Socket NETLINK

24 ottobre 2009
Linux Day

Sommario

Netlink è un sistema di comunicazione asincrono che permette di collegare e far comunicare task di varia natura sia kernel-space che user-space. Per instaurare la comunicazione occorre creare o aprire un socket detto appunto netlink socket. In altre parole netlink è un sistema comunemente detto IPC ossia "Inter-Process Comunication". Il seguente articolo è basato su [1] e mostra come creare un socket netlink a livello kernel e user space e mettere in comunicazione i vari task rispettando lo specifico protocollo scelto. Si vedrà inoltre come a livello kernel il sistema sia cambiato nel tempo partendo dal kernel 2.6.13 al 2.6.24: a partire da quest'ultima versione, netlink è rimasto fondamentalmente invariato fino ad ora (2.6.30 al momento in cui si scrive).

Il tutto sarà corredato da esempi software liberamente scaricabili da *www.lugman.org* nella sezione *Documentazione*.

1 Introduzione

In un sistema operativo complesso e strutturato come può essere Linux, nasce la necessità di poter comunicare con lo spazio utente. I metodi più comuni sono le interfacce dei file di dispositivo, system call, filesystem virtuali (proc, sysfs, ecc) e i socket NetLink. Questi socket in particolare permettono di realizzare a tutti gli effetti un sistema IPC (*Inter-Process Comunication*) principalmente per la comunicazione tra spazio kernel e spazio utente. In realtà, in quanto IPC, un socket NetLink permette la comunicazione anche tra semplici processi.

I socket NetLink permettono di instaurare una comunicazione asincrona tra i vari task definita da un protocollo specifico. A differenza di altri tipi di socket non è possibile aprire più di un socket NetLink dello stesso tipo (protocollo) e mantenere la comunicazione separata, ma ogni task che aprirà lo specifico socket in linea di massima avrà accesso a tutti i dati che fluiscono su questo protocollo NetLink; un esempio di socket che permettono di creare vari "canali" si comunicazione sono i socket INET con protocollo IP: il protocollo ethernet definisce una maschera di rete ed incapsula il protocollo IP definendo un indirizzo (IP appunto) diverso per ogni task messo in comunicazione sulla stessa maschera. NetLink invece definisce il proprio protocollo semplicemente assegnandogli un numero. Dopo l'header NetLink, l'area dati potrà essere formattata come si vuole, esattamente come potrebbe essere l'area dati (payload) di un unico pacchetto TCP non frammentato.

La comunicazione tramite socker NetLink è una comunicazione asincrona full-duplex. I messaggi possono essere di tipo unicast o multicast (broadcast). Questo è l'unico modo per discriminare chi deve ricevere lo specifico messaggio. Chiaramente l'accesso a NetLink richiede chiamate di sistema, ma nonostante ciò non sono richiesti speciali permessi se la comunicazione è unicast, mentre per le comunicazioni multicast il task deve avere i permessi di amministrazione.

Le applicazioni principali in cui viene usata la comunicazione con NetLink sono normalmente applicativi di rete come Firewall, gestione tabelle ARP, routing, ecc, ma anche sistemi di notifica di eventi come uDev. Infatti NetLink permette anche al kernel di comunicare un messaggio anche urgente allo spazio utente cosa che gli altri sistemi IPC a disposizione (come /proc, syscall, ecc) non possono fare se prima non è stato un processo in spazio utente ad accedere alla particolare risorsa a livello kernel (per esempio l'apertura di un device driver).

2 Socket NETLINK API

Vediamo ora come definire un socket NetLink in un sistema Linux a livello utente e kernel. Il kernle mette a disposizione dello user-space, tramite i suoi header, tutte le chiamate e le strutture necessarie e come si vedrà non è molto differente dal trattare un comune socket. Lo spazio kernel invece è molto più complesso dello spazio utente non solo perchè occorre rispettare un certo metodo di programmazione, ma soprattutto perchè l'evoluzione di questo codice è spesso molto rapida e porta a modifiche anche strutturali particamente in ogni cosa: dalle strutture alle funzioni alla gestine interna dei buffer. Al momento in cui si scrive, il kernel è alla versione 2.6.31 e NetLink è pressochè quello della versione 2.6.24. Gli esempi sviluppati però partono dal kernel 2.6.13 e questo permetterà di vederne l'evoluzione che, per il programmatore identifica fondamentalmente tre step: prima del 2.6.14, il 2.6.14 e il 2.6.24.

2.1 User Space

La base di partenza è *linux/netlink.h* sia per kernel che per user space. In questo file sono definiti tutti i codici numerici che identificano il protocollo che uno specifico socket NetLink può supportare. Se un nuovo protocollo venisse aggiunto ufficialmente, quel protocollo deve essere aggiunto in questo file. Il numero di protocolli supportati dal kernel 2.6.24 è di 19 e il numero massimo di protocolli supportabili è attualmente 32 (definito dalla macro `MAX_LINKS`).

Dal punto di vista dello spazio utente non vi è particolare differenza con gli altri tipi di socket. Un socket infatti viene aperto sempre nello stesso modo e quindi anche le chiamate a disposizione sono sempre:

```
int socket(int domain, int type, int protocol);
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

`socket()` richiede come primo parametro il tipo di comunicazione che intendiamo effettuare indicato come `PF_NETLINK` per poterci interfacciare con il kernel. Il tipo può essere `SOCK_DGRAM` o `SOCK_RAW`. In questo articolo si userà `SOCK_RAW` in quanto si vuole accedere ad ogni livello del messaggio. NetLink supporta comunque anche il trasporto dati di tipo Datagram[1] non è necessario instaurare una comunicazione Connection Oriented (come in TCP) e non è a priori rispettata la sequenza dei dati inviati. Chiaramente essendo implementato a livello kernel ed essendo utilizzato a livello locale, questo tipo di comunicazione risulta comunque molto affidabile. Infine vi è `protocol` che non è altro che il codice corrispondente al protocollo che intendiamo utilizzare sul socket appena aperto.

`bind()` invece richiede come primo parametro (`sockfd`) il descrittore del socket ottenuto dalla chiamata `socket()`. Segue la struttura che descrive il socket passata con un cast di tipo `sockaddr` e la rispetiva lunghezza come ultimo parametro. Il tipo di variabile `sockaddr` descrive una struttura generica del socket la quale dipende in realtà dal tipo di socket che stiamo configurando. La struttura per la configurazione del socket NetLink è `sockaddr_nl` così definita in *linux/netlink.h*:

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

Questa prima struttura definisce il tipo di famiglia del socket, dove andrà indicato chiaramente AF_NETLINK. Dopo di che vengono definiti `nl_pid` e `nl_groups`. Questi due parametri sono forse i principali; infatti grazie a questi è possibile instaurare una comunicazione dal momento in cui viene eseguito il bind del socket. Come si vedrà tra breve, il destinatario di un messaggio è identificato normalmente dal PID del processo. Se pari al campo `nl_pid` allora il messaggio viene accettato dal processo ricevente. `nl_groups` invece serve per definire un gruppo di appartenenza nel caso venga lanciato un messaggio multicast.

La seconda struttura che segue quella appena definita è l'header del pacchetto NetLink, definito come segue:

```
struct nlmsghdr {
        __u32   nlmsg_len;      /* Length of message including header */
        __u16   nlmsg_type;     /* Message content */
```

[1] Un esempio di socket `SOCK_DGRAM` è il protocollo UDP

```
        __u16   nlmsg_flags;    /* Additional flags */
        __u32   nlmsg_seq;      /* Sequence number */
        __u32   nlmsg_pid;      /* Sending process PID */
};
```

Questa struttura fa parte del messaggio e non del socket come la precedente. Qui vengono definite tutte le varie aree del messaggio il quale avrà come intestazione (header appunto) la struttura `nlmsghdr`. Di per se i vari campi hanno un significato limitato nel senso che il loro settaggio non pregiudica il funzionamento del sistema di comunicazione. I campi infatti sono quasi tutti informazioni accessorie. Probabilmente il campo più importante è `nlmsg_len` che definisce la lunghezza del messaggio e potrebbe essere utile per esempio alla funzione `skb_pull()` del kernel, ma di fatto resta una informazione accessoria come detto. Tramite questo valore è possibile determinare tutti gli offset e accedere all'area dati detta *payload* ("carico pagante").

2.2 Kernel Space

Nota: Quanto segue descrive NetLink basandosi sulla versione **2.6.24.1** del kernel. Le versioni precedenti verranno descritte brevemente in ultima analisi. Questa scelta è motivata dal fatto che parte degli esempi che verranno proposti si riferiscono a versioni precedenti del kernel ed è interessante vedere come il sistema si è evoluto.

Quanto elencato fino ad ora è principalmente legato al solo user-space, tranne `nlmsghdr` che viene usata anche a livello kernel. Il kernel è però più complesso perchè tende ad astrarre e gestire NetLink come se fosse un socket qualsiasi. Quindi in kernel space sarà sempre definita una variabile tipo `sock`. Questo tipo di struttura è definita in *net/sock.h* e permette di gestire completamente il socket[2]. A questa però va aggiunta una variabile necessaria per la gestione del buffer del socket, ossia la struttura `sk_buff` definita in *linux/skbuff.h*:

```
struct sk_buff {         /* These two members must be first. */
        struct sk_buff       *next;
        struct sk_buff       *prev;
        struct sk_buff_head  *list;
        struct sock          *sk;
        char                 cb[40];   // Control Buffer
        [...]
}
```

`sk_buff` nello specifico è una lista, come si può notare dai primi due elementi. Contiene inoltre l'indirizzo del socket di appartenenza e un campo `char` che permette di accedere ai vari campi di altre strutture per settarne alcuni parametri. In particolare, per il socket NetLink si può configurare il buffer in termini di PID e gruppo rispettivamente del mittente e del destinatario tramite la macro `NETLINK_CB`. Questi valori vengono scritti nell'area dati `cb`, ossia il *Control buffer* del socket che non è altro che un'area in cui inserire dei dati "provati" specifici per il tipo di socket e/o protocollo. In particolare la macro suddetta, definita come:

```
NETLINK_CB(skb)   (*(struct netlink_skb_parms*)&((skb)->cb))
```

definisce l'accesso alla struttura dati formattata secondo la struttura `netlink_skb_parms`:

```
struct netlink_skb_parms {
        struct ucred            creds;          /* Skb credentials      */
        __u32                   pid;
        __u32                   dst_groups;
        kernel_cap_t            eff_cap;
        __u32                   loginuid;       /* Login (audit) uid */
        __u32                   sid;            /* SELinux security id */
};
```

questa struttura, come quasi tutto il resto, non è documentata anche se dai nomi è possibile capire il significato dei campi. Come vedremo però il suo ruolo è relativamente marginale, quantomeno nella ricezione o trasmissione che sono interamente affidate a funzioni specifiche.

[2] Non viene postata in quanto enorme e non significativa per questo articolo

Le API del kernel mettono quindi a disposizione macro e funzioni per il management sia dei socket sia, soprattutto, del buffer. Le funzione di gestione specifiche per il socket NetLink sono definite in *linux/netlink.h* e implementate in *net/netlink/af_netlink.c*; le funzioni sono molteplici ma quelle rese effettivamente disponibili (`extern`) sono poche perchè tutte le funzioni necessarie sono note allo specifico socket e quindi contenute nella sua struttura. Queste funzioni sono identificate dal nome *netlink_* . Le principali funzioni messa a disposizione sono:

`struct sock netlink_kernel_create (struct net *net, int unit, unsigned int groups, void (*input)(struct sk_buff *skb), struct mutex *cb_mutex, struct module *modul)`

Questa funzione è fondamentale e serve per creare o aprire un socket NetLink a livello kernel; attenzione però che della versione 2.6.24, la creazione del socket può avvenire solo da parte del kernel, mentre è stato deciso che un programma in user-space non può aprire un socket NetLink senza che questo non sia stato prima creato a livello kernel. Nelle versioni passate era invece possibile creare una comunicazione NetLink tra due processi senza che il kernel avesse creato il socket precedentemente. Questa funzione è praticamente come la funzione `socket()` in user-space specifica per NetLink. `*net`, introdotta dalla versione 2.6.24, viene sempre settata a `&init_net` per qualsiasi tipo di socket NetLink all'interno del kenrel. `unit` identifica il numero del protocollo che dovrà supportare il socket, ossia un valore per differenziarlo da altri socket NetLink aperti. `groups` dovrebbe identificare il gruppo di appartenenza del kernel; di fatto questo valore viene scritto nella NetLink table (variabile `nl_table`) in corrispondenza dell'omonimo campo della struttura; nonostante ciò, modifiche di questo valore non hanno mai dato problemi funzionali. `input` è una funzione che verrà chiamata ogni qual volta verrà ricevuto un messaggio valido[3] per il kernel. In particolare il puntatore a questa funzione è contenuto nel campo `data_ready` della struttura `netlink_socket` (vedere l'implementazione in *net/netlink/af_netlink.c* della funzione `nlk_sk()` per ulteriori informazioni). Infine vi sono `*cb_mutex` e `*modul` i quali vengono, così come `groups`, inseriti nella struttura globale `nl_table` che identifica tutti i tipi di socket NetLink aperti. Il primo di questi due è un mutex che dovrebbe essere utilizzato per gestire gli accessi al socket ed evitare le race concurrency, mentre il secondo identifica il modulo corrente; tipicamente vengono settati rispettivamente a `NULL` e a `THIS_MODULE`.

`int netlink_unicast (struct sock *ssk, struct sk_buff *skb, __u32 pid, int nonblock)`

Una volta creato il socket occorre comunicare. Questa funzione invia un messaggio unicast da kernel indirizzato ad uno specifico processo, normalmente identificato dal suo PID. Per questa ragione i parametri indicati nei buffer sono abbastanza inutili: questa funzione ha infatti dominio assoluto sul send del messaggio. Ciò implica che la configurazione minuziosa delle strutture che compongono il messaggio è utile soprattutto al ricevente per capire chi gli ha spedito un messaggio. É chiaro che questi dati sono facilmente falsificabili. I parametri richiesti sono il socket (`*ssk`) creato tramite `netlink_kernel_create`, il buffer contenente i dati da inviare, il PID del processo ed infine `nonblock` che è un valore che indica come effettuare il tipo di chiamata. Tipicamente viene settato a `MSG_DONTWAIT` in modo che questa funzione non sia interruttiva; in altre parole il messaggio viene inviato nella speranza che ci sia qualcuno a prelevarlo. In caso contrario andrà perso.

`int netlink_broadcast (struct sock *ssk, struct sk_buff *skb, __u32 pid, __u32 group, int allocation)`

Analogamente a prima questa funzione invia un messaggio di tipo multicast. Fino a `pid` i termini sono uguali alla funzione precedente[4]. `group` identifica i gruppi a cui è destinato un messaggio. Si noti che nelle versioni precedenti alla 2.6.24 il valore di `group` era un valore digitale pari all'OR dei gruppi di destinazione all'ascolto. Per esempio se un processo ha settato il suo gruppo pari a 1 e un secondo processo ha settato 2, se si vuole mandare a entrambi il messaggio basta che `group` sia settato a 1|2=3; dalla versione 2.6.24 invece il numero di group funziona "al contrario": se il messaggio viene inviato con gruppo pari a "1", il messaggio sarà inviato al gruppo che ha il primo bit a "1" (come per esempio i processi dispari "1", "3", ecc); in altre parole adesso l'OR dei gruppi è impostato nei processi in ricezione e non nel messaggio spedito. `allocation` invece serve al kernel per sapere come eseguire kmalloc per allocare la memoria per i messaggi. Il kernel infatti crea un messaggio broadcast inviando tanti messaggi unicast e deve quindi allocare l'area

[3] Con "valido" si intende destinato al kernel

[4] Nonostante ciò `pid` non ha vaore in questa funzione, o comunque nei test fatti non si è mai riusciti a inviare un messaggio multicast a un gruppo più un processo specifico, cosa che invece dalla documentazione trovata sembrava possibile.

dati per ciascuno di questi. Anche in questo caso le scelte si riducono normalmente a `GFP_KERNEL` se non sono richieste prestazioni troppo elevate. `GFP_ATOMIC` è sicuramente più efficiente.

Queste sono le chiamate principali per gestire NetLink a livello kernel. Una volta creato il socket quindi è possibile già interagire con esso. Questo è compito del kernel, ma la funzione di gestione deve essere definita all'interno del modulo ed è la funzione `input` citata in precedenza. Questa funzione viene invocata dal kernel ed è quindi senza contesto il che implica che non è possibile indicizzare un buffer da user-space da qui. Proprio perchè viene chiamata dal kernel questa funzione è bene che sia il più rapida possibile, al limite atomica. Se le operazioni sono molte normalmente è bene ricorrere a delle segnalazioni come spinlock, semafori, mutex, ecc sbloccati da questa funzione i quali "risvegliano" uno specifico kernel-task delegato a svolgere le operazioni implementate. Negli esempi proposti, per semplicità tutto verrà fatto in questa funzione.

Ma cosa accade quindi quando arriva un messaggio? Il kernel lo riceve e valuta per chi è il messaggio in questione. Se il messaggio è destinato al kernel, il valore del campo `sockaddr_nl` impostato da chi trasmette è 0. A differenza di ciò che avveniva nelle versioni precedenti, il contenuto del messaggio che è memorizzato nel buffer (`sk_buff`) relativo al socket NetLink, viene "estratto" e passato alla funzione `input` che verrà immediatamente invocata. Qui vi sono i primi cambiamenti strutturali nella gestione dei messaggi NetLink apportati nella versione 2.6.24. Per prima cosa la definizione della funzione `input` così definita:

```
void (*input)(struct sk_buff ~*skb)
```

Tale funzione potrà elaborare il buffer `*skb` nel senso che è permessa la lettura e la scrittura del buffer esclusivamente nel contesto della funzione `input`. Quindi a differenza di ciò che avveniva in passato non è possibile reinviare subito questo buffer, ma occorre crearne un altro. Utilizzare questo buffer comporta crash e instabilizzazioni gravi di tutto il sistema; questo perchè all'uscita di questa funzione il sistema rilascia le risorse legate a questo buffer secondo l'implementazione interna di NetLink e dei socket in generale. Questo non implica che non esistano più le funzioni `skb_recv_datagram()` e `skb_dequeue()` utilizzate maggiormente nelle vecchie versioni del kernel, ma vengono semplicemente usate in altre sezioni di codice più internamente. L'uso non ne è quindi pregiudicato.

Ad ogni modo deve essere possibile inviare un messaggio. Come detto ora occorre crearlo e per farlo entrano in gioco le funzioni taggate con *skb_* per la gestione del buffer del socket (`struct sk_buff` citato in precedenza) e, come si può intuire, queste funzioni sono generiche per ogni tipo di socket. Esse sono definite in *linux/skbuff.h* e implementate in *net/core/datagram.c* e *net/core/skbuff.c*. Le funzioni principalmente usate negli esempi sono:

`static inline struct sk_buff *alloc_skb (unsigned int size, gfp_t priority)`
: Alloca il buffer del socket chiamando la funzione `__alloc_skb()`. La dimensione del buffer deve essere comprensiva della dimensione del payload e dell'header NetLink. Questo almeno negli esempi che verranno proposti visto che la formattazione dell'header può essere abbastanza arbitraria. Per esempio il NetLink di comunicazione con uDev crea una formattazione dell'header mediante delle sue regole interne (per maggiori informazioni vedere il file *lib/kobject_uevent.c*).

`static inline unsigned char *skb_put (struct sk_buff *skb, unsigned int len)`
: Questa funzione è fondamentale: indica al buffer `*skb` che è stato inserito una nuova area dati di lunghezza `len`. Solo dopo questa funzione sarà possibile agire sui dati del socket (header e/o payload che sia). Questa funzione modifica il puntatore `skb->data` e lo ritorna come risultato segnalando sempre all'interno di `*skb` che la lunghezza del buffer è variata. Attenzione che questa funzione può eccedere anche le dimensioni massime del buffer mandando in panic o in instabilità il kernel.

Esistono poi altre funzioni di gestione come `skb_pull` o `skb_push` le quali rimuovono o aggiungono dati a partire dall'inizio del buffer. Negli esempi proposti non verranno utilizzate.

Rimane quindi l'ultima funzione legata all'uscita e quindi alla distruzione del socket. Per fare ciò si usano le API fornite dal kernel per la gestione di un socket generico, ossia:

`void sock_release(struct socket *sock)` Questa funzione permette di eliminare dallo stack il socket normalmente chiamando una funzione apposita puntata da `sock->ops->release`. La funzione è implementata in *net/socket.c*.

2.2.1 Kernel inferiore al 2.6.24

Fino al kernel 2.6.23.x, NetLink si è mantenuto pressochè invariato dalla versione 2.6.14. La struttura `netlink_skb_parms` è la stessa rispetto al kernel 2.6.24, diversa rispetto al kernel inferiore alla versione 2.6.14, ma ciò che cambia principalmente è la funzione di creazione del socket:

`struct sock netlink_kernel_create (int unit, unsigned int groups, void (*input)(struct sk_buff *skb), struct mutex *cb_mutex, struct module *modul)`
: Questa funzione rispetto a quella precedente a quella del kernel 2.6.14 introduce solo il mutex e l'identificatore del modulo.

la funzione di input rimane invece sempre la stessa rispetto alle versioni del kernel inferiori alla 2.6.14.

2.2.2 Kernel inferiore al 2.6.14

Il codice di seguito riportato è preso dal kernel 2.6.13.1. In questo kernel le modifiche sono strutturali. Verranno riportate quelle individuate, ma si consideri che in queste versioni del kernel (e quindi in queste versioni di NetLink) il modulo non viene segnalato "in uso" nel caso in cui un processo abbia aperto un socket NetLink. Inoltre un processo poteva creare un canale NetLink anche senza che il kernel avesse creato il socket specifico. Ad ogni modo, la struttura `netlink_skb_parms` per kernle 2.6.13.x (o inferiore) è la seguente:

```
struct netlink_skb_parms {
        struct ucred            creds;          /* Skb credentials      */
        __u32                   pid;
        __u32                   groups;
        __u32                   dst_pid;
        __u32                   dst_groups;
        kernel_cap_t            eff_cap;
        __u32                   loginuid;       /* Login (audit) uid */
};
```

Anche in questa versione del kernel il suo ruolo di questi parametri è relativamente marginale al corretto funzionamento.

Seguono quindi le funzioni di base per la gestione del socket:

`struct sock netlink_kernel_create (int unit, void (*input)(struct sock *sk, int len))`
: Questa funzione è fondamentale e serve per aprire o creare un socket NetLink a livello kernel. `unit` identifica il numero del protocollo che dovrà supportare il socket, ossia un valore per differenziarlo da altri socket NetLink aperti, mentre `input` è una funzione che verrà chiamata ogni qual volta verrà ricevuto un messaggio valido[5] per il kernel. In questa versione la funzione input accedeva a tutto il socket e quindi si prendeva la briga di esaminarne il contenuto, estrarre i dati, modificarli e/o reinviarli con le opportune funzioni che vedremo di seguito.

Una volta creato il socket è possibile già interagire con esso. Questo è compito del kernel, ma la funzione di gestione deve essere definita all'interno del modulo ed è la funzione `input` citata in precedenza. Quando il kernel riceve un messaggio e il messaggio è destinato al kernel, il valore del campo `sockaddr_nl` impostato da chi trasmette è 0. Il contenuto del messaggio verrà memorizzato nel buffer relativo al socket NetLink creato e la funzione puntata (`input`) verrà chiamata. In queste versioni del kernel tale funzione è definita come segue:

```
void (*input)(struct sock *sk, int len)
```

A questo punto è presumibile che vi siano dei messaggi contenuti nel socket. Per accedervi entrano in gioco le funzioni taggate con *skb_* per la gestione del buffer del socket (`struct sk_buff` citato in precedenza) e, come si può intuire, queste funzioni sono generiche per ogni tipo di socket. Esse sono definite in *linux/skbuff.h* e implementate in *net/core/datagram.c* e *net/core/skbuff.c*. Le due funzioni principalmente usate negli esempi sono:

[5] Con "valido" si intende destinato al kernel

`struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags, int noblock, int *err)` Preleva solo il primo messaggio dalla coda, mantenendo gli altri messaggi. Per fare ciò si avvale comunque di `skb_dequeue()`. Il primo parametro è chiaramente il socket di tipo NetLink, seguono dei flags che attualmente[6], se diversi da 0, possono avere solo il valore `MSG_PEEK`. Quindi vi sono `noblock` e `*err`; `err` è semplicemente una variabile che permette di discriminare un eventuale errore. La presenza dell'errore è legata al ritorno (`NULL`) della funzione. `noblock` invece definisce il tempo di attesa del messaggio. Se 0 la chiamata non è interruttiva.

`struct sk_buff *skb_dequeue(struct sk_buff_head *list)` Decrementa la testa della lista e ritorna `NULL` nel caso non vi siano messaggi da leggere. Questa funzione esegue semplicemente le funzioni di una macro (o meglio una funzione `inline`) definita nell'header *linux/skbuff.h* ossia `__skb_dequeue()`. Questa macwait_for_packetro non fa altro che decrementare la lista riducendo il campo `list->qlen` e riportando come risultato il puntatore all'elemento desiderato. Il valore aggiunto dalla funzione `skb_dequeue()` è che essa esegue il tutto ponendo in lock l'operazione, evitando quindi i rischi di race concurrency.

Piccola nota sull'implementazione di `skb_recv_datagram()`: per eseguire il timeout sulla lettura del messaggio, viene eseguita la funzione `wait_for_packet()`. Questa funzione imposta il task come interruttibile (`TASK_INTERRUPTIBLE`) e setta un timeout tramite `schedule_timeout()` la quale è una funzione altamente ottimizzata dal punto di vista del sistema perchè non ruba tempo inutilmente alla CPU in quanto il processo viene gestito dallo scheduler. Il problema apparente è che quest'ultima funzione blocchi il task esattamente per il tempo impostato. In realtà il semaforo per eseguire il lock è interno alla struttura del socket (`sock`) ed è identificato dal campo `sk_sleep`. Se un messaggio dovesse arrivare in un tempo inferiore a quello di attesa, questo lock verrà liberato da un altro task del kernel facendo così in modo che `skb_recv_datagram()` si sblocchi. Tutto ciò è eseguito grazie al fatto che un socket generico (`struct sock`) contiene al suo interno alcuni puntatori a funzione, tra cui `sk_state_change`. Questo è un puntatore a funzione che viene configurato di default in modo che punti alla funzione `sock_def_wakeup()` la quale è implementata in questo modo (kernel 2.6.13.1, *net/core/sock.c*):

