Linux NAPI-compliant network device driver

L'articolo parla delle NAPI: l'architettura dei network device driver di Linux adatta a supportare gli adattatori di rete di nuova generazione.

di Antonino Calderone

Torniamo a discutere di network device driver (NDD) di Linux ricollegandoci a un precedente articolo ([1]) centrato sulle VPN. Alcuni degli argomenti esposti in quell'articolo, in particolare la struttura di un generico NDD, sono propedeutici alla lettura di questo, ma fruibili anche da molte altre fonti, vi segnaliamo in particolare [2].

Questa volta ci concentreremo sulle New API (*NAPI*), ovvero sul meccanismo di packet processing pensato per supportare al meglio i driver per gli adattatori di rete "veloci" - come i sempre più diffusi adattatori Gigabit-Ethernet - introdotto nella versione 2.6 del Kernel e successivamente portato "indietro" nella versione 2.4.20 (e successive).

L'architettura dei network device driver non-NAPI non è adatta a supportare dispositivi capaci di generare migliaia di interrupt al secondo, e questi d'altra parte possono rappresentare una potenziale causa di "starvation" per l'intero sistema. Alcuni dispositivi hanno funzionalità avanzate di *interrupt coalescing* (si veda a titolo di esempio [3]), ovvero la capacità di raggruppare più pacchetti per interruzione attraverso una logica per l'*interrupt mitigation*.

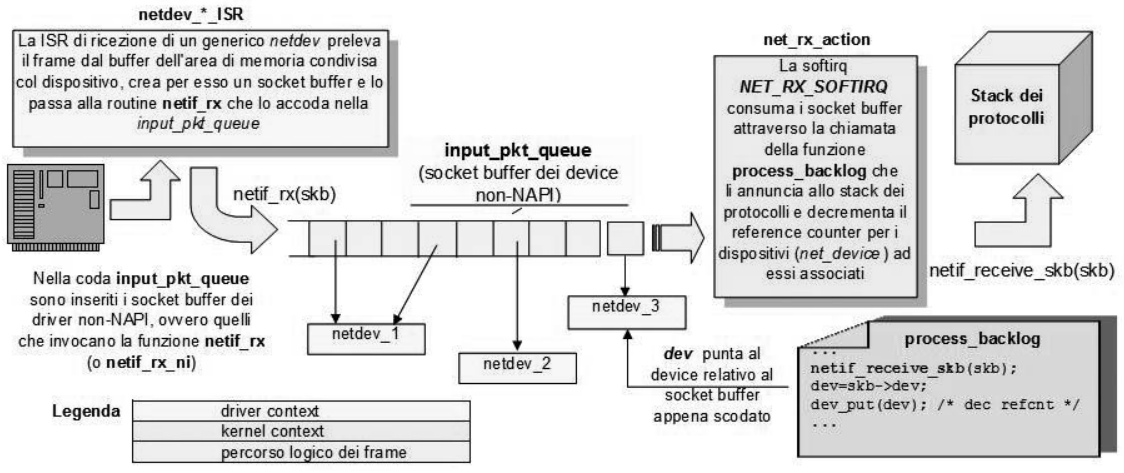
Senza l'ausilio delle NAPI, quindi senza un sostanziale supporto del Kernel, queste funzionalità dovrebbero essere interamente implementate nel driver, unite a meccanismi più o meno efficaci di prelazione (basati per esempio su timer-interrupt), e ancora su metodi di polling gestiti in contesti differenti da quello della ISR di ricezione (vale a dire kernel thread, tasklet, ecc.).

Come vedremo, il nuovo modello di NDD aggiunge al sottosistema di networking di Linux caratteristiche per supportare in modo efficace *interrupt mitigation* e *packet throttling*, e cosa importante, consente al Kernel di distribuire il "carico" mediando l'occupazione di risorse attraverso una politica round-robin tra i device.

Antonino Calderone acalderone@infomedia.it

Lavora nella R&D per Ericsson, con la qualifica di Technical Development Engineer. Progetta e sviluppa software, device driver e firmware per i sistemi per le telecomunicazioni, sistemi di controllo e automazione. Lo interessano in modo particolare gli internals dei sistemi operativi, la programmazione nello spazio di nucleo, e gli aspetti implementativi del software per il networking, in relazione soprattutto alle problematiche di interprocess communication per sistemi eterogenei, multiprogrammati e real-time.

