

Programmieren für das Internet

Martin Gasbichler Eric Knauel

26. Oktober 2005

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

Studenten der Informatik dürfen dieses Dokument für ihre persönlichen Lehrzwecke verwenden.

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skript nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Martin Gasbichler, Eric Knauel 2002, 2003, 2005

Inhaltsverzeichnis

1	Motivation	5
2	Netzwerk-Grundlagen	7
2.1	Protokolle und das ISO/OSI-Schichtenmodell	7
2.2	Das Internet-Protokoll	9
2.2.1	Classless InterDomain Routing (CIDR)	10
2.2.2	IPv6 — das Internet-Protokoll der nächsten Generation	11
2.3	TCP und UDP	12
2.4	TCP-Programmierung mit Sockets	13
2.4.1	Funktionen zur Socket-Programmierung	13
2.4.2	Verwendung von Sockets	15
2.5	UDP-Programmierung mit Sockets	16

Kapitel 1

Motivation

Das rasante Wachstum des Internets und die steigende Nachfrage nach „Internet-Anwendungen“ haben in der letzten Dekade einen maßgeblichen Einfluß auf die Entwicklung von Software gehabt. Die Palette der Anwendungen, die heute über das Internet nutzbar sind erstreckt sich fast über die komplette Bandbreite der relevanten Software-Anwendungen. Ausschließlich über das Internet nutzbare Software findet eine immer höhere Akzeptanz bei den Benutzern — und ersetzt damit in einigen Bereichen traditionelle Software. E-Mail-Clients sind hier ein Beispiel. Im Jahr 1995 kam zum ersten Mal die Idee auf, einen E-Mail-Client in Form einer webbasierten Anwendung zu realisieren [Bro98] — heute für viele eine Selbstverständlichkeit.

Aus technischer Sicht hält die Entwicklung von Anwendungen für das Internet einige Herausforderungen bereit. Der Entwickler sieht sich mit Problemen aus verschiedenen Themenkomplexen gleichzeitig konfrontiert, die schon für sich allein genommen schwer in den Griff zu bekommen sind. Zu diesen Themenkomplexen zählen:

Netzwerkprogrammierung Ein Netzwerk verbindet mehrere Rechner. Das Internet ist ein Netzwerk, das nahezu alle Rechner verbindet. Eine Protokollhierarchie teilt die verschiedenen Probleme der Programmierung mit Netzwerken in mehrere separate Aufgaben und Protokolle: Finden des Kommunikationspartners, Ablauf der Kommunikation, Datendarstellung, Fehlerbehandlung. Netzwerkanwendungen, die diese Protokolle benutzen unterscheiden sich trotzdem noch stark von Programmen die ohne Kommunikation über das Netz auskommen.

Nebenläufige Programmierung Viele Internet-Anwendungen sind nach dem Client-Server-Paradigma programmiert: Ein einzelner Server kommuniziert hierbei parallel mit mehreren Clients. Internet-Anwendungen sind also nebenläufige Programme.

Interaktive GUI-Programmierung Webserver erzeugen den Inhalt von Webseiten heutzutage nur noch selten, indem sie den Inhalt einer Datei ausliefern. Viel häufiger wird die Webseite gemäß der Benutzereingaben und externen Quellen berechnet. Webbrowser stellen den Inhalt nicht mehr einfach nur dar, sondern können programmiert werden, um mit dem Benutzer

zu interagieren. Anwendungen mit Benutzerinteraktion bieten in der Regel einen wesentlich größeren Raum an möglichen Kontrollabläufen. Dem Benutzer bleibt diese Komplexität hinter einer einfach zu bedienenden graphischen Oberfläche verborgen, der Programmierer muß sich jedoch stets über alle möglichen Interaktionspfade im Klaren sein.

Programmieren in heterogener Umgebung Durch das Netz werden Komponenten verbunden, die nur wenig von einander wissen: Microsoft-Clients treffen auf Unix-Server, Scriptsprachen sind in C-Applikationen integriert, kommerzielle und Open-Source-Programme kommunizieren miteinander. Probleme entstehen, sobald Protokolle nicht eingehalten werden oder zu viel Spielraum bieten.

Die Verbindung dieser Bereiche zur „Programmierung für das Internet“ ergibt ein Gebiet, das vom Programmierer strukturiertes Vorgehen und Erfahrung erfordert.

Diese Vorlesung widmet sich der Entwicklung von web-basierten Internet-Anwendungen, also über Webbrowser nutzbarer Software. Web-basierte Anwendungen sind nur eine Spielart von Internet-Anwendungen, aber mit Abstand die wichtigste. Von Informatikern wird heute — zu Recht — verlangt, Anwendungen dieser Art entwerfen und implementieren zu können.

Kapitel 2

Netzwerk-Grundlagen

In diesem Kapitel werden die ISO/OSI-Protokollhierarchie und die für das Internet wichtigen Protokolle IP, TCP und UDP vorgestellt.

2.1 Protokolle und das ISO/OSI-Schichtenmodell

Ein Protokoll definiert Regeln und Formate für die Nachrichten, die die Kommunikationspartner austauschen. Viele Protokolle definieren unter anderem auch ein Adress-Format. Die durch dieses Format beschriebenen Adressen dienen dazu, den gewünschten Kommunikationspartner im Netzwerk zu spezifizieren.

