A hálózati környezet beállítása RTNETLINK-kel

Az RTNETLINK felhasználása a hálózati beállításokat befolyásoló alkalmazások fejlesztésére.

A *Linux* felhasználó terében futó alkalmazások a *NETLINK* segítségével kommunikálhatnak a rendszermaggal. A *NETLINK* a standard *socket* implementációjának kiterjesztése. Használatával üzeneteket válthatunk a rendszermag különféle komponenseivel, pl. a hálózatkezelő résszel, és befolyásolhatjuk is azok működését.
A cikkben azt fogom bemutatni, hogyan használható az *RTNETLINK*, a *NETLINK* hálózatkezelésért felelős része a hálózati környezet programozására. Áttekintjük az *RTNETLINK* leghasznosabb funkcióit, a releváns socketkezelő függvényeket, az *RTNETLINK* üzenetei összeállításának módját és végül néhány példakódot.
Az *RTNETLINK IPv4*-es részének neve *NETLINK_ROUTE*, az *IPv6*-os részé pedig *NETLINK_ROUTE6*.
A cikkben szereplő leírások mindkét verzióra érvényesek.
A hálózatiprotokoll-kezeléssel foglalkozó fejlesztők az *RTNETLINK* segítségével különféle hálózati komponenseket módosíthatnak és figyelhetnek meg, pl. az útválasztótáblákat és a hálózati interfészeket. Az internettechnológiák szabványosítását végző *IETF* (*Internet Engineering Task Force*) sok olyan specifikációt készített és készít, amely megvalósítható a felhasználói térben. Ezek többnyire megkövetelik az útválasztás módosítását és az értesülést a más folyamatok által eszközölt változtatásokról. Ezen protokollok egy részét a következő csoportokba sorolhatjuk:

- *Dinamikus útválasztó protokollok* – Ebbe a családba tartozik többek között a *Routing Information Protocol (RIP)*, az *Open Shortest Path First (OSPF)* és az *Exterior Gateway Protocol (EGP)*. Az útválasztáshoz szükséges információkat aktívan kezelik, miközben velük egyenrangú gépekkel és útválasztókkal kommunikálnak egy hálózatban vagy az Interneten.
- *Mobilitásprotokollok* – A mobil gépek különböző időpontokban különböző hálózatokhoz csatlakoznak, például a *Mobile IP (MIP)*, a *Session Initiation Protocol (SIP)* és a *Network Mobility (NEMO)* protokoll alkalmazásával, ami lehetővé teszi az útvonalválasztási információk folyamatos karbantartását és a kommunikáció folytonosságának biztosítását.
- *Ad hoc hálózati protokollok* – Ha nincs kiépített mobil hálózati infrastruktúra, vagyis pl. nincsenek útválasztók és *WLAN* hozzáférési pontok, a mobil eszközök közvetlenül, egyenrangú (*peer-to peer*) módon kapcsolódhatnak egymáshoz, miközben beállításaik eltérők. Egy földrengés súlytotta környéken vagy más vészhelyzetben praktikus lehet az ilyen protokollok, például az *Ad hoc On-demand Distance Vector (AODV)* vagy az *Optimized Link State Routing (OLSR)* használata, melyek megkövetelik az útválasztási információ karbantartását a más gépekkel való kommunikációhoz, miközben a szomszédos eszközöket útválasztóként vagy átjáróként használják.

Ezeknek a protokolloknak a felhasználói térben való megvalósítása segít a rendszermag kódjának túlzottan bonyolulttá válását elkerülni. Emellett egyszerűbb a fejlesztés és a tesztelés is, mivel számos fejlesztőeszköz áll rendelkezésünkre a felhasználói térben történő programozáshoz, és a rendszermag összeomlásától sem kell tartani a tesztelés vagy a végső felhasználás során.

