Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Konfiguration à la Redhat

Redhat und Fedora benutzen klassisch Konfigurationsdateien für Netzwerkschnittstellen unter `/etc/sysconfig/network.scripts` mit dem Namen `ifcfg-devicename`, also z. B. `ifcfg-eth0`:

```
DEVICE="eth0"
ONBOOT="yes"
HWADDR=00:0C:29:46:C1:12
TYPE=Ethernet
IPADDR0=10.10.103.123
PREFIX0=20
GATEWAY0=10.10.96.1
DEFROUTE=yes
NAME="Broadcom Ethernet Card"
```

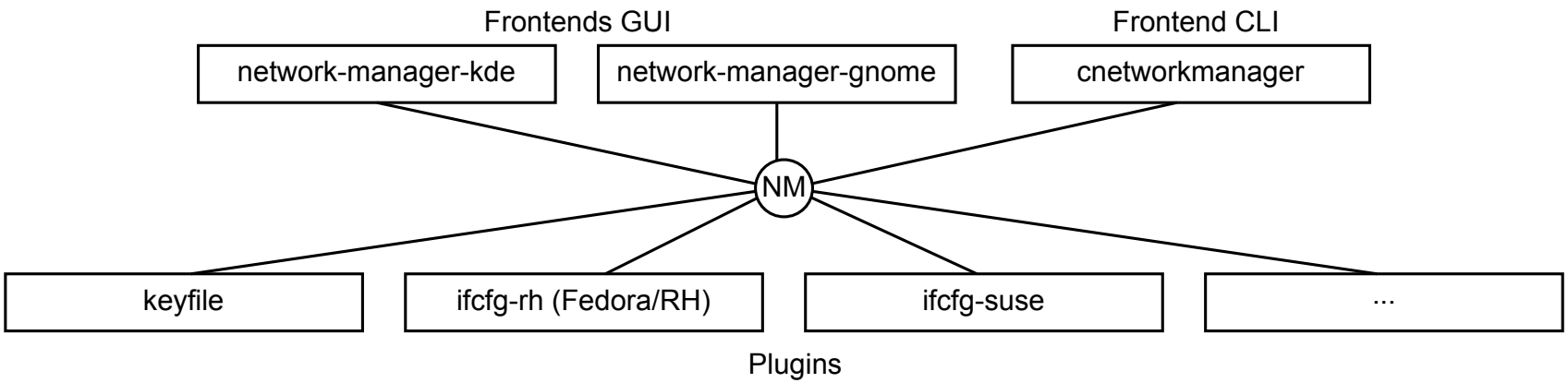
Die Zukunft: NetworkManager

Moderne Distributionen nutzen seit einiger Zeit als Ersatz des bisherigen Konfigurations- und Administrations-Systems den `NetworkManager`, der folgende Vorteile bietet:

- Unterstützung diverser Typen von Netzwerkschnittstellen: kabelgebundenes Ethernet, Wireless, Breitbankverbindungen (z. B. UMTS), VPN
- Frontends für diverse Desktops bzw. Windowmanager (kompatibel zum Standard von freedesktop.org)
- Plugins, die die Konfiguration entweder auf das eigene Schema von NetworkManager oder auf die Konfigurations-Prinzipien von Distributionen (wie Redhat, Suse) abbilden.

NetworkManager wurde 2004 von Redhat ins Leben gerufen, wird aber zumindest optional inzwischen auch u. a. von Debian und Suse unterstützt.

