Network Device Driver e VPN in Linux

L'articolo tratta dei network device driver di Linux usati per l'implementazione di un piccolo framework denominato Tiny-VPN utilizzabile per la creazione di Virtual Private Network.

di Antonino Calderone

Una virtual private network (VPN) è in termini pratici l'estensione di reti private attraverso reti condivise o pubbliche. Il tipico scenario di impiego di una VPN è quello di alternativa vantaggiosa, soprattutto in termini economici, alle connessioni dedicate tra reti private.

In una VPN le connessioni fisiche sono rimpiazzate da connessioni virtuali dette *tunnel*. I dati della rete privata sono incapsulati in *datagram* che contengono le informazioni d'instradamento necessarie per l'inoltro attraverso la rete pubblica. Ogni tunnel è in genere associato a un'interfaccia virtuale usata dal sistema operativo come una comune interfaccia di rete.

L'implementazione di una VPN è caratterizzata per difetto dai seguenti elementi: un protocollo usato per trasportare i dati attraverso i tunnel; una rappresentazione degli *end-point* delle istanze dei tunnel attraverso le interfacce di rete (pertanto dette) virtuali; un meccanismo di cifratura ed eventualmente compressione dei dati inviati attraverso i tunnel.

Esistono implementazioni di protocolli di *tunneling* e più in generale implementazioni di VPN, basate su standard quali *PPTP*, *L2TP* e/o *IPSec* (descritti più in dettaglio nel **Riquadro 1**), per citare i più noti. Non è nostro obiettivo dilungarci su questi protocolli o affrontare una dettagliata trattazione sulle tecniche usate per l'autenticazione, l'integrità e la sicurezza. Abbiamo scelto di discutere una nostra implementazione che potremmo definire embrionale, ponendo l'accento sugli aspetti strutturali che riguardano i driver delle interfacce virtuali. Il software del progetto, che abbiamo denominato TVPN (dove T sta per tiny), è disponibile attraverso il server ftp di Infomedia, e può essere liberamente scaricato e compilato in una qualunque distribuzione GNU/Linux basata su kernel a partire dalla versione 2.6.8.1. I sorgenti, dei quali riporteremo alcuni frammenti, sono distribuiti con licenza GPL, e forniranno in ultimo il riferimento per i dettagli implementativi che qui saranno solo accennati.

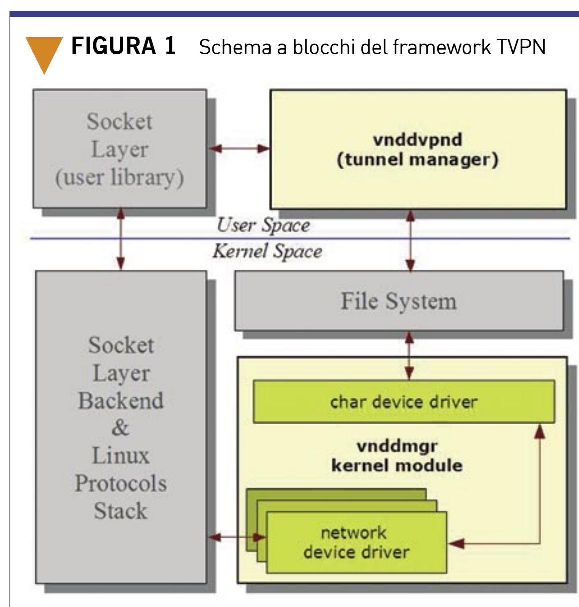
Antonino Calderone acalderone@infomedia.it

Lavora come software engineer presso Marconi S.p.A. (societá del gruppo Ericsson) nella Business Unit Broadband Networks. È socio fondatore della Digifacta Sw Engineering Srl. Si occupa principalmente di sistemi e protocolli per le telecomunicazioni, device driver, internals di sistemi operativi, firmware.

L'architettura di TVPN

Lo *stack dei protocolli* di Linux per completare la fase di comunicazione (ovvero trasmettere su un mezzo fisico o ricevere da esso) si basa sui *network device driver* (NDD). Un NDD ha lo scopo di fornire l'implementazione dei metodi per le interfacce di livello 2 che il sistema usa per la comunicazione attraverso il mezzo fisico. TVPN si basa su NDD che rappresentano interfacce *ethernet*. Trasportare frame ethernet in tunnel

FIGURA 1 Schema a blocchi del framework TVPN



non richiede accorgimenti particolari. Inoltre Linux fornisce un supporto di base per l'implementazione di molte delle operazioni degli *ethernet device driver,* per i quali non sia richiesta una specializzazione nel driver.

Normalmente un NDD pilota un adattatore di rete. Nel caso di una VPN il mezzo fisico è sostituito da istanze di tunnel. Il concetto di tunnel e di interfaccia virtuale associata a esso è alla base dell'architettura di TVPN, schematizzata in **Figura 1**.

Il protocollo di tunneling è gestito da un componente denominato *tunnel manager* (TM) eseguito nell'applicazione utente *vnddvpnd*. Il TM assume il ruolo di gestore delle istanze dei tunnel, interagendo con il *kernel module* chiamato *vnddmgr* (*virtual network device driver manager*), responsabile della gestione delle istanze dei NDD.

La comunicazione tra kernel space e user space è realizzata mediante un *char device driver* (CDD) componente dello stesso modulo *vnddmgr*.

Il char device driver

Un CDD fornisce un'interfaccia che mette in corrispondenza diretta le operazioni su file e quelle sul dispositivo. Esiste una struttura denominata *file_operations* che un CDD usa per tenere i puntatori alle funzioni definite dal driver per le varie operazioni da effettuare sul dispositivo. Se le operazioni di un driver non sono implementate, mettendo a *NULL* il relativo puntatore nella struttura *file_operations* si ottiene dal kernel un'implementazione di base.