Socketműveletek

A *socketek* lehetővé teszik két végpont kommunikációját, a programozói interfész standard adatstruktúra- és függvénykészletével. Az *RTNETLINK* esetében az egyik végpont a felhasználói térben, a másik a rendszermagban van. A hálózati környezet *RTNETLINK*-kel történő befolyásolásához a következő függvényhívássorozatra van szükség:

1. Socket megnyitása
2. A socket helyi címhez kötése (folyamatazonosító felhasználásával)
3. Üzenet küldése a másik végpontnak
4. Üzenet fogadása a másik végponttól
5. Socket bezárása

A `socket()` függvény megnyit egy végpontot, amely egyelőre sehová sem csatlakozik. A függvény prototípusa:

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

A `domain` adja meg a *socket* típusát, amely az *RTNETLINK* esetén *AF_NETLINK* (`PF_NETLINK`). A `type` a protokoll típusát határozza meg, lehet egyszerű (`SOCK_RAW`) vagy *datagram* (`SOCK_DGRAM`). A típus az *RTNETLINK-socket* szempontjából

irreleváns, bármelyiket megadhatjuk. A `protocol` határozza meg a socket ***NETLINK*** mivoltát, ami esetünkben a `NETLINK_ROUTE`. Ha sikerrel járt, a függvény pozitív egész számmal, a socketleíróval tér vissza, amit minden további ***RTNETLINK*** függvényhívásban használni fogunk, a socket lezárását is beleértve. Hiba esetén a visszatérési érték negatív, és az okot az ***errno.h*** fájlból elérhető `errno` változóból tudhatjuk meg.
A következő példa bemutatja, hogyan nyissunk meg egy ***RTNETLIK***-socketet:

```
int fd;
...
fd = socket(AF_NETLINK,
➥SOCK_RAW, NETLINK_ROUTE);
```

Ezt követően a socketet egy helyi címhez kell kötnünk. A felhasználói alkalmazások egyedi 32 bites azonosítót használhatnak erre a célra.
A függvény prototípusa:

```
int bind(int fd, struct
➥sockaddr *my_addr, socklen_t
➥addrlen);
```

A helyi címet a `sockaddr_nl` struktúrában kell megadni, amelynek deklarációja a ***linux/netlink.h*** fájlban található:

```
struct sockaddr_nl
{
  sa_family_t     nl_family; //
➥AF_NETLINK
  unsigned short  nl_pad;    //
➥zero
  __u32           nl_pid;   //
➥process pid
  __u32           nl_groups; //
➥multicast grps mask
};
```

Az `nl_pid` egyedi azonosító legyen, amire épp megfelelő a `getpid()` függvény eredménye, vagyis a socketet megnyitó folyamat azonosítója. Amennyiben több szál is fut, és mindegyik saját socketet nyit, módosított azonosítóra lehet szükség.
A struktúra kitöltését követően a kötés végrehajtható. Ha a `bind()` függvény hívása sikeres, a visszatérési érték nulla, ellenkező esetben negatív, és a hibakód a rendszerhibát tároló `errno` változóba kerül. Íme egy példa a `bind()` hívására:

```
struct sockaddr_nl la;
...
bzero(&la, sizeof(la));
la.nl_family = AF_NETLINK;
la.nl_pad = 0;
la.nl_pid = getpid();
la.nl_groups = 0;
rtn = bind(fd, (struct
➥sockaddr*) &la, sizeof(la));
```

Többescímzés esetén az `nl_groups` tagot is ki kell tölteni, úgy hogy a kívánt ***RTNETLINK*** művelet csoportjához tartozó csatlakozás megtörténjen.
Ha pl. értesülni szeretnénk arról, hogy más folyamatok piszkálták az útválasztótáblát, akkor az `RTMGRP_IPV4_ROUTE` és az `RTMGRP_NOTIFY` értékek vagy (|) kombinációját kell beállítanunk.
Az útválasztással kapcsolatos ***RTNETLINK*** üzeneteket a standard `sendmsg()` függvénnyel küldhetjük el a rendszermagnak. A függvény prototípusa:

```
ssize_t sendmsg(int fd, const
➥struct msghdr *msg, int flags);
```

Az `msg` mutató egy `msghdr` struktúrára mutat, melynek deklarációja így néz ki:

```
struct msghdr
{
  void *msg_name;        //
➥Címzett címe
  socklen_t msg_namelen; //
➥A címzett címének hossza
  struct iovec *msg_iov; //
➥Küldendő adatok tömbje
  size_t msg_iovlen;     //
➥Küldendő adatok száma
  void *msg_control;     //
➥Segédadatok
  size_t msg_controllen; //A
➥segédadatokat tároló buffer
➥hossza
  int msg_flags;         //
➥Kapcsolók a fogadott
➥üzenetekhez
};
```