Prinzip von NetworkManager



Netzwerkdienste unter Linux

- Automatische Konfiguration von Netzwerkschnittstellen über dhcp
- Namensauflösung über DNS
- Internetzeit über NTP
- Mailtransport mit Sendmail bzw. Alternativen wie postfix, exim usw.
- HTTP
- SSH
- FTP

Netzwerkschichten

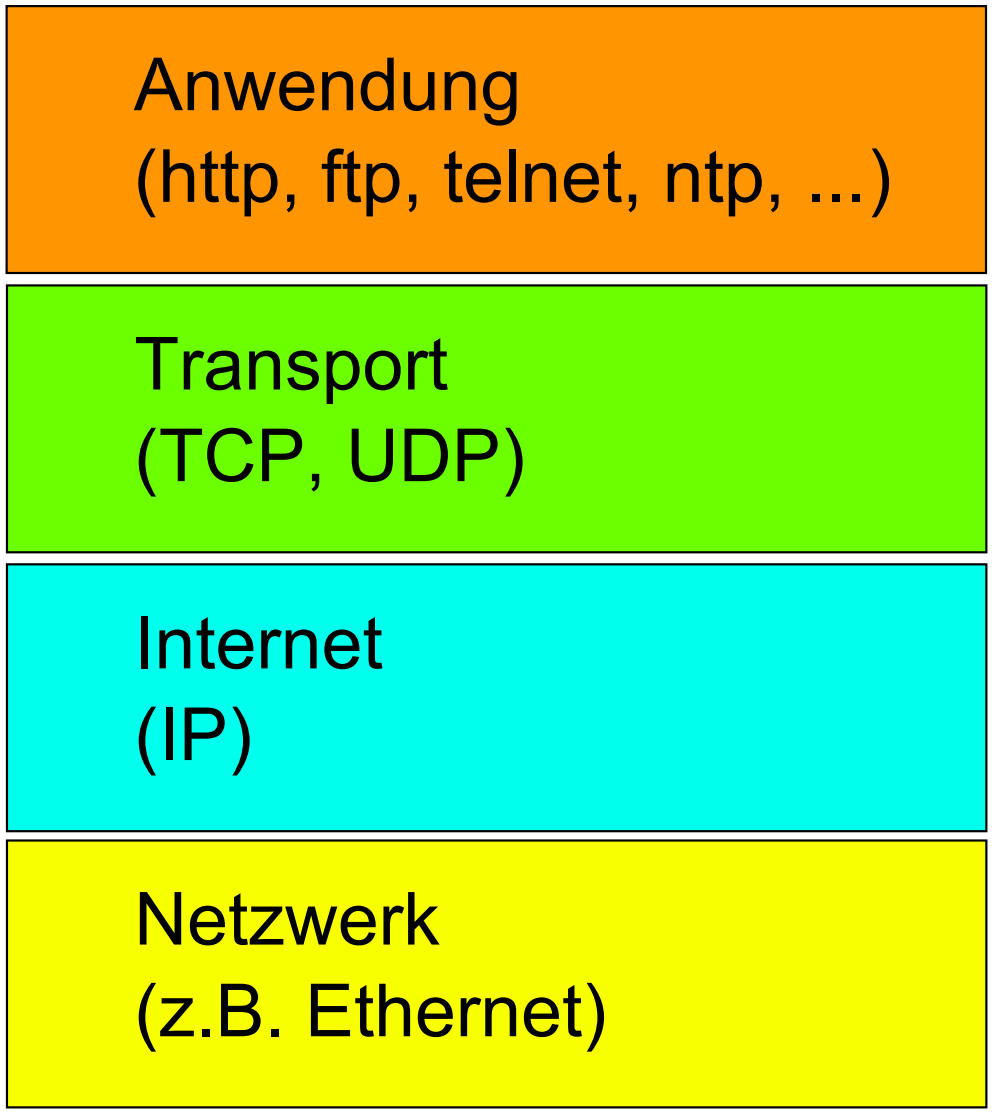
Die Kommunikation über Netzwerk besteht aus einem komplexen Zusammenspiel von

- Hardware: Netzwerkkarte im Rechner, Switche/Hubs/Router, Kabel
- Software: Netzwerktreiber, Datenverifikation/Fehlerkorrektur, Kommunikationssicherheit (Handshakes), Protokolle für Datenpakete, Datenaustauschprotokolle

Klassisch werden die einzelnen Schichten der Kommunikation über Netzwerke durch das recht granulare *OSI-Schichtenmodell* beschrieben.