In *vnddmgr*, la dichiarazione e l'assegnazione delle operazioni del CDD è fatta nel seguente modo:

```
struct file_operations cdev_vnddmgr_fops = {
  .owner = THIS_MODULE,
  .open  = cdev_vnddmgr_open,
  .release  = cdev_vnddmgr_release,
  .read  = cdev_vnddmgr_read,
  .write  = cdev_vnddmgr_write,
  .ioctl  = cdev_vnddmgr_ioctl
};
```

Un'estensione del compilatore *gcc* consente di assegnare i valori agli attributi di una struttura specificando i nomi dei campi per i quali si voglia fornire un valore non nullo.

Dei metodi registrati, *read* e *write* sono fondamentali, poiché forniscono il meccanismo di comunicazione tra lo spazio utente del modulo TM e quello delle

Il protocollo **PPTP** (*Point-to-Point Tunneling Protocol*), sviluppato da Microsoft e altri vendor, descritto nella *RFC* informativa 2637, è un'estensione del protocollo *PPP* definito nella *RFC* 1661. *PPP* è multi protocollo ed offre autenticazione, sicurezza e compressione dei dati. *PPTP* sfrutta una sessione *PPP* attraverso una connessione *IP*. Questo protocollo fornisce l'incapsulamento e l'adattamento dei pacchetti di informazione (*IP*, *IPX* o *NetBEUI*) all'interno dei pacchetti *IP* per la trasmissione su rete pubblica o condivisa.
Utilizza la porta *TCP* 1723 per l'autenticazione e lo scambio di informazioni fra client e server e il protocollo di trasporto *GRE* per lo scambio e la cifratura dei dati.
I client vengono validati utilizzando il metodo di autenticazione *PAP* (*Password Authentication Protocol*) o *CHAP* (*Challenge/Handshake Authentication Protocol*) di Microsoft e hash *MD4*.
La sessione viene cifrata con l'algoritmo *RC-4*.

IPsec (*Internet Protocol Security*) è un framework per una suite di protocolli per la sicurezza e l'integrità di dati. Nato come elemento integrante dell'*IPv6* è utilizzato per fornire uno strato "sicuro" allo stack dei protocolli *IPv4*. L'architettura di *IPsec* viene descritta nell'RFC 2401.
*IPsec*fornisce due possibili servizi per la sicurezza: l'*Autentication Header* (*AH*) e l'*Encapulating Sercurity Payload* (*ESP*). *AH* fornisce i soli servizi di autenticazione e integrità, mentre l'*ESP* fornisce i servizi di autenticazione, integrità e riservatezza. Sia *AH* che *ESP* possono essere utilizzati in modalità trasporto oppure in modalità tunnel: nel primo caso (non utilizzabile tra *security gateway*) si aggiungono gli *header* dei protocolli utilizzati (*AH* e/o *ESP*) tra l'header *IP* e l'*header* del protocollo di trasporto, mentre nel secondo il pacchetto *IP* originario viene interamente incapsulato in un nuovo *IP datagram*. *Internet key exchange* (*IKE*) è il protocollo che *IPsec* utilizza per stabilire una *security association* (*SA*). Una *SA* è una connessione tra due parti; è costituita dai seguenti parametri:

- un intero a 32 bit chiamato *Security Parameter Index* (*SPI*): un valore arbitrario che identifica univocamente la *SA*;
- gli indirizzi dei due peer coinvolti nella comunicazione;
- il protocollo (in modo esclusivo *AH* o *ESP*) usato per il *tunnel*;
- il tipo di cifratura e le relative chiavi utilizzate.

Tutte le *security association* attive su un host (o *security gateway*) sono contenute in un database detto *Security Association Database*. Esiste un altro database detto *Security Policy Database* che contiene le politiche di sicurezza. Le *SA* possono essere combinate tra loro, sia nel caso che i nodi terminali siano gli stessi, sia nel caso siano diversi.
Data la complessità dei protocolli usati da *IPSec*, si è reso necessario un protocollo denominato *NAT traversal* (*RFC* 3947). Questo protocollo fornisce la capacità di stabilire un tunnel IPsec anche quando i peer del tunnel subiscono un'operazione di traduzione di indirizzi ("*natting*").

Il protocollo **L2TP** (*Layer 2 Tunnelling Protocol*), definito nella *RFC* 2661, è una proposta di standard *IETF*. L'integrazione di *L2TP* con *IPSec* è definita nella *RFC* 3193. *L2TP/IPSec* incanala il traffico, mantenendo l'intera semantica *end-to-end* delle comunicazioni trasmesse all'interno del tunnel e fornendo supporto completo per le tecnologie di configurazione *host* e degli indirizzi *legacy* (*IPCP*). Questo protocollo viene in genere utilizzato negli ambienti *multi-vendor* e supporta l'autenticazione tramite *password* basata su *PAP*, *CHAP*, *MS-CHAP* e *MS-CHAP 2*, nonché l'autenticazione avanzata basata su *EAP*.

RIQUADRO 1 Protocolli e standard per VPN

interfacce virtuali. Abbiamo parzialmente alterato la semantica di questi metodi, fatto reso ammissibile data la particolare natura del CDD che non pilota un normale *stream device.* Il TM "scrive" sul descrittore di file associato al CDD per annunciare al driver l'arrivo di frame (ricevuti a sua volta dall'istanza di un tunnel), mentre "legge" da esso i frame provenienti dallo stack dei protocolli di Linux.

> **Una VPN è un'alternativa economicamente vantaggiosa alle connessioni dedicate tra reti private**

Il TM accede il CDD grazie allo speciale file creato con il comando *mknod* (per convenzione nella directory */dev*). A questo file sono associati due valori denominati *major* e *minor*. Il primo valore identifica univocamente il driver; benché ignorato nel CDD, in genere il secondo valore può essere usato dal driver che lo ricava dal parametro *inode* della funzione *open,* per modificare il proprio comportamento in relazione al significato specifico attribuito a questo numero.

