Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

VSNLib: UN' INTERFACCIA UNICA DI CONFIGURAZIONE PER STACK VIRTUALI TRAMITE PACCHETTI NETLINK

Relatore:
Chiar.mo Prof.
RENZO DAVOLI

Presentata da:
SIMONE PREITE

Sessione II
Anno Accademico 2016/17

Introduzione

Non vi è dubbio che i nodi nella rete internet siano cresciuti in maniera esponenziale fino ad oggi. Nonostante l'enorme dimensione raggiunta, questa rete ha mantenuto gli stessi principi costruttivi che impiegava quando collegava poche macchine sperimentali: i nodi sono considerati ancora le interfacce di rete degli elaboratori. Questa situazione non è più coerente con il funzionamento odierno di internet, dove il ruolo fondamentale non è più quello delle macchine che erogano servizi bensì dei servizi stessi.
La crescita smisurata dei nodi di rete e la necessità di passare da un'indirizzamento di nodi a quello di processi genera il bisogno di avere una grande quantità di indirizzi per poter accedere a questi processi individualmente. La famiglia di protocolli oggi più in uso, IP vesione 4 (IPv4), risulta insufficiente per questo scopo, infatti vengono usati solamente 4 byte per indirizzare tutti i nodi accedibili direttamente.
Per questo, già da tempo, è in atto la migrazione verso la versione 6 del procollo IP (IPv6) che consente di avere una quantità di indirizzi maggiore, sia per la problematica che abbiamo appena visto sia perché sempre più si affacciano sulla rete dispositivi automatici capaci di agire come entità autonome, oggetti provvisti di collegamenti di rete: è il fenomeno del così detto Internet of Things o IoT.
In questa ottica si colloca il presente lavoro di tesi. La realizzazione di strumenti dell'Internet of Threads (IoTh), cioè la possibilità di assegnare indirizzi anche a singoli processi e thread, richiede l'utilizzo di stack di rete a livello del singolo processo, ovvero implementati come librerie e funzionanti in spa-

zio utente.

Se l'interfaccia di uso comune da parte delle applicazioni della rete ha come standard la API di Berkeley socket non ne esiste una altrettanto standardizzata per la configurazione e la gestione delle interfacce di rete, essendo questa, nell'ottica degli elaboratori come nodi di rete, prerogative riservate agli amministratori di sistema e non comunemente accessibili come API da parte di processi.

In questa tesi si studierà come realizzare una libreria che sia in grado di interfacciarsi ad applicazioni che usano il protocollo netlink, usato dal kernel linux per la configurazione e la gestione delle interfacce di rete, in modo da poter utilizzare, all'interno dei programmi, gli strumenti per gli amministratori di rete.

Il primo capitolo presenta una rassegna dello stato dell'arte nell'ambito dei protocolli di rete e del paradigma di IoTh; nel secondo capitolo verrà trattato il progetto VSNLib, ovvero la libreria oggetto della tesi; infine il terzo capitolo descriverà alcuni esempi di utilizzo della stessa.

Indice

Elenco delle figure

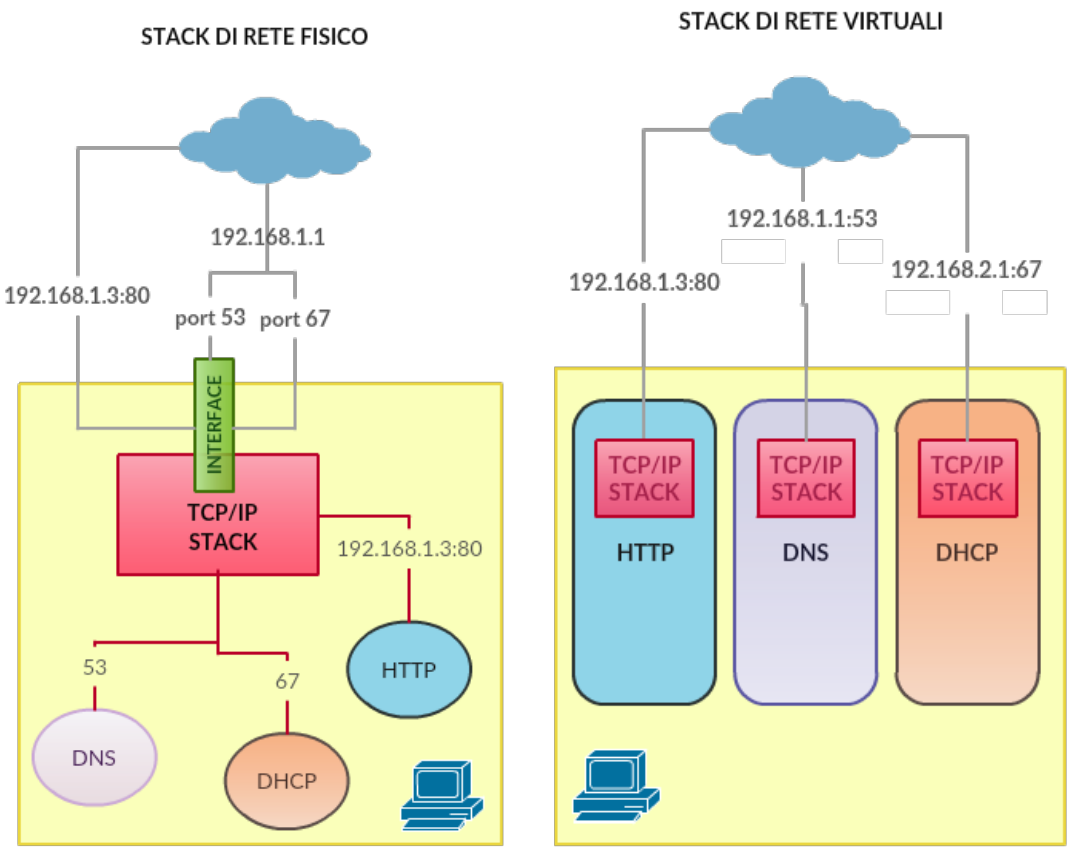
Elenco delle tabelle

Capitolo 1

Internet of Threads

1.1 Definizione

L'Internet of Things, ovvero l'internet delle cose, vuole rappresentare la diffusione dei sistemi embedded come veri e propri nodi di reti. Nell'ottica dell'IoT, oggi, apparecchi comuni in un'abitazione, quali condizionatore, la caldaia o addirittura il tostapane, possono essere nodi di rete ed essere quindi configurati e controllati attraverso di essa.
Il concetto di Internet of Threads è una evoluzione di questa astrazione. L'idea alla base di IoTh è quella di avere non solo oggetti ma anche processi come nodi di rete.
Per poter capire pienamente i vantaggi dell'IoTh si può pensare ad un'analogia con quanto è successo nel mondo della telefonia. In passato la telefonia collegava apparecchi fissi connessi via cavo. In questo modello il numero di telefono individuava un luogo e non una persona e quindi era comune, per trovare una persona, doversi interrogare sul dove questa potesse essere; al contrario l'avvento dei telefoni cellulari ha cambiato l'idea di significato di numero di telefono, non più associato a luoghi fisici ma ad una persona infatti i telefoni cellulari sono normalmente telefoni personali[1, 2].
Come si può notare infatti, con questa tecnologia lo stack è direttamente integrato nel programma. Nello stesso modo IoTh indirizza i processi che



(a) stack associato all'interfaccia fisica (b) stack associato ai processi

Figura 1.1: Differenze tra stack reali e virtuali

forniscono un servizio. Significa che questi non sono più vincolati a risiedere in un luogo specifico, o in un elaboratore specifico nell'ambito delle reti, ma è possibile trasparentemente farli migrare da una macchina all'altra senza che questo crei difficoltà di indirizzamento.
Una nota va fatta in termini di sicurezza. La possibilità che ogni servizio utilizzi propri indirizzi IP e uno stack di rete privato rende molto più difficile, se non praticamente impossibile, sfruttarne le fragilità per scalare l'intrusione ad altri servizi presenti su di esso.