Für ein Verständnis der Netzwerkprogrammierung reichen aber im Wesentlichen vier Schichten aus: *Netzwerkschicht*, *Internetschicht*, *Transportschicht*, *Anwendungsschicht*

Schaubild


Anwendung
(http, ftp, telnet, ntp, ...)
Transport
(TCP, UDP)
Internet
(IP)
Netzwerk
(z.B. Ethernet)

Bedeutung der Schichten

Netzwerkschicht Eigentliche Physik der Übertragung, im Wesentlichen Hardware

Internetschicht Low-Level-Kommunikationsprotokoll: Definition von Adressen, Ports, Format von Datenpaketen, heute i. A. IP(v4), in Zukunft evtl. IPv6

Transportschicht Legt den wesentlichen Typ der Kommunikation fest: Punkt-zu-Punkt-Verbindung (TCP), verbindungslos (UDP)
Analogien: Telefongespräch, Rundfunksendung
Frage: Ist ein klassischer Post-Brief verbindungslos oder nicht?

Anwendungsschicht realisiert ein anwendungsbezogenes Protokoll: z. B. SMTP/POP für Mail, http für WWW usw.

Übersicht Socket-API

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

int socket(int domain, int type, int protocol);

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

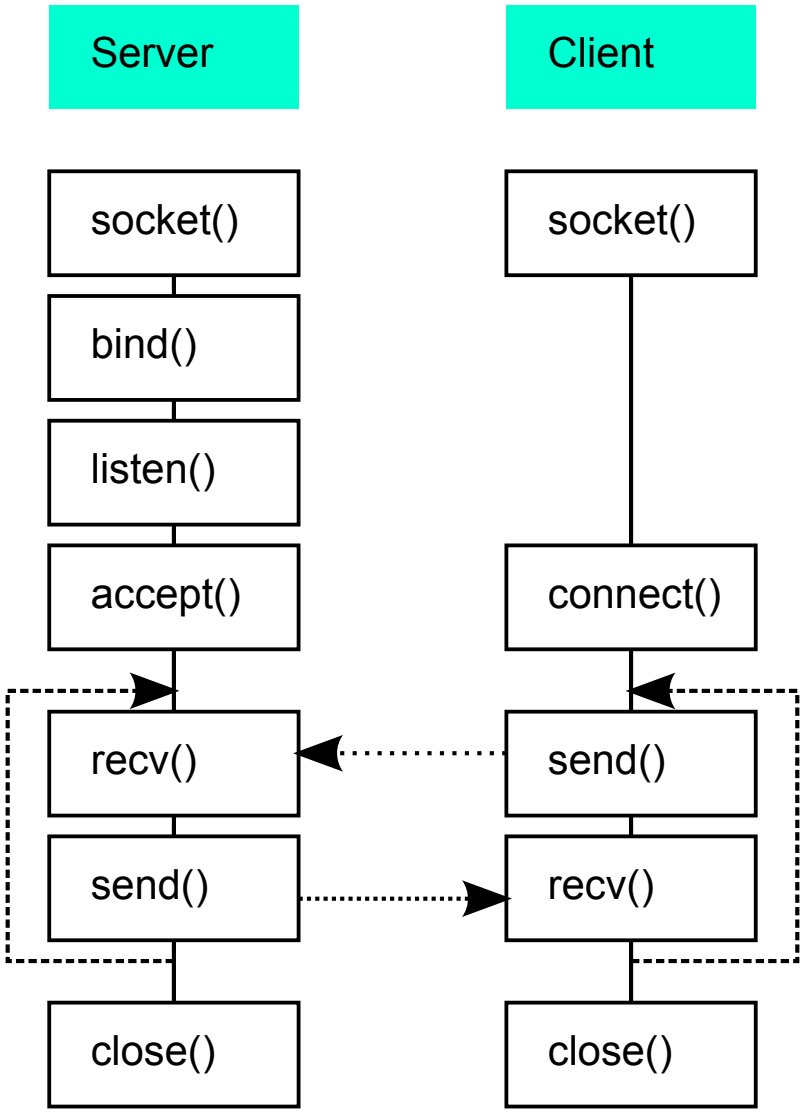
Client-Server

Für die Programmierung einer Netzwerkverbindung zwischen zwei Rechnern (also TCP/IP) soll ein Client-Server-Paar realisiert werden, wobei der Server die empfangenen Daten einfach als Echo zurück gibt.

Was muss der Server hierfür leisten:

- Einen Socket anlegen und
- diesen an eine/mehrere IP-Adresse(n) und einen Port binden.
- Auf dem Port lauschen und
- Verbindungen von Clients annehmen und mit diesen kommunizieren.

Schaubild Client-Server



Implementierung Echo-Server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define MAX 1024

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock<0) {
        printf("Fehler %d beim Anlegen des sockets: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }
    int yes = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
```

Implementierung Echo-Server (Forts.)

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(9999);
address.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *) &address, sizeof(address))<0) {
    printf("Fehler %d beim bind: %s\n", errno, strerror(errno));
    exit(EXIT_FAILURE);
}

listen(sock, 1);

char buf[MAX];
struct sockaddr_in client_address;
size_t addrlen = sizeof(client_address);
```