Un driver può riservare uno o più numeri di device, registrando un intervallo predeterminato o lasciando al kernel il compito di allocarne uno. Nel primo caso, la registrazione può avvenire mediante la funzione *register_chrdev_region*, nel secondo usando la funzione *alloc_chrdev_region*.

Il prototipo delle due funzioni è il seguente:

```
int alloc_chrdev_region(dev_t *dev, unsigned
        baseminor, unsigned count, const char *name);
int register_chrdev_region(dev_t from, unsigned count,
                              const char *name);
```

A titolo di esempio riportiamo il frammento di codice usato per la registrazione del CDD in *vnddmgr*. Il parametro *VNDDMGR_CDEV_MAJOR* è attribuito in *compile time* per selezionare l'uno o l'altro meccanismo di registrazione:

```
_vnddmgr_cdev_no = MKDEV(VNDDMGR_CDEV_MAJOR, VNDDMGR_
                                      CDEV_MINOR);
result = VNDDMGR_CDEV_MAJOR == 0 ?
  alloc_chrdev_region(&_vnddmgr_cdev_no, VNDDMGR_CDEV_
                          MINOR, 1, VNDDMGR_CDEV_NAME) :
  register_chrdev_region(_vnddmgr_cdev_no, 1, VNDDMGR_
                                       CDEV_NAME);
```

La creazione del device driver richiede la registrazione della struttura *file_operations*. Questa operazione può essere svolta usando la funzione del kernel *cdev_add* dopo che il device sia stato correttamente inizializzato mediante la funzione *cdev_init*. I prototipi delle due funzioni sono i seguenti:

```
void cdev_init(struct cdev *cdev, struct file_
                                 operations *fops);
int cdev_add(struct cdev *p, dev_t dev, unsigned
                                          count);
```

In alternativa, quando il driver non richieda più di un numero di dispositivo può essere registrato tramite la funzione *register_chrdev,* che sostituisce le precedenti e il cui prototipo è:

```
int register_chrdev(unsigned int major, const char
              *name, struct file_operations *fops);
```

Registrando un CDD viene creata una entry nel *sys file system*, nel percorso "*/sys/cdev*", col nome usato per la registrazione del driver. Nelle recenti versioni del kernel, questa entry dovrebbe contenere informazioni sul device utili per la diagnostica. Il driver viene anche incluso nella lista pubblicata dal kernel mediante la entry del *proc file system* "*/proc/devices*", dove è riportato l'elenco completo dei device driver registrati con il relativo *major number* associato. Se il *major* è assegnato dinamicamente, può essere ricavato, per la creazione del file associato al device, proprio da questa entry.

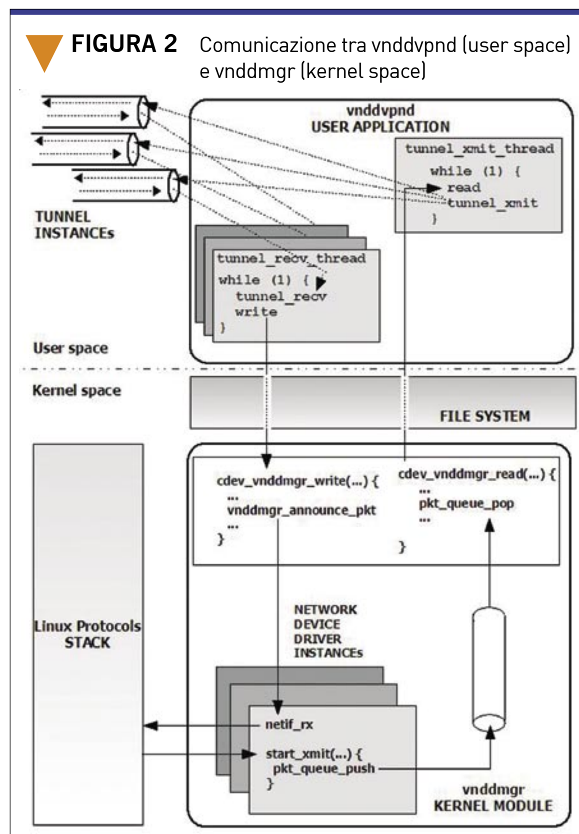


FIGURA 2 Comunicazione tra vnddvpnd (user space) e vnddmgr (kernel space)

Virtual interface e network device driver

Ogni frame trasmesso attraverso un'interfaccia virtuale viene imbustato in un *datagram UDP* (con *payload* opzionalmente cifrato) e inoltrato attraverso la rete "pubblica". I frame provenienti da un tunnel e destinati a un'interfaccia virtuale affrontano il percorso duale. In **Figura 2** è schematizzata la comunicazione tra TM e device driver, attraverso l'interfaccia del CDD.

La creazione e la rimozione delle interfacce, in TVPN, è stata affidata a un'applicazione denominata *vnd-dconfig*. Quest'utility invia le proprie richieste al CDD sempre tramite la funzione *write*.

Lanciando il programma *vnddconfig* senza alcun argomento si ottiene l'elenco completo dei parametri accettati, e viene prodotto un output simile a quello che riportiamo di seguito:

```
vnndconfig add|remove <ifname> [...]
- add command parameters:
        mtu <u16> (default 1500)
        mac <u8:u8:u8:u8:u8:u8>
         (default: 00:00:00:00:00:00)
- cdev (optional) parameter:
        cdev <string>
         (default /dev/vnddmgr)
```

Possiamo pensare un qualunque device driver costituito da due parti: una dipendente dal dispositivo che pilota (il dominio del driver); l'altra assai generalizzabile, che è l'implementazione del meccanismo d'interfacciamento verso il sistema operativo.