Az `msg_name` egy `sockaddr_nl` struktúrára mutat, ami a `sendmsg()` függvény címzettjét tartalmazza. Mivel az üzenet a rendszermagnak szól, az `nl_family` tagot kivéve a `sockaddr_nl` struktúra minden mezője nulla lesz. Az `msg_namelen` tag a `sockaddr_nl` struktúra méretét kapja értékül.
Az `msg_iov` egy `iovec` struktúrára mutat, amelybe a művelet szempontjából releváns ***RTNETLINK*** üzenetet vagy üzeneteket töltjük. Az üzenetek számát az `msg_iovlen` tag határozza meg.
A többi mezőt nullára inicializáljuk.
RTNETLINK üzenetek vételére a `recv()` függvényt használjuk, melynek prototípusa:

```
ssize_t recv(int fd, void *buf,
➥size_t len, int flags);
```

A második változó egy `buffer` mutatója, ahová az érkező bájtok kerülnek, a harmadik ennek a buffernek a hosszát adja meg. ***RTNETLINK*** esetén a bájtok üzenetsorozatot rejtenek, melyet a ***netlink.h*** és az ***rtnetlink.h*** fájlokban definiált makrókkal fedhetünk fel. A `flags`-ben lévő kapcsolók a vétel természetét befolyásolják, de ***RTNETLINK***-nél bátran használhatjuk a nulla értéket.
A kommunikáció végén a socketet le kell zárni a `close()` függvénnyel, melynek prototípusa:

```
int close(int fd);
```

Az RTNETLINK funkciói

Az ***RTNETLINK***-et használó alkalmazások fejlesztőinek legalább a következő fejállományokat kell beemelniük:

```
#include <bits/sockaddr.h>
#include <asm/types.h>
#include <linux/rtnetlink.h>
#include <sys/socket.h>
```

Ezek a fájlok tartalmazzák a különféle adattípusok, struktúrák deklarációját, amik az ***RTNETLINK*** függvényhívásokhoz nélkülözhetetlenek. Következzék egy rövid leírás arról, hogy milyen, az ***RTNETLINK*** szempontjából releváns deklarációkat tartalmaznak ezek a fájlok:

- A ***bits/sockaddr.h*** fájlban találhatók a socket-függvényekben használatos címek deklarációi.
- Az ***asm/types.h*** fájl tartalmazza a ***NETLINK*** és az ***RTNETLINK*** állományaiban szereplő adattípusok deklarációját.

- A *linux/rtnetlink.h* fájl tartalmazza az *RTNETLINK*-ben használt makrókat és struktúrákat. Minthogy az *RTNETLINK* a *NETLINK*-re épül, a fájl beemeli a *linux/netlink.h* fejállományt. A *netlink.h*-ban találjuk meg a *NETLINK* általános makróit és struktúráit.
- A *sys/socket.h*-ban találhatók a socket megvalósításhoz kötődő függvényprototípusok és adatstruktúrák.

Az *RTNETLINK* használatával végrehajtható műveletek az *rtnetlink.h* állományban vannak felsorolva. Minden egyes művelet háromféle beavatkozást tesz lehetővé: hozzáadást, illetve frissítést (*NEW*), törlést (*DEL*) és lekérdezést (*GET*). A támogatott műveletek a következők:
A hálózati környezetet befolyásoló általános szolgáltatások:

- Adatkapcsolati szintű interfészbeállítások: `RTM_NEWLINK`, `RTM_DELLINK` és `RTM_GETLINK`
- Hálózati (*IP*) szintű interfészbeállítások: `RTM_NEWADDR`, `RTM_DELADDR` és `RTM_GETADDR`
- Hálózati (*IP*) szintű útvonalválasztási beállítások: `RTM_NEWROUTE`, `RTM_DELROUTE` és `RTM_GETROUTE`
- Az adatkapcsolati és hálózati címzés párosítására szolgáló gyorsítótár műveletei: `RTM_NEWNEIGH`, `RTM_DELNEIGH` és `RTM_GETNEIGH`

Forgalomszabályozó szolgáltatások:

- A hálózati réteg csomagjainak irányítása: `RTM_NEWRULE`, `RTM_DELRULE` és `RTM_GETRULE`
- A hálózati interfészek sorkezelésének beállítása: `RTM_NEWQDISC`, `RTM_DELQDISC` és `RTM_GETQDISC`
- A sorokban használt forgalomosztályok beállítása: `RTM_NEWTCLASS`, `RTM_DELTCLASS` és `RTM_GETTCLASS`
- A sorokban használt forgalomszűrők beállítása: `RTM_NEWTFILTER`, `RTM_DELTFILTER` és `RTM_GETTFILTER`

