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

HandsOn 8: Netlink

Creare un'interfaccia di rete via Netlink

Last Update: April 9, 2026

Perché esiste Netlink

Quando un programma vuole creare un'interfaccia di rete, aggiungere una rotta o assegnare un indirizzo IP, deve in qualche modo comunicarlo al kernel. Prima di Netlink, l'unico meccanismo disponibile era `ioctl()`.

Il problema con `ioctl()` è strutturale: ogni chiamata trasporta una struttura dati **fissa**, definita a compile time nel kernel. Aggiungere un nuovo campo alla struttura significa cambiare la sua dimensione, rompendo la compatibilità binaria con tutti i programmi già compilati.

Netlink risolve questo con messaggi a **lunghezza variabile** composti da attributi opzionali. Se il kernel aggiunge un nuovo attributo, i programmi vecchi lo ignorano semplicemente passando al successivo. L'interfaccia è estensibile per definizione, senza mai rompere la compatibilità.

▷ Cosa usa Netlink nella pratica

Ogni volta che esegui `ip link add`, `ip addr add`, `ip route add`, o avvii un container Docker, il programma costruisce un messaggio Netlink e lo manda al kernel. `NetworkManager`, `systemd-networkd` e `iproute2` usano tutti la stessa interfaccia.

Netlink come socket

Netlink si apre con la normale chiamata `socket()`, usando la famiglia `AF_NETLINK` al posto di `AF_INET`.

Listing 1: Aprire un socket Netlink

```c
#include <sys/socket.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>

// Terzo argomento: sottosistema del kernel a cui parlare.
// NETLINK_ROUTE gestisce interfacce, rotte, indirizzi.
int nl = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);

// Il bind specifica l'identita' del processo lato Netlink.
// nl_pid=0 dice al kernel di usare il PID del processo.
struct sockaddr_nl sa = {
    .nl_family = AF_NETLINK,
    .nl_pid    = 0,        // 0 = usa il PID del processo
    .nl_groups = 0,        // 0 = non iscriversi a gruppi multicast
};
bind(nl, (struct sockaddr *)&sa, sizeof(sa));
```

Dopo il `bind()`, si usano `send()` e `recv()` normali. Il contenuto e' un protocollo binario strutturato descritto nelle sezioni successive.

I sottosistemi Netlink

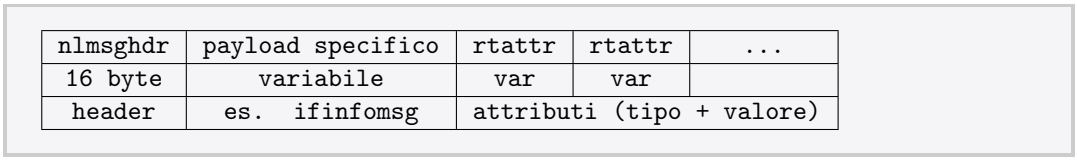
Il terzo argomento di `socket()` seleziona il sottosistema kernel a cui si vuole parlare. I principali sono:

Costante	Intestazione	Gestisce
`NETLINK_ROUTE`	`linux/rtnetlink.h`	Interfacce, rotte, indirizzi
`NETLINK_NETFILTER`	`linux/netfilter.h`	`iptables`, `nftables`
`NETLINK_KOBJECT_UEVENT`	`linux/netlink.h`	Eventi dispositivi (`udev`)
`NETLINK_AUDIT`	`linux/audit.h`	Audit di sicurezza
`NETLINK_GENERIC`	`linux/genetlink.h`	Sottosistemi personalizzati

In questo documento usiamo esclusivamente `NETLINK_ROUTE`.