Un NDD pensato per pilotare interfacce virtuali è per molti aspetti meno complesso che un normale driver, proprio perché non deve supportare dispositivi hardware.

I NDD implementati nel kernel module *vnddmgr* non gestiscono dispositivi, quindi la ricezione dei pacchetti non è sollecitata da un *IRQ* ovvero non parte da una *interrupt service routine* (come avverrebbe nella maggior parte dei driver per veri dispositivi di rete), ma dal CDD, che ricordiamo riceve dal TM i frame provenienti dalle istanze dei tunnel associate alle interfacce gestite. Questi frame vengono annunciati allo stack dei protocolli esattamente come avverrebbe se provenissero da un dispositivo di rete.

FIGURA 3 Sequenza delle chiamate effettuate per la ricezione di frame

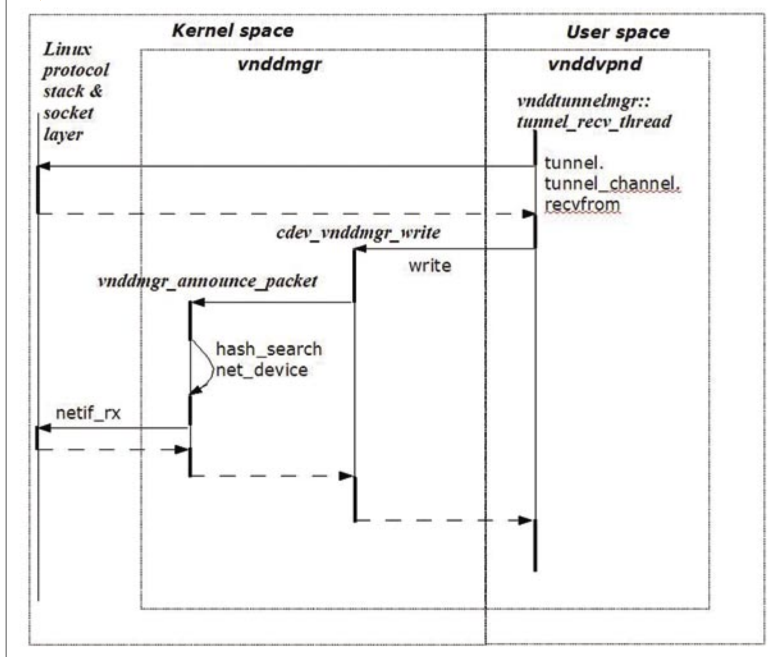
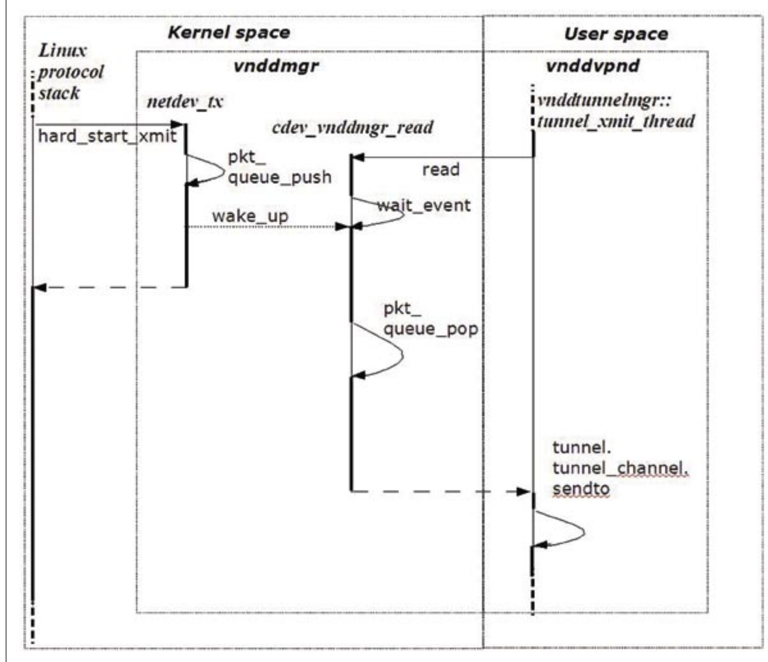


FIGURA 4 Sequenza delle chiamate effettuate per la trasmissione di frame



In particolare il metodo *write* (implementato nella funzione del CDD *cdev_vnddmgr_write*) riceve i frame ethernet, trasportati nei datagram UDP dei tunnel, dal processo utente che "scrive" sul CDD. La funzione *cdev_vnddmgr_write* costruisce un *socket buffer* per il frame e lo annuncia allo stack dei protocolli attraverso la funzione di Linux *netif_rx* secondo la sequenza rappresentata in **Figura 3**.

Quando è lo stack dei protocolli a trasmettere, il

NDD è invocato mediante il metodo *hard_start_xmit* che nel modulo *vnddmgr* è implementato dalla funzione *netdev_tx*. I frame provenienti da Linux sono consegnati al TM. Un *thread* dedicato alla trasmissione sul tunnel esegue la funzione *read*, usando il descrittore associato al CDD. Questo thread di norma è sospeso in attesa che la coda dei pacchetti sia popolata. Tale coda viene alimentata dalla funzione *netdev_tx* che riceve il socket buffer da Linux e con esso il puntatore alla struttura *net_device* che rappresenta l'interfaccia virtuale usata per la trasmissione. Se l'interfaccia risulta abilitata, *netdev_tx* inserisce il frame nella coda dei pacchetti uscenti, quindi sveglia il thread sospeso sulla *read*, invocando la funzione di sistema *wake_up_interruptible*. Il *thread* che esegue il metodo *read* del CDD (*cdev_vnddmgr_read*) sospeso sulla *wait_event_interruptible*, quando risvegliato preleva dalla coda dei pacchetti i frame e li passa al TM, premettendo un header costituito dal nome dell'interfaccia di provenienza. La sequenza delle principali operazioni appena descritta è tracciata nel diagramma di **Figura 4**.

