Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 16

Kern-Technik

Verbindungslose Kommunikation über UDP ist im Kernel mit weniger Aufwand zu programmieren als das verbindungsorientierte TCP. Eine geschickte Einstellung der Socket-Parameter lässt eigene Module auf Broadcasts hören und steigert die Performance. Eva-Katharina Kunst, Jürgen Quade

Nachdem die letzte Folge der Kern-Technik die Verwendung von TCP im Kernel demonstriert hat [1], geht es dieses Mal um UDP. Dabei handelt es sich um ein verbindungsloses Protokoll, was sich bereits am einfachen Programmierinterface bemerkbar macht: Der Server öffnet einen Socket, bindet ihn an einen Port und wartet auf die an ihn geschickten Nachrichten. Kommt eine an, nutzt der Server die übermittelte Absenderinformation, um dem Client zu antworten. Dazu braucht er nicht einmal einen neuen Socket anzulegen.

Der Client öffnet ebenfalls einen Socket und schickt seine Nachricht an den angegebenen Empfänger, also den Server. Dessen Identifikation erfolgt wie bei TCP über die Kombination von IP-Adresse und Port. Nicht mehr benötigte Sockets gibt das Modul schließlich wieder frei. Innerhalb des Kernels stehen für UDP-Kommunikation insgesamt fünf Funktionen zur Verfügung (siehe Abbildung 1): »sock_create()«, »bind()«, »sock_recvmsg()«, »sock_sendmsg()« und »sock_release()«.

Listing 1 zeigt, wie diese Funktionen zu verwenden sind. In Zeile 56 erzeugt »sock_create()« den Socket, wobei die Symbole »SOCK_DGRAM« und »IPPROTO_UDP« festlegen, dass es sich um einen UDP-Socket handelt. Den in der Struktur »struct sockaddr_in« spezifizierten Empfangsport bindet die Funktion »bind()« an den Socket (Zeile 63).

Etwas komplizierter ist der Aufruf der Empfangsfunktion selbst. Sie benötigt neben dem Speicher für die Daten nämlich noch drei Datenstrukturen (siehe Abbildung 2): eine Struktur »struct iovec« für die Adresse und Größe des Empfangspuffers, eine Struktur vom Typ »struct sockaddr_in« für den Paketabsender (IP-Adresse und Portnummer) und eine Struktur »struct msghdr msg«. Sie speichert die Adressen der anderen Datenstrukturen (Zeilen 25 und 29). Zur Initialisierung der Datenstruktur »struct iovec« müssen die Adresse und Größe des Empfangs-Datenspeichers zudem spezifiziert sein (Zeilen 32 und 33).

Wiederholt initialisieren

Den Inhalt der Datenstruktur »struct iovec« modifiziert der Kernel. Deshalb ist sie vor jedem Aufruf von »sock_recvmsg()« neu zu initialisieren. Das ist auch der Grund, warum die Initialisierung in Listing 1 innerhalb der »while«-Schleife (Zeile 31) stattfindet.

Zum Aufruf von »sock_recvmsg()« noch zwei Bemerkungen: Erstens braucht die Funktion einen Prozesskontext. Deshalb erzeugt Listing 1 einen Kernel-Thread (Zeile 70). Zweitens muss dafür gesorgt sein, dass man in den vorgesehenen Speicher schreiben darf. Da dieser im Kernelspace liegt, die Kopierfunktion aber vom Userspace ausgeht, ist die entsprechende Überprüfung auszuschalten. Dafür sorgt der Code der Zeilen 34 und 35. Der Aufruf »set_fs()« in Zeile 37 stellt den alten Wert wieder her.

Der Client, bitte

Ein UDP-Paket vom Kernel aus senden ist einfacher, als eins zu empfangen. Listing 3 zeigt dazu ein Kernelmodul, das die Aufgabe des vorgestellten Client-Programms übernimmt. Um das Modul kompakt zu halten, verschickt es das UDP-Paket gleich bei der Initialisierung. Der Code kann allerdings auch an fast jeder anderen Stelle im eigenen Kernelcode auftauchen. Zunächst erzeugt »sock_create()« den UDP-Socket, siehe Listing 3, Zeile 19. Dann folgt die Definition des Paketempfängers in der Struktur »struct sockaddr_in«.