Bei der Kommunikation über ein Netzwerk müssen sehr viele Details durch Protokolle geregelt werden. Dies fängt bei der Repräsentation der Daten als Spannungsdifferenzen auf der Hardware-Ebene an und geht über die Länge von Datenpaketen bis hin zur Fehlerbehandlung. Die Implementierung eines zuverlässigen Protokolls, wie es etwa für die Verbindung zu WWW-Servern verwendet wird, als monolithischer Block ist eine herkulische Aufgabe. In der Praxis werden daher anstatt eines Protokolls mehrere in Schichten organisierte Protokolle verwendet — man spricht von einem Schichtenmodell. Die Idee des Schichtenmodells besteht darin, die genannten Aufgaben über mehrere Schichten zu verteilen und für jede Schicht ein Protokoll zu spezifizieren, das sich einzig der Lösung der Schicht zugedachten Aufgabe widmet. Bei der Implementierung eines Protokolls kann also davon ausgegangen werden, daß die Aufgaben der unterliegenden Schichten bereits durch die entsprechenden Protokolle gelöst wurden.

Neben der einfacheren Implementierung von Protokollen liegt ein weiterer Vorteil des Schichtenmodells in der Austauschbarkeit von Schichten: Für spezielle Anwendungen mag es sinnvoll sein, die Protokolle der einzelnen Schichten nach bestimmten Eigenschaften der Implementierung auszuwählen. Etwa wenn für die Anwendung die Verbindung nicht unbedingt zuverlässig sein muß und der gelegentliche Verlust von Paketen akzeptabel ist (Dies ist zum Beispiel bei der Übertragung von Video-Daten der Fall).

Die *International Standards Organization* (ISO) hat einen Standard für die Organisation von Netzwerkprotokollen in Schichten definiert, das *open systems interconnection*-Modell. Das ISO/OSI-Modell umfaßt sieben Schichten [Tan96]:

1. *physical layer* (Bitübertragungsschicht) Die Bitübertragungsschicht stellt die Übertragung roher Bits über einen Kommunikationskanal zur Verfügung und wird deshalb von den Treibern der Netzwerkhardware implementiert. Beispiele für diese Schicht sind: Ethernet, Lichtwellenleiter und Breitband-ISDN im ATM.
2. *data link layer* (Sicherheitsschicht) Die rohe Übertragungsmöglichkeit für Bits wird durch die Sicherheitsschicht als eine Leitung dargestellt. Die Sicherheitsschicht garantiert, daß die Daten frei von Übertragungsfehlern sind und keine Daten verloren gehen. Dazu werden die Daten in *data frames* (Datenrahmen) bestimmter Länge verpackt, deren korrekter Empfang jeweils bestätigt wird. Im Falle eines Fehlers werden die Daten erneut gesendet. Beispiele für Protokolle der Sicherheitsschicht sind Protokolle der IEEE-802-Norm für lokale Netze, PPP und SLIP für serielle Wählleitungen (hauptsächlich) und HDLC.
3. *network layer* (Vermittlungsschicht) Wichtigste Aufgabe der Vermittlungsschicht ist das *Routing*, also die Auswahl eines geeigneten Übertragungsweges für Datenpakete über mehrere Zwischenstationen zum Bestimmungsort. Diese Wege werden häufig in statisch festgelegten *Routingtabellen* beschrieben. Das IP-Protokoll ist das bekannteste Beispiel für ein Protokoll dieser Ebene. Andere Protokolle sind IPX und AppleTalk.
4. *transport layer* (Transportschicht) Die Transportschicht stellt dem Benutzer einen Punkt-zu-Punkt-Kanal zur fehlerfreien Übertragung von Daten in ihrer Sendereihenfolge zur Verfügung. Im Internet kommen hier das verbindungsorientierte TCP und das paketorientierte UDP zum Einsatz.
5. *session layer* (Sitzungsschicht) Die Sitzungsschicht erweitert die Funktionalität der Transportschicht. Eine wichtige Funktion der Sitzungsschicht ist die *Dialogue control*, mit deren Hilfe festgelegt wird, ob Daten immer nur in eine Richtung fließen oder auch gleichzeitig in beide Richtungen transportiert werden dürfen. Das *Token-Management* verhindert, daß beide Seiten eine bestimmte Operation gleichzeitig ausführen. Außerdem bietet die Sitzungsschicht die Möglichkeit *Check Points* in den Datenfluß einzufügen und unterbrochene Übertragungen ab dem letzten erfolgreich übertragenen Check Point wieder aufzunehmen. Ein Beispiel für ein Protokoll auf der Sitzungsschicht sind die UNIX-RPC.
6. *presentation layer* (Darstellungsschicht) Auf dieser Ebene der Protokollhierarchie werden Daten in einer vereinbarten und standardisierten Weise dargestellt, so daß Sender und Empfänger die Daten in gleicher Weise interpretieren (Beispiel: Darstellung von Gleitkommazahlen). XDR und ASN.1 sind Beispiele für Protokolle dieser Schicht.
7. *application layer* (Verarbeitungsschicht) Auf dieser Protokollebene findet üblicherweise die Kommunikation zwischen Applikationen statt. Dementsprechend findet sich hier eine Fülle von Protokollen. Einige sehr bekannte Vertreter sind HTTP, FTP und SMTP. NFS ist ein Beispiel für ein Protokoll, das auf alle Dienste unterliegender Ebenen (XDR und RPC) zurückgreift.

