

# Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2011/2012

RPMs

Socket-Programmierung

TCP

## RPMs: Grundlagen

Die meisten Linux-Distributionen benutzen das Format des *Redhat Package Managers* für die Installation und Deinstallation von Softwarepaketen (Alternative: *APT – Advanced Packaging Tool auf Debian & Co.*).

Was leistet RPM:

- Prüfen auf Abhängigkeiten/Konflikten mit anderen Paketen vor der Installation (z. B. wird die grafische CD-Brennoberfläche K3b nur installiert, wenn auch die eigentlichen (Kommandozeilen-)Brennprogramme installiert sind),
- Installation und Deinstallation von Paketen (mit Vor- und Nacharbeiten, z. B. Ausführen von `ldconfig`),
- Nachverfolgen, welche Dateien zu welchem Software-Paket gehören, Prüfung auf Integrität,
- Update von Paketen

## Das Programm rpm

Auf der Kommandozeile wird mit dem Befehl `rpm` gearbeitet:

`rpm -i dateiname.rpm` zum Installieren eines RPM (falls die Abhängigkeiten erfüllt sind).

`rpm -e paket` zum Deinstallieren eines Paketes.

`rpm -U dateiname.rpm` installiert die angegebene RPM-Datei, ältere Versionen (so vorhanden) werden deinstalliert.

`rpm -q paket` zeigt an, ob und in welcher Version das Paket installiert ist.

`rpm -qi paket` gibt Informationen zum Paket aus.

`rpm -ql paket`, `rpm -qd paket`, `rpm -qc paket`

Liste aller zum Paket gehörenden Dateien bzw. Liste der Dokumentations- und Konfigurationsdateien.

`rpm -V paket` zum Prüfen der Integrität des Pakets.

## Bauen von RPMs

Zum Erstellen von RPMs für ein Projekt benötigt man

- die Quellen des Projekts
- evtl. Patches für die Anpassung an die Distribution/Architektur
- eine *SPEC*-Datei mit den Anweisungen, wie der RPM erzeugt werden soll.

Der RPM-Build erfordert eine bestimmte Dateistruktur:

```
topdir/SPECS  
    /SOURCES  
    /BUILD  
    /RPMS  
    /SRPMS
```

wobei das Toplevelverzeichnis in `~/ .rpmmacros` definiert wird:

```
%_topdir /home/klaus/rpms
```

# Einfache SPEC-Datei

Summary: Demon to ping syslog messages

Name: logdaemon

Version: 1.0

Release: 1

License: Public Domain

Group: System/Services

Prefix: /usr

Source: logdaemon-1.0.tar.gz

Packager: Santa Claus <sclaus@northpole.com>

BuildRoot: /tmp/logdaemon-1.0-root

%description

This is just a test RPM

%prep

%setup

## Einfache SPEC-Datei (Forts.)

```
%build
```

```
./configure --prefix=%{_prefix} --sysconfdir=%{_sysconfdir}  
make all
```

```
%install
```

```
make install DESTDIR=${RPM_BUILD_ROOT}
```

```
%files
```

```
%doc README
```

```
%{_sbindir}/log
```

```
%{_initrddir}/log
```

## Anatomie der SPEC-Datei

Wie man sieht, besteht die SPEC-Datei aus verschiedenen Abschnitten:

- %prep** Auspacken der Quellen, evtl. Anwenden von Patches
- %build** Kompilieren des Projektes
- %install** Installieren der Dateien (unterhalb von *BuildRoot*)
- %files** Dateien, die in den RPM übernommen werden sollen (ggfs. mit Markierung als Dokumentation oder Konfigurationsdatei)

Dabei kann eine SPEC mehrere RPM-Pakete definieren, in die die Dateien aufgesplittet werden (z. B. Trennung Programm und Headerdateien). Befehle vor oder nach der (De-)Installation können in den Abschnitten **%pre**, **%post** bzw. **%preun**, **%postun** definiert werden.

## Aufruf von `rpmbuild`

Der oder die RPMs werden dann mit `rpmbuild` erzeugt:

- bb Erzeugen des (binary) RPMs,
- bs erzeugen des Source-RPMs (SRPM),
- ba erzeugen von allen beiden.
- bp Nach `%prep` aufhören,
- bc nach `%build` aufhören,
- bi nach `%install` aufhören.

`--short-circuit` sorgt in Verbindung mit `-bc` bzw. `-bi` dafür, dass die Schritte *vorher* übersprungen werden.