Wie beim Empfang enthält auch beim Senden die Struktur »struct msghdr« alle wesentlichen Parameter (Abbildung 2). Sie besteht im Wesentlichen aus der Netzadresse des Empfängers (»struct sockaddr_in«) und der Speicheradresse des Sendepuffers (definiert über eine »struct iovec«). Das Feld »msg_flags« muss vor dem Senden auf »0« stehen.

Eine andere Variante, den Empfänger eines Paketes zu spezifizieren, macht die

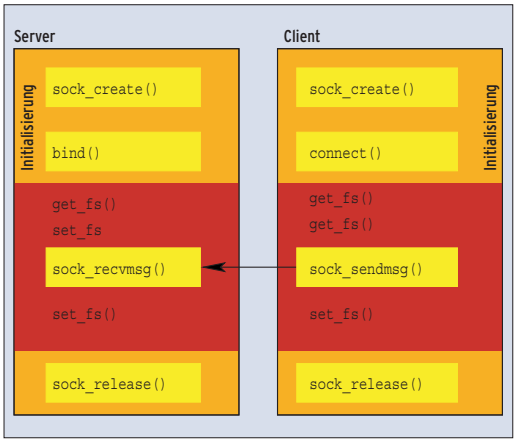

Abbildung 1: Zur Kommunikation über UDP bringt der Kernel sechs Netzwerk- und zwei Hilfsfunktionen mit.

Compile und Test

Der Code von Listing 1 lässt sich auf einem System mit Kernel 2.6 mit Hilfe des Makefile [2] übersetzen. Der Befehl »insmod udprcv.ko« lädt das resultierende Kernelmodul. Jetzt muss der Programmierer noch eine UDP-Nachricht an den Port 5555 des Rechners schicken, auf dem er das Modul geladen hat. Im Syslog dieses Rechners erfolgt dann eine Ausgabe, die die Nachricht selbst und den Absender enthält. Wichtig ist, auf diesem Wege nur Ascii-Strings zu schicken, da das Programm nicht überprüft, ob es sich um Binärdaten handelt.

Vom Userspace testen

Für einen Test eignet sich beispielsweise die Applikation aus Listing 2. Sie sendet ein einfaches „Hallo vom User-Space" an Port 5555 auf dem eigenen Rechner (IP-Adresse »localhost«, »127.0.0.1«). Die Testapplikation lässt sich jetzt am einfachsten durch den Aufruf von »make apudpsend« generieren. Es handelt sich ja um eine normale Applikation, sodass kein besonderes Makefile benötigt wird. Vor dem Aufruf der Applikation muss noch sichergestellt sein, dass der UDP-Port 5555 nicht durch eine Firewall blockiert ist.

Software zum Zeitpunkt des Verschickens besonders performant: Der Eintrag der Adresse der »struct sockaddr_in«, die die Empfänger-IP-Adresse und Empfänger-Portnummer enthält, erfolgt nicht in der »struct msghdr«. Stattdessen bindet sie der Aufruf der Methode »connect()« an den Socket. Das Feld »msg_name« setzt man dann vor dem Aufruf von »sock_sendmsg()« auf »0«, siehe Listing 4. Um Missverständnissen vorzubeugen: UDP ist und bleibt ein verbindungsloses Protokoll.