1.2 Stack Implementati

Diversi sono i progetti che si occupano di offrire ai processi il proprio stack di rete. Tra questi ne verranno presi in esame tre, considerati più indicati per uno studio in quanto open source. Ognuno di essi offre la propria implementazione di stack di rete.
Uno stack di rete legato al processo permette di connettere lo stesso ad una rete reale, tramite le interfacce fornite dal sistema, o virtuale (ad esempio VDE) come se fosse una macchina fisica a sè stante. Grazie a questo ogni processo può gestire la rete con le proprie regole che possono differire da quelle in uso sulla macchina che lo ospita.

1.2.1 PicoTCP

PicoTCP[4] è supportato da Tass Belgium (Altran); è lo stack più conosciuto e diffuso per sistemi embedded ed esistono anche dei progetti, basati su IP, di reti mesh realizzati con picoTCP[14].
PicoTCP presenta una struttura modulare (schema molto utilizzato per questo tipo di progetti come si vedrà in seguito), che permette di avere il minimo indispensabile all'interno dell'applicazione.
Vanta la compatibilità con un gran numero di dispositivi costruiti con diversi processori e interfacce di rete, può essere compilato con diversi compilatori ed è già integrato in alcuni sistemi nati per ridurre al minimo il consumo di risorse.
Purtroppo a causa di incompatibilità con la licenza GPL alcune parti del codice non sono state rese pubbliche anche se rappresentano una piccola percentuale dell'intero progetto.

1.2.2 LWIP

Nasce come progetto di Adam Dunkels pensato per sistemi embedded ma che inizialmente non forniva supporto ad IPv6.
Light-weight IP (LWIP[13]) vanta un core molto piccolo e la possibilità di

eseguire anche senza sistema oprativo (single-threaded) e con il minor consumo di RAM possibile, inoltre è modulare ed offre la possibilità di aggiungere protocolli come DHCP e DNS solo in caso di necessità.
Supporta sia little che big endian e gira su processori a 8 e 32 bit, quindi funziona anche su un semplice ATMEGA.
Il protocollo IPv6 è ancora in fase sperimentale: il supporto IPv6 è implementato solamente nella versione disponibile da GITHub.

1.2.3 LWIPv6

LWIPv6[6] nasce per colmare la mancanza del supporto IPv6 in LWIP, col tempo si è differenziato al punto tale da essere considerato un progetto a sè stante.
Quando al team di virtualsquare[1][15] è sopravvenuta la necessità di avere uno stack per la rete VDE non esisteva ancora nulla di maturo o comunque plug and play, da questa esigenza il team ha pensato di realizzare uno stack versatile in versione libreria tale per cui ogni processo potesse avere il proprio stack di rete.
Caratteristica principale di LWIPv6 è il suo *motore* ibrido che, di fatto funziona solo con IPv6.
Gli indirizzi usati dal LWIPv6 sono descritti nel RFC 4291[17] al paragrafo 2.5.5.2. IPv4-Mapped IPv6 Address. La conversione degli indirizzi ip da IPv4 ad IPv6 è particolare ma intuitiva. I 32 bit finali dell'indirizzo IPv6 rappresentano quello in versione 4, i 16 bit antecedenti sono settati a 1 e la parte restante, i primi 80 bit, a 0. In tabella 1.1 un esempio sull'indirizzo 192.168.1.2.

Per quanto riguarda le maschere però questa configurazione non può essere adottata. Se i primi 80 bit venissero settati a 0 non sarebbero considerati e nel caso in cui due reti differissero proprio nei primi 80 bit non si potrebbero

[1] un progetto ideato da Renzo Davoli e Michael Goldweber. Comprende una gamma di tool pensati per sperimentare e creare strumenti di uso comune in tema di virtualizzazione nell'ambito dei sistemi operativi e delle reti.

IP $\langle version \rangle$	80 bit	16 bit	32 bit
$IPv4$	$unused$	$unused$	192.168.1.2
$IPv6$	00000000.00000000.0000	$FFFF$	192.168.1.2
$IPv6$	00000000.00000000.0000	$FFFF$	$C0A80102$

Tabella 1.1: rappresentazione di un indirizzo IPv4 in IPv6

più smistare i pacchetti verso la corretta rete di destinazione.
LWIPv6 si caratterizza perché, tramite una modifica della conversione, non ha bisogno di usare due *motori* diversi per IPv4 ed IPv6 ma uno che è ibrido. La modifica è semplice ed è stato sufficiente ribaltare i primi 80 bit. Come prima, suggeriamo un esempio di conversione di una maschera all'interno di LWIPv6 nella tabella 1.2

IP $\langle version \rangle$	80 bit	16 bit	32 bit
$IPv4$	$unused$	$unused$	255.255.255.0
$IPv6$	$FFFFFFFF.FFFFFFFF.FFFF$	$FFFF$	255.255.255.0
$IPv6$	$FFFFFFFF.FFFFFFFF.FFFF$	$FFFF$	$FFFFFF00$

Tabella 1.2: rappresentazione delle maschere in LWIPv6

1.3 Netlink

1.3.1 Inter Process Communication(IPC)

Molti processi necessitano di scambiare informazioni per i motivi più disparati, dalla comunicazione di rete a quella interna ad una sola macchina. È banalmente limitativo pensare di costruire solo programmi fini a se stessi e che quindi non prevedano nessuna interazione con altri programmi.
Con IPC intendiamo quindi l'insieme delle tecnologie adottate per permettere ai processi di comunicare tra essi, sia che siano ospitati sulla stessa macchina

sia che siano distribuiti sulla rete. Tra queste tecnologie esiste appunto quella utilizzata all'interno della libreria VSNLib di cui parleremo nello specifico del prossimo capitolo.