FIGURA 1 Percorso logico dei frame ricevuti dai dispositivi pilotati da driver non-NAPI



Ricezione dei frame nel modello non-NAPI

Parleremo dei meccanismi di packet processing per la ricezione di frame interni al Kernel, senza la pretesa di essere esaustivi. Riteniamo sia necessario conoscere tali meccanismi per caratterizzare gli elementi che rendono comprensibili le differenze tra i due diversi modelli: NAPI e non-NAPI (consigliamo di consultare [4] per ulteriori approfondimenti).

Nel modello non-NAPI (schematizzato in **Figura 1**) la consegna dei frame allo stack dei protocolli avviene attraverso la funzione di Linux *netif_rx* che di norma è invocata nell'handler dell'IRQ per la ricezione di frame. Una variante di questa funzione, che può essere usata fuori dal contesto di interrupt, è la routine *netif_rx_ni*.

La funzione *netif_rx* mette nella coda di ricezione del sistema (*input_pkt_queue*) i pacchetti ricevuti dal dispositivo (imbustati in *socket buffer*), a patto che la lunghezza della coda non sia più grande di *netdev_max_backlog*: questo e altri parametri collegati sono esportati nel */proc file system* a partire dal percorso */proc/sys/net/core* e utilizzabili per il tuning.

La *input_pkt_queue* è contenuta in una struttura denominata *softnet_data* (**Listato 1**) definita nel file *netdevice.h* ([5]); della struttura *softnet_data* ne esiste un'istanza per CPU.

Quando il frame ricevuto non è scartato per il congestionamento della *input_pkt_queue*, il compito di processarlo è affidato alla softirq denominata *NET_RX_SOFTIRQ*, schedulata tramite la funzione *netif_rx_schedule*, a sua volta invocata internamente dalla routine *netif_rx*.

La softirq *NET_RX_SOFTIRQ* è implementata nella routine *net_rx_action*.

Al momento ci accontenteremo di dire che questa funzione ha il compito di passare i frame prelevati dalla *input_pkt_queue* e consegnarli alle routine dei protocolli affinché possano essere processati.

LISTATO 1 Definizione della struttura softnet_data

```
/*
 * Incoming packets are placed on per-cpu
                                queues so that
 * no locking is needed.
 */

struct softnet_data
{
   struct net_device       *output_queue;
   struct sk_buff_head     input_pkt_queue;
   struct list_head        poll_list;
   struct sk_buff          *completion_queue;

   struct net_device       backlog_dev;
#ifdef CONFIG_NET_DMA
   struct dma_chan         *net_dma;
#endif
}
```

Il nuovo modello di ricezione dei frame

Nel nuovo modello (schematizzato in **Figura 2**), nel caso di interrupt per ricezione di nuovi pacchetti, il driver informa il sottosistema di networking della disponibilità dei nuovi frame

(anziché processarli immediatamente) in modo che questi possa consumarli, attraverso un opportuno "metodo di polling", al di fuori del contesto di esecuzione dell'ISR.

I dispositivi che possono supportare le NAPI devono soddisfare dunque dei prerequisiti ([6]): il driver non può più utilizzare la coda di input dei pacchetti, pertanto il dispositivo pilotato deve essere in grado di accumulare i frame in ricezione mantenendo un buffer per quelli ricevuti anche con interrupt di ricezione disabilitati.

Questa logica riduce l'insorgenza di interruzioni a fronte di eventi generati dal dispositivo e delega al driver il compito di scartare i frame in caso di burst, evitando di saturare le code del sottosistema di networking.

Dal punto di vista dell'implementazione, le parti che differiscono sostanzialmente con il vecchio modello sono la routine di interrupt (di ricezione) e del nuovo metodo poll (attributo di *net_device*), il cui prototipo è definito nel seguente modo:

```
int (*poll)(struct net_device *dev,
                                int *budget);
```

Oltre a questo, fanno parte della struttura *net_device* due nuovi attributi (di tipo *int*), denominati *quota* e *weight*, usati per attuare il meccanismo di prelazione nel ciclo di polling (come vedremo tra poco).