2.2 Das Internet-Protokoll

Im Internet wird die Kommunikation über die so genannte TCP/IP-Protokoll-Suite geregelt. Diese Suite setzt auf existierende Schicht-2-Protokolle, wie zum Beispiel Ethernet, auf. Da TCP/IP vor dem ISO/OSI-Schichtenmodell definiert wurde, läßt sich TCP/IP nicht immer eindeutig den verschiedenen Schichten zuordnen.

Das *Internet Protocol* (IP) implementiert Schicht drei des ISO/OSI-Modells. IP ist ein verbindungsloses, nicht-zuverlässiges Protokoll zur Kommunikation zwischen Rechnern (Hosts). Es bietet Routing und Fragmentierung. Verbindungslos bedeutet, daß Nachrichten einzeln übermittelt werden. Sie enthalten deswegen auch immer alle Adressinformationen. Nicht-zuverlässig bedeutet, daß das Protokoll keine erfolgreiche Übertragung der Nachricht garantiert.

Ein IP-Rahmen (oder IP-Paket) besteht aus einem Header und dem Datenteil. Der Header enthält unter anderem folgende Felder:

- Quelladresse
- Zieladresse
- Protokollbezeichner für das darüberliegende Protokoll
- Checksumme
- Länge des gesamten Pakets (max. 65535 Bytes)

IP-Adressen sind 32 Bit lang und kodieren die ID des Netzes sowie die ID des Hosts. Dabei bestimmen die ersten Bits, wie die Adresse aufgeteilt ist:

- Class A:

0	7 Bits für Netz-ID	24 Bits für Host-ID
---	--------------------	---------------------
- Class B:

10	14 Bits für Netz-ID	16 Bits für Host-ID
----	---------------------	---------------------
- Class C:

110	21 Bits für Netz-ID	8 Bits für Host-ID
-----	---------------------	--------------------

Will ein Host ein Paket versenden, so überprüft er zunächst, ob die Netz-ID der Zieladresse mit seiner Netz-ID übereinstimmt. Ist das der Fall, so verschickt er das Paket mit Hilfe des Schicht-2-Protokolls direkt an diesen Host. Liegt die Zieladresse in einem anderem Netz, so versendet er das Paket mit Hilfe des Schicht-2-Protokolls an einen Router. Dieser kennt entweder das Zielnetzwerk und kann das Paket dorthin senden, oder er kennt einen übergeordneten Router, an den er das Paket dann verschickt.

Router, die ein Paket durch das Internet weiterreichen betrachten nur die Netz-ID des Pakets; erst wenn das Paket im richtigen physikalischen Netz angekommen ist spielt die Host-ID wieder eine Rolle. Eine weitere Aufteilung der Host-ID in Subnetz-ID und eigentliche Host-ID ist in RFC 950 [MP85] spezifiziert. Dazu wird eine *Subnetz-Maske* verwendet. Mit ihr ist es dem Host möglich zu bestimmen, ob ein anderer Rechner im gleichen Subnetz liegt.

Beispiel Die Netz-ID der Universität Tübingen lautet 134.2, binär dargestellt als

10000110	00000010
----------	----------

Die nächsten 5 Bit kodieren die Subnetz-ID. Für das Wilhelm-Schickard-Institut lautet die Subnetz-ID 1. Alle IP-Adressen innerhalb des WSI haben den folgenden Präfix:

10000110	00000010	00001
----------	----------	-------

Dementsprechend wird eine Subnetz-Maske definiert, die genau diesen Teil der Adresse erhält, wenn man eine binäre UND-Verknüpfung durchführt. Für das WSI hat die Subnetz-Maske die folgende Binärdarstellung (in dezimaler Darstellung 255.255.248.0):

11111111	11111111	11111000	00000000
----------	----------	----------	----------

Ein Host, der ein IP-Paket versenden möchte, prüft zunächst ob die Ziel-Adresse innerhalb seines eigenen Subnetzes liegt. Denn in diesem Fall ist das Paket direkt, d. h., ohne Umweg über einen Router zu versenden. Um die Zugehörigkeit zu einem Subnetz zu prüfen, geht der Host folgendermaßen vor: Zuerst wird die Quelladresse (also die eigene IP-Adresse) mit der Subnetz-Maske UND-verknüpft. Dies führt für einen Host im Netz des WSI immer zu dem folgenden Ergebnis:

10000110	00000010	00001000	00000000
----------	----------	----------	----------

Nun wird die Zieladresse mit der Subnetz-Maske UND-verknüpft. Stimmen diese beiden Ergebnisse überein, handelt es sich um einen Host im selben Subnetz und das Paket kann direkt versendet werden, andernfalls schickt der Host das Paket an den Router mittels des Schicht-2-Protokolls.