Az RTNETLINK üzeneteinek összeállítása és értelmezése

Az *RTNETLINK* kérés-válasz mechanizmussal cserél információt a hálózati környezetet befolyásolásához.
Az *RTNETLINK* kérése és válasza egyaránt üzenetstruktúrák sorozatából áll. Kérés esetén a struktúrát a hívó tölti fel, míg válasznál a rendszermag. Az *RTNETLINK* egy sor (`#define`) makrót kínál ezeknek a struktúráknak a feltöltésére, illetve a tartalmuk kinyerésére. Minden kérés az alábbi struktúrával kezdődik:

```
struct nlmsghdr
{
  __u32  nlmsg_len;   //Az
➥üzenet hossza, beleértve
➥a ezt a struktúrát is
  __u16  nlmsg_type;  //Az
➥üzenet típusa
  __u16  nlmsg_flags; //További
➥kapcsolók
  __u32  nlmsg_seq;   //
➥Sorozatszám
  __u32  nlmsg_pid;   //A küldő
➥folyamat azonosítója
}
```

Ez a *NETLINK* fejlécnek is nevezett struktúra határozza meg, hogy a kérés további részében milyen típusú *RTNETLINK* üzenetre számítsunk.
A mezők jelentése a következő:

- `nlmsg_len`: A teljes *RTNETLINK* üzenet hossza, beleértve az `nlmsghdr` struktúrát is. Kitöltését az `NLMSG_ALIGN(len)` makróval végezhetjük el, ahol len az `nlmsghdr` struktúrát követő üzenet hossza.
- `nlmsg_type`: 16 bites azonosító az üzenet típusának meghatározásához, például `RTM_NEWROUTE`.
- `nlmsg_flags`: 16 bites kapcsoló, amely tovább pontosítja az `nlmsg_type` mezőben meghatározott műveletet, például `NLM_F_REQUEST`.
- `nlmsg_seq` és `nlmsg_pid`: Ez a két mező egyértelműen azonosít egy *RTNETLINK* kérést. A hívó itt megadhat egy sorozatszámot és a folyamat azonosítóját.

Az `nlmsghdr` fejlécet a kérésben szereplő művelet szempontjából lényeges struktúrák követik. A művelet típusától függően, a hívónak az alábbi, *RTNETLINK* műveleti fejléceknek nevezett struktúrákból egyet vagy többet kell az üzenetben elhelyeznie:

- `rtmsg`: Ezt a struktúrát használjuk az útválasztótábla elemeinek megváltoztatására és lekérdezésére.
- `rtnexthop`: Az útválasztási bejegyzés következő csomópontja (next hop) a célállomás felé vezető úton elsőként szóba jövő gépet jelenti, amelyből egy bejegyzéshez több is tartozhat. Minden következő csomópontnak sokféle attribútuma lehet, például az *IP*-címe mellett a hálózati interfész is megadható.
- `rta_cacheinfo`: Minden útválasztási bejegyzéshez tartozik, többnyire a felhasználással kapcsolatos állapotinformáció, melyet a rendszermag rendszeresen frissít. Ezzel a struktúrával a felhasználó tájékoztatást kaphat az állapotinformációkról.
- `ifaddrmsg`: Ezzel a struktúrával módosíthatók vagy kérdezhetők le a hálózati interfészeknek a hálózati réteggel kapcsolatos attribútumai.
- `ifa_cacheinfo`: Az útválasztási bejegyzéshez hasonlóan, a hálózati interfészek is tárolnak magukról állapotinformációkat, amit a rendszermag frissít. Ezzel a struktúrával ezek az állapotinformációk kérdezhetők le.
- `ndmsg`: A struktúra a szomszédkeresés (*neighbor discovery*) nyomán létrejövő, a szomszédos gépek adatkapcsolati és hálózati rétegének címzése közötti kapcsolatok lekérdezésére és módosítására szolgál.
- `nda_cacheinfo`: A rendszermag által frissített szomszédkeresési bejegyzések adatainak tárolására szolgál.
- `ifinfomsg`: Ezzel a struktúrával a hálózati interfészek adatkapcsolati attribútumai kérdezhetők le és változtathatók meg.
- `tcmsg`: Ezt a struktúrát a forgalomszabályozás attribútumainak lekérdezésére és módosítására használjuk.