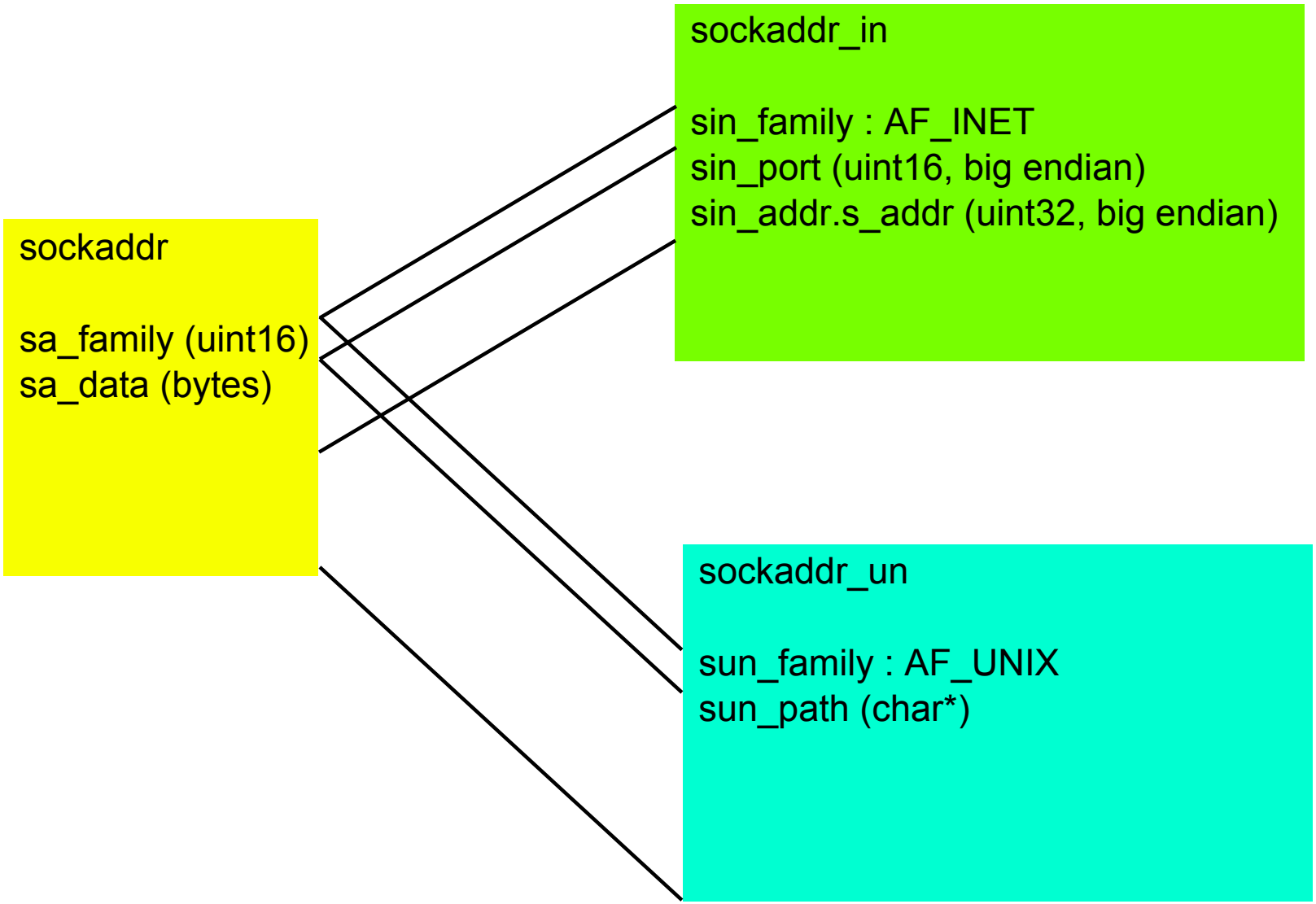
Implementierung Echo-Server (Forts.)

```
    while(1) {
        int client_sock = accept(sock,
                (struct sockaddr *) &client_address, &addrlen);
        if (client_sock<0) {
            printf("Fehler %d beim accept: %s\n",
                   errno, strerror(errno));
            exit(EXIT_FAILURE);
        }
        printf("Verbindung von %s, Port %d\n",
               inet_ntoa(client_address.sin_addr),
               ntohs(client_address.sin_port));
        sprintf(buf, "Bereit fuer Echo (quit fuer Ende)\n");
        send(client_sock, buf, strlen(buf), 0);
        do {
            int size = recv(client_sock, buf, MAX-1, 0);
            buf[size] = '\0';
            printf("%d Zeichen gelesen: +++%s+++\n", size, buf);
            send(client_sock, buf, strlen(buf), 0);
        } while (strcmp(buf,"quit"));
        close(client_sock);
    }
    return(EXIT_SUCCESS);
}
```

Besonderheiten bei Sockets

