Netlink

Comunicare con il Kernel via Socket

Roberto Marino, PhD

Università di Messina
CdL in Informatica — Reti e Sistemi Distribuiti Mod.B

April 13, 2026



UniMe
1548

Cos'è Netlink?

Definizione

Netlink è una **famiglia di socket** del kernel Linux — identificata da `AF_NETLINK` — progettata per la comunicazione **bidirezionale** tra processi in spazio utente e sottosistemi del kernel, e tra sottosistemi del kernel stesso.

È una famiglia di socket

- Non un driver, non un file speciale, non una syscall dedicata
- Si apre con la stessa `socket()` di TCP/IP
- I messaggi non escono dalla macchina — restano in memoria kernel

È bidirezionale

- **Sincrona**: il processo manda una richiesta e riceve una risposta (ACK o errore)
- **Asincrona**: il processo si iscrive a gruppi multicast e riceve *notifiche* dal kernel senza averle chieste

Una famiglia con più sottosistemi

"Famiglia" significa più canali

Il terzo argomento di `socket()` seleziona il sottosistema kernel a cui parlare. Ogni sottosistema è un canale separato.

Sottosistema	Gestisce
`NETLINK_ROUTE`	iface, rotte, IP
`NETLINK_NETFILTER`	iptables, nftables
`NETLINK_AUDIT`	audit di sicurezza
`NETLINK_KOBJECT_ UEVENT`	eventi dispositivi
`NETLINK_GENERIC`	sottosistemi custom

Cosa non è Netlink

- **Non** è un protocollo di rete
  i messaggi non hanno IP, non vengono instradati, non escono dalla macchina
- **Non** è una pipe o un file speciale
  a differenza di `/proc` e `/sys`, supporta messaggi strutturati con attributi variabili e notifiche asincrone in broadcast
- **Non** è solo request/response
  supporta multicast verso più processi contemporaneamente

Definizione

Netlink è una famiglia di socket (`AF_NETLINK`) per la comunicazione strutturata tra spazio utente e kernel, basata su messaggi binari a lunghezza variabile con attributi opzionali.

Le parole chiave della definizione

- **famiglia di socket** — API familiare, non una syscall nuova
- **strutturata** — messaggi con header e attributi tipizzati, non byte arbitrari
- **lunghezza variabile** — si aggiungono attributi senza compromettere la compatibilità

E' il meccanismo usato dal comando `ip`

`ip link add`, `ip addr add`, `ip route add` fanno **esattamente la stessa cosa**, con più gestione degli errori.

Il problema: come parlare con il kernel

`ioctl()`: struttura fissa

- Una costante `SIOC*` + una `struct` fissa (parametri 2 e 3 di ioctl)
- `struct ifreq` = sempre 40 byte
- Impossibile aggiungere parametri opzionali

Netlink: attributi variabili

- Messaggi a **lunghezza variabile**
- Attributi opzionali: aggiungi quelli che servono
- Attributo sconosciuto → saltato silenziosamente
- Nessuna rottura di compatibilità
- Comunicazione **bidirezionale** e asincrona

Netlink è un socket: API familiare

Aprire un socket Netlink

```
int nl = socket(
AF_NETLINK,   /* famiglia */
SOCK_RAW,
NETLINK_ROUTE); /* sottosistema */
```

Poi si usano le syscall normali

- `bind()`: associa al processo
- `send()`: invia richiesta al kernel
- `recv()`: legge risposta/notifica
- `close()`: chiude

I sottosistemi principali

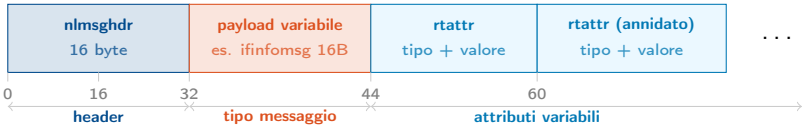
`NETLINK_ROUTE`	iface, rotte, IP
`NETLINK_NETFILTER`	iptables/nftables
`NETLINK_AUDIT`	audit sicurezza
`NETLINK_KOBJECT_ UEVENT`	eventi hw
`NETLINK_GENERIC`	sottosistemi custom

Chi lo usa ogni giorno

`ip(8)`, `ss(8)`, Docker, Podman, NetworkManager, systemd-networkd, WireGuard, nftables, iw(8)...

Tutto il networking moderno su Linux parla Netlink.

Anatomia di un messaggio Netlink



`struct nlmsghdr`: sempre presente

- `nlmsg_len`: lunghezza totale (aggiornata ad ogni attributo aggiunto)
- `nlmsg_type`: `RTM_NEWLINK`, `RTM_NEWADDR`...
- `nlmsg_flags`: `NLM_F_REQUEST | NLM_F_ACK | NLM_F_CREATE`

`struct rtattr`: ogni attributo

- `rta_type`: tipo (`IFLA_IFNAME`, ...)
- `rta_len`: header (4B) + valore
- Il valore segue immediatamente in memoria
- **Allineamento a 4 byte** obbligatorio: usare sempre le macro `RTA_*`

Le macro: che problema risolvono?