Die Methode »connect()« bindet in diesem Fall nur die Adresse des Empfängers an den Port und führt keinen wirklichen Verbindungsaufbau durch. Auch dieses Modul lässt sich mit Hilfe des Makefile [2] übersetzen und mit »insmod udpsend.ko« zusätzlich zum bereits geladenen Modul laden. Bei Erfolg erscheint im Syslog eine Meldung über den Empfang des gesendeten »hallo«. Natürlich ist es möglich, das Modul auch auf einem an-

Listing 1: UDP-Server »udprcv.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/in.h>
04 #include <net/sock.h>
05
06 #define SERVER_PORT 5555
07 static struct socket *udpsocket=NULL;
08 static DECLARE_COMPLETION( threadcomplete );
09 static int com_thread_pid;
10
11 static int com_thread( void *data )
12 {
13   struct sockaddr_in client;
14   struct sockaddr *address;
15   unsigned char buffer[100];
16   int len;
17   struct msghdr msg;
18   struct iovec iov;
19   mm_segment_t oldfs;
20
21   daemonize( "udpserver" );
22   allow_signal( SIGTERM );
23   if( udpsocket->sk==NULL )
24     return 0;
25   msg.msg_name = &client;
26   msg.msg_namelen = sizeof( struct
       sockaddr_in );
27   msg.msg_control = NULL;
28   msg.msg_controllen = 0;
29   msg.msg_iov     = &iov;
30   msg.msg_iovlen = 1;
31   while( !signal_pending(current) ) {
32     iov.iov_base = buffer;
33     iov.iov_len  = sizeof(buffer);
34     oldfs = get_fs();
35     set_fs( KERNEL_DS );
36     len = sock_recvmsg( udpsocket, &msg,
          sizeof(buffer), 0 );
37     set_fs( oldfs );
38     if( len>0 ) {
39       address = (struct sockaddr *)&client;
40       printk(KERN_INFO "msg \"%s\" from
            %u.%u.%u.%u\n", buffer,
41         (unsigned char)address->sa_data[2],
42         (unsigned char)address->sa_data[3],
43         (unsigned char)address->sa_data[4],
44         (unsigned char)address->sa_data[5] );
45     }
46   }
47   complete( &threadcomplete );
48   return 0;
49 }
50
51 static int __init server_init( void )
52 {
53   struct sockaddr_in server;
54   int servererror;
55
56   if( sock_create( PF_INET,SOCK_DGRAM,
         IPPROTO_UDP,&udpsocket)<0 ) {
57     printk( KERN_ERR "server: Error creating
          udpsocket.\n" );
58     return -EIO;
59   }
60   server.sin_family = AF_INET;
61   server.sin_addr.s_addr = INADDR_ANY;
62   server.sin_port = htons( (unsigned
       short)SERVER_PORT );
63   servererror = udpsocket->ops->bind(
        udpsocket,
64     (struct sockaddr *) &server, sizeof(
       server ) );
65   if( servererror ) {
66     sock_release( udpsocket );
67     return -EIO;
68   }
69
70   com_thread_pid=kernel_
      thread(com_thread,NULL, CLONE_KERNEL );
71   if( com_thread_pid < 0 ) {
72     sock_release( udpsocket );
73     return -EIO;
74   }
75   return 0;
76 }
77
78 static void __exit server_exit( void )
79 {
80   if( com_thread_pid ) {
81     kill_proc( com_thread_pid, SIGTERM, 0 );
82     wait_for_completion( &threadcomplete );
83   }
84   if( udpsocket )
85     sock_release( udpsocket );
86 }
87
88 module_init( server_init );
89 module_exit( server_exit );
90 MODULE_LICENSE("GPL");
```

deren Rechner zu installieren. Dann ist jedoch an Stelle von »127.0.0.1« die richtige IP-Adresse des Empfängers im Sourcecode einzutragen.

Socket-Parameter

Die höheren Ligen der Netzwerk-Programmierung beginnen, wenn beispielsweise mehrere Applikationen gleichzeitig auf ein und demselben Port warten oder wenn es um den Empfang von Multicast- oder Broadcast-Nachrichten geht. Der Applikationsprogrammierer weiß: Mit Hilfe von »setsockopt()« und »getsockopt()« lassen sich die dafür notwendigen Parameter einstellen.

Innerhalb des Kernels stehen entsprechende Funktionen zur Verfügung: »sock_setsockopt()« und »sock_getsockopt()«. Diese Funktionen erwarten ihr Argument im Userspace. Die Verwendung der Makros »set_fs()« und »get_fs()« – falls die Funktionen direkt im Kernel aufgerufen werden – ist damit obligatorisch (siehe [3]).

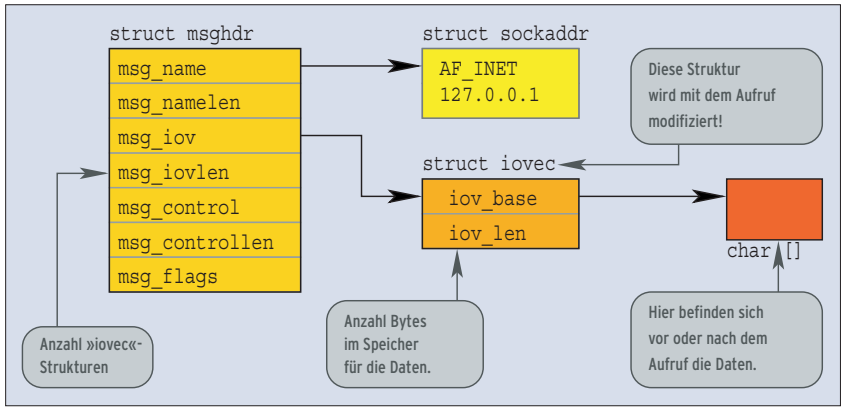

Abbildung 2: Die Datenstruktur vom Typ »struct msghdr« speichert die wesentlichen Parameter für das Senden und Empfangen der UDP-Pakete.

Allerdings gibt es für viele Optionen eine effizientere Variante. Vielfach wird nämlich mit dem Aufruf von »sock_setsockopt()« nur ein Element oder auch nur ein Flag der Socket-Datenstruktur gesetzt oder gelöscht. Ein Element liest oder schreibt man im Kernel aber besser direkt. Für das Setzen eines Flags kann der Programmierer auf die Inline-Funktionen (definiert in »net/sock.h«) »sock_set_flag()«, »sock_reset_flag()« und »sock_valbool_flag()« zurückgreifen.

Der bei diesen Makros verwendete erste Parameter »struct sock« ist Teil von

Listing 2: »apudpsend.c«

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <string.h>
05 #include <sys/types.h>
06 #include <sys/socket.h>
07 #include <netinet/in.h>
08 #include <arpa/inet.h>
09
10 int main( int argc, char **argv )
11 {
12   struct sockaddr_in to;
13   char *buf;
14   int sd, ret;
15
16   sd = socket( AF_INET, SOCK_DGRAM, 0 );
17   if( sd <= 0 ) {
18     perror( "socket" );
19     exit( -1 );
20   }
21   bzero( &to, sizeof(to) );
22   to.sin_family = AF_INET;
23   inet_aton( "127.0.0.1", &to.sin_addr );
24   to.sin_port = htons( (unsigned short)5555 );
25   buf = "Hallo vom User-Space";
26   ret = sendto( sd, buf, strlen(buf)+1, 0,
        (struct sockaddr *)&to,
27                 sizeof(to) );
28   close( sd );
29   return 0;
30 }
```