```
static void sock_def_wakeup(struct sock *sk) {
        read_lock(&sk->sk_callback_lock);
        if (sk->sk_sleep && waitqueue_active(sk->sk_sleep))
                wake_up_interruptible_all(sk->sk_sleep);
        read_unlock(&sk->sk_callback_lock);
}
```

Detto ciò quindi, è chiaro che si può effettuare anche una sostituzione delle funzioni di default direttamente usate dal kernel, per esempio per eseguire il debug o per aggiungere qualche funzione specifica. Si noti che la stessa funzione `skb_recv_datagram` non fa altro che settare la funzione puntata dal campo `sk_data_ready` della struttura `sock` pari a input, sostituendola a quella di default (`sock_def_readable`). A titolo informativo, la funzione che configura il socket `sock` all'interno del kernel è `sock_init_data()` implementata sempre in *net/core/sock.c*.

3 Specifiche Software

Il software distribuito con questo articolo è scaricabile dal sito *www.lugman.org* nella sezione *Documentazione* ed è diviso in directory chiamate *nlXX* dove *XX* è un numero progressivo. All'interno vi sono i sorgenti sia dei moduli che dei programmi per effettuare i test di comunicazione. Segue quindi la realizzazione di alcuni esempi. Gli esempi sono divisi per directory e in ogni directory è contenuta l'evoluzione di moduli e programmi.

Gli esempi sviluppati, sono riportati cronologicamente al loro sviluppo e in generale coprono queste funzionalità:

nl.ko modulo del kernel che mostra la creazione di un socket NetLink e la gestione per trasmissione e ricezione.

[6] Controllato fino al kernel 2.6.23.8

nl_mcom programma user-space di test che permette di porsi o in ricezione o in trasmissione dopo aver creato il socket.

Seguono quindi le specifiche di implementazione del modulo e del programma.

3.1 Specifiche del modulo del kernel *nl*

Per prima cosa esamineremo il modulo del kernel. Si noti fin d'ora che per il test di ricezione e trasmissione non serve avere il modulo se il kernel è inferiore al 2.6.24, mentre è obbligatorio caricare il modulo per le versioni successive. I programmi user-space possono già dialogare con il protocollo non standard scelto. I sorgenti di questo modulo si trovano nella directory *nl04/* a disposizione sul sito *www.lugman.org*.

Lo scopo di questo modulo è creare un interfaccia tra spazio kernel e user. I file che compongono il modulo sono:

nl.c è il driver in questione. Ogni directory *nlxy* (con xy indicante un numero) contiene una versione. Le versioni di test per i kernel descritti vanno dalla 04 alla 06.

nl.h è il file header che contiene le macro necessarie alla configurazione del socket NetLink. Questo file è fondamentale anche per user-space per poter conoscere come opera il kernel.

Specifiche di implementazione

Le specifiche del software su cui eseguire l'implementazione, sono scritte avendo in mente che il modulo deve poter comunicare con lo user-space e viceversa. La cosa più complessa è probabilmente istrupanicire il kernel per mandare il messaggio verso lo user-space. Per fare questo è stata fatta la scelta (arbitraria) di mandare dallo user-space un messaggio contenente le istruzioni per il kernel in modo che quest'ultimo possa creare un messaggio da rimandare allo user-space. In sintesi il comportamento del modulo deve:

- creare il socket NetLink in fase di load del modulo e settare la funzione di gestione del socket
- rilasciare il socket in fase di unload del modulo
- La funzione di gestione del socket (`nl_input` come si vedrà poi) chiamata quando viene ricevuto in messaggio, deve:
  - prelevare dalla coda dei messaggi il primo buffer (fatto in automatico dal kernel 2.6.24)
  - estrarre dal buffer la struttura NetLink (ossia l'header `nlmsghdr`) e il payload
  - analisi del payload per capire quale tipo di messaggio il kernel dovrà inviare
  - modifica di dati del payload e reinvio in broadcast o unicast in funzione di quanto richiesto

3.2 Specifiche per *nl_mcom* versione 01

Anche questo esempio si trova nella directory *nl04/*. *nl_mcom* è un programma in spazio utente che permette di settarsi in ricezione o trasmissione. Una volta configurato il socket, è possibile istruire il programma in modo che invii un messaggio unicast a un processo specifico o multicast a più processi.

Il programma contiene fondamentalmente tutte le strutture descritte in 2. Ciò che vi è in più sono alcune funzioni di trasmissione e ricezione che sono standard per i sistemi *NIX e consultabili dalle pagine di *man* ([8]). In accordo con [1] si è scelto di mantenere la stessa struttura per la trasmissione. Questo implica che vengano introdotte le strutture

```
struct msghdr    msg;   // richiesto per la funzione 'sendmsg()'
struct iovec     iov;   // serve a msg
```

La spiegazione di queste strutture esula leggermente lo scopo del documento, soprattutto perchè sono legate allo specifico uso della funzione `sendmsg()`. Se si utilizzassero altre chiamate, come la più comune `send()`, non ce ne sarebbe bisogno.

Tutti i parametri e strutture necessarie alla gestione dei messaggi vengono importate dal kernel includendo il file *nl.h*.

Specifiche di implementazione

Come detto il programma deve porsi o in ricezione o in trasmissione. Per farlo basta controllare se il primo parametro passato è un valore numerico. Se lo è, tale valore verrà settato come gruppo del programma e il processo si porrà in ricezione dei messaggi. Se nulla verrà passato dal prompt dei comandi, il programma si porrà automaticamente in trasmissione. Le specifiche sono:

- Il programma apre un socket in modalità RAW con il protocollo di test scelto arbitrariamente e definito in *nl.h.*
- Vengono azzerate tutte le strutture per precauzione.
- Vengono quindi configurate le variabili per ricezione e trasmissione ossia `dest_addr`, `nlh`, `iov` e `msg`. Fondamentalmente se ne alloca la memoria e si settano i principali parametri
- Si controlla se si è in ricezione o trasmissione e ci si setta in quello stato
- Trasmissione:
  - si esegue il bind del socket passando come gruppo 0 e come pid il valore del PID del processo
  - si entra in un loop che terminerà o con un Ctrl-C o inviando un messaggio che comincia per '0'
  - verrà chiesto il PID e il gruppo a cui si vuole inviare il messaggio
  - si procede con il riempimento dei campi del payload. Se il messaggio è per il kernel, verrà chiesto di compilare tutti i campi, altrimenti solo il campo `str`.
  - si procede all'invio del messaggio
- Ricezione:
  - si esegue il bind del socket passando come gruppo il valore passato da linea di comando.
  - si entra in un loop che terminerà o con un Ctrl-C o se il primo carattere del messaggio ricevuto è '0'.
  - la chiamata è interruttiva e, una volta ricevuto il messaggio, verrà stampato chi ha inviato il messaggio e il contenuto di `str`.

4 Implementazione del modulo *nl.c*

Il modulo *nl.c* è stato scritto per più versioni del kernel e quindi verranno proposte le varie soluzioni. Chiaramente tutte queste faranno sempre riferimento alle specifiche.

4.1 Esempi per kernel $<$ 2.6.14: directory *nl04/*

nl.h

Il file header messo a disposizione definisce principalmente la struttura del payload, fondamentale per sapere come sono organizzati i dati. Oltre a questo è fondamentale la macro che indica il protocollo usato dal nostro socket NetLink. Si noti che le macro che indicano i protocolli possono variare con il crescere del kernel. Per esempio il kernel 2.6.13.1 supporta 15 protocolli, mentre il kernel 2.6.23.8 ne supporta 19. Quindi se si vuole mantenere una certa portabilità di questo software di esempio, conviene scegliere valori alti dei protocolli, magari vicini a 31. Il valore scelto (arbitrariamente) è 30. Il file *nl.h* è il seguente:

```
#ifndef __NL_H__
#define __NL_H__
#define NETLINK_TEST        30    // protocollo di test inventato da noi
#define MAX_PAYLOAD_CHAR    100   // dimensione della stringa del paylod nl_payload_t
#define MAX_PAYLOAD_LENGTH  sizeof(struct nl_payload_t)   // dimensione dei dati utili
#ifndef __KERNEL__
 #define str_l(x) "%"#x"s"
 #define str_lim(x) str_l(x)
#endif
```

La struttura `nl_payload_t` che definisce la formattazione dei dati è definita arbitrariamente. É costituita da un messaggio `str` che legge chiunque riceve il messaggio. `pid` e `multicast` invece vengono usati dal kernel per costruire il suo messaggio.