Struttura di un messaggio Netlink

Ogni messaggio Netlink ha la seguente struttura in memoria:



nlmsghdr — l'header di ogni messaggio

Listing 2: Struttura nlmsghdr (linux/netlink.h)

```c
struct nlmsghdr {
    __u32 nlmsg_len;    // lunghezza totale del messaggio (header incluso)
    __u16 nlmsg_type;   // tipo: RTM_NEWLINK, RTM_NEWADDR, RTM_NEWROUTE, ...
    __u16 nlmsg_flags;  // NLM_F_REQUEST | NLM_F_ACK | NLM_F_CREATE | ...
    __u32 nlmsg_seq;    // numero di sequenza (per abbinare request/risposta)
    __u32 nlmsg_pid;    // PID del mittente (0 = kernel)
};
```

I tipi di messaggio piu' usati con `NETLINK_ROUTE`:

Tipo	Operazione
`RTM_NEWLINK`	Crea o modifica un'interfaccia
`RTM_DELLINK`	Elimina un'interfaccia
`RTM_GETLINK`	Leggi informazioni su un'interfaccia
`RTM_NEWADDR`	Aggiungi un indirizzo IP
`RTM_DELADDR`	Rimuovi un indirizzo IP
`RTM_NEWROUTE`	Aggiungi una rotta
`RTM_DELROUTE`	Rimuovi una rotta

I flag principali per `nlmsg_flags`:

Flag	Significato
NLM_F_REQUEST	Questo e' un messaggio di richiesta (obbligatorio)
NLM_F_ACK	Voglio una conferma (ACK o errore)
NLM_F_CREATE	Crea la risorsa se non esiste
NLM_F_EXCL	Errore se la risorsa esiste gia'
NLM_F_REPLACE	Sostituisci se esiste gia'

Il payload specifico

Dopo `nlmsghdr` segue una struttura che dipende dal tipo di messaggio. Per i messaggi `RTM_*LINK` il payload e' `struct ifinfomsg`:

Listing 3: Struttura ifinfomsg (linux/rtnetlink.h)

```
struct ifinfomsg {
    unsigned char  ifi_family;  // AF_UNSPEC per RTM_NEWLINK generico
    unsigned char  __ifi_pad;
    unsigned short ifi_type;    // tipo hardware (ARPHRD_ETHER, ...)
    int            ifi_index;   // indice dell'interfaccia (0 = nuovo)
    unsigned int   ifi_flags;   // IFF_UP, IFF_RUNNING, ...
    unsigned int   ifi_change;  // maschera dei flag da cambiare
};
```

Gli attributi rtattr

Dopo il payload specifico seguono zero o piu' attributi. Ogni attributo ha un header `rtattr` seguito dal valore:

Listing 4: Struttura rtattr (linux/rtnetlink.h)

```
struct rtattr {
    unsigned short rta_len;   // lunghezza totale (header + valore)
    unsigned short rta_type;  // tipo: IFLA_IFNAME, IFLA_LINKINFO, ...
    // seguito dal valore, allineato a 4 byte
};
```

[!] Allineamento obbligatorio

Ogni attributo deve iniziare a un offset multiplo di 4 byte rispetto all'inizio del messaggio. Le macro `RTA_LENGTH(len)`, `RTA_DATA(a)`, `RTA_ALIGN(len)` gestiscono questo automaticamente — usarle sempre invece di calcolare gli offset a mano.

Gli attributi possono essere **annidati**: un attributo di tipo `IFLA_LINKINFO` contiene al suo interno altri attributi (`IFLA_INFO_KIND`, `IFLA_INFO_DATA`). Per aprire un attributo annidato si aggiunge un `rtattr` con valore vuoto; al termine si aggiorna la sua `rta_len` con la dimensione totale di tutto il contenuto.

Leggere la risposta del kernel

Dopo `send()`, il kernel invia sempre una risposta se si e' impostato `NLM_F_ACK`. La risposta e' un messaggio di tipo `NLMSG_ERROR`:

Listing 5: Struttura di risposta nlmsgerr

```c
struct nlmsgerr {
    int           error;    // 0 = successo, negativo = -errno
    struct nlmsghdr msg;    // copia dell'header della richiesta
        originale
};
```

Listing 6: Pattern standard per leggere l'ACK

```c
char risposta[256];
ssize_t n = recv(nl, risposta, sizeof(risposta), 0);

struct nlmsghdr *h = (struct nlmsghdr *)risposta;

if (h->nlmsg_type == NLMSG_ERROR) {
    struct nlmsgerr *e = (struct nlmsgerr *)NLMSG_DATA(h);
    if (e->error == 0)
        printf("Successo.\n");
    else
        fprintf(stderr, "Errore: %d (%s)\n", -e->error, strerror(-e
            ->error));
}
```