- IP-Adressen (32-Bit Ganzzahl) und die Portnummer (16-Bit Ganzzahl) sind in *Network Byte Order* (Big Endian), unabhängig von der Byte-Order des Host-Rechners. Zur Konvertierung existieren die Funktionen `ntohs`, `ntohl`, `htons` und `htonl` (d. h. auf Little-Endian-Systemen drehen diese Funktionen die Byte-Reihenfolge um).
- Die Struktur `sockaddr` ist generisch und besteht aus Angabe der Familie und Binärdaten, die für die Familie spezifische Daten enthalten. Zur Erleichterung existieren binär kompatible Strukturen wie `sockaddr_in` für IP-Sockets und `sockaddr_un` für Unix Domain Socket Files.

Schaubild


sockaddr_in
sin_family : AF_INET
sin_port (uint16, big endian)
sin_addr.s_addr (uint32, big endian)
sockaddr
sa_family (uint16)
sa_data (bytes)
sockaddr_un
sun_family : AF_UNIX
sun_path (char*)

Aufgaben des Clients

Die Aufgaben des Clients sind etwas einfacher. Theoretisch könnte auch für diesen ein `bind` ausgeführt werden, damit dieser einen festen Port verwendet, praktisch wird dieses i. A. weg gelassen, da dann automatisch ein freier Port ($>$ 1024) verwendet wird.
Somit muss also

- Ein Socket angelegt werden,
- eine Verbindung mit der IP des Server-PC und dem Port, auf dem der Server lauscht, angelegt werden.
- Anschließend findet dann die Kommunikation statt.