1.3.2 Il Sistema IPC Netlink

Netlink è un sistema IPC adottato dal kernel linux perché in grado di mettere in comunicazione diversi task. Solitamente serve per far comunicare task in user-space con task in kernel-space ma può far interagire anche processi entrambi in user-space.
Studiato per essere il successore di ioctl per le configurazioni ed il monitoraggio, si propone di essere più flessibile. Ma come avviene esattamente la comunicazione attraverso netlink?
Netlink utilizza un sistema di socket indicizzato in base ai processi. Ogni processo può definire socket diversi di tipo netlink (AF_NETLINK, NETLINK_GENERIC, NETLINK_XFRM) per inviare e ricevere messaggi con e da altri processi; implementa un sistema di porte basato sul process ID. Ad esempio per la comunicazione con il kernel viene usata la porta 0.
In figura 1.2 possiamo vedere uno schema esplicativo di come avvengono le comnucazioni netlink, siano esse tra processi o tra processi e kernel

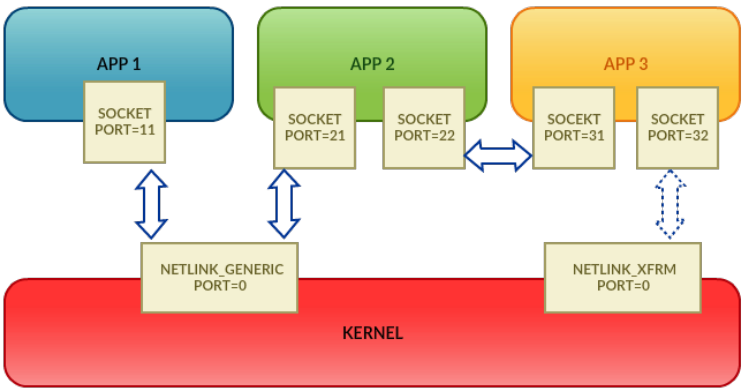


Figura 1.2: scambio di messaggi attraverso socket netlink

L'interfaccia di comunicazione è abbastanza standardizzata ma il payload è personalizzabile ed è possibile definire la propria struttura da utilizzare per inviare dati (informazioni), tra processi diversi, attraverso i socket netlink.

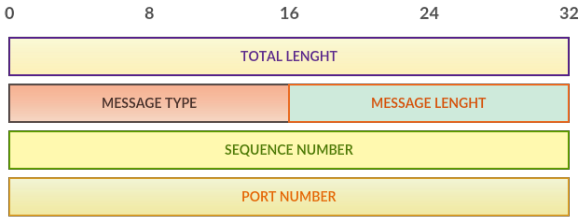


Figura 1.3: schema della struttura nlmsghdr

1.3.3 Libnl

Netlink Protocol Library Suite (libnl) [10], è una raccolta di librerie ed utility che forniscono API di comunicazione netlink basate su quelle del kernel linux e comprende:

libnl Core della libreria, offre uno strato di unificazione delle interfacce sulle quali poi si basano le altre librerie che pertanto dipendono da questa;

libnl-route Questa libreria si occupa di fornire API per la configurazione degli elementi della famiglia NETLINK_ROUTE;

libnl-genl genl significa generic netlink e questa libreria offre una versione estesa del protocollo netlink;

libnl-nf API per configurazioni netlink basate su netfilter.

Capitolo 2

VSNLib

Interfaccia di configurazione degli stack di rete attraverso l'utilizzo di tecnologia netlink.

2.1 Il Progetto

Gli stack analizzati in precedenza offrono a grandi linee la stessa tipologia di servizio seppur ognuna con le proprie caratteristice.
Nessuno dei progetti ha però tenuto in considerazione l'idea di utilizzare un'interfaccia di configurazione che fosse standard, è pertanto necessario usare le funzioni specifiche per ognuno di questi, costringendo il programmatore a cambiare modalità di interazione da stack a stack.
L'idea di creare questa VSNLib nasce proprio dall' esigenza di uniformare le interfacce di comunicazione in modo tale che l'utente non sia costretto ad adattarsi ogni qual volta decida di cambiare stack.

2.2 VSNLib

2.2.1 Preambolo

Per configurare uno stack di rete programmi come `ip`[1] generano un pacchetto netlink con le informazioni necessarie lo inviano al kernel che lo elabora, esegue le operazioni e genera un messaggio contenente un flag attraverso il quale viene comunicato il tipo di errore (0 in caso di successo). A questo messaggio è associato un payload di risposta generalmente usato per comporre un feedback da mostrare all'utente che sta interagendo con il programma, in figura 2.1 lo schema dell'header per il messaggio di risposta.

VSNLib è un progetto in fase di sviluppo scritto in C e completamente open-

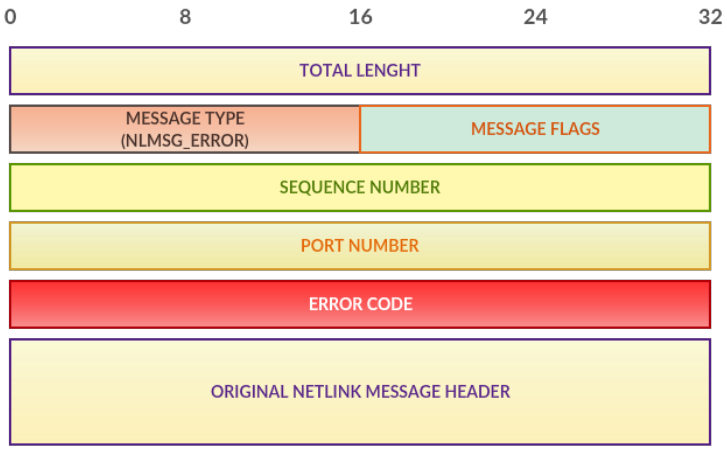


Figura 2.1: schema della struttura per il messaggio di risposta

source, liberamente scaricabile attraverso la piattaforma github al seguente link `https://github.com/simonepreite/vsnlib`.

[1] il programma usato per configurare lo stack di rete tramite comandi quali `ip addr...`, `ip link...`, `ip route...` ecc.

2.2.2 Ambiente di Sperimentazione

Per poter sperimentare e valutare le funzionalita' della libreria VSNLib è necessario intercettare le comunicazioni Netlink fra i processi e il kernel. Per farlo si è reso necessario intercettare la system call di invio della richiesta contenente il payload; ovvero l'esecuzione della sendto, verso il socket netlink sotto la nostra attenzione, contenente il buffer della richiesta.
In questo punto interviene la libreria che si interpone e, di fatto, fa le veci del kernel, al quale il pacchetto netlink non arriverà mai realmente.
In partenza VSNLib utilizzava come *motore* di cattura delle system call la libreria purelibc[8], in seguito però si è giunti alla conclusione che costruire un modulo ad hoc all'interno del sistema vuos fosse un'alternativa più versatile, immediata e pulita, inoltre vuos ha un sistema di debug built in che rende più semplice l'individuazione dei problemi legati allo sviluppo.
Vuos[5] è un progetto di virtualsquare, ha come obiettivo quello di virtualizzare parti del sistema operativo senza negare all'utilizzatore la scelta degli elementi da virtualizzare, è costruito con un sistema di moduli che possono essere aggiunti a discrezione dell'utente.
La versione di sviluppo ufficiale è liberamente scaricabile attraverso la piattaforma github al seguente link `https://github.com/virtualsquare/vuos`.
Per il testing è stato creato un fork di vuos contenente il modulo relativo alla cattura delle system call di interesse per VSNLib, anche questa versione reperibile risiede su github `https://github.com/simonepreite/vuos`.
Il modulo in questione si chiama unrealvsnlib per analogia alla libreria ma ognuno è libero di costruire un proprio modulo che si adatti alle esigenze di sviluppo personali.