Gli errori piu' comuni:

errno	Costante	Causa tipica
1	`EPERM`	Privilegi insufficienti (`CAP_NET_ADMIN`)
17	`EEXIST`	Interfaccia gia' esistente (`NLM_F_EXCL` impostato)
19	`ENODEV`	Interfaccia non trovata
22	`EINVAL`	Messaggio malformato o attributo non valido

Creare una interfaccia dummy (cioè che non fa nulla). Analisi del codice.

`nl_dummy.c` crea un'interfaccia di rete di tipo `dummy` usando il protocollo Netlink direttamente, senza librerie di supporto. È il programma più semplice possibile che parla con il sottosistema di rete del kernel.

Il programma percorre esattamente il flusso che `ip link add dummy0 type dummy` esegue internamente:

1. Apre un socket `AF_NETLINK`
2. Costruisce un messaggio binario `RTM_NEWLINK` in un buffer
3. Lo invia al kernel con `send()`
4. Legge la risposta (ACK o errore) con `recv()`

Le strutture dati globali

Listing 7: Il buffer del messaggio — definizione globale

```c
static struct {
    struct nlmsghdr  hdr;      /* offset   0, 16 byte */
    struct ifinfomsg ifi;      /* offset  16, 16 byte */
    char             buf[256]; /* offset  32, spazio per gli
        attributi */
} msg;
```

Perché un'unica struttura e non tre variabili separate?

Netlink richiede che il messaggio sia un **blocco contiguo di memoria**. Il kernel legge `nlmsg_len` byte a partire dall'indirizzo del primo byte della struttura. Se `hdr`, `ifi` e `buf` fossero tre variabili separate allocate in punti diversi della memoria, `send()` non potrebbe mandarle come un unico messaggio.

Unirle in una `struct` anonima garantisce la contiguità — il compilatore le dispone in memoria nell'ordine dichiarato, senza padding tra i campi (le strutture Netlink sono già allineate a 4 byte).

Perché `static`?

Il qualificatore `static` a livello di file ha due effetti:

- **Visibilità**: la variabile è visibile solo in questo file sorgente. Il linker non la espone. Se un altro file definisse un `msg` omonimo, non ci sarebbe conflitto.
- **Inizializzazione**: le variabili `static` vengono inizializzate a zero automaticamente prima del `main()`. Il `memset(&msg, 0, sizeof(msg))` nel `main` è quindi ridondante ma esplicito — utile per chiarezza didattica.

Le strutture interne

`struct nlmsghdr` (da `<linux/netlink.h>`) è l'header presente in ogni messaggio Netlink, qualunque sia il tipo:

Listing 8: Definizione di nlmsghdr nel kernel

```
struct nlmsghdr {
    __u32 nlmsg_len;    /* lunghezza totale del messaggio (header
       incluso) */
    __u16 nlmsg_type;   /* tipo: RTM_NEWLINK, RTM_NEWADDR, ...
                   */
    __u16 nlmsg_flags;  /* NLM_F_REQUEST, NLM_F_ACK, NLM_F_CREATE,
       ...     */
    __u32 nlmsg_seq;    /* numero di sequenza per abbinare req/
       risposta     */
    __u32 nlmsg_pid;    /* PID del mittente (0 = kernel lo imposta)
                  */
};
```

`struct ifinfomsg` (da `<linux/rtnetlink.h>`) è il payload specifico per i messaggi `RTM_*LINK`:

Listing 9: Definizione di ifinfomsg nel kernel

```
struct ifinfomsg {
    unsigned char   ifi_family;   /* AF_UNSPEC per uso generico
                   */
    unsigned char   __ifi_pad;    /* padding
                                     */
    unsigned short ifi_type;     /* tipo hardware (ARPHRD_ETHER,
       ...)      */
    int             ifi_index;    /* indice interfaccia (0 = nuovo)
                */
    unsigned int    ifi_flags;    /* IFF_UP, IFF_RUNNING, ...
                  */
    unsigned int    ifi_change;   /* maschera dei flag da modificare
                */
};
```