Senza macro: pericoloso

- Gli offset si calcolano a mano
- Un errore di 1 byte → `EINVAL` dal kernel
- Su ARM/MIPS: bus error (accesso non allineato)
- Difficile da leggere e mantenere

Il vincolo

Ogni attributo deve iniziare a un offset **multiplo di 4** nel buffer.

`"dummy0\0"` = 7 byte.

`4 (header) + 7 = 11` ? non allineato.

Serve 1 byte di padding → occupa 12 byte.

Con macro: sicuro

```
/* allinea len al multiplo di 4 superiore */
NLMSG_ALIGN(11) = 12
NLMSG_ALIGN(10) = 12
NLMSG_ALIGN(12) = 12 /* gia' ok */

/* come funziona internamente */
(len + 3) & ~3
```

`NLMSG_ALIGN`	lunghezze messaggio
`RTA_LENGTH`	hdr(4B) + valore
`RTA_ALIGN`	arrotonda a 4 byte
`RTA_DATA`	puntatore al valore

Le macro nel dettaglio

Macro	Calcolo	Usata per
`NLMSG_LENGTH(x)`	`16 + x`	lunghezza iniziale del messaggio
`NLMSG_ALIGN(len)`	`(len+3) & ~3`	posiziona prossimo attr
`RTA_LENGTH(len)`	`4 + len`	valore per `rta_len`
`RTA_ALIGN(len)`	`(len+3) & ~3`	incrementa `nlmsg_len`
`RTA_DATA(a)`	`a + 4`	puntatore al valore
`NLMSG_DATA(h)`	`h + 16`	payload dopo header
`RTA_OK(a,len)`	controlla validità	scansione attributi
`RTA_NEXT(a,len)`	avanza di `rta_len`	scansione attributi

Regola assoluta

Non calcolare mai offset Netlink a mano — usare sempre le macro. Un byte fuori posto causa `EINVAL` o corruzione silenziosa su ARM/MIPS.

Costruire il buffer: struttura C

Listing 1: Il buffer del messaggio: tutto contiguo in memoria

```
/* Una struct anonima garantisce che hdr, ifi e buf siano contigui.
   send() manda esattamente nlmsg_len byte a partire da &msg.
   Se fossero tre variabili separate, send() non potrebbe mandarle insieme. */
static struct {
    struct nlmsghdr  hdr;      /* offset  0, 16 byte ? header Netlink        */
    struct ifinfomsg ifi;      /* offset 16, 16 byte ? payload fisso per link */
    char             buf[256]; /* offset 32, spazio per gli attributi rtattr  */
} msg;
```

Perché `static`?

- Visibile solo in questo file sorgente
- Nessun conflitto con altri file
- Inizializzata a zero automaticamente (il `memset` è esplicito ma ridondante)
- Le funzioni helper operano su questa variabile globale del file

Perché struct anonima?

`send(nl, &msg, msg.hdr.nlmsg_len, 0)`

Il kernel riceve un blocco contiguo di byte. `nlmsg_len` dice quanti byte leggere. La contiguità è garantita dalla struct, non da tre allocazioni separate.

add_attr(): aggiungere un attributo

Listing 2: add_attr(): posiziona un rtattr nel buffer e aggiorna nlmsg_len

```
static struct rtattr *add_attr(int type, const void *data, int len)
{
    /* 1. Posizione: subito dopo la fine attuale del messaggio (allineata) */
    struct rtattr *a = (struct rtattr *)
        ((char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len)); // (@\color{blue!70!black}offset corretto@)

    /* 2. Header dell'attributo */
    a->rta_type = type;
    a->rta_len  = RTA_LENGTH(len);   // (@\color{blue!70!black}4 + len ? valore reale, non allineato@)

    /* 3. Copia il valore (NULL per attributi annidati ? apre il contenitore) */
    if (data) memcpy(RTA_DATA(a), data, len); // (@\color{blue!70!black}RTA\_DATA(a) = a + 4@)

    /* 4. Avanza il cursore di nlmsg_len (con padding per allineamento) */
    msg.hdr.nlmsg_len =
        NLMSG_ALIGN(msg.hdr.nlmsg_len) + RTA_ALIGN(a->rta_len); // (@\color{blue!70!black}arrotondato@)

    return a; /* restituito solo per gli attributi annidati (close_attr) */
}
```

La differenza tra `rta_len` e il contributo a `nlmsg_len`

`rta_len = 11` (valore reale, scritto nell'header) `nlmsg_len += RTA_ALIGN(11) = 12` (con padding nel buffer)

close_attr(): chiudere un attributo annidato

Listing 3: close_attr(): corregge rta_len a posteriori

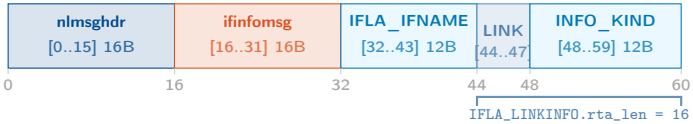
```
static void close_attr(struct rtattr *a)
{
    /* rta_len = fine del messaggio
                 - inizio dell'attributo
       Copre: header(4B) + tutti i figli */
    a->rta_len =
        (char *)&msg
        + NLMSG_ALIGN(msg.hdr.nlmsg_len)
        - (char *)a;
}
```

Perché è necessaria?