2.2.3 Schema

La libreria costituisce uno strato intermedio di compatibilità tra netlink e le specifiche configurazioni degli stack.

Vediamo come è composta:

VSNLib: Il primo strato si occupa di inizializzare la libreria in base allo stack che si intende utilizzare (informazione richiesta all'utente) e di mediare la comunicazione tra netlink ed il modulo scelto.
Indicare esplicitamente il modulo desiderato è una scelta implementativa dettata dal principio di avere un core che sia il più contenuto possibile.

Modules: Sono la parte specifica della libreria, essi contengono l'intestazione delle funzioni generiche che si occupano di chiamare quelle particolari per ogni stack.

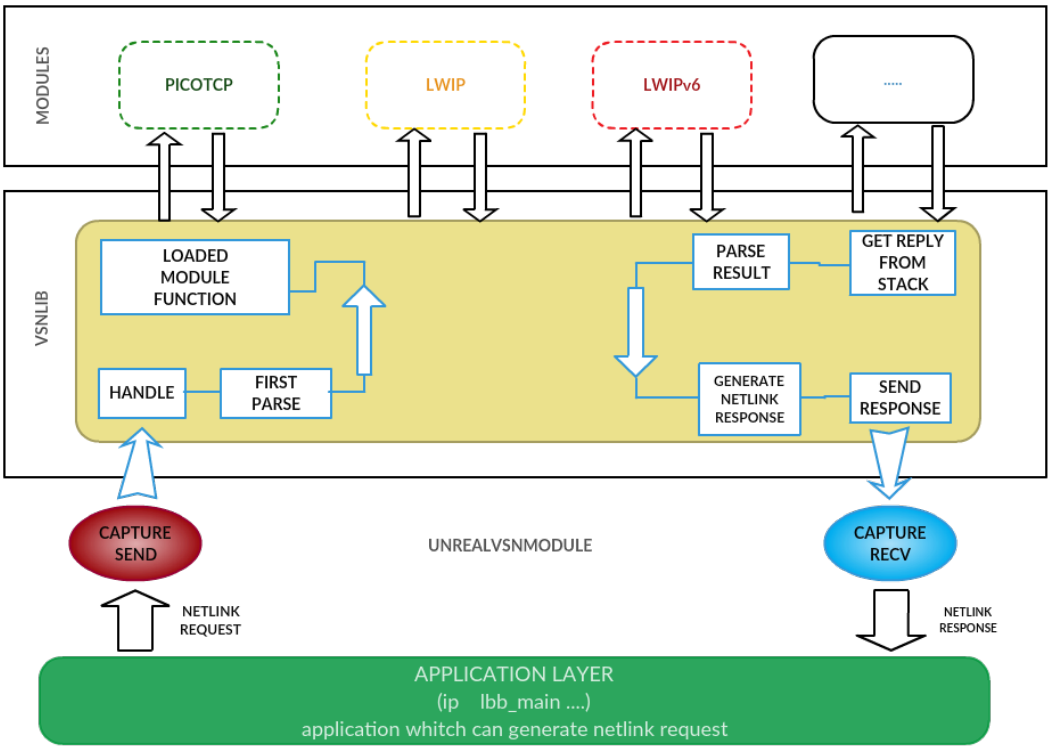


Figura 2.2: mappa concettuale libreria

2.2.4 VSNLib

Espone le interfacce di comunicazione della libreria, che di fatto sono due, una per inizializzare l'ambiente ed una per inviare il pacchetto da gestire.

```
int init_vsnlib(char* stack);
void handle_vsnlib(void* buf, size_t len, void* nif,
   void* stack);
```

La prima funzione è di inizializzazione della libreria e tramite dlopen ed il nome del modulo da caricare si incarica di popolare l'array di puntatori con i corrispettivi delle funzioni nel modulo.
La seconda funzione contiene un loop che cicla sui messaggi del paccheto netlink, che di fatto potrebbero essere più di uno. Per ogni messaggio ne viene analizzata la tipologia ed in base al tipo si accede all'elemento dell'array che contiene il puntatore alla funzione adeguata a svolgere quel compito.
L'utente non ha accesso ad altre funzioni perché è la libreria stessa ad analizzare il pacchetto, riempire i campi della struttura generica ed inviarlo al modulo relspecifico dello stack.
In risposta si riceverà una struttura adatta alla costruzione del paccehetto netlink da spedire alla system call recvfrom del programma in attesa, questo è l'ultimo compito della libreria.

2.2.5 Moduli

La presenza di moduli per gestire i vari stack lascia la libertà a chiunque necessiti di uno stack non supportato, o personalizzato, di creare il proprio modulo rispettando le specifiche della libreria.
Ogni modulo, infatti, ha la stessa struttura e contiene azioni generiche comuni in tutti gli stack. Vengono proposti i prototipi di queste funzioni qui di seguito.

```
struct response* adddeladdr(struct config* cfg);
struct response* getaddr(struct config* cfg);
```

```
struct response* adddellink(struct config* cfg);
struct response* getsetlink(struct config* cfg);
struct response* adddelroute(struct config* cfg);
struct response* getroute(struct config* cfg);
```

In breve ogni modulo è composto dallo stesso numero di funzioni che si occupano di mediare la comunicazione tra la libreria e il vero stack. Ognuna di queste ha un'implementazione diversa a seconda del modulo.
Grazie alla tecnica dei puntatori a funzione, si riesce ad essere abbastanza generali e possiamo evitare di specificarne il nome completo.
L'idea è stata quella di inizializzare, in fase di caricamento della libreria (che avviene con dlopen), un array di puntatori a funzione (attraverso dlsym) in modo tale da poter effettuare le chiamate attraverso di esso. Dlopen e dlsym permettono di caricare dinamicamente solo il modulo di cui si necessita.
All' interno dei moduli le funzioni ricevono in input lo stesso tipo di struttura. Durante l'esecuzione di un'azione loro compito è la raccola di informazioni dalle funzioni reali dello stack che verranno poi usate per riempire i campi della struttura di risposta che a sua volta servirà a creare il payload per il pacchetto netlink di risposta.

Capitolo 3

Casi d'Uso

Di seguito alcuni esempi di utilizzo della libreria. Le dimostrazioni seguenti sono state effettuate su una macchina con architettura amd64 e con installato il sistema operativo debian in versione sid (quindi unstable) ma sono state testate e riprodotte anche su altre configurazioni e distribuzioni GNU/Linux.

3.1 Environment

Come precedentemente accennato è possibile integrare la libreria in vuos attraverso il modulo unrealvsnlib o creandone uno alternativo che dipenda da VSNLib.

Creazione moduli: La creazione di un modulo è molto semplice e basta seguire le linee guida di quelli preesisteni, inoltre grazie ad una routine in cmake non è necessario aggiungere alcuna riga manualmente al makefile e sarà, appunto, cmake ad occuparsi della compilazione.