La creazione delle interfacce e l'eventuale rimozione sono richieste al driver anch'esse mediante il metodo *write* del CDD. Creare un'interfaccia vuol dire a più basso livello registrare una nuova istanza della struttura *net_device*, che può essere allocata e inizializzata attraverso la funzione di Linux *alloc_netdev*. La struttura *net_device* nella concezione ADT (*abstract data type*) o se preferite "a oggetti", rappresenta la "classe interfaccia" ovvero l'insieme degli attributi e delle operazioni (o metodi) che il driver implementa per essa.

La funzione *alloc_netdev* accetta tre parametri: la dimensione in byte della parte "*private*" dell'istanza del driver, usata per gli attributi specifici del "dispositivo" controllato; il nome dell'interfaccia costituito da una stringa alfanumerica (di lunghezza massima *IFNAMSIZ*) e il puntatore a una funzione "costruttore" che in *vnddmgr* è denominata *netdev_setup* (**Listato 1**), invocata per l'inizializzazione degli attributi dell'interfaccia stessa. La routine *netdev_setup* del driver è usata per inizializzare l'oggetto *net_device*, in particolare per la registrazione dei metodi associati alle operazioni sull'interfaccia.

I metodi dei NDD implementati dal modulo *vnddmgr* sono i seguenti:

- *open*: invocato dal sistema quando l'interfaccia viene abilitata;
- *stop*: invocato quando l'interfaccia viene disabilitata;
- *hard_start_xmit*: usato da Linux quando lo stack dei protocolli decide di trasmettere;
- *get_stats*: è invocata dal sistema per ottenere le informazioni sulle statistiche di trasmissione e ricezione dei pacchetti;

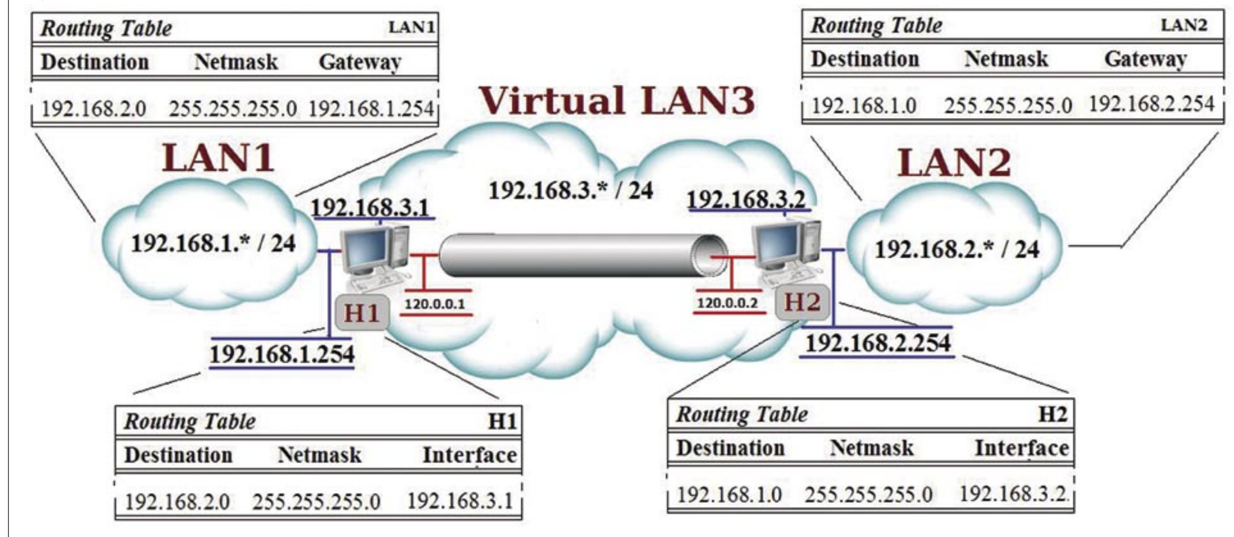
LISTATO 1 Implementazione del costruttore dell'istanza di net_device

```
/* net_device :: constructor */
static
void netdev_setup(struct net_device *ndev) {
  netdev_priv_t *priv;
  BUG_ON( ! ndev );
  priv = netdev_priv(ndev);
  BUG_ON( ! priv );
  memset(priv, 0, sizeof(netdev_priv_t));
  spin_lock_init(&priv->lock);
  ether_setup(ndev);
  ndev->open            = netdev_open;
  ndev->stop            = netdev_release;
  ndev->hard_start_xmit = netdev_tx;
  ndev->get_stats       = netdev_stats;
  ndev->change_mtu      = netdev_change_mtu;
  ndev->features |= NETIF_F_NO_CSUM;
  ndev->hard_header_cache = NULL;  /* Disable caching */
}
```

LISTATO 2 Implementazione dei metodi open e stop del NDD

```
typedef
struct _netdev_priv {
/* ... */
  u32 flags;
/* ... */
} netdev_priv_t;
#define NETDEV_TX_ON 1
#define NETDEV_RX_ON 2
/* ... */
#define netdev_tx_enable(_FLGS) { _FLGS |= NETDEV_TX_
                                                  ON; }
#define netdev_rx_enable(_FLGS) { _FLGS |= NETDEV_RX_
                                                  ON; }
#define netdev_tx_disable(_FLGS) { _FLGS &= ~NETDEV_
                                               TX_ON; }
#define netdev_rx_disable(_FLGS) { _FLGS &= ~NETDEV_
                                               RX_ON; }
/* ... */
/* net_device :: open */
static
int netdev_open(struct net_device *dev) {
  /* enable tx and rx for current interface */
  netdev_priv_t *priv = netdev_priv(dev);
  netdev_tx_enable(priv->flags);
  netdev_rx_enable(priv->flags);
  /* Interface UP */
  netif_start_queue(dev);
  return 0;
}
/* net_device :: stop */
static
int netdev_release(struct net_device *dev) {
  /* can't transmit any more */
  netdev_priv_t *priv = netdev_priv(dev);
  netdev_tx_disable(priv->flags);
  netdev_rx_disable(priv->flags);
  /* Interface DOWN */
  netif_stop_queue(dev);
  return 0;
}
```

FIGURA 5 Un esempio di VPN tra reti private



- *change_mtu*: è invocata dal sistema se si fa richiesta di modificare la dimensione massima dell'unità di trasmissione (MTU).