La routine di interrupt nel modello NAPI delega al metodo *poll* la consegna dei frame allo stack dei protocolli. In pratica il suo lavoro si "riduce" a disabilitare l'interrupt di ricezione del dispositivo (che continuerà ad accumulare i frame entranti), effettuare le operazioni specifiche di acknowledgment dell'IRQ e infine schedulare (usando la routine *netif_rx_schedule*) la softirq *NET_RX_SOFTIRQ* associata alla funzione *net_rx_action*.

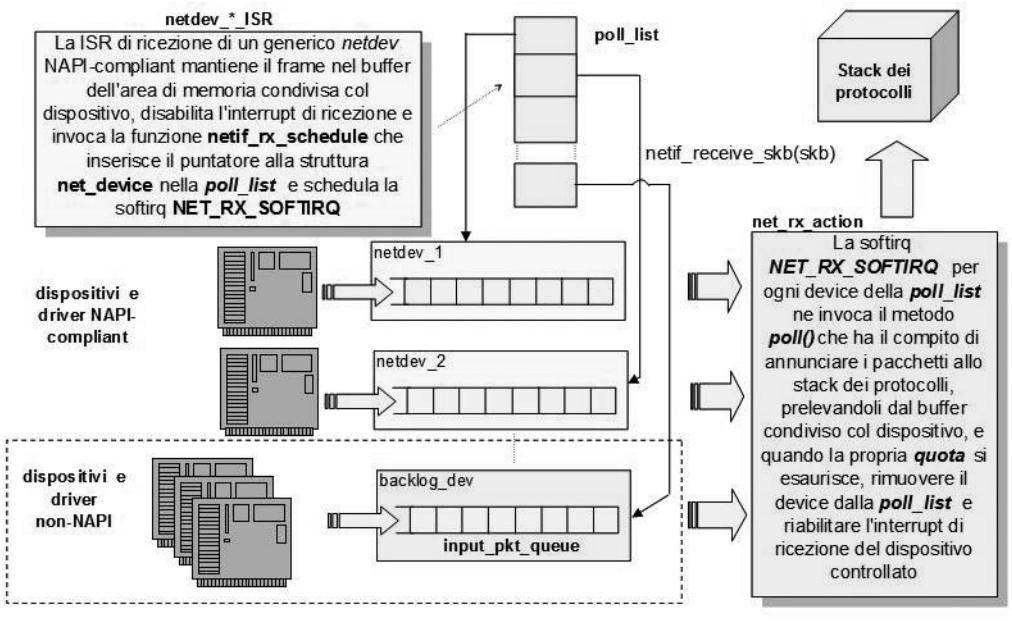
I driver in attesa di essere sottoposti a *polling* sono inseriti in una lista (*poll_list*) dalla routine *netif_rx_schedule* che prende come argomento il puntatore all'istanza di *net_device*.

La *poll_list* viene scandita durante l'esecuzione della softirq *NET_RX_SOFTIRQ* nella routine *net_rx_action*, che per ogni driver invoca il relativo metodo *poll* e questo imbusta i frame in *socket buffer* e li notifica allo stack dei protocolli.

Andando più in dettaglio la routine *net_rx_action* effettua le seguenti operazioni:

- Recupera il riferimento alla *poll_list* per il processore corrente;
- Salva nella variabile *start_time* il valore di *jiffies*;
- Imposta il parametro *budget* (passato come riferimento a *poll*) al valore iniziale di *netdev_budget* (configurabile da */proc/sys/net/core/netdev_budget*);
- Per ogni device nella *poll_list* oppure finché non si esaurisce il *budget*, oppure ancora se non è

FIGURA 2 Modello di packet processing NAPI - compliant



trascorso più di un "*jiffies*" dallo *start_time*:

- se la *quota* è positiva invoca il metodo *poll* del "device" (ovvero il suo riferimento instanza della struttura *net_device*), altrimenti somma alla quota il valore *weight* e riaccoda il device nella *poll_list*;
- se il metodo *poll* restituisce un valore non nullo, ripristina la *quota* in base all'attributo *weight* e riaccoda il device nella *poll_list*;
- se il metodo *poll* restituisce zero, assume che il device sia stato rimosso dalla lista e quindi che non sia più in *polling-state*.