Quando si apre un attributo annidato con `add_attr(tipo, NULL, 0)`, `rta_len = 4` (solo header).

Non si sa ancora quanto saranno lunghi i figli. `close_attr()` misura la distanza reale **a posteriori**.

Il messaggio dummy: analisi byte per byte




```
[0..3]   3c 00 00 00   nlmsg_len=60      [4..5]   10 00   RTM_NEWLINK=16      [6..7]   05 06   REQUEST|ACK|CREATE|EXCL
[16..31] 00 00 ...  00   ifinfomsg tutto zero (nuova interfaccia, indice libero)
[32..33] 0b 00   rta_len=11   [34..35] 03 00   IFLA_IFNAME   [36..42] 64 75 6d 6d 79 30 00   "dummy0\0"   [43] 00 pad
[44..45] 10 00   rta_len=16 (close_attr)                                                    [46..47] 12 00   IFLA_LINKINFO
[48..49] 0a 00   rta_len=10 [50..51] 01 00   IFLA_INFO_KIND [52..56] 64 75 6d 6d 79   "dummy\0" [57..59] 00 00 00 pad
```

Il messaggio dummy: analisi byte per byte

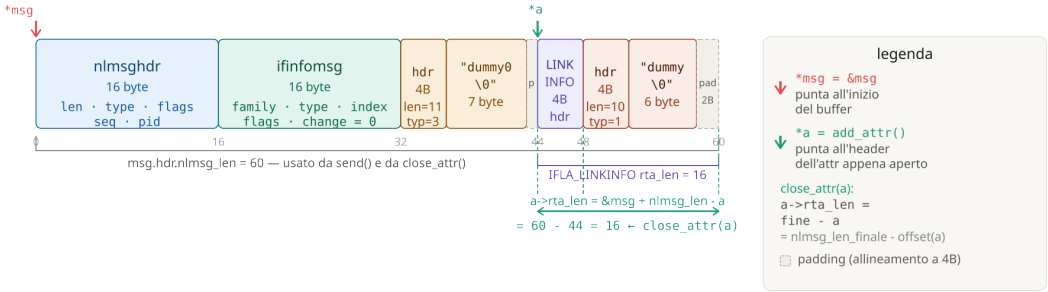


Figure: nl_dummy.c: Pacchetto Netlink e dettaglio dei puntatori

Costruzione del messaggio: crescita di `nlmsg_len`

Operazione	Contributo	nlmsg_len
`NLMSG_LENGTH(sizeof(ifinfomsg))`	$16 + 16 = 32$	32
`add_attr(IFLA_IFNAME, "dummy0", 7)`	`RTA_ALIGN(4+7)=12`	44
`add_attr(IFLA_LINKINFO, NULL, 0)`	`RTA_ALIGN(4+0)=4`	48
`add_attr(IFLA_INFO_KIND, "dummy", 6)`	`RTA_ALIGN(4+6)=12`	60
`close_attr(info)`	0 (aggiorna solo `rta_len`)	60
Totale inviato al kernel		**60 byte**

La distinzione chiave

`rta_len = 11`: valore reale scritto nell'header `rtattr`.

`nlmsg_len += 12`: spazio occupato nel buffer (con padding incluso).

`close_attr` non tocca `nlmsg_len`

Aggiorna solo `rta_len` di `IFLA_LINKINFO`, ricalcolandola come differenza tra posizione finale del messaggio e inizio dell'attributo:

Inviare la richiesta e leggere la risposta

Listing 4: Passo 3 e 4 del programma nl_dummy.c

```
/* 3. Invia al kernel ? send() manda nlmsg_len byte contigui */
send(nl, &msg, msg.hdr.nlmsg_len, 0);

/* 4. Leggi la risposta ? sempre NLMSG_ERROR se si era impostato NLM_F_ACK */
char risposta[256];
recv(nl, risposta, sizeof(risposta), 0);

struct nlmsghdr *h = (struct nlmsghdr *)risposta;
struct nlmsgerr *e = (struct nlmsgerr *)NLMSG_DATA(h); // (@\color{blue!70!black}payload dopo l'header@)

if (h->nlmsg_type == NLMSG_ERROR && e->error != 0) {
    fprintf(stderr, "Errore: %d", -e->error);          // (@\color{blue!70!black}e->error = -errno@)
    if (-e->error == 1)  fprintf(stderr, " (EPERM)");  // (@\color{blue!70!black}serve sudo@)
    if (-e->error == 17) fprintf(stderr, " (EEXIST)"); // (@\color{blue!70!black}gia' esiste@)
}
```

Errori comuni

1	EPERM	Serve CAP_NET_ADMIN
17	EEXIST	Interfaccia già esiste
22	EINVAL	Messaggio malformato
19	ENODEV	Modulo non trovato

struct nlmsgerr

- error == 0: successo
- error < 0: -errno del kernel
- msg: copia header originale