Guardando i sorgenti di Linux (consultabili per alcune versioni anche all'URL [3]), si può osservare come la "classe" *net_device* preveda un numero assai più elevato di operazioni e attributi. Infatti, per le operazioni non implementate nel driver, Linux fornisce una propria implementazione di base.

La registrazione di un'interfaccia può essere fatta mediante la funzione di Linux *register_netdev*, che prende come unico parametro il puntatore all'oggetto istanza di *net_device*. Prima di effettuare la chiamata alla funzione *register_netdev* tutti gli oggetti utilizzati da un NDD dovrebbero essere già inizializzati, poiché da qual momento il sistema potrebbe invocare i metodi del driver.

Il frammento di codice che segue mostra l'uso delle funzioni *alloc_netdev* e *register_netdev* appena descritte:

```
const char* network_device_name = "eth0";
struct net_device* netdev;
/* ... */
netdev = alloc_netdev( sizeof(netdev_priv_t),
                       network_device_name,
                       netdev_setup );
register_netdev(netdev);
```

La rimozione di un'interfaccia può essere fatta usando la routine *unregister_netdev* e la memoria associata all'istanza della stessa può essere liberata usando la primitiva *free_netdev*. Queste funzioni accettano come parametro il puntatore alla struttura *net_device*.

Nel **Listato 2** riportiamo l'implementazione dei metodi *open* e *stop* usati in *vnddmgr*. L'implementazione si riduce essenzialmente ad abilitare o disabilitare la trasmissione e la ricezione agendo sui flag dell'oggetto istanza di *net_device* attraverso le due funzioni *netif_start_queue* e *netif_stop_queue*.

Il tunnel manager

Un tunnel in TVPN è realizzato mediante l'imbustamento di frame ethernet in datagram UDP.

Un'istanza di tunnel è rappresentata dagli indirizzi dei due peer (*ingress* ed *egress*) e delle porte UDP usate per l'indirizzamento a livello di trasporto. Le interfacce e le istanze dei tunnel sono rigidamente collegate tra loro. In particolare, per i frame "in uscita" dall'interfaccia, quindi diretti all'egress del tunnel, il nome dell'interfaccia stessa individua rigidamente un'istanza di tunnel. Per i frame provenienti dal tunnel, quindi consegnati a un NDD, la coppia indirizzo locale e porta locale dell'interfaccia pubblica individuano in modo univoco la relativa interfaccia virtuale.

Un tunnel in TVPN è realizzato mediante l'imbustamento di frame Ethernet in datagram UDP

I frame trasmessi dallo stack dei protocolli di Linux verso le interfacce virtuali sono gestiti da un unico thread denominato *tunnel_xmit_thread* (**Listato 3**). Questo preleva dal CDD del modulo *vnddmgr* i frame accodati dai NDD. Ciascuno di questi frame è preceduto da un header che contiene il nome dell'interfaccia sulla quale lo stack dei protocolli ha trasmesso. Il nome è usato come chiave di una mappa incapsulata nella classe *vnddtunnelmgr*, nella quale sono memorizzate le

istanze dei tunnel rappresentate da oggetti della classe *vnddtunnel*.

I frame provenienti dal tunnel e diretti allo stack dei protocolli di Linux attraverso una certa interfaccia sono ricevuti dal thread *tunnel_recv_thread* (**Listato 4**). A differenza della trasmissione, esiste una distinta istanza del thread di ricezione per ogni corrispondente istanza di tunnel.

Il TM può opzionalmente cifrare il *payload* dei datagram UDP usando l'algoritmo *DES*. Quest'algoritmo usa una chiave privata condivisa. La scelta di codificare o meno i dati viene fatta per istanza di tunnel. Nel caso di dati codificati è usata una chiave di (almeno) 8 caratteri alfanumerici, fornita al TM come parametro di configurazione della specifica istanza di tunnel.

Creiamo una "tiny" VPN

Con TVPN è possibile creare tunnel (anche cifrati), per collegare tra loro LAN private attraverso un rete IP condivisa o pubblica (eventualmente Internet). Immaginiamo uno scenario come quello rappresentato in **Figura 5**. *H1* e *H2* sono due host Linux sui quali sia installato il framework TVPN, in particolare:

- *LAN 1* è una sotto-rete di classe C con indirizzo 192.168.1.0/24;
- *LAN 2* è una sotto-rete di classe C con indirizzo 192.168.2.0/24 ;
- *H1* un host con due interfacce di rete: una configurata con indirizzo pubblico 120.0.0.1 e l'altra con indirizzo privato 192.168.1.254;