Az *RTNETLINK* művelet fejlécét a vonatkozó attribútumok követik, pl. az interfész száma és *IP*-címe, melyeket az `rtattr` struktúrában adunk meg. Minden attribútumhoz külön struktúra tartozik. Az `rtattr` típusa például a következő:

```
struct rtattr
{
  unsigned short  rta_len;
  unsigned short  rta_type;
};
```

Közvetlenül utána következik az attribútum értéke. Az *IPv4*-es cím pl. 4 bájt helyet foglal. Az `rta_len` ezt és az attribútumleíró struktúra hosszát is magában foglalja. Az `rta_type` az attribútum típusát határozza meg, értékeként az *rtnetlink.h* fájlban található felsoroló típusok tagjait veheti fel. Az `rtattr_type_t` és más felsoroló típusok adják meg azokat az attribútumazonosítókat, amelyek az `rta_type` mező értékei lehetnek, például `IFA_ADDRESS` és `NDA_DST`. Az egymás után fűzhető attribútumok számát az `RTATTR_MAX` makró korlátozza. Íme egy példa attribútum hozzáadására:

```
rtap->rta_type = RTA_DST;
rtap->rta_len = sizeof(struct
➥rtattr) + 4;
inet_pton(AF_INET, dsts,
     ((char *)rtap) +
➥sizeof(struct rtattr));
```

Az *RTNETLINK*-socketből származó információ szintén struktúrák sorozata. Legegyszerűbben úgy nyerhetjük ki az adatokat, hogy a bájtsorozat mentén egy mutatót mozgatunk, amit az éppen feldolgozott struktúra méretével mindannyiszor megnövelünk. Az eljárás egyszerűsítésére az *RTNETLINK* egy csokor makrót is nyújt:

- `NLMSG_NEXT(nlh, len)`: Eredménye a következő struktúrára mutat. `nlh` a legutoljára visszakapott fejléc mutatója, len pedig a teljes üzenet mérete. Ciklusban hívva, az összes üzenet kiolvasásárát lehetővé teszi.
- `NLMSG_DATA(nlh)`: A *NETLINK* fejlécben megadott műveletre vonatkozó *RTNETLINK* fejléc mutatóját eredményezi. Útválasztási bejegyzés manipulációja esetén a visszatérési érték egy `rtmsg` struktúra mutatója lesz.
- `RTM_RTA(r)`, `IFA_RTA(r)`, `NDA_RTA(r)`, `IFLA_RTA(r)` és `TCA_RTA(r)`: Eredményük az *RTNETLINK* üzenet `r` fejlécében megadott műveletre vonatkozó attribútumsor kezdetére mutat.
- `RTM_PAYLOAD(n)`, `IFA_PAYLOAD(n)`, `NDA_PAYLOAD(n)`, `IFLA_PAYLOAD(n)` és `TCA_PAYLOAD(n)`: Eredményük a *NETLINK* üzenet fejlécére mutató ***n*** által megadott *RTNETLINK* műveletet követő attribútumok teljes hossza.
- `RTA_NEXT(rta, attrlen)`: Az előzőleg megkapott attribútumot (`rta`) és a maradék attribútumok hosszát (`attrlen`) megadva a következő attribútum mutatóját adja eredményül.

Ha például egy útválasztótáblát kértünk le egy *RTNETLINK* kéréssel, a választ a következő módon dolgozzuk fel:

```
char *buf; // mutató az
➥RTNETLINK-adatra
int nll; // az összes adat
➥hossza bájtban
struct nlmsghdr *nlp;
struct rtmsg *rtp;
int rtl;
struct rtattr *rtap;
nlp = (struct nlmsghdr *) buf;
for(;NLMSG_OK(nlp, nll);
➥nlp=NLMSG_NEXT(nlp, nll))
{
  // az RTNETLINK-üzenet
➥fejlécének azonosítása
  rtp = (struct rtmsg *)
➥NLMSG_DATA(nlp);
  // az attribútumok kezdetének
➥megállapítása
  rtap = (struct rtattr *)
➥RTM_RTA(rtp);
  // az attribútumok hosszának
➥meghatározása
  rtl = RTM_PAYLOAD(nlp);
  // ciklus az összes
➥attribútum kiolvasásához
  for(;RTA_OK(rtap, rtl);
rtap=RTA_NEXT(rtap, rtl))
  {
    // attribútum feldolgozása
  }
}
```

Egy RTNETLINK példa sorról sorra

A most bemutatásra kerülő példakód az útválasztótáblán végrehajtható három műveletre koncentrál:

- `get_routing_table`: a rendszer fő útválasztótáblájának kiolvasása
- `set_routing_table`: új bejegyzés elhelyezése az útválasztótáblában
- `mon_routing_table`: a tábla változásainak megfigyelése

Mindhárom példa main() függvénye egy kaptafára készült, néhány további függvényt hív az *RTNETLINK* üzenetek létrehozásához, küldéséhez és a fogadott üzenetek feldolgozásához. Az egyszerűség kedvéért a hibakezelést teljesen mellőztem. A példák a *IPv4*-es környezetben működnek (`AF_INET`). Íme a main() függvény:

```
int main(int argc, char
➥*argv[])
{
  // socket megnyitása
  fd = socket(AF_NETLINK,
➥SOCK_RAW, NETLINK_ROUTE);
  // helyi cím beállítása
➥és kötése
  bzero(&la, sizeof(la));
  la.nl_family = AF_NETLINK;
  la.nl_pid = getpid();
  bind(fd, (struct sockaddr*)
➥&la, sizeof(la));
  // alfüggvények az RTNETLINK-
➥üzenetek létrehozására,
  // socketen küldésére, válasz
➥fogadására és feldolgozására
  form_request();
  send_request();
  recv_reply();
  read_reply();
  // socket lezárása
  close(fd);
}
```

Hasonlóképpen, a socketkommunikációt végző két függvény is majdnem teljesen azonos a példákban. Az egyik üzeneteket küld a rendszermagnak, a másik pedig üzeneteket fogad a rendszermagtól. A kivételt a `set_routing_table` és `mon_routing_table` példák jelentik. Az előbbiben nincs üzenetfogadási szakasz, míg az utóbbiban a küldés hiányzik, hiszen mindössze az útválasztási környezetben bekövetkező változások megfigyelése a feladat. A kérdéses adatokat a rendszermag többesküldéssel az összes olyan *RTNETLINK* socketnek eljuttatja, amely a megfelelő állapotban vár erre. Elsőként lássuk a kérelem küldését végző `send_request()` függvény kódját:

```
void send_request()
{
  // a távoli cím létrehozása
  bzero(&pa, sizeof(pa));
  pa.nl_family = AF_NETLINK;
  // a sendmsg() függvény
➥bemenetét képező msghdr
  // struktúra létrehozása és
➥inicializálása
  bzero(&msg, sizeof(msg));
  msg.msg_name = (void *) &pa;
  msg.msg_namelen = sizeof(pa);
  // az RTNETLINK-üzenet
➥mutatóját és méretét
  // az msghdr struktúrába
➥töltjük
  iov.iov_base = (void *)
➥&req.nl;
  iov.iov_len =
➥req.nl.nlmsg_len;
  msg.msg_iov = &iov;
  msg.msg_iovlen = 1;
  // az RTNETLINK-üzenet
➥küldése a rendszermagnak
  rtn = sendmsg(fd, &msg, 0);
}
```

Párja a fogadást végző recv_reply():

```
void recv_reply()
{
  char *p;
  // a socket kimeneti
➥bufferének inicializálása
  bzero(buf, sizeof(buf));
  p = buf;
  nll = 0;
  // addig olvasunk
➥a socketből, amíg NLMSG_DONE
  // típusú üzenetet nem
➥kapunk, vagy monitorozó
  // socketről van szó
  while(1) {
    rtn = recv(fd, p,
➥sizeof(buf) - nll, 0);
    nlp = (struct nlmsghdr *)
➥p;
    if(nlp->nlmsg_type ==
➥NLMSG_DONE)
      break;
    // a buffer mutatóját
➥a következő
    // üzenetre pozícionáljuk
    p += rtn;
    // a teljes hosszt
➥megnöveljük az utoljára
    // kapott üzenet méretével
    nll += rtn;
    if((la.nl_groups &
➥RTMGRP_IPV4_ROUTE)
                        ==
➥RTMGRP_IPV4_ROUTE)
      break;
  }
}
```