La funzione `add_attr()`

Listing 10: add_attr() — aggiunge un attributo rtattr al messaggio

```
1  static struct rtattr *add_attr(int type, const void *data, int len)
2  {
3      struct rtattr *a = (struct rtattr *)
4          ((char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len));
5      a->rta_type = type;
6      a->rta_len  = RTA_LENGTH(len);
7      if (data) memcpy(RTA_DATA(a), data, len);
8      msg.hdr.nlmsg_len =
9          NLMSG_ALIGN(msg.hdr.nlmsg_len) + RTA_ALIGN(a->rta_len);
10     return a;
11 }
```

Riga 1 — calcolo della posizione

```
1  struct rtattr *a = (struct rtattr *)
2      ((char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len));
```

Il nuovo attributo va posizionato subito dopo la fine del messaggio corrente. `msg.hdr.nlmsg_len` contiene la lunghezza attuale del messaggio — ogni chiamata precedente ad `add_attr()` l'ha già aggiornata.

`NLMSG_ALIGN(x)` arrotonda `x` al multiplo di 4 più vicino:

```
#define NLMSG_ALIGN(len) (((len)+NLMSG_ALIGNTO-1) & ~(NLMSG_ALIGNTO-1))
#define NLMSG_ALIGNTO 4
```

Il cast `(char *)&msg` è necessario per fare aritmetica per byte — sommare un intero a un puntatore a `struct` sommerebbe multipli di `sizeof(struct)`, non byte.

Riga 2-3 — compilazione dell'header rtattr

```
1  a->rta_type = type;
2  a->rta_len  = RTA_LENGTH(len);
```

`struct rtattr` ha solo due campi da 2 byte ciascuno: `rta_len` (lunghezza header + valore) e `rta_type` (tipo).

`RTA_LENGTH(len)` calcola la lunghezza totale dell'attributo:

```
#define RTA_LENGTH(len) (RTA_ALIGN(sizeof(struct rtattr)) + (len))
```

Quindi `RTA_LENGTH(6)` $= 4 + 6 = 10$. Non è un multiplo di 4, ma `rta_len` può avere un valore non allineato — l'allineamento viene applicato solo quando si calcola la posizione del *prossimo* attributo.

Riga 4 — copia del valore

```
1  if (data) memcpy(RTA_DATA(a), data, len);
```

`RTA_DATA(a)` restituisce il puntatore al valore, cioè i byte immediatamente dopo l'header `rtattr`:

```
#define RTA_DATA(rta) ((void *)(((char *)(rta)) + RTA_LENGTH(0)))
```

Il controllo `if (data)` gestisce il caso degli attributi annidati: quando si apre un contenitore come `IFLA_LINKINFO`, si passa `NULL` come dati — l'attributo viene creato con solo l'header, senza valore, e il suo `rta_len` verrà corretto da `close_attr()`.

Riga 5-6 — aggiornamento di `nlmsg_len`

```
1 msg.hdr.nlmsg_len =
2     NLMSG_ALIGN(msg.hdr.nlmsg_len) + RTA_ALIGN(a->rta_len);
```

`nlmsg_len` viene aggiornato aggiungendo la dimensione dell'attributo appena scritto, arrotondata a 4 byte con `RTA_ALIGN`. Questa è la riga che "muove il cursore" in avanti nel buffer.

Nota la differenza tra `rta_len` e il contributo a `nlmsg_len`:

- `rta_len = 10` (valore esatto, header + stringa)
- contributo a `nlmsg_len` = `RTA_ALIGN(10) = 12` (arrotondato, per garantire che il prossimo attributo sia allineato)

Riga 7 — valore di ritorno

```
1 return a;
```

Il puntatore all'attributo viene restituito **solo per gli attributi annidati**: il chiamante lo salva e lo passa poi a `close_attr()`. Per gli attributi foglia il valore di ritorno viene ignorato.