LISTATO 3 Implementazione del thread tunnel_xmit_thread

```
int vnddtunnelmgr::tunnel_xmit_thread(unsigned long
              des_key_,  unsigned long tvm_, unsigned long
                                    vnddmgr_, unsigned long)
{
  assert ( tvm_ );
  assert ( vnddmgr_ );
  bool use_des = des_key_ != 0;
  std::string sKey;
  if (use_des) {
    sKey = std::string ( reinterpret_cast<const char*>
                                         ( des_key_ ) );
  }
  std::string if_name;
  vnddtunnelmgr* tvm_ptr = reinterpret_cast
                             <vnddtunnelmgr*> ( tvm_ );
  vnddmgr* vnddmgr_ptr = reinterpret_cast<vnddmgr*>
                                           ( vnddmgr_ );
  try {
    while (1) {
      char buf[CDEV_REQUEST_MAX_LENGTH] = {0};
      /* Get a new packet from vndd manager kernel driver
       */
      size_t buflen = vnddmgr_ptr->get_packet(buf, sizeof
                                       (buf), if_name);
      if (buflen>0 && buflen <= sizeof(buf)) {
        try {
        /* Search for tunnel instance corrisponding to the
                                        interface if_name
         */
          const vnddtunnel & tunnel = tvm_ptr->get_tunnel_
                                    instance( if_name );
          ip_address remote_ip = tunnel.get_remote_ip();
          udp_socket::port_t remote_port = tunnel.get_
                                          remote_port();
          printf("%s: packet announced by %s, xmit to the
              peer %s:%i\n", __FUNCTION__, if_name.c_str(),
                        std::string( remote_ip ).c_str(),
                                    remote_port & 0xffff
          );
          size_t byteSent = 0;
          if (use_des) {
            crypt_buf crypted_buf(buf, buflen, sKey.c_
                                                 str());
            /* Crypted packet is sent across the tunnel to
                                         the remote peer
             */
            byteSent = tunnel.tunnel_channel().sendto
                   (crypted_buf.get_buf(), crypted_buf.get
                     _buf_len(), remote_ip, remote_port );
          }
          else {
            /* The packet is sent across the tunnel to the
                                             remote peer
             */
            byteSent = tunnel.tunnel_channel().sendto(buf,
                          buflen, remote_ip, remote_port );
          }
          if ( byteSent<=0 ) {
            fprintf(stderr, "%s: tunnel.tunnel_channel
               ().sendto " "error sending sending to %s:%i
                    (thread terminating)\n", __FUNCTION__,
                         std::string( remote_ip ).c_str(),
                                     remote_port & 0xffff
            );
            return -1;
          }
        }
        catch (vnddtunnelmgr::exception & e) {
          switch (e) {
            case vnddtunnelmgr::TUNNEL_INSTANCE_NOT_FOUND:
              break;
            default:
              assert(0);
          }
        }
      }
    } // ... while (1)
  }
  catch (...) {
    assert( 0 );
  }
  return 0;
}
```

- *H2* un host con due interfacce di rete: una configurata con indirizzo pubblico 120.0.0.2 e l'altra con indirizzo privato 192.168.2.254 (la scelta dei due indirizzi pubblici è casuale e serve solo per fissare le idee).
- *H1* è in grado di raggiungere *H2* attraverso l'interfaccia IP 120.0.0.1. Così come *H2* può raggiungere *H1* attraverso la propria interfaccia pubblica.
- I due host sono i *default gateway* per le rispettive sotto-reti private.

Tenendo conto dello scenario illustrato, mostriamo nel seguito del paragrafo un esempio di configurazione degli host *H1* e *H2* per la realizzazione della rete denominata *virtual LAN 3* attraverso la creazione di un tunnel con TVPN.

Per prima cosa si creano le interfacce virtuali che denominiamo *vlan3* su entrambi gli host usando il medesimo comando:

```
vnddconfig add vlan3
```

La precedente operazione è lecita dal momento che lo spazio dei nomi delle interfacce è locale all'host.

Dopo si configurano le interfacce virtuali come interfacce IPv4 *punto-punto*, e si disabilita per queste il protocollo *ARP*. Per configurare *H1* si esegue il comando:

```
ifconfig vlan3 192.168.3.1 pointopoint 192.168.3.2 -
                                                  arp
```

LISTATO 4 Implementazione del thread tunnel_recv_thread

```
int vnddtunnelmgr::tunnel_recv_thread(unsigned long name_,
               unsigned long tvm_, unsigned long vnddmgr_,
                                   unsigned long des_key_)
{
  assert ( name_ );
  assert ( tvm_ );
  assert ( vnddmgr_ );
  bool use_des = des_key_ != 0;
  std::string sKey;
  if (use_des) {
    sKey = std::string ( reinterpret_cast<const char*>
                                         ( des_key_ ) );
  }
  std::string name = std::string ( reinterpret_cast<const
                                    char*> ( name_ ) );
  vnddtunnelmgr* tvm_ptr = reinterpret_cast
                             <vnddtunnelmgr*> ( tvm_ );
  vnddmgr* vnddmgr_ptr = reinterpret_cast<vnddmgr*>
                                          ( vnddmgr_ );
  try {
    const vnddtunnel & tunnel =
      tvm_ptr->get_tunnel_instance( name );
    if (! tunnel.try_lock()) {
      return -1;
    }
    while (1) {
      if (tunnel.is_closing()) {
        tunnel.unlock();
        return 0;
      }
      struct timeval timeout = {0};
      timeout.tv_usec = 0;
      timeout.tv_sec = 5;
      udp_socket::poll_state_t poll_state =
        tunnel.tunnel_channel().poll(timeout);
      if (poll_state == udp_socket::TIMEOUT_EXPIRED) {
        continue;
      }
      else if (poll_state == udp_socket::ERROR_IN_
                                     COMMUNICATION) {
        tunnel.unlock();
        return -1;
      }
      char buf[CDEV_REQUEST_MAX_LENGTH];
      ip_address remote_ip;
      udp_socket::port_t remote_port;
      size_t recvdBytesCnt = tunnel.tunnel_
               channel().recvfrom( buf, sizeof(buf),
                           remote_ip, remote_port);
      if (recvdBytesCnt > 0) {
        if (use_des) {
          decrypt_buf* decrypted_buf = new decrypt_
                buf(buf, recvdBytesCnt, sKey.c_str());
          vnddmgr_ptr->announce_packet(name.c_str(),
                  decrypted_buf->get_buf(), decrypted_
                                buf->get_buf_len());
          delete decrypted_buf;
        }
        else {
          vnddmgr_ptr->announce_packet(name.c_str(), buf,
                                     recvdBytesCnt);
        }
        printf("%s announced packet from tunnel (remote
                peer %s:%i) for interface %s\n",
           __FUNCTION__, std::string( remote_ip ).c_str(),
                remote_port & 0xffff, name.c_str() );
      }
      else {
        tunnel.unlock();
        fprintf(stderr, "%s failed receiving a packet from
        tunnel for interface %s" "(thread terminating)\n",
                             __FUNCTION__, name.c_str() );
        return -1;
      }
    }
  }
  catch (exception) {
    return -1;
  }
  catch (...) {
    assert( 0 );
  }
  return 0;
}
```