Implementierung des Clients

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define MAX 1024

int main() {
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock<0) {
        printf("Fehler %d beim Anlegen des sockets: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }
```

Implementierung des Clients

```
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(9999);
inet_aton("127.0.0.1", &address.sin_addr);

if (connect(sock, (struct sockaddr *) &address, sizeof(address))<0) {
    printf("Fehler %d beim connect: %s\n",
           errno, strerror(errno));
    exit(EXIT_FAILURE);
}

char buf[MAX];

int size = recv(sock, buf, MAX-1, 0);
buf[size] = '\0';
printf(buf);
```

Implementierung des Clients

```
    do {
        printf("Zeile eingeben: \n");
        gets(buf);
        send(sock, buf, strlen(buf), 0);
        printf("Gesendet wurde: +++%s+++\n", buf);
        int size = recv(sock, buf, MAX-1, 0);
        buf[size] = '\0';
        printf("Empfangen wurden %d Zeichen: +++%s+++\n", size, buf);
    } while (strcmp(buf,"quit"));
    close(sock);

    return(EXIT_SUCCESS);
}
```

Verwenden von Puffern

Sockets sind relativ normale Filedeskriptoren (mit Einschränkungen, so ist eine absolute Positionierung natürlich nicht möglich). Analog zu Pipes können Sockets als gepuffert werden.

Hierbei gilt als Faustregel, dass ein Socket niemals mit einem `FILE` gleichzeitig zum Lesen und Schreiben gepuffert werden darf. Daher ist folgender Code typisch:

```
FILE* in = fdopen(sock, "r");
FILE* out = fdopen(dup(sock), "w");
```

Beispiel-Client

Im folgenden Beispiel wird meine Webseite über eine gepufferte Verbindung angefordert und gelesen (und nebenbei demonstriert, wie Namen per DNS aufgelöst werden können):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define MAX 100

int main() {
    char * hostname = strdup("www.klaus-hoeppner.de");
    struct hostent * host = gethostbyname(hostname);
```

Beispiel-Client (Forts.)

```
if (host==NULL) {
    printf("Kann host nicht aufloesen\n");
    exit(EXIT_FAILURE);
}

struct in_addr * host_addr = (struct in_addr *) host->h_addr_list[0];
printf("IP: %s\n", inet_ntoa(*host_addr));

int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock<0) {
    printf("Fehler %d beim socket-Erzeugen: %s\n",
           errno, strerror(errno));
    exit(EXIT_FAILURE);
}

struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
address.sin_addr = *host_addr;
```

Beispiel-Client (Forts.)

```
    if (connect(sock, (struct sockaddr *) &address, sizeof(address))<0) {
        printf("Fehler %d beim connect: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }

    FILE* in = fdopen(sock, "r");
    FILE* out = fdopen(dup(sock), "w");
    fprintf(out, "GET / HTTP/1.1\r\nHost: %s\r\n\r\n", hostname);
    fflush(out);

    char buf[MAX];
    while (1) {
        if (fgets(buf, MAX-1, in)==NULL) break;
        printf(buf);
    }
    close(sock);
    return(EXIT_SUCCESS);
}
```

Parallele Server

Der bisherige Echo-Server kann nur einen Client gleichzeitig bedienen, d. h. während ein Client verbunden ist, müssen alle anderen warten.

Nun wird der Echo-Server so modifiziert, dass er für jeden von max. 5 Clients einen neuen Serverprozess per `fork` erzeugt. Dies entspricht z. B. dem üblichen Vorgehen beim Apache-HTTPD (wo zur Beschleunigung meist schon per *prefork* mehrere Server im Voraus angelegt werden).