La funzione `close_attr()`

Listing 11: close_attr() — corregge rta_len di un attributo annidato

```
1 static void close_attr(struct rtattr *a)
2 {
3     a->rta_len = (char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len) - (
          char *)a;
4 }
```

Quando si apre un attributo annidato con `add_attr(tipo, NULL, 0)`, il suo `rta_len` viene impostato a `RTA_LENGTH(0) = 4` (solo l'header, nessun valore). Questo valore è **provvisorio**: non si sa ancora quanto saranno lunghi i figli.

`close_attr()` corregge `rta_len` a posteriori:

```
rta_len = (fine del messaggio) - (inizio dell'attributo)
```

- `(char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len)`: indirizzo della fine attuale del messaggio
- `(char *)a`: indirizzo dove inizia l'attributo annidato
- la differenza copre: header dell'attributo + tutti i figli

▷ **Analogia**

È come misurare una scatola dopo averla riempita: non sai quanto è grande finché non hai messo dentro tutti gli oggetti. `close_attr()` è il momento in cui prendi il metro e misuri.

Il `main()` — i quattro passi

Passo 1 — Aprire il socket Netlink

Listing 12: Apertura socket e bind

```
int nl = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
struct sockaddr_nl sa = { .nl_family = AF_NETLINK };
bind(nl, (struct sockaddr *)&sa, sizeof(sa));
```

`socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE)` apre un socket verso il sottosistema `NETLINK_ROUTE` del kernel, che gestisce interfacce, indirizzi IP e rotte.

Il `bind()` associa il socket al processo corrente. `nl_pid = 0` dice al kernel di usare il PID del processo come identificatore — il kernel lo sostituisce automaticamente. `nl_groups = 0` significa nessuna iscrizione a gruppi multicast (non vogliamo ricevere notifiche, solo mandare richieste).

▷ **Nota**

La gestione degli errori è omessa per brevità didattica. In codice di produzione ogni chiamata andrebbe verificata.

Passo 2 — Costruire il messaggio

Listing 13: Inizializzazione dell'header e di ifinfomsg

```
memset(&msg, 0, sizeof(msg));
msg.hdr.nlmsg_len   = NLMSG_LENGTH(sizeof(msg.ifi));
msg.hdr.nlmsg_type  = RTM_NEWLINK;
msg.hdr.nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK
                    | NLM_F_CREATE  | NLM_F_EXCL;
msg.ifi.ifi_family  = AF_UNSPEC;
```

`NLMSG_LENGTH(x)` calcola la lunghezza iniziale del messaggio: header Netlink (16 byte) più il payload fisso `x`:

```
NLMSG_LENGTH(sizeof(ifinfomsg)) = 16 + 16 = 32
```

I flag combinati con OR:

- `NLM_F_REQUEST` — è una richiesta (obbligatorio)
- `NLM_F_ACK` — voglio conferma dal kernel
- `NLM_F_CREATE` — crea l'interfaccia se non esiste
- `NLM_F_EXCL` — restituisci errore se esiste già (senza questo flag, il kernel modificherebbe l'interfaccia esistente invece di segnalare un errore)

`ifi_family = AF_UNSPEC` significa nessuna famiglia specifica — stiamo creando un'interfaccia generica, non IPv4 né IPv6 in particolare. Tutti gli altri campi di `ifinfomsg` restano a zero: `ifi_index = 0` dice al kernel di assegnare un indice libero, `ifi_flags = 0` significa interfaccia DOWN alla creazione.

Listing 14: Aggiunta degli attributi

```
add_attr(IFLA_IFNAME, "dummy0", strlen("dummy0") + 1);

struct rtattr *info = add_attr(IFLA_LINKINFO, NULL, 0);
    add_attr(IFLA_INFO_KIND, "dummy", strlen("dummy") + 1);
close_attr(info);
```

Perché `strlen() + 1`? La stringa viene inviata con il suo terminatore `0`. Il kernel legge il valore dell'attributo come stringa C e ha bisogno del terminatore per sapere dove finisce.

Struttura dell'annidamento: `IFLA_LINKINFO` è un attributo contenitore. Al suo interno si trovano attributi figli che descrivono il tipo di interfaccia. `IFLA_INFO_KIND` contiene la stringa `"dummy"` — il kernel legge questa stringa, cerca il modulo `dummy.ko`, e lo chiama per creare l'interfaccia.

