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

Hands-On #9

Interfacce virtuali: creare una veth pair via Netlink da C

Last Update: 13 aprile 2026

Scenario

Nell'HO precedente abbiamo imparato a costruire messaggi Netlink per creare un'interfaccia `dummy` — un'operazione semplice: 60 byte, nessun parametro extra, nessun annidamento profondo.

Oggi affrontiamo il caso reale: la **veth pair**. Una veth pair è una coppia di interfacce di rete virtuali collegate tra loro come un cavo — quello che entra da un lato esce dall'altro. È il meccanismo su cui si basa tutto il networking dei container: ogni container Docker, LXC o Podman ha una veth pair che lo collega al bridge dell'host.

Il messaggio Netlink per creare una veth pair è più complesso: 96 byte invece di 60, quattro livelli di annidamento, e un caso speciale che rompe il pattern normale di `add_attr()`. Costruirlo a mano ci darà una comprensione profonda di come funziona il networking a livello kernel.

Cos'è una veth pair

Il lato interno e il lato esterno

Ogni interfaccia di rete Linux ha due lati. Il **lato interno** è rivolto verso il kernel: espone funzioni standard (`ndo_start_xmit`, `ndo_open`, `ndo_get_stats64`) identiche per tutti i tipi di interfaccia. Il **lato esterno** è ciò che il driver fa quando il kernel chiama `ndo_start_xmit`:

Tipo	Cosa fa `ndo_start_xmit`
`eth0`	scrive nel registro DMA → cavo fisico
`lo`	chiama `netif_rx(skb)` → ritorna al kernel
`tun0`	mette in coda → `read(fd)` del processo
`dummy0`	chiama `dev_kfree_skb(skb)` → nulla
`veth0`	chiama `veth1->netif_rx(skb)` → altra interfaccia

Per il kernel sono tutti `net_device` identici. Routing, firewall, Netlink e socket funzionano allo stesso modo su tutti.

La veth e i namespace

Senza namespace le due estremità condividono lo stesso stack IP e il traffico non simula macchine distinte. Lo scopo reale di una veth pair è collegare due network namespace:

```
namespace ns-A                                    namespace ns-B
Stack IP
[Stack IP]
veth0 (10.0.1.1) ⟷ veth1 (10.0.1.2)
```

I frame entrano da un lato ed escono dall'altro.
È esattamente ciò che fa Docker per ogni container.

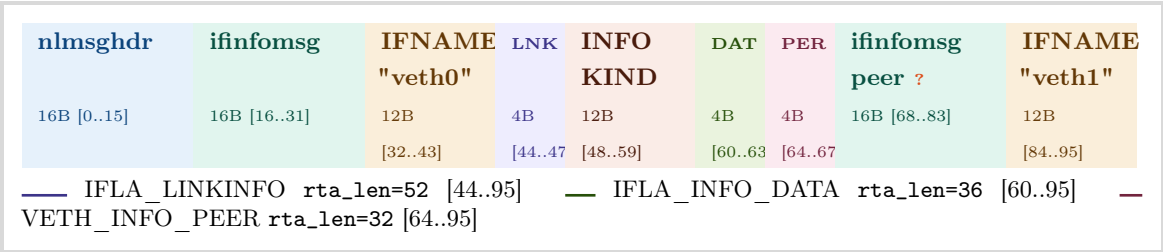
Confronto: shell vs Netlink

Il comando `ip` fa esattamente quello che faremo noi in C: costruisce un messaggio Netlink di 96 byte e lo manda al kernel.

Shell (`ip`)	**Netlink (C)**
`ip link add veth0 type veth peer name veth1`	`RTM_NEWLINK + attributi annidati`
`ip netns add ns-A`	`unshare(CLONE_NEWNET)`
`ip link set veth1 netns ns-A`	`RTM_NEWLINK + IFLA_NET_NS_PID`
`ip addr add 10.0.1.1/30 dev veth0`	`RTM_NEWADDR`
`ip addr add 10.0.1.2/30 dev veth1`	`RTM_NEWADDR`

Struttura del messaggio Netlink per la veth pair

Il messaggio occupa **96 byte** in memoria. Ogni blocco ha larghezza proporzionale alla sua dimensione reale.



I tre livelli di annidamento (come le matrioske)

La caratteristica più importante del messaggio veth è la struttura ad albero degli attributi. Ci sono tre livelli sovrapposti:

Livello 1. `IFLA_LINKINFO` — il primo contenitore [offset 44..95]. Raggruppa tutto ciò che riguarda il tipo di interfaccia. Il kernel lo riconosce e scansiona il contenuto ricorsivamente. `rta_len = 52` = scritto da `close_attr()` a posteriori: $96 - 44 = 52$. Contiene direttamente: `IFLA_INFO_KIND` e `IFLA_INFO_DATA`.