```
typedef struct nl_payload_t {
   int pid;                    // Proce ID a cui il kernel invierà il messaggio
   unsigned multicast;         // Bitmask per multicast: OR dei gruppi interessati al messaggio
   char str[MAX_PAYLOAD_CHAR]; // Stringa dati
} nl_payload_t;
#endif // __NL_H__
```

nl.c

Questo è il vero modulo del kernel. Inizialmente vengono definite alcune macro informative e successivamente vengono definiti tre parametri per il modulo, modificabili o in fase di load o agendo tramite sysfs:

`kernel_group` indica il gruppo di appartenenza del kernel. Serve solo a dimostrare che il kernel non ha gruppi e può leggere solo il messaggio unicast. Questo parametro verrà configurato nel buffer solo dopo che è affenuta una ricezione di un messaggio valido per il kernel.

`actual_kernel_group` parametro read-only che indica l'attuale gruppo settato nel buffer del socket NetLink a livello kernel.

`use_skb_dequeue` usa la funzione `skb_dequeue()` se settato a 1 al posto di `skb_recv_datagram()`.

Il modulo è poi strutturato in funzioni. Ogni funzione qui implementata comincia col suffisso *nl_* :

`nl_init_module` inizializzazione del modulo e creazione del socket NetLink.

`nl_cleanup_module` clear del socket

`nl_input` funzione principale di gestione dei dati

`nl_change_str_case` funzione per la modifica del payload. Nulla di fondamentale per il funzionamento

Il codice del modulo è il seguente:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/types.h>
//--- NetLink ---
#include <linux/socket.h>  // gestione dei socket e strutture relative
#include <linux/netlink.h> // implementato in net/netlink/af_netlink.c (2.6.13.1)
#include <net/sock.h>      // implementa la struttura sock
//--- nl.h ---
#include "nl.h"            // macro e variabili necessarie al kernel quanto allo user space
MODULE_LICENSE("GPL");
MODULE_VERSION("0.4");
MODULE_AUTHOR("Calzo");
MODULE_DESCRIPTION("NL creates a NetLink socket and test the comunication between "
                   "user and kernel space");
static unsigned kernel_group = 0;        // Gruppi di appartenenza in attesa di definizione
static unsigned actual_kernel_group = 0; // Gruppo attuale di appartenenza reale attuale
static unsigned use_skb_dequeue = 0;     // Usa skb_dequeue() al posto di skb_recv_datagram()
module_param(kernel_group, uint, S_IRUGO|S_IWUSR);    // see LDD3 cap 2, pag 36 e linux/stat.h
module_param(actual_kernel_group, uint, S_IRUGO);     // see LDD3 cap 2, pag 36 e linux/stat.h
module_param(use_skb_dequeue, uint, S_IRUSR|S_IWUSR); // see LDD3 cap 2, pag 36 e linux/stat.h
MODULE_PARM_DESC(kernel_group, "Kernel Group for multicast comunications (field .dest_groups)"
                               " ready to the nest step");
MODULE_PARM_DESC(actual_kernel_group, "Actual Kernel Group for multicast comunications"
                                      " (fielad .dest_groups");
MODULE_PARM_DESC(use_skb_dequeue, "Use ‘skb_dequeue()’ instead of ‘skb_recv_datagram()’");
static struct sock    *nl_sk  = NULL;
```

La funzione non fa altro che cambiare il case della stringa passatagli scambiando le lettere tra maiuscole e minuscole. Questa funzione serve solo per rielaborare il payload per dimostrare che il kernel ha effettivamente ricevuto il messaggio.

```
void nl_change_str_case(char *str, int maxstr_len) {
   int i;

   for(i=0; str[i] && i<maxstr_len; i++) {
        if(str[i]>='A' && str[i]<='Z')
          str[i]+=0x20;
        else
          if(str[i]>='a' && str[i]<='z')
            str[i]-=0x20;
     }
}
```

La funzione `nl_input` è la funzione principale. É settata all'interno del socket sostituendola alla funzione di default per l'elaborazione dei dati (campo `data_ready`). Si noti subito che questa funzione è bene che sia il più rapida e contenuta possibile. L'esempio seguente non è quindi dei migliori, ma serve solo per studiare le API. La funzione in questione preleva un messaggio proveniente dallo user-space all'interno del quale vi sono i dati per creare un altro messaggio e inviarlo a uno o più nodi. Il messaggio reinviato viene modificato tramite `nl_change_str_case()`. La funzione subito comunica se è arrivato un messaggio dallo user-space. Viene poi prelevato il buffer dal quale vengono estratte le varie struttura (header, payload, ecc).

```
void nl_input(struct sock *sk, int len) {
   struct sk_buff *buf = NULL;
   struct nlmsghdr *nlh;
   int err;
   struct nl_payload_t *payload;

   printk(KERN_ALERT "Message received by user-space\n");
```

In funzione della variabile `use_skb_dequeue` viene scelto quale funzione utilizzare. In particolare viene usata la funzione `use_skb_dequeue()` per prelevare tutti i messaggi. come si nota però viene messa in una specie di loop. Tecnicamente ciò potrebbe essere fatto anche con `skb_recv_datagram()` la quale si appoggia cmunque a `use_skb_dequeue()`. Il problema di quest'ultima funzione è che non discrimina gli errori se ve ne sono.

```
   if(use_skb_dequeue == 0)
     buf = skb_recv_datagram(sk,    // socket NETLINK
                             0,     // Flags=0? chiama skb_dequeue(). Può essere 0 o MSG_PEEK
                             0,     // tempo di blocking della funzione (see sock_rcvtimeo())
                             &err); // discrimina l'errore se ritorna NULL
   else
     {
__redo_skb_dequeue:
        buf = skb_dequeue(&sk->sk_receive_queue);
        if(!buf)  // se il buffer è nullo esco
          return; // e non devo liberare la memoria
     }
```

Ricevuto il messaggio, estraggo i dati utili, ossia header e payload.

```
nlh = (struct nlmsghdr *)buf->data;                // prelevo la struttura del messaggio dell'header NETLINK
   payload = (struct nl_payload_t *)NLMSG_DATA(nlh);  // prelevo il payload, ossia i dati effettivi
   printk(KERN_ALERT "  sender pid: %d > %s\n",
          nlh->nlmsg_pid, payload->str);
```

A questo punto utilizzo il buffer ricevuto per rispedirlo. Così facendo non ne creo un altro. Quanto segue risetta alcune informazioni relative al buffer in modo che chi riceverà il messaggio saprà da dove esse proviene. In realtà questa parte di codice è funzionalmente inutile. Viene però mantenuta in accordo con la documentazione [1].

```
    NETLINK_CB(buf).groups = kernel_group;  // gruppo di appartenenza
    NETLINK_CB(buf).pid = 0;                // Il processo mittente è il kernel
    NETLINK_CB(buf).dst_pid = payload->pid; // PID del destinatario richiesto da user
    NETLINK_CB(buf).dst_groups = payload->multicast; // Multicast communication mask: da user
    actual_kernel_group = NETLINK_CB(buf).groups;    // Setto il vero gruppo del kernel notificando
                                                     //   l'eventuale cambiamento (da sysfs)
```

L'`if` che segue è molto importante. Se lo user-space richiede di inviare un messaggio unicast e il messaggio è indirizzato al processo 0 (ossia al kernel), il messaggio va scartato. Se così non dovesse essere, il kernel si chiuderebbe in loop e subentrerebbe un freeze del sistema che obbligherebbe a spegnere forzatamente la macchina.

```
    // Reinvio un messaggio unicast al pid
    if(!payload->multicast)
      if(payload->pid == 0)  // Se viene richiesto di inviare il messaggio AL
        {                    //  kernel, ritorno per evitare loop  ;)
           printk(KERN_ALERT "Message sent by kernel: avoid processing\n");
           goto __return;    // se non ritorno, il sistema va in freeze... ma di quelli potenti
        }

    // Rielaboro il Payload tanto per fare vedere qualche cosa
    nl_change_str_case(payload->str, MAX_PAYLOAD_CHAR);
    nlh->nlmsg_pid = 0;

    if(payload->multicast)  /// controllo se mi è richiesto di spedire in Multicast...
      {
         payload->pid = 0;                      // lo azzero per precauzione
         netlink_broadcast(nl_sk, buf,
                           payload->pid,        // PID di destinazione
                           payload->multicast,  // Multicast di destinazione
                           GFP_KERNEL);         // Modalità di allocazione
         printk(KERN_ALERT "  MULTICAST Msg '%s' sent with group mask 0x%X\n",
                payload->str, payload->multicast);
      } else                /// ...ma se sono qui, spedisco in Unicast
      {
         netlink_unicast(nl_sk, buf,
                         payload->pid,   // PID di destinazione
                         MSG_DONTWAIT);  // Chiamata non interruttiva
         printk(KERN_ALERT "  UNICAST Msg '%s' resent to %d process\n",
                payload->str,
                NETLINK_CB(buf).dst_pid);
      }

__return:
    if(use_skb_dequeue)
      goto __redo_skb_dequeue;
}
```

Funzione di inizializzazione in cui viene creato il socket e indicato il protocollo da usare.

```
int nl_init_module(void)
{
    nl_sk = netlink_kernel_create(NETLINK_TEST, // protocollo arbitrario
                                  nl_input);    // Funzione di gestione dei dati
    if(nl_sk)
      printk(KERN_INFO "Socket created: %u\n", (unsigned)nl_sk);
    else
      printk(KERN_INFO "Socket not created: %u\n", (unsigned)nl_sk);
    return 0;
}
```

Funzione di scaricamento in cui viene creato il socket e indicato il protocollo da usare.

```
void nl_cleanup_module(void)
{
```

```
    printk(KERN_INFO "Close NETLINK socket %u...\n", (unsigned)nl_sk);
    sock_release(nl_sk->sk_socket);
}
```

Set delle funzioni di load e clean up per il kernel

```
module_init(nl_init_module);
module_exit(nl_cleanup_module);
```

