierzu das Feld *Time to Live*, welches die Zeit in Sekunden angibt, die ein Paket überleben darf. Allerdings wird dieses Feld von den meisten Routern nicht so interpretiert. In IPv6 wurde das Feld deshalb umbenannt, um die tatsächliche Nutzung wiederzugeben.

Source Address, Destination Address

Die beiden Felder *Quell-* und *Zieladresse* dienen zur Identifizierung des Senders und Empfängers eines IP-Datengramms. IPv6 verwendet zur Adressierung 4 mal so große Adressen wie IPv4: 128 Bit statt 32 Bit. Eine genaue Beschreibung der IPv6-Adressen folgt im Abschnitt IPv6-Adressierung.

Ein Vergleich des IPv4-Headers mit dem IPv6-Basis-Header veranschaulicht, welche Felder bei IPv6 weggelassen wurden:

- Das Feld *Length (Internet Header Length - IHL)* ist nicht mehr vorhanden, da der IPv6-Basis-Header eine feste Länge von 40 Byte hat. Bei IPv4 ist dieses Feld notwendig, da der Header aufgrund der Optionen eine variable Länge hat.
- Das Feld *Protocol* wird nicht mehr benötigt, da das Feld *Next Header* angibt, was nach dem letzten IP-Header folgt (z.B. TCP oder UDP).
- Alle Felder die bisher zur Fragmentierung eines IP-Datengramms benötigt wurden (*Identification, Flags, Fragment Offset*), sind im IPv6-Basis-Header nicht mehr vorhanden, da die Fragmentierung in IPv6 gegenüber IPv4 anders gehandhabt wird. Alle IPv6 kompatiblen Hosts und Router müssen Pakete mit einer Größe von 1280 Byte (RFC 1883 legte diese Größe noch auf 576 Byte fest) unterstützen. Durch diese Regel wird eine Fragmentierung im Prinzip nicht notwendig. Empfängt ein Router ein zu großes Paket, so führt er keine Fragmentierung mehr durch, sondern sendet eine Nachricht an den Absender des Pakets zurück. In dieser Nachricht wird der sendende Host angewiesen, alle weiteren Pakete zu diesem Ziel aufzuteilen. Das bedeutet, daß von den Hosts "erwartet" wird, daß sie von vornherein eine Datengrammgröße wählen, die keine Fragmentierung voraussetzt. Dadurch wird eine größere Effizienz bei der Übertragung erreicht, als wenn Pakete von Routern auf dem Weg fragmentiert werden müssen. Die Steuerung der Fragmentierung erfolgt bei IPv6 über den *Fragment Header*.
- Das Feld *Checksum* ist nicht mehr vorhanden, da die Berechnung der Prüfsumme sich nachteilig auf die Leistung der Datenübertragung ausgewirkt hat. Das entfernen der Prüfsumme aus dem Internet Protokoll hat zu heftigen Diskussionen geführt [Ta96]. Die eine Seite kritisierte heftig das entfernen der Prüfsumme, während die andere Seite argumentierte, daß Prüfsummen etwas sind, das auch von Anwendungen übernommen werden kann, sofern sich die Anwendung tatsächlich um Datenintegrität

kümmert. Ein weiteres Gegenargument war, daß eine Prüfsumme auf der Transportschicht bereits vorhanden ist, weshalb innerhalb der Vermittlungsschicht keine weitere Prüfsumme notwendig sei. Letztendlich fiel die Entscheidung, daß IPv6 keine Prüfsumme enthält.

3.3.4. Erweiterungs-Header

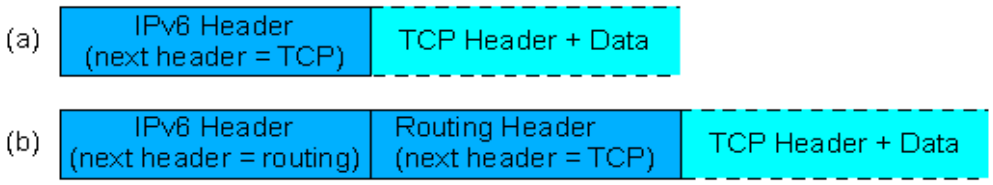
IPv6 nutzt das Konzept der Erweiterungs-Header, um a) eine effiziente Datenübertragung und b) eine Erweiterung des Protokolls zu ermöglichen. Der erste Punkt ist leicht ersichtlich: Der Basis-Header enthält nur Felder, die unbedingt für die Übermittlung eines Datengramms notwendig sind, erfordert die Übertragung weitere Optionen, so können diese über einen Erweiterungs-Header angegeben werden. IPv6 sieht vor, das einige Merkmale des Protokolls nur gezielt benutzt werden. Ein gutes Beispiel ist hier die Fragmentierung von Datengrammen. Obwohl viele IPv4-Datengramme nicht fragmentiert werden müssen, enthält der IPv4-Header Felder, für die Fragmentierung. IPv6 gliedert die Felder für die Fragmentierung in einen separaten Header aus, der wirklich nur dann verwendet werden muß, wenn das Datengramm tatsächlich fragmentiert werden muß. Ein weiterer wesentlicher Vorteil des Konzepts der Erweiterungs-Header ist, daß das Protokoll um neue Funktionen erweitert werden kann. Es genügt, für das Feld *Next Header* einen neuen Typ und ein neues Header-Format zu definieren. IPv4 erfordert hierzu eine vollständige Änderung des Headers.
Derzeit sind 6 Erweiterungs-Header definiert. Alle Erweiterungs-Header sind optional. Werden mehrere Erweiterungs-Header verwendet, so ist es erforderlich, sie in einer festen Reihenfolge anzugeben.

Header	Beschreibung
IPv6-Basis-Header	Zwingend erforderlicher IPv6-Basis-Header
Optionen für Teilstrecken (Hop-by-Hop Options Header)	Verschiedene Informationen für Router
Optionen für Ziele (Destination Options Header)	Zusätzliche Informationen für das Ziel
Routing (Routing Header)	Definition einer vollständigen oder teilweisen Route
Fragmentierung (Fragment Header)	Verwaltung von Datengrammfragmenten
Authentifikation (Authentication Header)	Echtheitsüberprüfung des Senders
Verschlüsselte Sicherheitsdaten (Encapsulating Security Payload Header)	Informationen über den verschlüsselten Inhalt
Optionen für Ziele (Destination Options Header)	Zusätzliche Informationen für das Ziel (für Optionen, die nur vom endgültigen Ziel des Paketes verarbeitet werden müssen)
Header der höheren Schichten	Header der höheren Protokollschichten (TCP,

(Upper Layer Header)	UDP, ...)

IPv6 Erweiterungs-Header.

Die ersten 5 Header sind in RFC 2460 (bzw. RFC 1883). Der Authentifikations-Header sowie der Header für Sicherheitsdaten werden in RFC 2402 (RFC 1826) und RFC 2406 (RFC 1827) beschrieben.



IPv6 Datengramme. (a) IPv6-Basis-Header und Nutzdaten; (b) IPv6-Basis-Header mit einem Zusatz-Header für Routing-Informationen, gefolgt von Nutzdaten.