Mind az eddig bemutatott, mind a most következő függvények használnak globális változókat. Ezeket egyaránt használjuk a socketműveletekhez, illetve az ***RTNETLINK*** üzenetek elkészítéséhez és feldolgozásához:

```
// az RTNETLINK-kéréseket
➥tároló buffer
struct {
  struct nlmsghdr nl;
  struct rtmsg    rt;
  char            buf[8192];
} req;
// socketkommunikációhoz
➥használt változók
int fd;
struct sockaddr_nl la;
struct sockaddr_nl pa;
struct msghdr msg;
struct iovec iov;
int rtn;
// az RTNETLINK-válasz(oka)t
tároló buffer
char buf[8192];
// az RTNETLINK-üzenetek
➥feldolgozásánál
// használt üzenetmutatók
➥és -hosszak
struct nlmsghdr *nlp;
int nll;
struct rtmsg *rtp;
int rtl;
struct rtattr *rtap;
```

A get_routing_table példa az ***IPv4***-es hálózati környezet fő útválasztótábláját kérdezi le. A kérést összeállító form_request() függvény a következő:

```
void form_request()
{
  // a kérés bufferének
➥inicializálása
  bzero(&req, sizeof(req));
  // a NETLINK-fejléc
➥beállítása
  req.nl.nlmsg_len
              = NLMSG_LENGTH
➥(sizeof(struct rtmsg));
  req.nl.nlmsg_flags =
➥NLM_F_REQUEST | NLM_F_DUMP;
  req.nl.nlmsg_type =
➥RTM_GETROUTE;
  // az útválasztófejléc
➥beállítása
  req.rt.rtm_family = AF_INET;
  req.rt.rtm_table =
➥RT_TABLE_MAIN;
}
```

Az ***RTNETLINK*** kérésre érkező választ a buf változóban találjuk. Ezt a read_reply() függvénnyel feldolgozva megkapjuk az útválasztótábla tartalmát. A függvény kódja:

```
void read_reply()
{
  // az útválasztótábla
➥tartalmának (azaz egy
  // bejegyzésnek a tárolására
➥szolgáló karakterláncok
  char dsts[24], gws[24],
➥ifs[16], ms[24];
  // külső ciklus: bejárja az
➥összes NETLINK-fejlécet,
  // így az útválasztási
➥bejegyzését is
  nlp = (struct nlmsghdr *)
➥buf;
  for(;NLMSG_OK(nlp, nll);
➥nlp=NLMSG_NEXT(nlp, nll))
  {
    // az útválasztási
➥bejegyzés fejlécének
➥meghatározása
    rtp = (struct rtmsg *)
➥NLMSG_DATA(nlp);
    // csak a fő táblára
➥vagyunk kíváncsiak
    if(rtp->rtm_table !=
➥RT_TABLE_MAIN)
      continue;
    // a karakterláncok
➥inicializálása
    bzero(dsts, sizeof(dsts));
    bzero(gws, sizeof(gws));
    bzero(ifs, sizeof(ifs));
    bzero(ms, sizeof(ms));
    // belső ciklus: bejárja
➥egy bejegyzés összes
    // attribútumát
    rtap = (struct rtattr *)
➥RTM_RTA(rtp);
    rtl = RTM_PAYLOAD(nlp);
    for(;RTA_OK(rtap, rtl);
➥rtap=RTA_NEXT(rtap,rtl))
    {
      switch(rtap->rta_type)
      {
        // destination IPv4
```

```
➥address
        case RTA_DST:
          inet_ntop(AF_INET,
➥RTA_DATA(rtap),
                    dsts, 24);
          break;
        // next hop IPv4
➥address
        case RTA_GATEWAY:
          inet_ntop(AF_INET,
➥RTA_DATA(rtap),
                    gws, 24);
          break;
        // unique ID associated
➥with the network
        // interface
        case RTA_OIF:
          sprintf(ifs, "%d",
                  *((int *)
          RTA_DATA(rtap)));
        default:
          break;
      }
    }
    sprintf(ms, "%d", rtp->
➥rtm_dst_len);
    printf("dst %s/%s gw %s if
➥%s\n",
                    dsts, ms,
➥gws, ifs);
  }
}
```