4.2 Esempi per kernel < 2.6.24: directory *nl05*

Come detto al paragrafo 2.2.1, le modifiche a NetLink operate fino a qui non sono molto diverse rispetto alle versioni del kernel 2.6.13 e inferiori. Le modifiche principali sono solo relative alla funzione di creazione del kernel. Ne consegue che, per mantenere la retrocompatibilità, la funzione di caricamento del modulo (`nl_init_module()`) definita in *nl.c* sia scritta come segue:

```
int nl_init_module(void) {
#if LINUX_VERSION_CODE < 132622              // Se il kernel è < del 2.6.14...
    nl_sk = netlink_kernel_create(NETLINK_TEST, // TEST: valore arbitrario deciso da noi
                                  nl_input);    // Funzione di gestione del socket
#else                                         // se siamo nel kernel 2.6.14 o +...
    nl_sk = netlink_kernel_create(NETLINK_TEST, // TEST: valore arbitrario deciso da noi
                                  0,
                                  nl_input,     // Funzione di gestione del socket
                                  NULL, THIS_MODULE);
#endif
    if(nl_sk)
      printk(KERN_INFO "Socket created: %u\n", (unsigned)nl_sk);
    else
      printk(KERN_INFO "Socket not created: %u\n", (unsigned)nl_sk);
    return 0;
}
```

Di differente vi è solo la presenza della macro di compilazione che, in funzione del kernel per cui si compila, sceglie quale funzione adoperare. Si noti che per poter leggere la macro `LINUX_VERSION_CODE` occorre includere il file *linux/version.h*. Per qualche motivo ignoto ai più la versione del kenrnel 2.6.14.x *non* ha questo file e definisce questa macro in fase di compilazione calcolandola nel *Makefile*. Quindi per questo kernel l'inclusione del file *version.h* andrebbe commentata.

A parte questo il resto è strutturalmente uguale.

4.3 Esempi per kernel > 2.6.24: directory *nl06*

In questo caso il codice è strutturalmente diverso. In questa versione del modulo è stata eliminata la retrocompatibilità legata alla rilevazione della versione del kernel in quanto avrebbe portato ad avere due funzioni di input differenti da passare a funzioni di creazione del socket differenti. Tutto questo a scapito della chiarezza.

Il codice che segue è quindi relativo alle sole due funzioni di inizializzazione e di input:

```
int __init nl_init_module(void) {
    nl_sk = netlink_kernel_create(&init_net,    // così fan tutti
                                  NETLINK_TEST, // TEST: valore arbitrario deciso da noi
                                  kernel_group, // definito in fase di loading
                                  nl_input,     // Funzione di gestione del socket
                                  NULL, THIS_MODULE);
    if(nl_sk)
      printk(KERN_INFO "Socket created: %u (Group=%d)\n",
             (unsigned)nl_sk, kernel_group);
    else {
      printk(KERN_INFO "Socket not created: %u\n", (unsigned)nl_sk);
      return -ENOSPC;
    }
    return 0;
}
```

in questa funzione si vede che è presente il parametro `kernel_group`. Questo parametro va settato all'inizio del caricamento del modulo, ma dalle prove non ha avuto effetto sul funzionamento.

La funzione di input è invece completamente riscritta rispetto alle versioni precedenti:

```
void nl_input(struct sk_buff *skb) {
   struct sk_buff *buf = NULL;   // nuovo buffer che andremo a creare
   struct nlmsghdr *nlh;         // message header
   int len;
   struct nl_payload_t *payload_r, *payload_t; // payload ricevuto e da trasmettere

   printk(KERN_ALERT "Message received by user-space\n");

   // Comincio analizzando il buffer
   nlh = (struct nlmsghdr *)skb->data;                 // prelevo la struttura header+mesaggio
   payload_r = (struct nl_payload_t *)NLMSG_DATA(nlh); // prelevo il payload dopo l'header
   printk(KERN_ALERT "  sender pid: %d > %s\n",        // comunico chi e cosa è stato spedito
          nlh->nlmsg_pid, payload_r->str);

   // Controllo che il messaggio non arrivi dal kernel per precauzione
   if(!payload_r->multicast)
     if(NETLINK_CB(skb).pid == 0)
       {
          printk(KERN_ALERT "Message sent by kernel: avoid processing\n");
          return;
       }
```

Con i dati ottenuti posso creare il nuovo buffer. Per farlo calcolo la lunghezza del messaggio data dal payload maggiorato della dimensione dell'header (ad oggi 16 byte).

```
len = NLMSG_LENGTH(sizeof(struct nl_payload_t)); // calcolo la lunghezza totale del netlink
   buf = alloc_skb(len, GFP_KERNEL);   // creo il buffer
   printk("BUF = 0x%X\n", (int)buf);
   if(!buf)                            // se il buffer è nullo esco
     {
        printk(KERN_ERR "Buffer sk_buff created is null...\n");
        return; // e non devo liberare la memoria
     }
```

A questo punto chiamo `skb_put` che configura il buffer in modo da poter accedere a `len` byte di dati. Questa chiamata è fondamentale. Senza questa il sistema può andare in instabilità o addirittura in panic:

```
nlh = (struct nlmsghdr *)skb_put(buf, len);    // prelevo la struttura header+messaggio
```

l'header va resettato o ricompilato in modo che il processo utente possa capire da chi proviene il messaggio. Chiaramente si potrebbero anche inviare dati fasulli.

```
   memset(nlh, 0, len);
   nlh->nlmsg_pid = 0;    // sono info tendenzialmente inutili che il ricevente leggerà
   nlh->nlmsg_len = len;  // gli passo anche la lunghezza del messaggio. Può servire a skb_pull()
   payload_t = (struct nl_payload_t *)NLMSG_DATA(nlh);  // prelevo il payload (dati effettivi)

   // riempio il buffer
   strcpy(payload_t->str, payload_r->str); // copio il contenuto dell'uno nell'altro
   // rielaboro il payload tanto per far vedere che faccio qualche cosa
   nl_change_str_case(payload_t->str, MAX_PAYLOAD_CHAR);
```

L'istruzione `NETLINK_CB()` che setta il gruppo qui di seguito viene sempre fatta e sembra obbligatorio il suo utilizzo nel caso di comunicazioni broadcast (vedere per esempio *lib/kobject_uevent.c*). In realtà anche facendola variare non si è visto nulla di particolarmente differente nella trasmissione delle informazioni. Da questa versione inoltre ci si è accorti che passando un valore di PID alla funzione `netlink_broadcast` tramite `payload_t->pid` si esclude questo processo dalla ricezione del messaggio.

```
NETLINK_CB(buf).dst_group = payload_r->multicast; // informazione apparentemente superflua
   NETLINK_CB(buf).pid    = payload_r->pid;       // altra info superflua (mai visto usarla)
   payload_t->pid = 0;       // lo azzero per precauzione...  non si sa mai col multicast
   payload_t->multicast=0;   // idem

   if(payload_r->multicast)  /// controllo se mi è richiesto di spedire in Multicast...
     {
        netlink_broadcast(nl_sk, buf,           // socket a cui spedire il buffer "buf"
                          payload_r->pid,       // PID di destinazione indicato da user-space
                          payload_r->multicast, // Multicast di destinazione (da user-space)
                          GFP_ATOMIC);          // Modalità di allocazione
        printk(KERN_ALERT "  MULTICAST Msg '%s' sent with group mask 0x%X - exclude pid %d\n",
               payload_t->str, payload_r->multicast);
     } else                   /// ...ma se sono qui, spedisco in Unicast
     {
        netlink_unicast(nl_sk, buf,
                        payload_r->pid,  // PID di destinazione indicato dallo user-space
                        MSG_DONTWAIT);   // MSG_DONTWAIT=Chiamata non interruttiva
        printk(KERN_ALERT "  UNICAST Msg '%s' resent to %d process\n",
               payload_t->str,
               payload_r->pid);
     }
}
```

Si ricorda che da questa versione del kernel i gruppi non sono più OR del valore del gruppo dei processi in ascolto ma definisce un preciso valore che identifica univocamente un solo gruppo a cui può appartenere il processo ricevente.

5 Implementazione di *nl_mcom.c*

Il programma *nl_mcom.c* rimane pressochè invariato nelle varie versioni dei moduli. Il principale scopo di questo programma è solo testare il modulo stesso. Anch'esso si deve appoggiare al file *nl.h* per conoscere i protocolli e le strutture di formattazione del payload. Segue quindi l'implementazione del programma e test dei vari moduli.

5.1 Implementazione

nl_mcom.c

Questo programma di fatto è l'evoluzione di *nl_com.c* presente comunque nella directory in quanto ha permesso di testare il software quando era agli inizi.

Si noti che dal punto di vista funzionale, se viene inviato un messaggio a un processo indicando il suo PID e inoltre viene anche indicato un gruppo a cui questo PID appartiene, al processo in questione arriverà comunque un solo messaggio. Inoltre se il processo di trasmissione non ha i permessi di root, non potrà inviare messaggi multicast.

Discorso analogo per il bind del socket con un gruppo diverso da 0: non è possibile settarlo a meno che il processo non abbia i permessi di amministrazione. Questa non è una scelta implementativa arbitraria, ma deriva dall'implementazione del socket a livello di sistema.

Il programma è costituito principalmente dalle seguenti funzioni:

`nl_config_dest` funzione di configurazione del socket NetLink.

`nl_receiver` funzione di ricezione.