2.2.1 Classless InterDomain Routing (CIDR)

Der Zuschnitt der drei Adressklassen ist häufig Gegenstand von Kritik: Für viele Netzteilnehmer ist ein Class-C-Netz mit seinen maximal 256 Hosts zu klein, ein nächstgrößeres Class-B-Netz aber wiederum viel zu groß (max. $2^{16} = 65536$ Hosts). Der steigende Bedarf von IP-Adressen (vgl. Abschnitt 2.2.2) und der ungünstige Zuschnitt der Adressklassen führt zu zwei Problemen: Knappheit an IP-Adressen, da die Adressbereiche großer Netzwerke nur selten voll ausgeschöpft werden und eine Explosion der Größe von Routingtabellen. Für die Kodierung der Netz-Id eines Class-C-Netzes stehen 21 Bits zur Verfügung, es können also $2^{21} \approx 2$ Millionen Netze dieser Größe eingerichtet werden. Für jede Netz-Id wird eine Routing-Information in den Routingtabellen eines Routers hinterlegt. Mit steigender Zahl von Class-C-Netzen steigt die Zahl der Einträge, damit die Größe der Tabellen und schließlich der gesamte Verwaltungsaufwand. Bei einigen verwendeten Routing-Algorithmen wird diese Menge an Informationen zum Problem.¹

Seit dem Beginn der 1990er Jahre werden Class-C-Netze daher anders verwendet, man spricht vom *Classless InterDomain Routing (CIDR)* [FLYV93]. Besteht ein Bedarf für mehr als 256 Adressen, wird dafür kein neues Class-B-Netz eingerichtet, sondern eine entsprechende Anzahl an von der Adresse her aneinander grenzenden Class-C-Netzen. Um nicht weltweit auf jedem Router die

¹Etwa, wenn die Tabellen regelmäßig zwischen Routern übertragen werden oder die Komplexität des Algorithmus mehr als linear mit der Größe der Tabelle wächst.[Tan96]

Routing-Informationen für alle Class-C-Netze speichern zu müssen, wurde der Adressbereich für Class-C-Netze in vier Blöcke aufgeteilt (siehe Abbildung 2.1). Die Routingtabelle enthält im wesentlichen die Informationen für das Routing innerhalb eines dieser Blöcke — nämlich nur dem Block, dem der Router zugeordnet ist. Ist ein IP-Paket an einen Host aus einem anderen Block adressiert, reicht es den Präfix zu betrachten, um die Region zu erkennen. Alle für diesen Bereich bestimmte Pakete werden dann an einen Router der entsprechenden Region gesendet.

Anfang	Ende	Region
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 Pazifikraum

Abbildung 2.1: Aufteilung von Class-C-Netzwerken nach Regionen.

Durch CIDR orientiert sich die Adressvergabe im Netz also stärker an einer Hierarchie. Diese Idee ist nicht nur auf Class-C-Netzwerke anwendbar, sondern generell auf alle Klassen von IP-Adressen. Tatsächlich werden die in Abschnitt 2.2 beschriebenen Klassen heute nicht mehr zum Routing im Internet verwendet.

2.2.2 IPv6 — das Internet-Protokoll der nächsten Generation

Das Internet-Protokoll IP sieht 32 Bit zur Speicherung einer Adresse vor. Damit stehen 2^{32} , also mehr als 4 Milliarden Adressen zur Verfügung. Obschon diese Anzahl von Netzteilnehmern in der Praxis auch nicht nur annähernd erreicht ist, werden IP-Adressen knapp [Husct]. Die Gründe dafür sind vielfältig: Zu großzügige Vergabe großer Subnetze in den frühen Jahren des Internet², ein steigender Bedarf an Adressen, und schließlich sind noch einige Adress-Bereiche für spezielle technische Zwecke reserviert.

Dies war der entscheidende Grund für die *Internet Engineering Taskforce (IETF)* in den 1990er Jahren mit der Entwicklung eines Nachfolgers für das IP-Protokoll zu beginnen: IPv6. Eine Adresse im IPv6-Protokoll ist mit 128 Bit deutlich länger. Von dieser großzügigen Bemessung verspricht man sich neben der Lösung des Knappheitsproblems auch ein übersichtlicheres Routing [Tan96]. Neben dem Adressformat, wurden noch etliche andere Änderungen vorgenommen. Zum Beispiel wurde das Format der Header vereinfacht und das Protokoll um Funktionalität zur sicheren Kommunikation erweitert.

Hinweis Die hier gegebene Beschreibung des IP-Protokolls ist sehr knapp gehalten. Einige Protokolle, die im Zusammenhang mit dem IP-Protokoll wichtige Rollen spielen werden im folgenden genannt; für eine umfassende Auskunft empfiehlt sich die Lektüre von [Ste90, Tan96]. Eine besonders detaillierte Beschreibung dieser Protokolle findet sich unter anderem in [Com91]. Das *Internet Con-*

²Das amerikanische Massachusetts Institute of Technology zum Beispiel, verfügt über ein Class-A-Netzwerk, könnte also 2^{24} (ca. 16,7 Millionen) Rechner auf dem Campus aufstellen und adressieren.

*tr*ol Message Protocol (ICMP) dient zur Fehlerbehandlung und zur Steuerung innerhalb von IP. Die Adressabbildung von IP-Adressen auf darunterliegende Hardware-Adressen geschieht über das *Address Resolution Protocol* (ARP).

2.3 TCP und UDP

Das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP) implementieren Schicht vier des ISO/OSI-Modells. Beide Protokolle bauen auf IP auf und bieten Prozess-zu-Prozess-Kommunikation. Dazu ist eine zusätzliche Adressierung innerhalb eines Hosts notwendig, die so genannten *Ports*. Ports werden durch 16 Bit große Ganzzahlen repräsentiert. Protokolle der höherliegenden Schichten definieren *well-known Ports*, d.h. Portnummern über die die Kommunikation abläuft oder zumindest beginnt.