Start Vuos: Vuos si avvia attaverso il comando `umvu` ed è in grado di interpretare comandi successivi come argomenti, come mostrato nell'esempio infatti, `umvu` è affiancato dal comando konsole e pertanto verrà lan-

ciata una nuova istanza di terminale con ambiente vuos. In alternativa è possibile utilizzare altri comandi come xterm, bash o zsh.

Include dei Moduli vuos I moduli in vuos vengono inclusi attraverso la direttiva

```
vu_insmode <nome_modulo>
```

Un esempio di avvio e inclusione del modulo è quello della figura seguente.

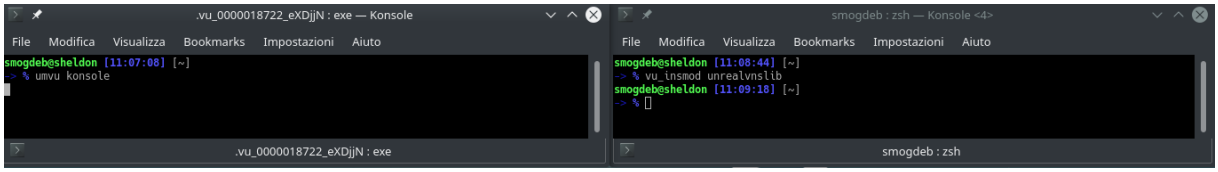


Figura 3.1: start and include vuos

3.2 Esempi

Nella parte successiva verrà mostrato un esempio di configurazione per LWIPv6 attraverso VSNLib.
Per semplificare la dimostrazione si è deciso di non utilizzare l'ambiente sopra descritto o altre tecniche di cattura delle system call ma i pacchetti netlink sono stati costruiti all'interno del codice di test. Sono esempi statici di come li creerebbe il programma `ip` ma utili a mostrare sia come è formato un pacchetto di questa tipologia sia per tracciare i passi compiuti per il parsing e la configurazione.

Init VSNLib: L'unico header di riferimento della libreria necessario è vsn-shared.h, all'interno sono dichiarate le funzioni necessarie all'utilizzo della libreria.

```
extern int init_vsnlib(char* stack);
extern void handle_vsnlib(void* buf, size_t len,
   void* nif, void* stack);
```

Come già spiegato la prima funzione serve per indicare alla libreria di quale modulo fare il load pertanto richiede come input la stringa con nome del file da caricare comprensivo di `.so` (si noti che è necessario specificare, tra i PATH di ricerca delle librerie, quello in cui risiede il modulo `.so`. Senza questo accorgimento il file non verrà trovato da dlopen provocando la terminazione del programma. La variabile dell'environment da settare è `LD_LIBRARY_PATH`, un esempio è mostrato in figura 3.2). La seconda è l'unico "handle" per passare il pacchetto netlink che dovrà essere poi interpretato, inoltre prende come argomeni un puntatore allo stack e all'interfaccia oggetto della modifica.

Strutture: I pacchetti netlink sono accomunati dallo stesso header, come mostrato in precedenza, ma presentano strutture diverse per il resto. In questo esempio sono stati creati i pacchetti per l'aggiunta e la rimozione di un indirizzo da un'interfaccia, l'aggiunta e la rimozione di una rotta e l'accensione/spegnimento di un'interfaccia.

```
struct info{
  struct rtattr attr;
  unsigned char ip6_address[sizeof(struct in6_addr)
     ];
};

  struct addr_payload{
  struct nlmsghdr header;
  struct ifaddrmsg ifa;
  struct info attribute[2];
};

struct route_payload{
```

```
  struct nlmsghdr header;
  struct rtmsg rtm;
  struct info attribute;
};

struct link_payload{
  struct nlmsghdr header;
  struct ifinfomsg ifi;
};
```

rispettivamente:

- info contiene il payload con gli indirizzi.
- addr_payload viene usato per aggiungere/rimuovere un indirizzo a/da un interfaccia.
- route_payload usato per aggiungere/rimuovere rotte.
- link_payload per attivare/disattivare l'interfaccia.

La descrizione è ristretta a questo esempio, ma queste strutture vengono impiegate per compiere più azioni di quelle presentate.

Creazione pacchetti: È eseguita dalle funzioni nel test.

```
void fill_buf_addr(struct link_payload* buffer, int
   mode, int modeIP);
void fill_buf_route(struct link_payload* buffer, int
    mode, int modeIP);
void fill_buf_link(struct link_payload* buffer, int
   mode);
```

Il compito di queste funzioni è riempire i campi delle strutture sopra elencate in base all'operazione richiesta, è possibile modificarle per ottenere comportamenti diversi e compiere altri test.
In pratica si occupano di fare quello che farebbe un programma co-

me `ip` in maniera semplificata, questo rende più semplice al lettore la comprensione, la modifica e l'utilizzo degli strumenti proposti.

VSNLib: Per tutte le operazioni viene chiamata sempre la stessa funzione "handle" di VSNLib cambiando solo il paccehtto inviato. Questo dimostra come la libreria sia in grado di interpretare in maniera autonoma i pacchetti netlink ed utilizzarne le informazioni per completare con successo (o con fallimento in caso le informazioni non siano coerenti) le operazioni richieste.

3.2.1 Test

Ispirato da un programma esplicativo per LWIPv6 questo test dovrebbe avere lo stesso comportamento di netcat ma la configurazione dello stack avviene attraverso VSNLib e i pacchetti netlink, come mostrato nell'Appendice C. Procediamo in sequenza:
Come prima operazione viene inizializzata VSNLib, successivamente viene eseguito un controllo per accertarsi che non sia necessario caricare LWIPv6 dinamicamente ed una volta superato questo step vengono creati stack e interfaccia di rete (nel caso specifico viene collegata ad uno switch vde). In ultima analisi vengono eseguite le configurazioni necessarie a rendere l'interfaccia attiva.

3.2.2 Replica dell'Esperimento

- Preparazione: Il contesto prevede che sulla macchina siano installati vde2 e la libreria LWIPv6, vde2 è reperibile nei repository ufficiali di debian mentre LWIPv6 è disponibile nella versione sid sotto il nome di liblwipv6-dev (si ricorda che tutto il codice è open source ed è disponibile online, ai link di riferimento in bibliografia, per lo studio, lo sviluppo e l'utilizzo).

- Installazione VSNLib:
  Si è utilizzato cmake per automatizzare il processo di compilazione e installazione della libreria.

```
git clone http://github.com/simonepreite/vsnlib.git
cd vsnlib
mkdir build
cd build
cmake ..
make
sudo make install
```

- Uso del test:
  All'interno della directory vsnlib se ne trova un'altra chiamata test-dir. Al suo interno troviamo due file di esempio di un client netcat che utilizza uno stack LWIPv6 sia in configurazione IPv4 sia IPv6. Per compilare il test basta eseguire i comandi:

```
cd test-dir
make
```

  I comandi generano gli eseguibili nella subdirectory build.