Listing 3: UDP-Client »udpsend.c«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/in.h>
04 #include <linux/inet.h>
05 #include <net/sock.h>
06
07 #define SERVERPORT 5555
08 static struct socket *clientsocket=NULL;
09
10 static int __init client_init( void )
11 {
12   int len;
13   char buf[64];
14   struct msghdr msg;
15   struct iovec iov;
16   mm_segment_t oldfs;
17   struct sockaddr_in to;
18
19   if( sock_create( PF_INET,SOCK_DGRAM,
       IPPROTO_UDP,&clientsocket)<0 ) {
20     printk( KERN_ERR "server: Error creating
          clientsocket.\n" );
21     return -EIO;
22   }
23   to.sin_family = AF_INET;
24   to.sin_addr.s_addr = in_aton( "127.0.0.1" );
       /* destination address */
25   to.sin_port = htons( (unsigned short)
       SERVERPORT );
26
27   msg.msg_name = &to;
28   msg.msg_namelen = sizeof(to);
29   memcpy( buf, "hallo", 6 );
30   iov.iov_base = buf;
31   iov.iov_len  = 6;
32   msg.msg_control = NULL;
33   msg.msg_controllen = 0;
34   msg.msg_iov    = &iov;
35   msg.msg_iovlen = 1;
36
37   oldfs = get_fs();
38   set_fs( KERNEL_DS );
39   len = sock_sendmsg( clientsocket, &msg, 6 );
40   set_fs( oldfs );
41   if( len < 0 )
42     printk( KERN_ERR "sock_sendmsg returned:
         %d\n", len);
43   return 0;
44 }
45
46 static void __exit client_exit( void )
47 {
48   if( clientsocket )
49     sock_release( clientsocket );
50 }
51
52 module_init( client_init );
53 module_exit( client_exit );
54 MODULE_LICENSE("GPL");
```

»struct socket«, siehe »net.h«. Um beispielsweise mehrere Kernel-Threads gleichzeitig auf einem Socket warten zu lassen, ist nach »sock_create()« folgender Code einzufügen:

```
oldfs = get_fs();
set_fs(KERNEL_DS);
optval = 1;
sock_setsockopt(sock,SOL_SOCKET, ➥
  SO_REUSEADDR,&optval,sizeof(int));
set_fs(oldfs);
```

Alternativ erfüllt eine einzige Zeile denselben Zweck:

```
socket->sk->sk_reuse = 1;
```

Um Broadcast-Pakete senden oder empfangen zu können, sind noch Optionen zu setzen:

```
oldfs = get_fs();
set_fs(KERNEL_DS);
optval = 1;
sock_setsockopt(sock,SOL_SOCKET, ➥
  SO_BROADCAST,&optval,sizeof(int));
set_fs(oldfs);
```

Als Alternative für diesen Fall kommt noch in Frage:

```
sock_valbool_flag(sock, SOCK_BROADCAST, 1);
```

Damit nimmt der Kernel auch Pakete mit Broadcast-Adressen an.

Vorschau

Mit den vorgestellten Techniken lassen sich beliebige, verbindungslose Netzwerkfunktionen im Kernel programmieren, ob als Server oder Client. Die nächste Folge der Kern-Technik-Reihe beschäftigt sich mit der darunter liegenden Schicht des Netzwerk-Stacks und erklärt, wie Treiber für Netzwerkadapter aufgebaut sind. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 15: Linux-Magazin 10/04, S. 114.

[2] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/11/Kern-Technik]

[3] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik“, Folge 14: Linux-Magazin 9/04, S. 92

[4] Quellcode zur Funktion »sock_setsockopt()«: [http://lxr.linux.no/source/net/core/sock.c?v=2.6.8.1#L183]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Ihr gemeinsames Buch zum Kernel 2.6 heißt „Linux Treiber entwickeln“.

Listing 4: Optimierung