▷ **Confronto con veth**

Per una veth, dentro `IFLA_LINKINFO` andrebbero aggiunti anche `IFLA_INFO_DATA` e `VETH_INFO_PEER` con la descrizione del peer. Per dummy non c'è nessun dato extra — il tipo "dummy" non ha parametri.

Passo 3 — Inviare il messaggio

Listing 15: Invio al kernel

```
printf("Messaggio: %d byte\n", msg.hdr.nlmsg_len);
send(nl, &msg, msg.hdr.nlmsg_len, 0);
```

`send()` invia esattamente `msg.hdr.nlmsg_len` byte a partire dall'indirizzo di `msg`. Questo è il motivo per cui la struttura deve essere contigua in memoria: `send()` non sa niente di Netlink — manda byte consecutivi.

Il quarto argomento `0` indica nessun flag speciale. Si potrebbe usare `sendmsg()` per maggior controllo, ma `send()` è sufficiente per messaggi semplici.

Passo 4 — Leggere la risposta

Listing 16: Lettura dell'ACK

```
char risposta[256];
recv(nl, risposta, sizeof(risposta), 0);
struct nlmsghdr *h = (struct nlmsghdr *)risposta;
struct nlmsgerr *e = (struct nlmsgerr *)NLMSG_DATA(h);

if (h->nlmsg_type == NLMSG_ERROR && e->error != 0) {
    fprintf(stderr, "Errore: %d", -e->error);
    if (-e->error == 1)  fprintf(stderr, " (EPERM: serve sudo)");
    if (-e->error == 17) fprintf(stderr, " (EEXIST: dummy0 esiste
        gia')");
    fprintf(stderr, "\n");
```

```
    close(nl); return 1;
}
```

La risposta è sempre un messaggio di tipo `NLMSG_ERROR`. La struttura `nlmsgerr` contiene il campo `error`: se è 0 significa successo, se è negativo è il valore `-errno` del kernel.

`NLMSG_DATA(h)` è una macro che restituisce il puntatore al payload del messaggio (i byte dopo l'header `nlmsghdr`):

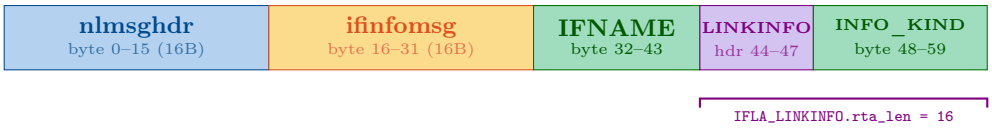
```
#define NLMSG_DATA(nlh) ((void *)(((char *)(nlh)) + NLMSG_HDRLEN))
```

Gli errori più comuni:

errno	Costante	Causa
1	EPERM	Mancano i privilegi (CAP_NET_ADMIN)
17	EEXIST	dummy0 esiste già (NLM_F_EXCL)
22	EINVAL	Messaggio malformato o attributo non valido

La struttura del messaggio in memoria

Il messaggio finale occupa **60 byte** nel buffer. Ogni sezione corrisponde a una struttura o a un attributo:



Il dump esadecimale completo, con annotazioni:

```
nlmsghdr (offset 0):
3c 00 00 00   nlmsg_len = 60
10 00           nlmsg_type = 16 = RTM_NEWLINK
05 06           nlmsg_flags = REQUEST|ACK|CREATE|EXCL
00 00 00 00   nlmsg_seq = 0
00 00 00 00   nlmsg_pid = 0

ifinfomsg (offset 16):
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
      tutto zero:  nuova interfaccia, indice libero, DOWN

rtattr IFLA_IFNAME (offset 32):
0b 00   rta_len = 11 (4 + 7 byte stringa)
03 00   rta_type = 3 = IFLA_IFNAME
64 75 6d 6d 79 30 00   "dummy0\0"
00         padding (11 → allineato 12)

rtattr IFLA_LINKINFO (offset 44):
10 00   rta_len = 16 (scritto da close_attr)
12 00   rta_type = 18 = IFLA_LINKINFO

rtattr IFLA_INFO_KIND (offset 48, figlio di LINKINFO):
0a 00   rta_len = 10 (4 + 6 byte stringa)
01 00   rta_type = 1 = IFLA_INFO_KIND
64 75 6d 6d 79 00   "dummy\0"
00 00 00   padding (10 → allineato 12)
```

Le macro Netlink fondamentali

Le macro definite in `<linux/netlink.h>` e `<linux/rtnetlink.h>` gestiscono tutto l'allineamento e i calcoli di offset. Vale la pena conoscerle perché appaiono in ogni programma che usa Netlink.