## Netzwerkschichten

Die Kommunikation über Netzwerk besteht aus einem komplexen Zusammenspiel von

- Hardware: Netzwerkkarte im Rechner, Switches/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);
```

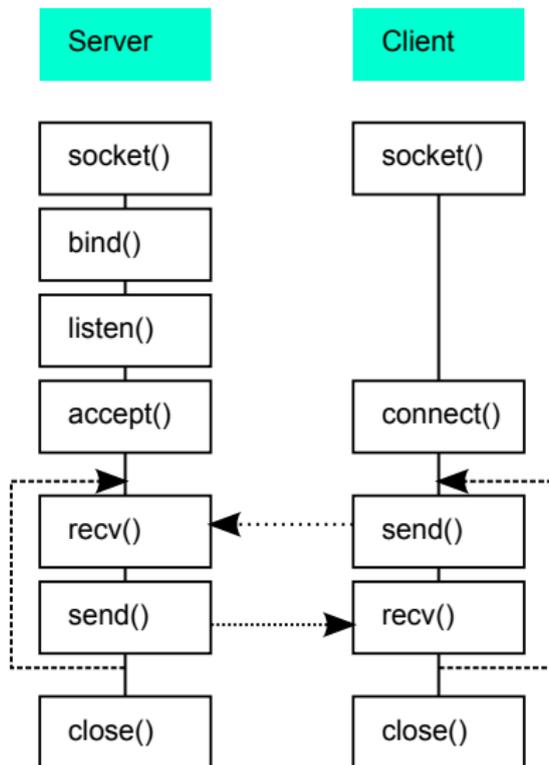
# Client-Server

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);
```

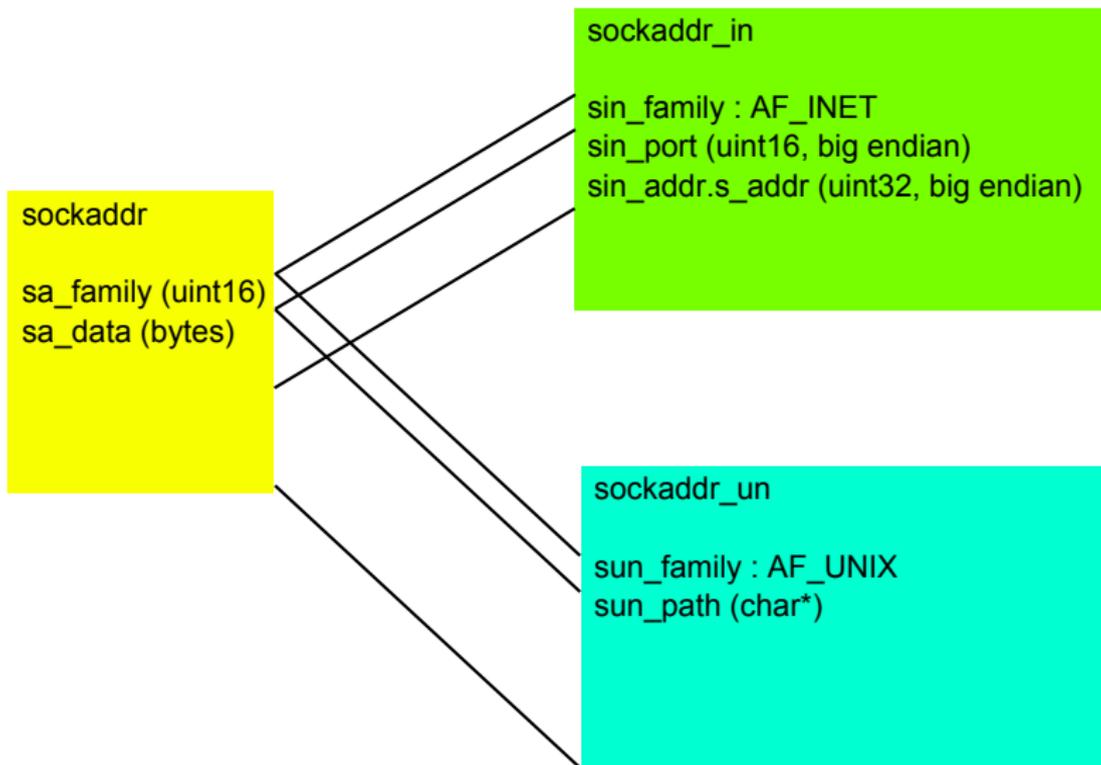
## Implementierung Echo-Server (Forts.)

```
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



## 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. weggelassen, 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);
}
```