Eine Verbindung im Internet ist damit durch folgendes Fünf-Tupel eindeutig gekennzeichnet:

- Das Protokoll (TCP oder UDP)
- Die IP-Adresse des lokalen Hosts
- Die lokale Port-Nummer
- Die IP-Adresse des entfernten Hosts
- Die Port-Nummer auf dem entfernten Host

UDP ist wie IP ein verbindungsloses Protokoll mit einer maximalen Paketgröße von 65535 Bytes. Als einziges Feature neben den Portnummern bietet es optionale Checksummen an. Stimmt die Checksumme beim Empfang nicht überein wird das Paket verworfen. UDP ist im RFC 768 [Pos80] definiert.

TCP ist ein verbindungsorientiertes, verlässliches, stromorientiertes Protokoll. Die Kommunikation über eine TCP-Verbindung ähnelt der Kommunikation über eine Telefonleitung: Nach Aufbau der Verbindung wird diese Verbindung von beiden Partnern so lange verwendet, bis einer der Kommunikationspartner die Verbindung beendet. Die Verlässlichkeit einer TCP-Verbindung äußert sich in mehreren Eigenschaften: Die Richtigkeit der übertragenen Daten wird durch Prüfsummen überprüft. Der Empfang von Daten ist dem Sender zu bestätigen, es gehen also keine Daten unbemerkt verloren. Die TCP-Implementierung garantiert, daß die Daten in der Reihenfolge beim Empfänger eintreffen, in der sie vom Absender verschickt wurden. TCP ist stromorientiert, d. h. die Länge der Nachrichten ist unbeschränkt. Kann ein TCP-Nachricht nicht mit einem IP-Paket versendet werden, verteilt die TCP-Implementierung die Daten automatisch auf mehrere IP-Pakete. Die TCP-Implementierung des Empfängers speichert ankommende IP-Pakete in einem Puffer zwischen und setzt die Pakete in der richtigen Reihenfolge zu einer kompletten TCP-Nachricht zusammen. TCP verfügt außerdem über eine Flußkontrolle: Kommuniziert ein schneller Host mit einem langsamen Host, regelt das TCP die Geschwindigkeit der Kommunikation entsprechend. TCP ist im RFC 793 [Pos81] definiert.

Die folgende Tabelle zeigt die Eigenschaften von TCP und UDP im Vergleich:

	UDP	TCP
verbindungsorientiert?	nein	ja
Längenbeschränkungen?	ja	nein
Checksumme	optional	ja
Empfangsbestätigung?	nein	ja
Timeout mit erneuter Übertragung?	nein	ja
Erkennung von Duplikaten?	nein	ja
Reihenfolge?	nein	ja
Flusskontrolle?	nein	ja

2.4 TCP-Programmierung mit Sockets

Verbindungen über TCP/IP werden unter Unix durch so genannte *Sockets* repräsentiert. Ein Socket stellt den Endpunkt einer Verbindung dar. Die Systemaufrufe für den Zugriff auf Sockets sind durch den POSIX-Standard [Gal95, Lew94] normiert. Eine Referenz der in diesem Abschnitt vorgestellten Funktionen findet sich auch im `man` Hilfe-System eines jeden Unix-Betriebssystems. Der Befehl `man 2 socket`, zum Beispiel zeigt einen Hilfe-Text mit allen Informationen, die für die Benutzung der `socket`-Funktion relevant sind. Eine ausführliche Beschreibung von Sockets findet sich in [Ste90] und [Lew94].

2.4.1 Funktionen zur Socket-Programmierung

Im folgenden Abschnitt werden die POSIX-Systemaufrufe für die Programmierung mit Sockets vorgestellt.

- `int socket(int domain, int type, int protocol)`

Erstellt ein Socket-Objekt. Das Argument *domain* ist eine symbolische Konstante, die festlegt welcher Protokollhierarchie der zu erzeugende Socket zugehört. Für die Kommunikation im Internet ist dies immer `AF_INET`.³ Die symbolischen Konstanten für weitere vom Betriebssystem unterstützte Protokollhierarchien finden sich in der Header-Datei `sys/socket.h` bzw. können auf der `man`-Seite zu `socket` nachgeschlagen werden.

Das Argument *type* legt den Typ des Sockets fest: `SOCK_STREAM` für eine stromorientierte Verbindung oder `SOCK_DGRAM` für eine paketorientierte Verbindung. Da sich aus dem Typ der Verbindung das Protokoll bereits ergibt (TCP für stromorientiert, UDP für paketorientiert), setzt man den *protocol*-Parameter in der Regel auf 0. Der Rückgabewert von `socket` ist ein *Socket-Deskriptor*, welchen das Betriebssystem auf ein Socket-Objekt abbildet. Der `socket`-Aufruf setzt das „Protokoll“-Feld des Fünf-Tupels, das eine Internet-Verbindung beschreibt (vgl. Abschnitt 2.3).