Änderung an der Implementierung

```
void child_exited(int signum) {
    int pid, status;
    while ( (pid=wait(&status)) != -1) {
        printf("Kindprozess %d beendet\n", pid);
    }
}

// Anpassungen in main()

listen(sock, 5);
sethandler(SIGCHLD, child_exited);

char buf[MAX];
struct sockaddr_in client_address;
size_t addrlen = sizeof(client_address);
while(1) {
    int client_sock = accept(sock,
            (struct sockaddr *) &client_address, &addrlen);
    if (client_sock<0) {
        printf("Fehler %d beim accept: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }
```

Änderung an der Implementierung

```
    int child_pid = fork();
    if (child_pid==0) {
        close(sock);
        printf("Verbindung von %s, Port %d\n",
               inet_ntoa(client_address.sin_addr),
               ntohs(client_address.sin_port));
        sprintf(buf, "Bereit fuer Echo (quit fuer Ende)\n");
        send(client_sock, buf, strlen(buf), 0);
        do {
            int size = recv(client_sock, buf, MAX-1, 0);
            buf[size] = '\0';
            printf("%d Zeichen gelesen: +++%s+++\n", size, buf);
            send(client_sock, buf, strlen(buf), 0);
        } while (strcmp(buf,"quit"));
        close(client_sock);
        exit(EXIT_SUCCESS);
    }
    close(client_sock);
}
```

Verbindungslose Kommunikation mit UDP

Bei der verbindungslosen Kommunikation per UDP schickt ein Client Pakete an einen Server (oder an alle), ohne dass über Handshakes der Erfolg der Übertragung sicher gestellt ist.

Da so ein großer Teil des Overheads wegfällt, ist die Kommunikation so deutlich Ressourcen-schonender. Der Nachteil liegt natürlich darin, dass man sich auf die Kommunikation nicht verlassen kann. D. h. eine Anwendung muss so implementiert sein, dass sie robust dagegen ist, dass eine verschickte Nachricht nicht ankommt.

Beispiel

Im folgenden Beispiel sollen ein oder mehrere Clients verbindungslos den Namen und die Position eines Spielers an einen Server übertragen.

Hierfür wird in `position.h` folgende Struktur definiert:

```
#ifndef _POSITION_H
#define _POSITION_H

typedef struct {
    char name[40];
    int pos;
} position_t;

#endif
```

Implementierung Server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include "position.h"

#define MAX 1024

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

    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(9999);
    address.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sock, (struct sockaddr *) &address, sizeof(address));
```

Implementierung Server (Forts.)

```
    position_t pos;
    struct sockaddr_in client_address;
    size_t addrlen = sizeof(client_address);
    while(1) {
        recvfrom(sock, &pos, sizeof(pos),
                 0, (struct sockaddr *) &client_address, &addrlen);
        printf("Daten von %s, Port %d, Name %s, Position %d\n",
               inet_ntoa(client_address.sin_addr),
               ntohs(client_address.sin_port),
               pos.name, pos.pos);
    }

    return(EXIT_SUCCESS);
}
```

Implementierung Client

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include "position.h"

#define MAX 1024

int main(int argc, char** argv) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in client_address;
    memset(&client_address, 0, sizeof(client_address));
    client_address.sin_family = AF_INET;
    client_address.sin_port = htons(0);
    client_address.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sock, (struct sockaddr*) &client_address, sizeof(client_address))
```

Implementierung Client (Forts.)

```
    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(9999);
    inet_aton("127.0.0.1", &address.sin_addr);

    position_t pos;
    strcpy(pos.name, argv[1]);
    int x;
    for (x=0; x<10; x++) {
        pos.pos = x;
        sendto(sock, &pos, sizeof(pos),
               0, (struct sockaddr*) &address, sizeof(address));
        sleep(1);
    }
    close(sock);

    return(EXIT_SUCCESS);
}
```