Livello 2. `IFLA_INFO_DATA` — il secondo contenitore [offset 60..95]. Esiste perché i parametri specifici del driver devono essere separati dal tipo. `IFLA_INFO_KIND` dice *quale* driver (`"veth"`); `IFLA_INFO_DATA` dice *come configurarlo*. Per `dummy` questo blocco non esiste. `rta_len = 36` $= 96 - 60 = 36$.

Livello 3. `VETH_INFO_PEER` — il terzo contenitore [offset 64..95]. Specifico del driver veth. Descrive la seconda interfaccia della coppia. Il kernel passa questo blocco direttamente al driver, che lo interpreta. `rta_len = 32` $= 96 - 64 = 32$. Contiene: un `ifinfomsg` (caso speciale) + `IFLA_IFNAME "veth1"`.

[!] Il caso speciale di `VETH_INFO_PEER`

Il driver veth si aspetta un `struct ifinfomsg` immediatamente dopo l'header di `VETH_INFO_PEER`, prima degli attributi del peer. Questo `ifinfomsg` **non** si aggiunge con `add_attr()` ma a mano:

```
msg.hdr.nlmsg_len += sizeof(struct ifinfomsg);
```

È l'unica eccezione al pattern uniforme.

Tabella completa dei campi

Offset	B	Campo	Significato	Valore
struct nlmsghdr [0..15, 16B]				
0	4	`nlmsg_len`	Lunghezza totale del messaggio	`96`
4	2	`nlmsg_type`	Tipo: crea/modifica interfaccia	`16=RTM_NEWLINK`
6	2	`nlmsg_flags`	REQUEST|ACK|CREATE|EXCL	`0x0605`
8	4	`nlmsg_seq`	Numero di sequenza	`0`
12	4	`nlmsg_pid`	PID mittente (kernel lo imposta)	`0`
struct ifinfomsg [16..31, 16B] — interfaccia veth0				
16	1	`ifi_family`	AF_UNSPEC: nessuna famiglia specifica	`0`
17	1	*`__ifi_pad`*	*Padding strutturale*	*0x00*
18	2	`ifi_type`	Tipo hardware: 0=kernel decide	`0`
20	4	`ifi_index`	0 = assegna indice libero	`0`
24	4	`ifi_flags`	0 = nasce DOWN	`0`
28	4	`ifi_change`	Maschera flag: 0 = nessun cambio	`0`
rtattr IFLA_IFNAME [32..43, 12B] — nome veth0				
32	2	`rta_len`	4+6=10; occupa RTA_ALIGN(10)=12B	`10`
34	2	`rta_type`	Nome interfaccia	`3=IFLA_IFNAME`
36	6	`valore`	strlen("veth0")+1=6	`"veth0\0"`
42	2	*`pad`*	*Allinea a offset 44*	*0x00×2*
rtattr IFLA_LINKINFO [44..95, 52B] — contenitore livello 1				
44	2	`rta_len`	96−44=52 (close_attr)	`52`
46	2	`rta_type`	Contenitore tipo interfaccia	`18=IFLA_LINKINFO`
rtattr IFLA_INFO_KIND [48..59, 12B] — figlio di LINKINFO				
48	2	`rta_len`	4+5=9; occupa RTA_ALIGN(9)=12B	`9`
50	2	`rta_type`	Tipo interfaccia: carica il driver	`1=IFLA_INFO_KIND`
52	5	`valore`	strlen("veth")+1=5	`"veth\0"`
57	3	*`pad`*	*Allinea a offset 60*	*0x00×3*
rtattr IFLA_INFO_DATA [60..95, 36B] — contenitore livello 2				
60	2	`rta_len`	96−60=36 (close_attr)	`36`
62	2	`rta_type`	Parametri specifici del driver	`2=IFLA_INFO_DATA`
rtattr VETH_INFO_PEER [64..95, 32B] — contenitore livello 3				
64	2	`rta_len`	96−64=32 (close_attr)	`32`
66	2	`rta_type`	Descrizione del peer veth1	`1=VETH_INFO_PEER`

continua nella pagina successiva

(continua)