A `set_routing_table` példában *RTNETLINK* kérést küldünk, hogy egy új bejegyzés kerüljön az útválasztótáblába. A kérdéses bejegyzés egy útvonal (32 bites hálózati előtaggal) egy privát *IP* címhez (192.168.0.100), a 2-es számú hálózati interfészen keresztül. Ezek az értékek sorra a `pn` (hálózati prefix hossza), `dsts` (cél *IP* cím) és `ifcn` (interfész száma) változókba kerülnek. Érdemes futtatni a `get_routing_table` példát, hogy képbe kerüljünk a rendszerünkben található interfészek azonosítószámát és az *IP* hálózatot illetően. A kérelem összeállítását végző `form_request()` kódja:

```
void form_request()
{
  // az útválasztási bejegyzés
➥attribútumai
  char dsts[24] =
➥"192.168.0.100";
  int ifcn = 2, pn = 32;
  // az RTNETLINK-kérés
➥bufferének inicializálása
  bzero(&req, sizeof(req));
  // a kérelem kezdeti
➥hosszának kiszámítása
  rtl = sizeof(struct rtmsg);
  // az első attribútum
➥hozzáadása:
  // a cél IP-cím beállítása
➥és az RTNETLINK-buffer
  // méretének növelése
  rtap = (struct rtattr *)
➥req.buf;
  rtap->rta_type = RTA_DST;
  rtap->rta_len = sizeof(struct
➥rtattr) + 4;
  inet_pton(AF_INET, dsts,
     ((char *)rtap) +
➥sizeof(struct rtattr));
  rtl += rtap->rta_len;
  // második attribútum
➥hozzáadása:
  // az interfész számának
➥beállítása és
  // a méret növelése
  rtap = (struct rtattr *)
➥(((char *)rtap)
            + rtap->rta_len);
  rtap->rta_type = RTA_OIF;
  rtap->rta_len = sizeof(struct
➥rtattr) + 4;
  memcpy(((char *)rtap) +
➥sizeof(struct rtattr),
          &ifcn, 4);
  rtl += rtap->rta_len;
  // a NETLINK-fejléc
➥összeállítása
  req.nl.nlmsg_len =
➥NLMSG_LENGTH(rtl);
  req.nl.nlmsg_flags =
➥NLM_F_REQUEST | NLM_F_CREATE;
  req.nl.nlmsg_type =
➥RTM_NEWROUTE;
  // a struct rtmsg fejléc
➥beállítása
  req.rt.rtm_family = AF_INET;
  req.rt.rtm_table =
➥RT_TABLE_MAIN;
  req.rt.rtm_protocol =
➥RTPROT_STATIC;
  req.rt.rtm_scope =
➥RT_SCOPE_UNIVERSE;
  req.rt.rtm_type =
➥RTN_UNICAST;
  // a hálózati prefix
➥nagyságának beállítása
  req.rt.rtm_dst_len = pn;
}
```

A `mon_routing_table` példa azokat az *RTNETLINK* üzeneteket fogadja, amik akkor keletkeznek, ha más folyamatok megváltoztatják a rendszer fő útválasztótábláját. Most is a korábban bemutatott `read_reply()` függvényt használjuk az üzenetek feldolgozására. A main() függvényben apró változtatást kell eszközölnünk. Mivel ez a művelet a rendszermag többesküldéssel továbbított üzeneteire figyel, a socket helyi címhez kötésénél az `RTMGRP_IPV4_ROUTE` és `RTMGRP_NOTIFY` kapcsolókat is használnunk kell:

```
la.nl_groups =
➥RTMGRP_IPV4_ROUTE |
➥RTMGRP_NOTIFY;
```

A `mon_routing_table` végrehajtása után adjuk ki a `route add` vagy `route del` parancsot egy másik parancssori értelmezőből, hogy lássuk az eredményt.

Összefoglalás

Az *RTNETLINK* egyszerű, ugyanakkor sokoldalú eszköz a *Linux* hálózati beállításainak manipulálásához. A felhasználó térben írt protokollimplementációk ideális alanyai az *RTNETLINK* használatának. Az *IPROUTE2*-nek is becézett, fejlett *IP* útválasztási parancsgyűjtemény is *NETLINK* alapú. Az *RTNETLINK* műveleteiről és kapcsolóiról többet is megtudhatunk a *NETLINK(7)* és az *RTNETLINK(7)* kézikönyvlapokból. A példakódok letölthetők az ➲ *ftp.ssc.com/pub/lj/listings/issue145/8498.tgz* címről.

Köszönetnyilvánítás

Hálásan köszönöm *Carmelita Goerg* professzor segítségét.

Asanga Udugama
(adu@comnets.uni-bremen.de) kutató, szoftverfejlesztő a Brémai Egyetem ComNets részlegén, Németországban. Jelenleg a mobilitással kapcsolatos, hálózati rétegbeli IETF-protokollok szabványosításában és implementációjában vesz részt. A nevét öregbíti néhány referenciaimplementációja is. Most épp álmai tolószékét várja (Meyra X3 szervóval).