Il riferimento alla variabile *budget* è passato alla routine *poll* assieme al puntatore alla struttura *net_device* del driver. La funzione *poll* deve decrementare questo valore del numero dei frame processati. I frame scodati dal buffer popolato dal dispositivo sono imbustati in *socket buffer* e consegnati allo stack dei protocolli attraverso la funzione *netif_receive_skb*.

Al criterio di prelazione basato sulla variabile *budget* è affiancato quello per device su base *quota*: il metodo *poll* deve farsi carico di verificare quanti frame possa consegnare al kernel in base alla massima *quota* assegnata al device. Quando questa si esaurisce nessun altro pacchetto dovrebbe essere inoltrato al kernel, consentendo a quest'ultimo di effettuare il polling di un altro device in attesa nella *poll_list*. Per questo la funzione *poll* deve decrementare il valore dell'attributo *quota* del numero di pacchetti processati dal driver, in modo del tutto analogo a quanto fatto per il *budget*.

Se il driver ha esaurito la propria *quota* prima di aver completato la consegna di tutti i frame accodati, allora il metodo *poll* deve cessare l'esecuzione e restituire un valore non nullo al Kernel.

Nel caso in cui tutti i pacchetti ricevuti siano stati consegnati allo stack dei protocolli, il driver deve riabilitare le interruzioni del dispositivo e arrestare il polling invocando la funzione di sistema *netif_rx_complete* (che estrae il device dalla *poll_list*), quindi deve cessare l'esecuzione e restituire zero alla funzione chiamante (che sappiamo essere *net_rx_action*).

Un altro importante attributo della struttura *net_device*, abbiamo visto, è il campo *weight*, usato per ripristinare la *quota* a ogni invocazione di *poll*.

Va da sé che il campo *weight* deve essere sempre inizializzato con un valore strettamente positivo.

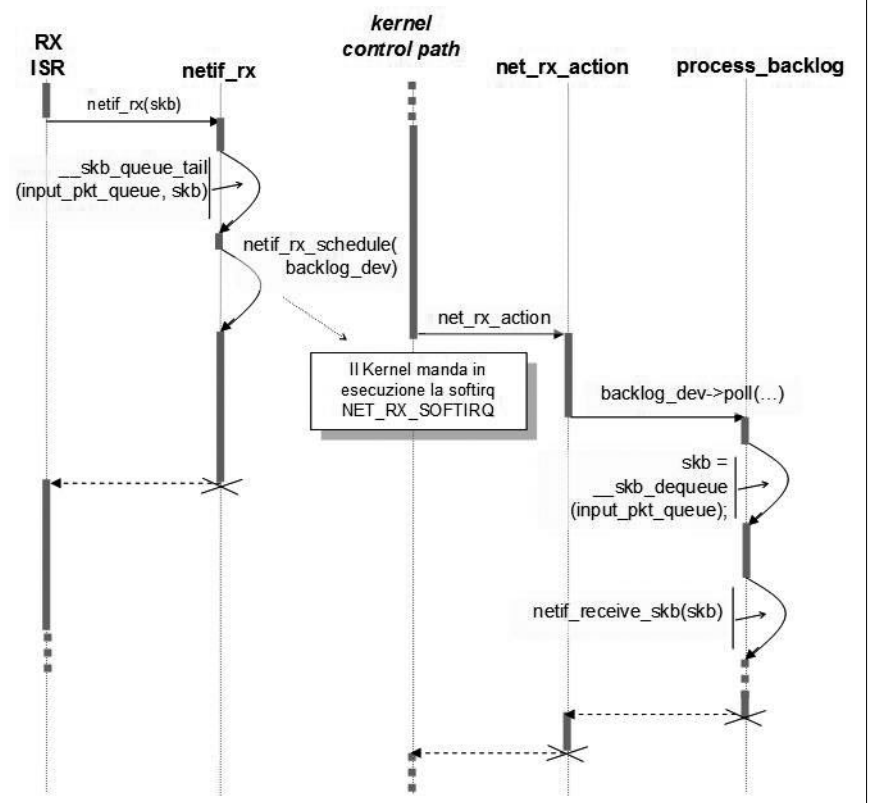
LISTATO 2 Esempio di implementazione di ISR di ricezione e del metodo poll, attributo della struttura net_device

```
static irqreturn_t sample_netdev_intr(int irq, void *dev)
{
   struct net_device *netdev = dev;
   struct nic *nic = netdev_priv(netdev);

   if (! nic->irq_pending())
      return IRQ_NONE;

   /* Ack interrupt(s) */
   nic->ack_irq();

   nic->disable_irq();

   netif_rx_schedule(netdev);

   return IRQ_HANDLED;
}

static int sample_netdev_poll(struct net_device *netdev,
                                                int *budget)
{
   struct nic *nic = netdev_priv(netdev);

   unsigned int work_to_do = min(netdev->quota, *budget);
   unsigned int work_done = 0;

   nic->announce(&work_done, work_to_do);

   /* If no Rx announce was done, exit polling state. */

   if(work_done == 0) || !netif_running (netdev)) {

      netif_rx_complete(netdev);
      nic->enable_irq();

      return 0;
   }

   *budget -= work_done;
   netdev->quota -= work_done;

   return 1;
}
```