Offset	B	Campo	Significato	Valore
struct ifinfomsg peer			**[68..83, 16B] caso speciale**	
68	1	`ifi_family`	AF_UNSPEC	`0`
69	1	*`__ifi_pad`*	*Padding strutturale*	*`0x00`*
70	2	`ifi_type`	0: kernel sceglie per veth1	`0`
72	4	`ifi_index`	0: indice libero per veth1	`0`
76	4	`ifi_flags`	0: veth1 nasce DOWN	`0`
80	4	`ifi_change`	0: nessun flag da cambiare	`0`
rtattr IFLA_IFNAME			**[84..95, 12B] — nome veth1 (figlio di PEER)**	
84	2	`rta_len`	4+6=10; occupa 12B	`10`
86	2	`rta_type`	Nome del peer	`3=IFLA_IFNAME`
88	6	`valore`	strlen("veth1")+1=6	`"veth1\0"`
94	2	*`pad`*	*Porta nlmsg_len a 96*	*`0x00`×2*

Codice di partenza

Il file `veth_create.c` contiene tutto il necessario tranne il cuore del programma: la costruzione del messaggio Netlink. Quella sezione è **omessa** — e bisogna implementarla (Riga 62)

Le funzioni `aggiungi_attr()` e `chiudi_attr()` sono già pronte e documentate. Il tuo compito è usarle in modo annidato per descrivere la struttura ad albero del messaggio.

Listing 1: `veth_create.c` — il codice di partenza (righe 100–135 oscurate)

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <linux/if_link.h>
#include <linux/veth.h>
#include <net/if.h>

struct {
    struct nlmsghdr  hdr;
    struct ifinfomsg ifi;
    char             buf[512];
} msg;

/*
 * Aggiunge un attributo rtattr al buffer del messaggio.
 * - tipo:   IFLA_IFNAME, IFLA_LINKINFO, IFLA_INFO_KIND, ...
 * - valore: puntatore ai dati (NULL per attributi contenitore)
 * - len:    lunghezza del valore (0 per contenitori)
```

```
 * Ritorna il puntatore all'attributo (serve per chiudi_attr).
 */
static struct rtattr *aggiungi_attr(int tipo, const void *valore,
    int len)
{
    struct rtattr *a = (struct rtattr *)
        ((char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len));
    a->rta_type = tipo;
    a->rta_len  = RTA_LENGTH(len);
    if (valore) memcpy(RTA_DATA(a), valore, len);
    msg.hdr.nlmsg_len =
        NLMSG_ALIGN(msg.hdr.nlmsg_len) + RTA_ALIGN(a->rta_len);
    return a;
}

/*
 * Chiude un attributo contenitore aggiornando il suo rta_len
 * a posteriori (quando si conosce la dimensione finale del
    contenuto).
 * Formula: rta_len = posizione_finale - inizio_attributo
 */
static void chiudi_attr(struct rtattr *a)
{
    a->rta_len =
        (char *)&msg + NLMSG_ALIGN(msg.hdr.nlmsg_len) - (char *)a;
}

int main(void)
{
    /* Passo 1: apri socket Netlink */
    int nl = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
    struct sockaddr_nl sa = { .nl_family = AF_NETLINK };
    bind(nl, (struct sockaddr *)&sa, sizeof(sa));

    /* Passo 2: inizializza header e 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_CREATE
                        | NLM_F_EXCL    | NLM_F_ACK;
    msg.ifi.ifi_family  = AF_UNSPEC;

    /** TODO - Creazione del messaggio **/

    /* Passo 3: invia */
    send(nl, &msg, msg.hdr.nlmsg_len, 0);

    /* Passo 4: leggi risposta */
    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 dal kernel: %d\n", -e->error);
        if (-e->error ==  1) fprintf(stderr, " (EPERM: serve sudo)\
           n");
        if (-e->error == 17) fprintf(stderr, " (EEXIST: veth0
```