- Set vde2:
  Per questo esempio è stato usato uno switch vde il cui path è passato al programma di test per creare l'interfaccia.
  Bisogna creare, in primo luogo, lo switch

```
vde_switch -s </path/of/switch>
```

  Questo path deve essere passato al programma test, ovviamente può essere modificato a piacimento.

- Set vdens:
  Per la creazione del server invece si è optato per il programma vdens

che da la possibilità di agganciarsi allo switch, per utilizzarlo è possibile scaricare vdens direttamente dal suo repo[16].
La sintassi è abbastanza intuitiva, per quanto riguarda il test:

```
vdens vde://</path/of/switch>
```

Si noti che se il path del file con lo switch parte da root gli "/" diventano 3, ad esempio

```
vdens vde:///tmp/switch1
```

Inoltre si ricordi, come spiegato nel readme di vdens, che l'uso dei namespace agli utenti non privilegiati deve essere abilitato in debian

```
sudo echo 1 > /proc/sys/kernel/
   unprivileged_userns_clone
```

Entrati nell'ambiente vdens bisogna configurarlo per il server che si vuole con il nostro test: IPv4:

```
ip addr add 192.168.250.1/24 dev vde0
ip link set vde0 up
nc -l -p 9999
```

IPv6:

```
ip -6 addr add 2001:db8:0:f101::1/64 dev vde0
ip link set vde0 up
nc -l6 -p 9999
```

Si presti attenzione alla versione di netcat installata in quanto netcat-traditional non supporta IPv6, si suggerisce netcat-openbsd.

- Start test:
  Ora non resta che lanciare il programma di test in base alla configurazione scelta e controllare che netcat permetta di scambiare messaggi come nelle figure 3.2 e 3.3.

```
./testX.exe STACK SWITCH
```

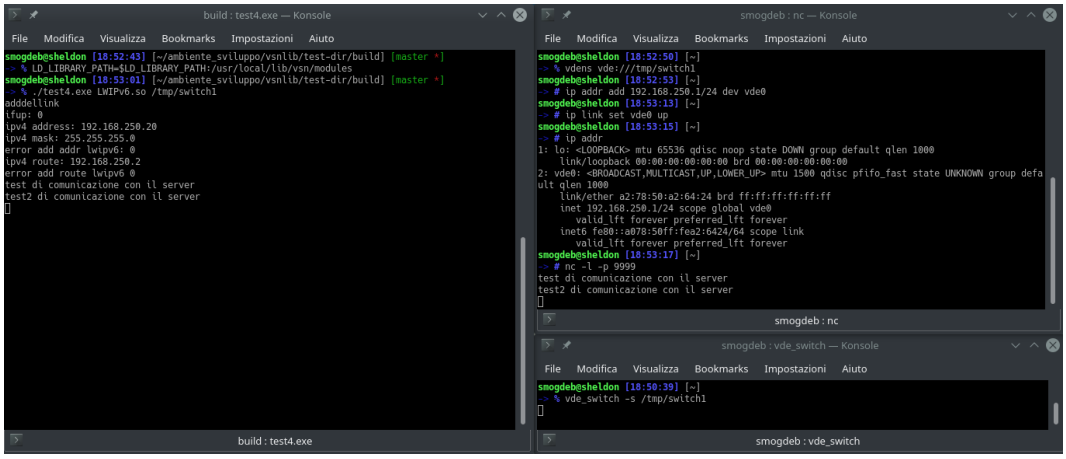


Figura 3.2: netcat IPv4

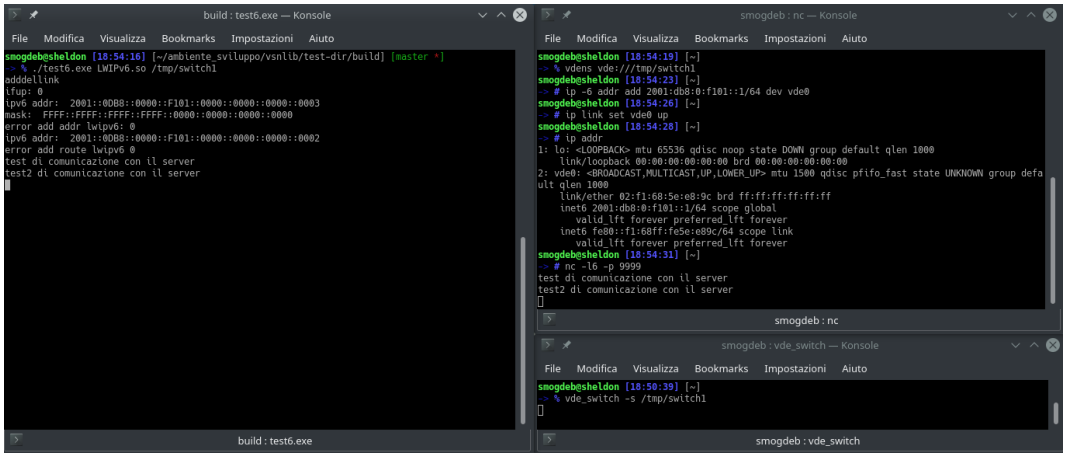


Figura 3.3: netcat IPv6

Conclusioni

Lo state dell'arte al momento in cui si scrive è una libreria in grado di configurare uno stack per LWIPv6, mentre gli altri moduli sono in fase di sviluppo ma esistono già dei proof of concept (moduli in fase sperimentale) all'interno del repo su github[1]. I risultati dei test effettuati, nonostante il progetto sia tutt'ora in fase di sviluppo, sono significativi: si riesce a configurare uno stack specifico in maniera generica ed usando le interfacce proposte dal kernel Linux.
L'utilizzo di queste tipologie di stack dipende strettamente dalla diffusione del protocollo IPv6 e dei sistemi embedded.
Lo sviluppo di tali tecnologie è un punto chiave per il testing del procollo IPv6.
La ricerca e lo sviluppo di questi meccanismi sono necessari per ottenere un sistema performante e stabile quando IPv6 sarà l'ordinario. Vanno inoltre considerati i vantaggi conseguenti nelle reti virtuali che già ora vengono utilizzate per la sperimentazione e che in futuro saranno sicuramente di larga diffusione.
L'indipendenza dalle infrastrutture fisiche è un vantaggio non indifferente, significa che le reti ed i sistemi possono essere creati, modificati o ricostruiti da zero con estrema facilità. Basti pensare ad una rete di macchine virtuali che, trattandosi di software, può essere ampliata con nuovi elaboratori virtuali, spenta e modificata senza dover invesitire in nuovo hardware.
Quelli esposti sono solo alcuni dei vantaggi ma sono sufficienti a lasciare

[1] http://github.com/simonepreite/vsnlib

intendere la direzione verso cui questo settore sta proseguendo.

Sviluppi Futuri

In fase di progettazione non sono stati posti limiti alla crescita del progetto e la sua modularità è legata principalmente a questo aspetto.
L'augurio è che questo progetto continui a crescere arricchendosi di funzionalità.
In questa fase ci si è occupati delle configurazioni e del routing che deve essere completata e testata prima di poter affermarne la stabilità.
Senz'altro aumentare il supporto agli stack meno conosciuti ed a quelli che in futuro saranno sviluppati è il prosieguo più ovvio per VSNLib. Tra le altre ipotesi di ampliamento c'è quella di gestione del filtering, l'intento in questo caso sarà fornire un'interfaccia identica a quella di iptables ma per settare le regole di filtraggio sugli stack virtuali.