```
01 to.sin_family = AF_INET;
02 to.sin_addr.s_addr = in_aton( "127.0.0.1" );
03 to.sin_port = htons( (unsigned short)serverport );
04 errorcode=clientsocket->ops->connect(clientsocket,
     (struct sockaddr *)&to,sizeof(to),0);
05 msg.msg_name = NULL;
```

TCP/UDP/IP-Schnittstellenfunktionen im Kernel

Socket-Parametrierung	

`int sock_setsockopt(struct socket ➥ *sock, int level, int optname, char ➥ __user *optval, int optlen)`
 	
`int sock_getsockopt(struct socket ➥ *sock, int level, int optname, char ➥ __user *optval, int __user *optlen)`
	Mit der Funktion »sock_setsockopt()« wird der Parameter »optname« des Socket »sock« mit dem Wert »optval« belegt, dessen Größe »optlen« festlegt. Die Funktion »sock_getsockopt()« liest den Parameter »optname« des Socket »sock« aus, wobei der Parameterwert in »optval« steht. Bei Aufruf der Funktion enthält die Variable »optlen« (deren Adresse übergeben wird) die Größe des mit »optval« zur Verfügung stehenden Speicherbereichs. Nach dem Aufruf ist »optlen« mit der Größe der kopierten Daten belegt.
	Der Kernel erwartet, dass »optval« und »optlen« im Userspace liegen. Wer die Funktionen innerhalb des Kernels aufruft, muss die Makros »get_fs()« und »set_fs()« einsetzen. Der Parameter »level« bestimmt die Protokollebene. Alle Parameter beziehen sich auf die Ebene »SOL_SOCKET«. Die Funktionen entsprechen den Applikationsfunktionen »setsockopt()« und »getsockopt()« (siehe dazu »man setsockopt«).
	Die verfügbaren Parameter lassen sich am einfachsten der Datei »/usr/src/linux/net/core/sock.c« entnehmen. Oft ist es effizienter, an Stelle der Funktionen »sock_setsockopt()« und »sock_getsockopt()« direkt die jeweils zugehörigen Codefragmente zu verwenden (siehe [4]). Im fehlerfreien Fall geben die Funktionen »0« zurück, sonst einen negativen Fehlercode.

`inline void sock_set_flag(struct sock ➥ *sk, enum sock_flags flag)`
 	
`inline void sock_reset_flag(struct ➥ sock *sk, enum sock_flags flag)`
 	
`inline void sock_valbool_flag(struct ➥ sock *sk, int bit, int valbool)`
	Die in der Headerdatei »net/sock.h« definierten Inline-Funktionen dienen zum Setzen beziehungsweise Zurücksetzen des Flag »flag« von »sk«. Mit diesen Funktionen sind die Flags »SOCK_DEAD«, »SOCK_DONE«, »SOCK_URGINLINE«, »SOCK_LINGER«, »SOCK_DESTROY«, »SOCK_ BROADCAST« und »SOCK_TIMESTAMP« ansprechbar. Mit der Inline-Funktion »sock_ valbool_flag« wird das Flag »bit« zurückgesetzt, falls »valbool« »0« ist, ansonsten wird »bit« gesetzt.

`inline int sock_flag(struct sock ➥ *sk, enum sock_flags flag)`	Die in der Headerdatei »net/sock.h« definierte Inline-Funktion liest das Flag »flag« des Sockets »sk« aus und gibt das Ergebnis der aufrufenden Funktion zurück.
Methoden des Socket-Objekts	
`int (*connect)(struct socket *sock,➥ struct sockaddr *vaddr, int ➥ sockaddr_len, int flags)`	Im Fall eines verbindungslosen Protokolls (»SOCK_DGRAM«, UDP) wird die Empfängeradresse »vaddr« an den Socket »sock« gebunden. Ist bei einem nachfolgenden »sock_sendmsg()« das Element »msg_name« der Struktur »struct msghdr« auf »NULL« gesetzt, wird das Paket an den Empfänger »to« geschickt. Bei einem nachfolgenden »sock_recvmsg()« werden nur Pakete vom angegebenen Absender »vaddr« akzeptiert. Im Fall eines verbindungsorientierten Protokolls (»SOCK_STREAM«, TCP) wird eine Verbindung zu »vaddr« aufgebaut. Ist »flags« nicht auf »O_NONBLOCK« eingestellt, ist der aufrufende Thread so lange blockiert, bis entweder die Verbindung hergestellt oder ein Fehler aufgetreten ist. Konnte »vaddr« erfolgreich an den Socket gebunden oder eine Verbindung aufgebaut werden, gibt die Funktion »0« zurück, sonst einen negativen Fehlercode.