```
            esiste)\n");
77          close(nl); return 1;
78      }
79
80      printf("veth0 <-> veth1 create.\n");
81      printf("Verifica: ip link show veth0 && ip link show veth1\n");
82      printf("Pulizia:  sudo ip link del veth0\n");
83      close(nl);
84      return 0;
85  }
```

Esercizi

Esercizio 1 — Analisi preliminare

Prima di scrivere codice, rispondi per iscritto a queste domande:

1. Perché nel messaggio veth ci sono tre livelli di annidamento invece di uno solo? Cosa conterrebbe un messaggio "piatto" senza `IFLA_LINKINFO`?
2. Qual è il valore di `nlmsg_len` dopo ogni chiamata ad `aggiungi_attr()`? Compilala nella tabella seguente:

Operazione	+byte	nlmsg_len
`NLMSG_LENGTH(sizeof(ifinfomsg))`	32	32
`aggiungi_attr(IFLA_IFNAME, "veth0", 6)`		
`aggiungi_attr(IFLA_LINKINFO, NULL, 0)`		
`aggiungi_attr(IFLA_INFO_KIND, "veth", 5)`		
`aggiungi_attr(IFLA_INFO_DATA, NULL, 0)`		
`aggiungi_attr(VETH_INFO_PEER, NULL, 0)`		
`nlmsg_len += sizeof(struct ifinfomsg)`		
`aggiungi_attr(IFLA_IFNAME, "veth1", 6)`		
Totale atteso		**96**

3. Perché `chiudi_attr()` va chiamato in ordine inverso rispetto all'apertura? Cosa succede se si chiama prima `chiudi_attr(linkinfo)` e poi `chiudi_attr(peer)`?
4. `rta_len` di `IFLA_LINKINFO` vale 52. Verifica che sia corretto: quanti byte occupa il contenuto di `IFLA_LINKINFO`?

Esercizio 2 — Implementa le righe oscurate

Completa `veth_create.c` implementando le righe 100–135.
Usa `aggiungi_attr()` e `chiudi_attr()` rispettando la struttura ad albero mostrata nel codice nl$_a$*ummy.c.Ricordailcasospecialedell'*`ifinfomsg`*delpeer*.
Compila ed esegui:

```
gcc -Wall -o veth_create veth_create.c
sudo ./veth_create
ip link show veth0
ip link show veth1
sudo ip link del veth0   # cancella entrambe
```

Verifica che `nlmsg_len` alla fine valga esattamente 96. Aggiunge una `printf` prima di `send()` per controllarlo:

```
printf("Dimensione messaggio:  %d byte\n", msg.hdr.nlmsg_len);
```

▷ Suggerimento

Segui la struttura dell'albero mostrato nelle righe oscurate. Ogni `aggiungi_attr(..., NULL, 0)` apre un contenitore e restituisce un puntatore da salvare. Ogni `chiudi_attr(ptr)` lo chiude calcolando `rta_len` a posteriori come *fine* − *inizio*.

▷ Caso speciale

Dopo `aggiungi_attr(VETH_INFO_PEER, NULL, 0)`, prima di aggiungere gli attributi del peer, inserisci:

```
msg.hdr.nlmsg_len += sizeof(struct ifinfomsg);
```

Questo aggiunge i 16 byte del secondo `ifinfomsg` al buffer senza usare `aggiungi_attr()`.

Esercizio 3 — Equivalente da riga di comando

Esegui gli stessi comandi che fa il tuo programma usando `ip`:

```
# 1.  Crea i namespace
sudo ip netns add ns-A
sudo ip netns add ns-B

# 2.  Crea la veth pair
sudo ip link add veth0 type veth peer name veth1

# 3.  Sposta le estremita' nei namespace
sudo ip link set veth0 netns ns-A
sudo ip link set veth1 netns ns-B

# 4.  Configura gli indirizzi IP
sudo ip netns exec ns-A ip addr add 10.0.1.1/30 dev veth0
sudo ip netns exec ns-B ip addr add 10.0.1.2/30 dev veth1

# 5.  Porta le interfacce UP
sudo ip netns exec ns-A ip link set veth0 up
sudo ip netns exec ns-A ip link set lo up
sudo ip netns exec ns-B ip link set veth1 up
sudo ip netns exec ns-B ip link set lo up

# 6.  Testa la connettivita'
sudo ip netns exec ns-A ping -c 3 10.0.1.2

# 7.  Pulizia
sudo ip netns del ns-A
sudo ip netns del ns-B
```

▷ Sbirciare i byte

Per vedere esattamente i 96 byte che `ip` manda al kernel, usa `strace`:

```
sudo strace -e trace=sendmsg -xx ip link add veth0 type veth peer
                                 veth1
```

Esercizio 4 — Nomi parametrici

Modifica `veth_create.c` per accettare i nomi delle due interfacce da riga di comando:

```
sudo ./veth_create myiface0 myiface1
```

Aggiungi la validazione: ogni nome deve essere al massimo `IFNAMSIZ - 1 = 15` caratteri. Usa `strnlen()` e stampa un errore se il limite è superato.

[!] Attenzione

Ricorda che `aggiungi_attr()` copia esattamente `len` byte. Se passi `strlen(nome)+1` il terminatore `\0` viene incluso — obbligatorio affinché il kernel legga la stringa correttamente.