In genere per le schede Ethernet o Fast questo valore è compreso tra 16 e 32, mentre per le Gigabit assume valori più elevati (tipicamente 64).

Guardando all'implementazione della funzione *net_rx_action* ([7]) possiamo osservare che se un driver utilizza un "extra-budget" della propria quota (fatto semanticamente ammissibile), nelle successive iterazioni deve attendere di tornare oltre una soglia positiva prima di essere nuovamente sottoposto a polling, e questo avviene tanto prima quanto più è elevato il valore attribuito al campo *weight*.

FIGURA 3 Sequence diagram relativo alle principali funzioni invocate durante il meccanismo di packet processing per driver non-NAPI



Nel **Listato 2** abbiamo riportato lo pseudo-codice relativo alla routine di interrupt di ricezione e l'implementazione del metodo *poll* di un immaginario *sample* device.

Nelle **Figure 3** e **4** sono riportati i *sequence diagram* relativi alla principali chiamate effettuate durante la fase di *packet processing* in ricezione, relativi rispettivamente al modello non-NAPI e quello NAPI-compliant.

Integrazione dei driver non-NAPI con la nuova architettura

Illustrato il meccanismo di packet processing della nuova architettura, completiamo la nostra descrizione parlando di come questo sia stato integrato con l'architettura non-NAPI.

Esiste un attributo della già discussa struttura *softnet_data* denominato *backlog_dev*. Questo attributo è istanza della struttura *net_device*.

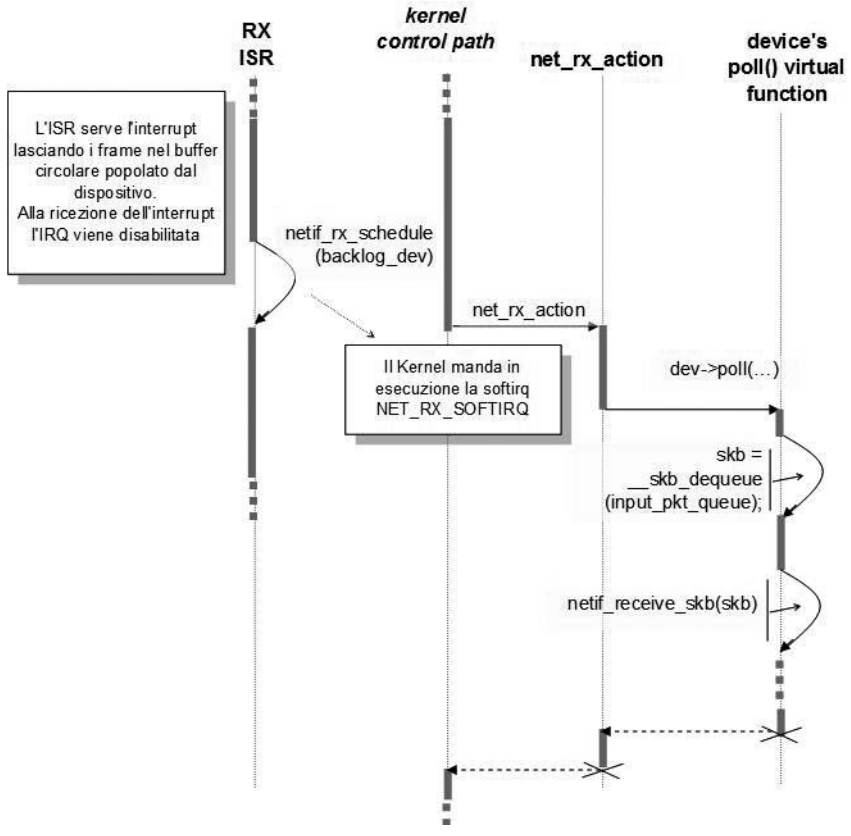
Al momento dell'inizializzazione del sottosistema di networking (in particolare nella routine *net_dev_init*) il puntatore *poll* di *backlog_dev* è inizializzato con l'indirizzo della funzione interna al kernel *process_backlog*.