- `int bind(int sockdes, struct sockaddr *localaddr, int addrlen)`

Konzeptionell füllt `bind` die Felder „IP-Adresse des lokalen Hosts“ und „lokale Port-Nummer“ des Fünf-Tupels zur Beschreibung einer Internet-Verbindung (vgl. Abschnitt 2.3). Diese Funktion wird in der Regel nur von einem Server

³Auf einigen Betriebssystemen ist auch der Name `PF_INET` üblich.

aufgerufen, um den lokalen Port (in der Regel ein well-known port) und die lokale IP-Adresse auf der Verbindungen angenommen werden sollen, festzulegen. Für einen Client ist der Aufruf von `bind` dagegen optional: fehlt der Aufruf, bleiben die entsprechenden Felder des Tupels unausgefüllt und werden zu einem späteren Zeitpunkt vom Betriebssystem automatisch gefüllt. Der Typ `struct sockaddr` dient nur als Platzhalter. Die Deklaration sieht folgendermaßen aus:

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

Jede Protokollhierarchie definiert einen eigenen Typ für Adressen, welcher dann beim Aufruf von `bind` in den typ `struct sockaddr*` gecastet wird. Für das Internet gelten die Definitionen:

```
#include <netinet/in.h>
struct in_addr {
    unsigned long s_addr;
};

struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

Das Feld `sa_family` des Platzhalters `sockaddr` enthält eine Ganzzahl, die identifiziert zu welcher Protokollhierarchie die Adresse gehört. Der Wert dieses Feldes entspricht den möglichen Werten für das `domain`-Argument der `socket`-Funktion. Diese Information muß auch im ersten Feld der Structs für die einzelnen Protokollhierarchien stehen. Das Feld `sa_data` ist der eigentliche Platzhalter. Das Struct `sockaddr_in` besteht aus einer Portnummer `sin_port` und einer IP-Adresse `sin_addr`. Das Array im Feld `sin_zero` füllt das Struct `sockaddr_in` auf die Größe von `sockaddr` auf. Es ist dringend zu empfehlen, die Structs für die Socket-Adressen vor der Benutzung mit der Funktion `bzero` zu initialisieren.

- `int connect(int sockdes, struct sockaddr* remoteaddr, int addrlen)`

Für verbindungsorientierte Protokolle (TCP) stellt `connect()` die Verbindung zur angegebenen Adresse her. Konzeptionell werden dabei die Felder „IP-Adresse des entfernten Hosts“ und „Port-Nummer auf dem entfernten Host“ des Fünf-Tupels für Internet-Verbindungen festgelegt. Nach dem diese Funktion aufgerufen wurde, kann die Kommunikation beginnen. Ein Socket-Deskriptor ist dabei genau so zu benutzen, wie ein File-Deskriptor: Mit den Funktionen `read()` und `write()` aus der C-Standardbibliothek können Daten aus dem Socket gelesen bzw. in den Socket geschrieben werden.

- `int listen(int sockdes, int backlog)`

Wird nur vom Server aufgerufen und legt eine Warteschlange für eingehende Verbindungen an. Die Länge der Warteschlange ist `backlog` und in der Regel vom

Betriebssystem auf fünf begrenzt. POSIX schreibt vor, die maximale Länge in der Konstanten `SOMAXCONN` abzulegen. Die Warteschlange wird verwendet, um weitere eingehende Verbindungen zu puffern, während eine Verbindung akzeptiert wird.

- `int accept(int sockdes, struct sockaddr* remoteaddr, int* addrlen)`

Dieser Aufruf wartet bis eine eingehende Verbindung vorhanden ist. Dann stellt `accept` die Verbindung her. Rückgabewert ist der Socket-Deskriptor, der zur Verbindung gehört. Die Adresse des Kommunikationspartners wird im Parameter `remoteaddr` abgelegt.

Fehlerbehandlung Die hier vorgestellten Funktionen geben im Falle eines Fehlers eine negative Ganzzahl zurück. Neben diesem Rückgabewert, der lediglich das Auftreten eines Fehlers kodiert, wird die globale Variable `errno` auf einen Fehlercode gesetzt, der die Ursache des Fehlers genauer beschreibt. Im Fehlerfall sollte also der Inhalt dieser Variable genauer betrachtet werden. Listen der jeweils möglichen Fehlercodes und deren Bedeutung finden sich auf den Manual-Seiten der entsprechenden Funktionen. Hilfreich ist in diesem Zusammenhang auch die Funktion `strerror` aus der C-Standardbibliothek, die einen `errno`-Fehlercode dekodiert und eine textuelle Beschreibung des Fehlers zurückgibt. (Siehe auch Abbildung 2.3)

2.4.2 Verwendung von Sockets