E analogamente per H2:

```
ifconfig vlan3 192.168.3.2 pointopoint 192.168.3.1 -
                                                  arp
```

In alternativa è possibile creare le interfacce come *broadcast*. In tal caso è necessario attribuire alle stesse un distinto *mac address*, lasciando inoltre abilitato il protocollo ARP oppure aggiornando staticamente la *ARP cache* di ciascun host.

Ultimata la creazione delle interfacce virtuali, si passa alla istanziazione del tunnel eseguendo su entrambi gli host il processo *vnddvpnd*, al quale devono essere forniti, per ogni interfaccia virtuale, gli indirizzi e le porte UDP degli end point (locale e remoto, rispettivamente) del tunnel stesso. Su H1 si può usare il comando:

```
vnddvpnd -tunnel vlan3 120.0.0.1 33000 120.0.0.2 33000
```

E in modo analogo, su H2:

```
vnddvpnd -tunnel vlan3 120.0.0.2 33000 120.0.0.1 33000
```

Anche se la porta *33000* è stata scelta in modo arbitrario, in generale tale scelta dovrebbe tenere conto della configurazione del sistema, della presenza di eventuali firewall, di meccanismi di *natting* e/o *transparent proxy*, e più estesamente dei meccanismi di traduzione di indirizzi tra rete privata e rete pubblica, omessi dal nostro esempio per ragioni di semplicità.

Il programma *vnddvpnd* può essere eseguito opzionalmente come servizio specificando il parametro *-daemonize*. Se invece viene eseguito come un normale processo, l'output prodotto fornisce informazioni utili per la diagnostica.

Per ottenere che la connessione del tunnel venga cifrata si può usare il parametro *-pwd* seguito dalla stringa usata come chiave dall'algoritmo DES, che deve essere la medesima per *H1* e *H2*.

L'elenco completo dei parametri accettati da *vnddvpnd* può essere ottenuto eseguendo questo programma senza argomenti; l'output prodotto è quello che riportiamo di seguito:

```
vnndvpnd -tunnel <tunnel_param>
         [-tunnel tunnel_param ...]
         [-cdev <cdevname>]
         [-daemonize]
where
  <tunnel_param> = ifname
                   local_ip local_port
                   remote_ip remote_port
                   [-pwd <password>]
default <cdevname> is /dev/vnddmgr
```

Dato che *H1* e *H2* fanno da gateway per le rispettive sotto-reti, è necessario che su questi due host venga abilitato l'*IP forwarding*. Questo può essere ottenuto scrivendo "1" nella entry "*/proc/sys/net/ipv4/ip_forward*" del *proc file system*, usando (per esempio) il comando:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

Infine, per completare la configurazione della VPN si devono aggiornare le tabelle di routing degli host delle reti private in modo che adottino come default gateway l'host 254 della rispettiva subnet.

Completata la configurazione nel modo descritto, un qualunque host della rete privata 192.168.1.0 potrà comunicare con un qualunque altro host della rete privata 192.168.2.0, attraverso la rete virtuale creata con *TVPN*.

Conclusioni

Gli argomenti affrontati in quest'articolo speriamo offrano uno spunto per coloro che abbiano interesse per GNU/Linux e il supporto che questo sistema offre per i protocolli di comunicazione, a partire dalla infrastruttura dei network device driver. Abbiamo implementato TVPN, non per assenza di strumenti e soluzioni (che anzi abbondano per Linux), ma per esercitare le nostre conoscenze e offrire al lettore il punto di vista, come sempre, dello sviluppatore.

Suggeriamo a chi non avesse molta familiarità con gli argomenti affrontati (in particolare con il mondo dei device driver) di consultare [1]. La seconda edizione di questo libro (che tratta fino alla versione 2.4 del kernel) è distribuita gratuitamente all'URL [2]. I socket buffer e la gestione della memoria nei NDD (che nell'articolo non sono stati approfonditi) sono descritti in un interessante articolo di Alan Cox pubblicato all'URL [4] al quale vi rimandiamo.

Crediamo che il lettore interessato potrà trarre giovamento dall'analisi dei sorgenti di TVPN, anzi lo sproniamo a sperimentare in prima persona modificando ed estendendo il progetto originale che potrà certamente essere migliorato.

Bibliografia & Riferimenti

[1] J. Corbet, G. Kroah-Hartman, A. Rubini – "Linux Device Drivers, 3rd Edition", O'Reilly, 2005
[2] http://safari.oreilly.com/?XmlId=0-59600-008-1
[3] http://lxr.linux.no/blurb.html
[4] http://www.tldp.org/LDP/khg/HyperNews/get/net/net-intro.html
[5] http://download.fedora.redhat.com/

CODICE ALLEGATO

ftp.infomedia.it **VPN**