La funzione *process_backlog* viene invocata nella softirq *NET_RX_SOFTIRQ* come qualunque altro metodo *poll* dei device NAPI-compliant della *poll_list*. Si ricorderà che *netif_rx* chiama la funzione *netif_rx_schedule*: il parametro passato a questa è proprio un puntatore a *backlog_dev* che è un attributo dell'istanza della struttura *softnet_data* associata alla CPU che esegue la *softirq*.

È la funzione *process_backlog* a scodare i frame della *input_pkt_queue* inseriti dalla funzione *netif_rx* e a consegnarli allo stack dei protocolli invocando la funzione *netif_receive_skb* (la stessa che è chiamata nel metodo *poll* di un generico driver NAPI-compliant).

Il numero massimo di frame che possono essere consegnati dipende anche in questo caso dal valore dell'attributo *quota* inizializzato con il parametro globale *weight_p* (modificabile da */proc file system*), oltreché dalla durata della funzione stessa che è limitata a non più di un tick del contatore *jiffies*. Esaurito questo "quanto", o esauriti i socket buffer da processare, la funzione *process_backlog* cede nuovamente il controllo alla funzione chiamante che, ormai dovrebbe essere

FIGURA 4 Sequence diagram relativo alle principali funzioni invocate durante il meccanismo di packet processing per driver NAPI-compliant



chiaro, è la funzione *net_rx_action*.

Avendo citato più volte il */proc file system* ci sembra opportuno segnalare che le impostazioni relative ai device driver (in particolare per i parametri della nuova infrastruttura NAPI) sono esportate nel *sysfs* a partire dalla directory */sys/class/net*.

Conclusioni

L'adozione delle NAPI comporta un certo numero di vantaggi.

Col nuovo modello si lascia al dispositivo il compito di gestire i buffer dei pacchetti in ingresso, delegando a questo o al software del driver il compito di scartare i frame in caso di congestione, per circoscriverne l'effetto ed evitare di interessare l'intero sistema.

Si sfruttano al meglio i meccanismi di interrupt mitigation di cui il dispositivo è capace.

Il vecchio criterio di packet processing FIFO è rimpiazzato da una politica di gestione round-robin che utilizza un meccanismo di prelazione che, affidandosi alla "cooperazione" dei driver, consente una migliore distribuzione delle risorse del sistema.

I sistemi SMP sono supportati in modo da garantire il miglior livello di concorrenza possibile.

In conclusione crediamo che l'architettura di networking di Linux basata sulle NAPI, purché supportata dal dispositivo da pilotare, sia da preferire rispetto il modello non-NAPI. Questo in effetti è l'orientamento nella comunità degli sviluppatori del "Pinguino".

BIBLIOGRAFIA & RIFERIMENTI

[1] A. Calderone - "Network Device Driver e VPN in Linux", Computer Programming N.159 - Luglio/Agosto 2006
[2] J. Corbet, G. Kroah-Hartman, A. Rubini - "Linux Device Drivers, 3rd Edition", O'Reilly, 2005
[3] http://lxr.linux.no/source/Documentation/networking/cxgb.txt
[4] C. Benvenuti - "Understanding Linux Network Internals", O'Reilly, 2006
[5] http://lxr.linux.no/source/include/linux/netdevice.h#L616
[6] http://lxr.linux.no/source/Documentation/networking/NAPI_HOWTO.txt
[7] http://lxr.linux.no/source/net/core/dev.c#L1904
[8] http://www.digifacta.com/~calderon/html/art5HTML/tvpn.gz