Abbildung 2.2 zeigt den typischen Ablauf einer Kommunikation zwischen Client und Server über TCP. Der Server erzeugt einen neuen Socket mit `socket()` und trifft durch den Aufruf von `bind()` und `listen()` die Vorbereitungen, um auf eingehende Verbindungen von Clients zu warten. Der Aufruf von `accept()` blockiert den Server, d. h., dieser Funktionsaufruf kehrt erst zurück wenn eine eingehende Verbindung vorhanden ist. Eine solche Verbindung kommt zustande, wenn der Client einen `connect()`-Aufruf (selbstverständlich mit den entsprechenden Argumenten, um diesen Server zu erreichen) tätigt. In diesem Moment kehrt der Funktionsaufruf `accept()` zurück und die eigentliche Kommunikation zwischen Client und Server kann beginnen. Ein Socket funktioniert aus Sicht des Programmierers analog zu einem Datei-Deskriptor: Er kann die Funktionen `read()` und `write()` benutzen, um Daten aus dem Socket zu lesen bzw. Daten in den Socket zu schreiben. Ein Aufruf von `close()` beendet die Verbindung. Auf Seite des Servers ist darauf zu achten, daß der Server den Socket schließt, den der Aufruf von `accept()` zurückgab. Der mit `socket()` erzeugte Socket bleibt geöffnet damit der Server weitere Verbindungen annehmen kann. Hierfür ist auch ein neuerlicher Aufruf von `accept()` notwendig. Deshalb werden die Aufrufe ab `accept()` im Server in der Regel in einer Schleife aufgerufen, in diesem Fall spricht man von einer *bind-listen-accept-Schleife*.

Die Abbildungen 2.3 und 2.4 zeigen den vollständigen C-Sourcecode für einen Client bzw. Server, der über TCP kommuniziert. Die Funktion `panic` wird im Fehlerfall aufgerufen und druckt eine Fehlermeldung aus bevor sie die Ausführung des Programms abbricht. In der Funktion `main()` wird ein neuer Socket erzeugt, ein Struct vom Typ `sockaddr_in` erzeugt und initialisiert, um sich zum Server an der IP-Adresse 127.0.0.1⁴ Port 8005 zu verbinden.

⁴Dabei handelt es sich um eine spezielle Adresse, die den lokalen Host identifiziert.

Client	Server
<code>socket()</code>	<code>socket()</code>
	<code>bind()</code>
	<code>listen()</code>
	<code>accept()</code>
<code>connect()</code>	(blockiert)
<code>read()/write()</code>	<code>read()/write()</code>
<code>close()</code>	<code>close()</code>

Abbildung 2.2: Ablauf einer Client-Server-Kommunikation über TCP

Die Portnummer wird von der Funktion `htons()` von der lokalen Repräsentation in die für das Netzwerk festgelegte Repräsentation konvertiert. Die Funktion `inet_addr()` der C-Standardbibliothek wandelt eine in lesbarer String-Notation gegebene IP-Adresse in die entsprechende 32-Bit-IP-Adresse um.

2.5 UDP-Programmierung mit Sockets

Um Nachrichten über UDP zu senden müssen sowohl Server als auch Client zunächst einen Socket mit Hilfe von `socket()` (vgl. Abschnitt 2.4.1) erstellen. Das Argument `domain` für den Aufruf von `socket()` ist, wie schon bei TCP, auf `AF_INET` zu setzen. Für das Argument `type` wird die Konstante `SOCK_DGRAM` eingesetzt. `Protocol` darf wiederum auf 0 gesetzt werden.

Danach müssen sowohl Server also auch Client `bind()` aufrufen, um einen Port zu reservieren. Dies ist hier auch für den Client wichtig, weil keine logische Verbindung aufgebaut wird, der Client muß sich selbst darum kümmern, stets an der gleichen Stelle zu horchen, damit der Server ihn erreicht.

Um im Server auf eingehende Nachrichten zu warten, ruft dieser die Funktion `recvfrom()` auf:

- `int recvfrom(int sockdes, char* buf, int nbytes, int flags, struct sockaddr* from, int* addrlen)`

Das Argument `sockdes` bezeichnet den Socket-Deskriptor, `buf` ist ein Zeiger auf einen Puffer der Länge `nbytes`. In diesen Puffer schreibt `recvfrom()` die empfangene Nachricht. Mit Hilfe des Argumentes `flags` können einige Optionen gesetzt werden. Kehrt `recvfrom()` zurück, so wird die im Argument `from` übergebene Struct `struct sockaddr` mit der Adresse des Absenders der UDP-Nachricht gefüllt. Die Länge dieses Structs wird über `addrlen` übergeben. Die Funktion `recvfrom()` kehrt erst zurück, wenn eine Nachricht empfangen wurde.

Der Client schickt eine Nachricht mit Hilfe der Funktion `sendto`:

- `int sendto(int sockdes, char* buf, int nbytes, int flags, struct sockaddr* to, int addrlen)`

Für die Argumente `sockdes`, `buf`, `nbytes` und `flags` entsprechen den entsprechenden Argumenten von `recvfrom()`. `Buf` enthält die zu versendende Nachricht und `nbytes` gibt die Länge der Nachricht an. Im `struct sockaddr` befinden sich IP-Adresse und Port des Empfängers. Die Länge des Structs ist im Argument `addrlen` zu übergeben.

Beide Funktionen geben die Länge der empfangenen bzw. gesendeten Daten zurück oder -1 falls ein Fehler aufgetreten ist.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int panic(char *where)
{
    fprintf(stderr, "error at %s '%s'\n", where, strerror(errno));
    exit(127);
}

int main(void) {

    int sd;
    struct sockaddr_in remote_addr;
    int addrlen = sizeof(remote_addr);

    bzero((void *) &remote_addr, addrlen);
    remote_addr.sin_family = AF_INET;
    remote_addr.sin_port = htons(8005);
    remote_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        panic("socket");

    if (connect(sd, (struct sockaddr*) &remote_addr, addrlen) < 0)
        panic("connect");

    printf("Connected to server\n");
    close (sd);
    return 0;
}
```