Macro	Significato
`NLMSG_HDRLEN`	Dimensione di `nlmsghdr` allineata (16)
`NLMSG_LENGTH(x)`	Lunghezza messaggio con payload `x`: `HDRLEN + x`
`NLMSG_ALIGN(x)`	Arrotonda `x` al multiplo di 4 superiore
`NLMSG_DATA(h)`	Puntatore al payload (byte dopo l'header)
`NLMSG_OK(h,n)`	Vero se il messaggio è valido e non troncato
`NLMSG_NEXT(h,n)`	Avanza al messaggio successivo nel buffer
`RTA_LENGTH(x)`	Lunghezza rtattr con valore di `x` byte
`RTA_ALIGN(x)`	Arrotonda `x` al multiplo di 4 superiore
`RTA_DATA(a)`	Puntatore al valore dell'attributo
`RTA_OK(a,n)`	Vero se l'attributo è valido
`RTA_NEXT(a,n)`	Avanza all'attributo successivo

[!] Non calcolare mai gli offset a mano

Usare sempre queste macro. L'allineamento a 4 byte è obbligatorio: un offset non allineato causa un `EINVAL` dal kernel o, peggio, dati corrotti letti silenziosamente.

Evoluzione del messaggio — `nlmsg_len` passo per passo

`nlmsg_len` cresce ad ogni chiamata ad `add_attr()`. Seguire il suo valore permette di capire esattamente quando e quanto cresce il messaggio:

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 (non modifica nlmsg_len)	60
Totale		**60 byte**

`close_attr()` non modifica `nlmsg_len` — aggiorna solo `rta_len` dell'attributo annidato, ricalcolandola come differenza tra la posizione finale del messaggio e la posizione di inizio dell'attributo: $60 - 44 = 16$.

Cosa fa il kernel quando riceve il messaggio

Il kernel riceve i 60 byte e li processa in sequenza:

1. Legge `nlmsg_type = RTM_NEWLINK` e instrada il messaggio alla funzione `rtnetlink_newlink()` in `net/core/rtnetlink.c`.
2. Legge `ifinfomsg`: `ifi_index = 0` significa interfaccia nuova, `ifi_family = AF_UNSPEC` significa nessun filtro per famiglia.
3. Scansiona gli attributi `rtattr` con `RTA_NEXT()`:
   - `IFLA_IFNAME`: salva il nome `"dummy0"`

- `IFLA_LINKINFO`: entra nel blocco annidato
- `IFLA_INFO_KIND`: legge la stringa `"dummy"`, cerca il modulo con `rtnl_link_ops_get("dummy")`

4. Trova il driver dummy registrato con `rtnl_link_register()`.
5. Alloca un `net_device`, lo inizializza con le operazioni del driver dummy (inclusa `dummy_xmit`).
6. Registra il dispositivo nella lista dei `net_device` del namespace corrente con `register_netdevice()`.
7. Invia l'ACK di successo (`nlmsgerr.error = 0`).

▷ Cosa succede dopo

Da questo momento `dummy0` esiste come `net_device` nel kernel. È visibile con `ip link show dummy0`, si può configurare con `ip addr add`, e il kernel la tratta esattamente come `eth0` o `lo` per tutto ciò che riguarda routing, firewall e socket. L'unica differenza è che il suo driver butta via ogni pacchetto in uscita con `dev_kfree_skb()`.

Riferimenti: `man 7 netlink`, `man 7 rtnetlink`, `include/uapi/linux/rtnetlink.h`, `net/core/rtnetlink.c`, `drivers/net/dummy.c`