`nl_sender` funzione di trasmissione.

`nl_bind` funzione di naming del socket.

Il listato è riportato di seguito. Si noti subito un problema che credo sia un potenziale bug: i file header *socket.h* e *netlink.h* vanno inclusi nell'ordine riportato altrimenti in programma non si compila. Questo è stato appurato su kernel 2.6.13.1 con compilatore gcc 3.3.6. Su altre configurazioni al momento non sono stati fatti test.

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>    // deve essere prima di netlink.h altrimenti non si compila nulla
#include <linux/netlink.h>
#include "nl.h"
#define QUIT_CHAR  '0'
int    sock_fd;
struct sockaddr_nl src_addr, dest_addr; // dest_ non ha influenza se siamo in ricezione
struct nlmsghdr  *nlh;  // header di spedizione e ricezione
struct msghdr    msg;   // richiesto per la funzione 'sendmsg()'
struct iovec     iov;   // serve a msg
struct nl_payload_t p;  // formattazione arbitraria del payload
void   nl_config_dest(unsigned, unsigned); // configurazione del messaggio di destinazione
void   nl_receiver(void);   // funzione di ricezione
void   nl_sender(void);     // funzione di trasmissione
int    nl_bind(int);        // funzione di bind con settaggio di src_addr
```

Si comincia con la creazione del socket di tipo RAW, l'azzeramento delle variabili e viene configurata l'area dati valida sia per la ricezione che per la trasmissione in cui si troveranno i dati (spediti o ricevuti), ossia `msg`.

```
int main(int argc, char **argv)
{
   int retval=0;

   sock_fd = socket(PF_NETLINK,    // il socket è un Kernel User Interface Device
                    SOCK_RAW,      // accesso grezzo (RAW)
                    NETLINK_TEST); // è lo stesso numero definito nel kernel

   memset(&p, 0, sizeof(p));       // azzero per precauzione il payload
   memset(&dest_addr, 0, sizeof(src_addr));
   nl_config_dest(0,0);
```

Controllo se sono in ricezione o trasmissione e in funzione del caso mi pongo in uno specifico stato.

```
   if(argc==1)     // se passo dei parametri sono in trasmissione...
     {
        if(retval=nl_bind(0))  // bind con gruppo 0, ma un valore vale l'altro qui
          goto exit_;
        nl_sender();
     }
   else            // ...altrimenti in ricezione
     {
        int bgroup;     // devo determinare il broadcast group

        bgroup = atoi(argv[1]); // il broadcast è passato come primo parametro
        if(bgroup<0)
          bgroup = 0;

        if(bgroup)      // controllo se il gruppo è veramente multicast (!=0)
          if (getuid() && geteuid()) // se sono qui sono in multicast e DEVO essere root
            {
               fprintf(stderr, "Multicast Bind must be run as root!\n");
               retval=-EPERM; // operazione non permessa
               goto exit_;
            }
        if(retval=nl_bind(bgroup))  // bind con gruppo 0 se sono in trasmissione
          goto exit_;
        nl_receiver();
     }

exit_:
   printf("Quit process %d...\n", getpid());
   if(nlh)
     free(nlh);
   close(sock_fd);

   return retval;
```

```
}
```

La funzione seguente configura il PID e il gruppo del destinatario. É fondamentale soprattutto per la spedizione del messaggio stesso. Fondamentalmente non fa altro che compilare la variabile `msg`.

```
void nl_config_dest(unsigned dest_pid, unsigned dest_group)
{
   dest_addr.nl_family = AF_NETLINK;
   dest_addr.nl_pid    = dest_pid;   // messaggio destinato al kernel
   dest_addr.nl_groups = dest_group; // unicast=0, multicat altrimenti

   nlh = (struct nlmsghdr*)malloc(NLMSG_SPACE(MAX_PAYLOAD_LENGTH));
   nlh->nlmsg_len = NLMSG_LENGTH(MAX_PAYLOAD_LENGTH);
   nlh->nlmsg_pid = getpid();  // self pid, ma non serve molto per il funzionamento
   nlh->nlmsg_flags = 0;

   iov.iov_base = (void *)nlh;
   iov.iov_len = nlh->nlmsg_len;
   msg.msg_name = (void *)&dest_addr;
   msg.msg_namelen = sizeof(dest_addr);
   msg.msg_iov = &iov;
   msg.msg_iovlen = 1;
}
```

Funzione di ricezione: si noti che questa funzione vuole in ingresso `msg` la quale basta che sia configurata una sola volta. Dopo di che ogni volata che i dati saranno ricevuti, i vari campi verranno sovrascritti. Questa funzione si chiude se il primo carattere della stringa contenuta nel payload è '0'.

```
void nl_receiver(void)
{
   printf("Receiver PID: %d\n", getpid());
   do {
      recvmsg(sock_fd, &msg, 0);

      memcpy(&p, NLMSG_DATA(nlh), MAX_PAYLOAD_LENGTH);
      printf("PID %d:\tReceiving message '%s' from %d process\n",
             getpid(), p.str,  nlh->nlmsg_pid);
   }
   while(p.str[0]!=QUIT_CHAR);
}
```

Funzione di trasmissione: inizialmente stampa l'attuale configurazione del messaggio e chiede di reimpostarla. Viene chiesto sia il PID del processo a cui si vuole spedire, sia il gruppo di appartenenza nel caso si voglia effettuare un messaggio broadcast. Se il PID è 0, il programma crede che si voglia inviare al kernel un messaggio e così è. Per questo nel seguito viene chiesto di compilare tutti i campi del payload per istruire il kernel.

```
void nl_sender(void)
{
   int c;
   unsigned dest_pid, dest_groups;

   dest_pid    = dest_addr.nl_pid;
   dest_groups = dest_addr.nl_groups;

   printf("Sender PID: %d\n\n", getpid());
   do {

      printf("---------------------------------\n"
             "Destination message configuration:\n"
             "  Destination PID:   %d\n"
             "  Destination Group: 0x%X\n",
             dest_addr.nl_pid, dest_addr.nl_groups);
      printf(" > change destination PID:\t");
```

```
        fflush(stdout);
        scanf("%d", &dest_pid);

        printf(" > change destination Group:\t");
        fflush(stdout);
        scanf("%d", &dest_groups);

        nl_config_dest(dest_pid, dest_groups);
        printf("New destination message configuration:\n"
               "  Destination PID:   %d\n"
               "  Destination Group: 0x%X\n",
               dest_addr.nl_pid, dest_addr.nl_groups);

        printf("Fill payload structure for %s:\n", dest_pid ? "process" : "kernel");
        memset(&p, 0, sizeof(p));        // azzero il buffer

        printf("Message:\t");
        fflush(stdout);
        scanf(str_lim(MAX_PAYLOAD_CHAR), p.str);
        p.str[MAX_PAYLOAD_CHAR-1]='\0'; // per sicurezza termino la stringa

        if(!dest_pid)     // se la destinazione è il kernel, ossia se dest_pid==0...
          {
             printf("Multicast mask:\t");
             fflush(stdout);
             scanf("%d", &p.multicast);

             printf("PID:\t");
             fflush(stdout);
             scanf("%d", &p.pid);
          }

        memcpy(NLMSG_DATA(nlh), &p, MAX_PAYLOAD_LENGTH);

        printf("Process %d send '%s' message...\n\n", nlh->nlmsg_pid, p.str);
        sendmsg(sock_fd, &msg, 0);

    } while(p.str[0]!=QUIT_CHAR);
}
```

Esegue il bind del socket settandone i parametri principali. Si ricorda che se il gruppo è diverso da 0, occorre avere i privilegi di root per eseguire il comando.

```
int nl_bind(int multicast_group)
{
    printf("Multicast Group: %d\n", multicast_group);

    src_addr.nl_family = AF_NETLINK;
    src_addr.nl_pid    = getpid();
    src_addr.nl_groups = multicast_group; // gruppo per messaggi multicast

