Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2012/2013

Grundlagen Netzwerk

Für Netzwerke wird heutzutage meist noch IPv4 benutzt, bei der einer Netzwerkschnittstelle eine IP-Adresse zugeordnet wird, die aus vier Bytes besteht. Der Übergang zu IPv6 findet derzeit statt, aufgrund des allmählich knapp werdenden Adress-Pools aber überraschend langsam.

Hierbei gibt es einige besondere Adressbereiche:

127.0.0.0/8 bezieht sich auf das Loopback-Interface. Dies ist keine reale Netzwerkkarte, sondern eine Rechner-interne Kommunikation.

10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 sind private IP-Bereiche, insbesondere für Intranets. Private IP-Adressen werden nicht innerhalb des öffentlichen Internets geroutet und sind daher unsichtbar. Die Verbindung zum Internet erfolgt über Proxys bzw. per NAT (*Network Address Translation*).

Subnetze und Gateways

Neben der IP-Adresse ist für eine Netzwerkschnittstelle noch bedeutsam, über welchen IP-Bereich sich das Subnetz erstreckt, in dem andere Netzwerkendpunkte direkt erreicht werden. Dies sind Rechner, bei denen ein Bit-And von IP mit Netmask zum selben Wert führen.

Beispiele:

- IP 192.168.129.27, Netmask 255.255.255.0 (Subnet 192.168.129.0/24): Rechner im IP-Bereich 192.168.129.0–255 sind im selben Subnet.
- IP 10.10.103.123, Netmask 255.255.240.0 (Subnet 10.10.96.0/20): Rechner im IP-Bereich 10.10.96–111.0–255 sind im selben Subnet.

Konfiguration der Netzwerkschnittstelle per Hand

- Konfiguration des Loopback-Interfaces:
  `ip addr add 127.0.0.1/8 dev lo`
  oder
  `ifconfig lo 127.0.0.1 netmask 255.0.0.0`
- Konfiguration der Netzwerkkarte
  `ip addr add 10.10.103.123/20`
  `broadcast 10.10.111.255 dev eth0`
  oder
  `ifconfig eth0 10.10.103.123`
  `netmask 255.255.240.0 broadcast 10.10.111.255`

Hinzufügen der Routen

Nun müssen noch die Routen hinzugefügt werden:

- Verbindungen nach 127.0.01/8 über das Loopback-Interface
- Verbindungen nach 10.10.96.0/20 direkt über eth0
- Verbindungen nach außerhalb über das Gateway 10.10.96.1

```
route add -net 127.0.0.0 netmask 255.0.0.0 dev lo
route add -net 10.10.96.0 netmask 255.255.240.0 dev eth0
route add default gw 10.10.96.1
```

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"
```

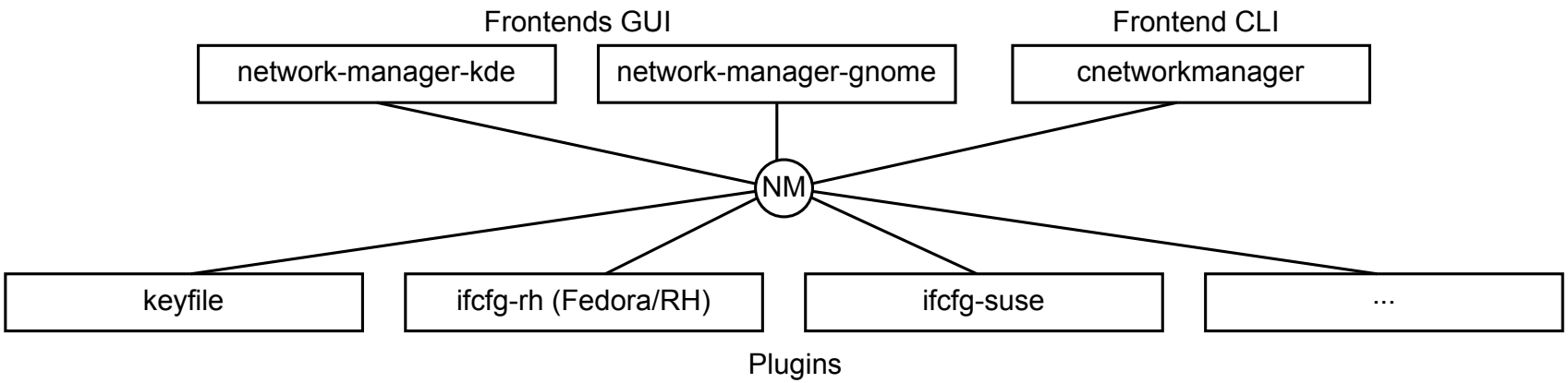
Die Zukunft: NetworkManager

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

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);
}
```

Unix Domain Sockets

Ein Socket muss nicht zwingend eine Kommunikation über Netzwerk beschreiben.

Eine bereits lange verbreitete Methode besteht in dem Verwenden von *Unix Domain Sockets*, bei denen die Kommunikation über eine Datei stattfindet. Dies kann also auf einem Rechner zur Inter-Prozess-Kommunikation benutzt werden.

Der Umstieg von IP-Kommunikation zu Socket-Files ist einfach:

- Änderung der Familie von `AF_INET` nach `AF_UNIX`
- Aufbau einer Struktur des Typs `sockaddr_un` statt `sockaddr_un`.

Grundgerüst Server

```
#include <sys/socket.h>
#include <sys/un.h>
...
int main() {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un address;
    memset(&address, 0, sizeof(address));
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "/tmp/myecho.sock");

    bind(sock, (struct sockaddr *) &address, sizeof(address));
    listen(sock, 1);

    struct sockaddr_un client_address;
    size_t addrlen = sizeof(client_address);
    while(1) {
        int client_sock = accept(sock,
                (struct sockaddr *) &client_address, &addrlen);
        ...
        close(client_sock);
    }
    return(EXIT_SUCCESS);
}
```

Grundgerüst Client

```
#include <sys/socket.h>
#include <sys/un.h>
...
int main() {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un address;
    memset(&address, 0, sizeof(address));
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "/tmp/myecho.sock");

    connect(sock, (struct sockaddr *) &address, sizeof(address));

    ...

    close(sock);

    return(EXIT_SUCCESS);
}
```