Abbildung 2.3: Ein Client, der über TCP kommuniziert

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

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

int panic(char *where)
{
    fprintf(stderr, "error at %s '%s'\n", where, strerror(errno));
    exit(127);
}

int main(void) {

    int sd;
    struct sockaddr_in local_addr;
    int local_addr_len = sizeof(local_addr);

    if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        panic("socket");

    bzero((void *) &local_addr, local_addr_len);
    local_addr.sin_family = AF_INET;
    local_addr.sin_port = htons(8005);
    local_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    if (bind(sd, (struct sockaddr *) &local_addr, local_addr_len) < 0)
        panic("bind");

    if (listen(sd, SOMAXCONN) < 0)
        panic("listen");

    while (1) {
        int new_sd;
        struct sockaddr_in client_addr;
        unsigned int client_addr_len = sizeof(struct sockaddr);

        new_sd = accept(sd, (struct sockaddr*) &client_addr, &client_addr_len);

        if (new_sd < 0)
            panic("accept");

        printf("New connection on socket descriptor %d\n", new_sd);
        close (new_sd);
    }

    close(sd);
    return 0;
}
```

Abbildung 2.4: Ein Server, der über TCP kommuniziert

Literaturverzeichnis

- [Bro98] Po Bronson. Hotmale. *Wired*, December 1998. 5
- [Com91] Douglas E. Comer. *Internetworking with TCP/IP*, volume 1. Prentice-Hall, Inc., 2nd edition, 1991. 11
- [FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519 (Proposed Standard), September 1993. 10
- [Gal95] Bill O. Gallmeister. *POSIX.4: Programming For The Real World*. O'Reilly Inc., 1995. 13
- [Husct] Geoff Huston. Ipv4 address space report, 2005 Oct. 11
- [Lew94] Donald. A Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1994. 13
- [MP85] J.C. Mogul and J. Postel. Internet Standard Subnetting Procedure. RFC 950 (Standard), August 1985. 9
- [Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980. 12
- [Pos81] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168. 12
- [Ste90] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall Software Series, 1990. 11, 13
- [Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall International, Inc., third edition, 1996. 7, 10, 11

Index

- AF_INET, 13, 16
- SOCK_DGRAM, 13, 16
- SOCK_STREAM, 13
- SOMAXCONN, 15
- bind(), 16
- bzero, 14
- errno, 15
- htons(), 16
- inet_addr(), 16
- read(), 14
- recvfrom(), 16
- sa_family, 14
- sendto, 16
- sin_port, 14
- sockaddr_in, 14
- socket(), 16
- strerror, 15
- struct sockaddr, 14, 16
- write(), 14
- accept(), 15
- bind(), 13
- connect(), 14
- listen(), 14
- socket(), 13
- socket.h, 13
- Address Resolution Protocol, 12
- Adress-Format, 7
- AppleTalk, 8
- application layer, 8
- ARP, 12
- ASN.1, 8
- bind-listen-accept-Schleife, 15
- Bitübertragungsschicht, 8
- Check Points, 8
- Checksumme, 9
- CIDR, 10
- Classless InterDomain Routing, 10
- Darstellungsschicht, 8
- data frames, 8
- data link layer, 8
- Datenpaket, 7
- Datenrahmen, 8
- Dialogue control, 8
- Ethernet, 8, 9
- FTP, 8
- GUI, 5
- Header, 9
- HTTP, 8
- ICMP, 12
- IEEE-802, 8
- ietf, 11
- International Standards Organization, 7
- Internet Control Message Protocol, 12
- Internet Engineering Taskforce (IETF), 11
- Internet Protocol, 9
- IP, 9, 12
- IP-Adresse, 9, 12
- IP-Protokoll, 8
- IP-Rahmen, 9
- IPv4, 9
- IPX, 8
- ISO, 7
- ISO/OSI, 7
- Kommunikationskanal, 8
- Nachricht, 7
- network layer, 8
- Netz-ID, 9
- Netzwerk, 7
- Netzwerkhardware, 8
- NFS, 8

- Nicht-zuverlässig, 9
- physical layer, 8
- Port, 12
- Ports
 - well-known, 12
- POSIX-Standard, 13
- PPP, 8
- presentation layer, 8
- Programmierung
 - interaktiv, 5
 - nebenläufige, 5
 - Netzwerk, 5
- Protokoll, 7
- Protokollhierarchie, 5

- Quelladresse, 9, 10

- Remote Procedure Calls, 8
- Repräsentation, 7
- Router, 9
- Routing, 8
- Routingstabelle, 8
- RPC, 8

- Schicht, 7
- Schichtenmodell, 7
- Sendereihenfolge, 8
- session layer, 8
- Sicherungsschicht, 8
- Sitzungsschicht, 8
- SLIP, 8
- SMTP, 8
- Socket-Deskriptor, 13
- Sockets, 13
- Subnetz, 9
- Subnetz-Maske, 9

- TCP, 8, 12
- TCP/IP, 9
- Token-Management, 8
- Transmission Control Protocol (TCP),
12
- transport layer, 8
- Transportschicht, 8

- UDP, 8, 12
- User Datagram Protocol (UDP), 12

- Verarbeitungsschicht, 8
- Verbindungslos, 9
- verbindungslos, 9
- Vermittlungsschicht, 8

- XDR, 8

- Zieladresse, 9, 10
- Zielnetzwerk, 9