Appendice A

Core

vsnshared.h

```
#ifndef _VSNSHARED_H
#define _VSNSHARED_H

#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>
#include <arpa/inet.h>

struct response{
  int error; //0 for success
};

extern void handle_vsnlib(void* buf, size_t len, void* nif, void* stack);
extern int init_vsnlib(char* stack);

#endif
```

vsnlib.h

```
#ifndef _VSNLIB_H
#define _VSNLIB_H

#include "vsnshared.h"

#define  API_TABLE_SIZE 6

/* struct definition */
struct config{
  char* addr;
  int mask;
```

```
  int family;
  void* interface;
  void* stack; // stack file descriptor
  int operation;
};

typedef struct response* (*generic_api)(struct config* cfg);
struct gen_api{
  char* fun_name;
  generic_api real_fun;
};

static struct gen_api api_table[]={
  {"adddellink", NULL},
  {"getsetlink",NULL},
  {"adddeladdr", NULL},
  {"getaddr", NULL},
  {"adddelroute", NULL},
  {"getroute", NULL}
};

/* client side */

struct nlmsghdr* rtm_newroute_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_delroute_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_getroute_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_newaddr_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_deladdr_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_getaddr_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_newlink_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_dellink_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_getlink_c(struct nlmsghdr* header, void* nif, void*
    stack);
struct nlmsghdr* rtm_setlink_c(struct nlmsghdr* header, void* nif, void*
    stack);

typedef struct nlmsghdr* vsn_cli(struct nlmsghdr* header, void* nif, void*
    stack);
static vsn_cli *vsncli_fun_table[]={
```

```
  /* NEW/DEL/GET/SET link */
  rtm_newlink_c,
  rtm_dellink_c,
  rtm_getlink_c,
  rtm_setlink_c,
  /* NEW/DEL/GET addr */
  rtm_newaddr_c,
  rtm_deladdr_c,
  rtm_getaddr_c,
  NULL,
  /* NEW/DEL/GET route */
  rtm_newroute_c,
  rtm_delroute_c,
  rtm_getroute_c,
  NULL
};

#endif
```

vsnlib.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>
#include <dlfcn.h>

#include <vsnlib.h>
#include <vsnshared.h>


struct ret_val{
  struct nlmsghdr* res;
  struct ret_val* next;
};

char* combine_stack_handler(char* fun_name, char* stack){
    //rimuove il .so
    char* combo;
    int len_stack = strlen(stack)-1;
    combo = malloc(sizeof(char)*(strlen(fun_name)+(len_stack-1)));
    strcpy(combo, stack);
    combo[len_stack-2] = '_';
    combo[len_stack-1] = '\0';
    strcat(combo, fun_name);
    return combo;
```

```
}


int init_vsnlib(char* stack){
  const char *error;
  char* f;
  void *stack_module;

  stack_module = dlopen(stack,RTLD_LAZY);
  if(!stack_module){
    fprintf(stderr, "Cannot open stack library: %s\n",
    dlerror());
    return -1;
  }
  for(int i=0; i < API_TABLE_SIZE; i++){
    api_table[i].real_fun = dlsym(stack_module, api_table[i].fun_name);
    if ((error = dlerror())) {
      fprintf(stderr, "Couldn't find %s: %s\n", api_table[i].fun_name, error
          );
      exit(1);
    }
  }
  return 0; //if success
}

void handle_vsnlib(void* buf, size_t len, void* nif, void* stack){
  struct nlmsghdr* header = (struct nlmsghdr*) buf;
  struct nlmsghdr* ret_pkg;
  while (NLMSG_OK(header, len)) {
    /*int err;*/
    int type;

    if (header->nlmsg_type == NLMSG_DONE)
      break;
    type=header->nlmsg_type - RTM_BASE;
    header->nlmsg_pid=getpid();
    if (type >= 0 && vsncli_fun_table[type] != NULL) {
        struct ret_val* app = malloc(sizeof(struct ret_val));
        (vsncli_fun_table[type](header, nif, stack));
    }
    header = NLMSG_NEXT(header, len);
  }
}

struct nlmsghdr* rtm_newlink_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifinfomsg* ifi = (struct ifinfomsg*)(NLMSG_DATA(header));
```

```
  struct config generic_conf;
  struct response* generic_resp;
  generic_conf.interface = nif;
  generic_conf.operation = ifi->ifi_flags;
  generic_resp = api_table[0].real_fun(&generic_conf);
}

struct nlmsghdr* rtm_dellink_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifinfomsg* ifi = (struct ifinfomsg*)(NLMSG_DATA(header));
  struct config* generic_conf;
  struct response* generic_resp;
  generic_resp = api_table[0].real_fun(generic_conf);
}

struct nlmsghdr* rtm_getlink_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifinfomsg* ifi = (struct ifinfomsg*)(NLMSG_DATA(header));
  struct config* generic_conf;
  struct response* generic_resp;
  generic_resp = api_table[1].real_fun(generic_conf);
}

struct nlmsghdr* rtm_setlink_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifinfomsg* ifi = (struct ifinfomsg*)(NLMSG_DATA(header));
  struct config* generic_conf;
  struct response* generic_resp;
  generic_resp = api_table[1].real_fun(generic_conf);
}

struct nlmsghdr* rtm_newaddr_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifaddrmsg* ifa = (struct ifaddrmsg*)(NLMSG_DATA(header));
  struct config generic_conf;
  struct response* generic_resp;
  char str6[INET6_ADDRSTRLEN];
  char* str4;
  generic_conf.operation=header->nlmsg_type;
  generic_conf.interface=nif;
  generic_conf.mask=ifa->ifa_prefixlen;
  generic_conf.family=ifa->ifa_family;
  struct rtattr* attr = (struct rtattr*)((void*)(((char*)ifa)+ sizeof(struct
      ifaddrmsg)));
  if(ifa->ifa_family == AF_INET6){
    struct in6_addr* ip6_a = (struct in6_addr*)RTA_DATA(attr);
    inet_ntop(ifa->ifa_family, ip6_a, str6, INET6_ADDRSTRLEN);
```