    return bind(sock_fd, (struct sockaddr*)&src_addr,
                sizeof(src_addr));
}
```

5.2 Test di *nl.ko* e *nl_mcom* in *nl06/*

Verranno documentati ora alcune modalità di test del modulo e del programma di interfaccia user-space. La versione in esame è quella contenuta nella directory *nl06/* e relativa al kernel 2.6.24 o superiore. I messaggi possono essere inviati da user a user o da user a kernel e viceversa. Le uniche cose da tener presente sono che *nl_mcom* va lanciato con 0 se si vuole che il processo si metta in ascolto con gruppo 0, ossia riceva solo messaggi unicast a lui indirizzati mediante il suo PID. Lanciato con un numero maggiore di 0, *nl_mcom* si porrà in ascolto di messaggi sia unicast che broadcast indirizzati al suo gruppo, ma solo se il programma ha i permessi di amministrazione. *nl_mcom* lanciato senza parametri permetterà di spedire messaggi in funzione di gruppi e/o processi.

Si ricorda che *nl_mcom* potrà funzionare in ricezione solo se il modulo *nl.ko* verrà caricato. Per caricarlo basta eseguire *insmod nl.ko* dalla directory corrente. Per ogni istanza del programma *nl_mcom*, il kernel aumenterà il flag che indica che il modulo è in uso; per vederlo basta eseguire *lsmod*. Si ricorda inoltre che per trasmettere o ricevere un messaggio multicast occorre avere i privilegi di root. Quando si indica come processo di destinazione il processo 0, si intende che il kernel riceverà il messaggio.

user to user: come inviare un messaggio unicast o multicast? Il messaggio unicast tra due processi senza passare per il kernel viene fatto settando *Destination PID* al valore del PID del processo in ascolto sul socket. Il messaggio multicast viene fatto settando un pid di destinazione a caso diverso da 0 e un gruppo (*Destination Group*) diverso da 0.

user to user: cosa accade se un pid appartiene anche al gruppo? Ogni processo riceve uno e un solo messaggio indipendentemente dal tipo multicast o unicast.

user to user: cosa accade inviando un messaggio al gruppo 1 con all'ascolto il gruppo 2 e 3? Nelle versioni precedenti del kernel il valore del gruppo indica a quali gruppi il messaggio deve arrivare, ossia se il messaggio è destinato al gruppo 2 e 3 è sufficiente indicare il gruppo 3 perchè entrambi ricevano il messaggio. Dal kernel 2.6.24.1 invece, il sistema ragiona al contrario ossia i processi in ascolto definiscono la maschera di bit su cui rimanere in ascolto. Per esempio definendo due istanze di *nl_mcom* con gruppo 2 e 3, se voglio che entrambi ricevono il messaggio dovrò lanciare un messaggio destinato al gruppo 2. Se il messaggio fosse destinato al gruppo 1, esso verrebbe ricevuto dal processo appartenente al gruppo 3, ma non al gruppo 2.

user to kernel to user: come inviare un messaggio a un altro passando per il kernel? Basta caricate il modulo *nl.ko* e avviare due nl_mcom, uno in ricezione (che avrà per esempio PID=1000) e uno in trasmissione. Nel momento della trasmissione impostare come destination pid il valore 0 e così anche per il gruppo; questo implica che il messaggio venga intercettato dal kernel. Quando si immettono i valori, inserire per esempio il messaggio "xxXxx", 0 al gruppo e, infine, il pid del processo ricevente (1000 in questo caso). Se tutto andrà a buon fine, il processo ricevente riceverà "XXxXX".

user to kernel to user: cosa accade inviando un messaggio al gruppo 1 passando per il kernel con all'ascolto il gruppo 2 e 3? La situazione è come la precedente (ossia due istanze di *nl_mcom* con gruppo 2 e 3). Settando come destination PID e group 0, il messaggio arriverà al kernel il quale dovrà redirezionare la stringa al gruppo 1 (Multicast mask = 1 e PID=-1¸o un numero a caso). Il messaggio arriverà al gruppo 3, esattamente come prima.

user to kernel & user: cosa accade se invio un messaggio broadcast ai processi e unicast al kernel? Per farlo si avviino tre o più *nl_mcom*, uno con gruppo 1 (pid=1001), un altro con gruppo 2 (pid=1003), l'ultimo con gruppo 3 (pid=1010) e un quarto per la trasmissione. Come destination pid inserire 0 e come gruppo 2. Il messaggio è per esempio "xXxXx", il gruppo per il kernel è 0, il PID=1001. Il risultato sarà per tutti e tre i processi il medesimo, ossia:

```
PID ????:       Receiving message 'XxXxX' from 0 process
```

Questo implica che il messaggio è stato accettato dal kernel e che il buffer è stat quindi subito modificato prima di rilasciarlo di nuovo allo spazio utente. Dopo di che è stato reinviato allo spazio utente in particolare al processo 1001. Gli altri processi hanno potuto leggere il messaggio perchè il gruppo era '2', quindi il processo 1003 (che ha gruppo 2) lo ha letto e così anche il processo 3 (con gruppo 3) perchè 3|2=3 ed è diverso da 0.

Rieseguendo quanto appena fatto lasciando inalterato il messaggio e il gruppo del payload per il kernel, ma modificando il PID a 1003, al primo processo non arriverà nulla, al secondo arriveranno due messaggi uguali pari a "XxXxX" e al terzo arriverà lo stesso messaggio, ma una sola volta.

5.3 Test di *nl.ko* e *nl_mcom* in *nl04/*

La versione del driver nella directory *nl04/* si può interfacciare con kernel 2.6.23.x o inferiore. Quanto segue mostra cosa accade su un kernel 2.6.13.1. Il programma *nl_mcom.c* è stato introdotto da questa versione del software (04 appunto) e anche in qesto caso deve essere eseguito da root per poter usufruire di tutte le potenzialità.

Di differente rispetto a prima non vi è molto, salvo il fatto che aprendo il socket, il modulo *nl* non indica che il socket è usato da qualcuno (per vederlo, lanciare *lsmod*) e, anche per questo, non è necessario caricare il driver per effettuare la comunicazione. Detto ciò seguono solo alcuni casi particolari:

user to user: cosa accade lanciando da user space un messaggio al gruppo 3? Supponendo di avere due processi con gruppo 1 e 2 e da user space si manda un messaggio non indirizzato al kernel (ossia indico un destination pid a caso, per esempio -1), il messaggio è ricevuto da entrambi i processi con gruppo 1 e 2 riceveranno il messaggio. Questo perchè nelle versioni più veccie del kernel, si ricorda che il gruppo è una maschera di bit e per mandare un messaggio broadcast basta settare il gruppo a `0xFFFF`.

user to user & kernel: cosa accade se mando un messaggio al gruppo 2 e al kernel dicendogli di reindirizzare tutto al gruppo 1? Per farlo occorre lanciare *nl_mcom* con gruppo 1 e grupo 2. Successivamente si lancia *nl_mcom* indicandogli come destination pid 0 (ossia messaggio per il kernel) e destination group 2. Segue il messaggio, per esempio XxXxX, Multicast Mask 1 e PID 0. Questo ha come risultato che sia il processo con gruppo 1 e 2 riceveranno xXxXx. Questo perchè probabilmente il messaggio viene individuato dal kernel ed elaborato subito. Inoltre il modulo non solo "cattura" il buffer, ma lo rielabora e lo reinvia subito. Questo probabilmente implica che quando il messaggio verrà dimesso a disposizione dello user-space, esso sarà modificato.

user to user & kernel: cosa accade se mando un messaggio al gruppo 1 e al kernel dicendogli di reindirizzare tutto al pid con gruppo 1? Ipotizzando la situazzione precedente, definendo Destination PID 0 e Group 1, e per il kernel Multicast mask 0 e come PID il valore del pid del gruppo 1, il processo di gruppo 1 riceverà due messaggi uguali. Le motivazioni sono le stesse del caso precedente

sysfs L'interfaccia sysfs permette di modificare i parametri del modulo. In realtà gli unici parametri modificabili sono `kernel_group` e `use_skb_dequeue`. Mentre il secondo sarà operativo fin da subito, il primo ha bisogno che un messaggio sia processato dal kernel per essere reso definitivo; per capirlo basta confrontare il valore di `actual_kernel_group`: se è uguale a `kernel_group`, il gruppo è stato impostato. I messaggi successivi vedranno impostato questo valore, ma il valore del gruppo per il kernel non ha significato. É stato implementato solo per verificare questo fatto.

Riferimenti bibliografici

[1] *Why and How to Use Netlink Socket* - Linux Journal articolo 7356 - http://www.linuxjournal.com/article/7356

[2] *Generic Netlink HOWTO* - The Linux Foundation - http://www.linuxfoundation.org/en/Net:Generic_Netlink_HOWTO (o http://lwn.net/Articles/211209/)

[3] *Understanding And Programming With Netlink Sockets* - Neil Horman Version 0.3 - http://people.redhat.com/nhorman/papers, file *netlink.pdf*

[4] *Netlink Socket Overview* - Gowri Dhandapani - http://qos.ittc.ku.edu/netlink/html/node1.html

[5] *Linux Device Driver 3th Edition* - Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman - O'REILLY

[6] *Understanding The Kernel Linux 3th Edition* - Daniel P. Bovet and Marco Cesati - O'REILLY

[7] *Man Pages* for `netlink` per le macro, `netlnk(7)` per la generazione del socket (http://www.kernel.org/doc/man-pages/online/pages/man7/netlink.7.html)

[8] *Man Pages* for `send()`, `sendto()` e `sendmsg()`

Info & Credits

Articolo scritto e presentato al LINUX DAY 2009 - nona giornata di Linux e del Software Libero - dall'associazione LUGMan (Linux Users Group Mantova). L'articolo è scaricabile in formato PDF ed è distribuito insieme ai sorgenti dei moduli e programmi descritti. Il tutto è scaricabile liberamente dal sito dell'associazione (*www.lugman.org* nella sezione "Documentazione"). L'articolo è stato scritto con LyX 1.4.3 sotto Slackware 10.2 da Calzoni Pietro aka *Calzo*.

Chiunque volesse altri formati del documento o i sorgenti e non riuscisse a trovarli in altro modo, può richiederli contattando l'associazione a *info@lugman.org* o all'autore *calzog@gmail.com*.

Chiunque è libero di correggere e modificare questo testo e il software con esso rilasciato nel rispetto delle licenze *Creative Commons* e *GPL*.