```
    inet_pton(ifa->ifa_family, str6, ip6_a);
    generic_conf.addr=str6;
  }
  else{
    struct in_addr ip4_a = *((struct in_addr*)RTA_DATA(attr));
    str4 = inet_ntoa(ip4_a);
    generic_conf.addr=str4;
  }
  generic_conf.stack=stack;
  generic_resp = api_table[2].real_fun(&generic_conf);
}

struct nlmsghdr* rtm_deladdr_c(struct nlmsghdr* header, void* nif, void*
    stack){
  rtm_newaddr_c(header, nif, stack);
}

struct nlmsghdr* rtm_getaddr_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct ifaddrmsg* ifa = (struct ifaddrmsg*)(NLMSG_DATA(header));
  struct config* generic_conf;
  struct response* generic_resp;
  generic_resp = api_table[3].real_fun(generic_conf);
}

struct nlmsghdr* rtm_newroute_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct rtmsg* rtm = (struct rtmsg*)(NLMSG_DATA(header));
  struct config generic_conf;
  struct response* generic_resp;
  char str6[INET6_ADDRSTRLEN];
  char* str4;

  generic_conf.operation=header->nlmsg_type;
  generic_conf.interface=nif;
  generic_conf.family=rtm->rtm_family;
  generic_conf.stack=stack;
  struct rtattr* attr = (struct rtattr*)((void*)(((char*)rtm)+ sizeof(struct
      rtmsg)));
  if(rtm->rtm_family == AF_INET6){
    struct in6_addr* ip6_a = (struct in6_addr*)RTA_DATA(attr);
    inet_ntop(rtm->rtm_family, ip6_a, str6, INET6_ADDRSTRLEN);
    generic_conf.addr=str6;
  else{
    struct in_addr ip4_a = *((struct in_addr*)RTA_DATA(attr));
    str4 = inet_ntoa(ip4_a);
    generic_conf.addr=str4;
```

```
  }
  generic_resp = api_table[4].real_fun(&generic_conf);
  }
}

struct nlmsghdr* rtm_delroute_c(struct nlmsghdr* header, void* nif, void*
    stack){
  rtm_newroute_c(header, nif, stack);
}

struct nlmsghdr* rtm_getroute_c(struct nlmsghdr* header, void* nif, void*
    stack){
  struct rtmsg* rtm = (struct rtmsg*)(NLMSG_DATA(header));
  struct config* generic_conf;
  struct response* generic_resp;
  generic_resp = api_table[5].real_fun(generic_conf);
}
```

Appendice B

Modules

vsnmodules.h

LWIPv6.c

```
#include <stdio.h>
#include <vsnlib.h>
#include <lwipv6.h>


struct response* adddeladdr(struct config* cfg){
  struct ip_addr addr;
  struct ip_addr mask;
  uint16_t* p_mask = NULL;

  if(cfg->family == AF_INET6){
    uint16_t mask_app[8]={0,0,0,0,0,0,0,0};
    struct in6_addr ip6;

    uint16_t* p = (uint16_t*)(&ip6);
    p_mask = (uint16_t*)(&mask_app);

    for(int i=0; i<8; i++){
      *p=0;
      p++;
    }
    p = (uint16_t*)(&ip6);
    inet_pton(cfg->family, cfg->addr, &ip6);

    while(cfg->mask > 0){
      *p_mask = 0xffff;
      cfg->mask-= 16;
```

```
      p_mask++;
    }

    p_mask = (uint16_t*)(&mask_app);

    printf("ipv6 addr: %04X::%04X::%04X::%04X::%04X::%04X::%04X::%04X\n",
        htons(*(p)),htons(*(p+1)),htons(*(p+2)),htons(*(p+3)),htons(*(p+4)),
        htons(*(p+5)),htons(*(p+6))htons(*(p+7)));
    IP6_ADDR(&addr,htons(*(p)),htons(*(p+1)),htons(*(p+2)),htons(*(p+3)),
        htons(*(p+4)),htons(*(p+5)),htons(*(p+6)),htons(*(p+7)));
    printf("mask: %04X::%04X::%04X::%04X::%04X::%04X::%04X::%04X\n",*(p_mask
        ),*(p_mask+1),*(p_mask+2),*(p_mask+3),*(p_mask+4),*(p_mask+5),*(
        p_mask+6),*(p_mask+7));
    IP6_ADDR(&mask,*(p_mask),*(p_mask+1),*(p_mask+2),*(p_mask+3),*(p_mask+4)
        ,*(p_mask+5),*(p_mask+6),*(p_mask+7));
}
else{
    struct in_addr ip4;
    uint16_t lwip_mask4[4] = {0,0,0,0};
    p_mask = (uint16_t*)(&lwip_mask4);
    while(cfg->mask > 0){
      *p_mask=255;
      cfg->mask -= 8;
      p_mask++;
    }
    inet_aton(cfg->addr, &ip4);
    unsigned char ip4_dot[4];
    int j=0;
    for(int i=0; i<4; i++){
      ip4_dot[i]=(ip4.s_addr >> j) & 0xFF;
      j=j+8;
    }
    printf("ipv4 address: %d.%d.%d.%d\n",ip4_dot[0],ip4_dot[1],ip4_dot[2],
        ip4_dot[3]);
    IP64_ADDR(&addr,ip4_dot[0],ip4_dot[1],ip4_dot[2],ip4_dot[3]);
    printf("ipv4 mask: %d.%d.%d.%d\n",lwip_mask4[0],lwip_mask4[1],lwip_mask4
        [2],lwip_mask4[3]);
    IP64_MASKADDR(&mask,lwip_mask4[0],lwip_mask4[1],lwip_mask4[2],lwip_mask4
        [3]);
}
if(cfg->operation == RTM_NEWADDR){
    printf("error add addr lwipv6: %d\n", lwip_add_addr((struct netif*)(cfg
        ->interface),&addr,&mask));
}
else if (cfg->operation == RTM_DELADDR) {
    printf("error del addr lwipv6: %d\n", lwip_del_addr((struct netif*)(cfg
        ->interface),&addr,&mask));
```

```
  }
}

struct response* getaddr(struct config* cfg){

}

struct response* adddellink(struct config* cfg){
  printf("adddellink\n");
  if(cfg->operation == 1)
    printf("ifup: %d\n", lwip_ifup((struct netif*)(cfg->interface)));
  else
    printf("ifdown: %d\n", lwip_ifdown((struct netif*)(cfg->interface)));
}

struct response* getsetlink(struct config* cfg){

}

struct response* adddelroute(struct config* cfg){
  struct ip_addr addr;

  if(cfg->family==AF_INET6){
    struct in6_addr address;
    uint16_t* p = (uint16_t*)(&address);
    p = (uint16_t*)(&address);
    inet_pton(cfg->family, cfg->addr, &address);
    printf("ipv6 addr: %04X::%04X::%04X::%04X::%04X::%04X::%04X::%04X\n",
        htons(*(p)),htons(*(p+1)),htons(*(p+2)),htons(*(p+3)),htons(*(p+4)),
        htons(*(p+5)),htons(*(p+6))htons(*(p+7)));
    IP6_ADDR(&addr,htons(*(p)),htons(*(p+1)),htons(*(p+2)),htons(*(p+3)),
        htons(*(p+4)),htons(*(p+5)),htons(*(p+6)),htons(*(p+7)));
  }
  else{
    struct in_addr ip4;
    inet_aton(cfg->addr, &ip4);
    unsigned char ip4_dot[4];
    int j=0;
    for(int i=0; i<4; i++){
      ip4_dot[i]=(ip4.s_addr >> j) & 0xFF;
      j=j+8;
    }
    printf("ipv4 route: %d.%d.%d.%d\n",ip4_dot[0],ip4_dot[1],ip4_dot[2],
        ip4_dot[3] );
    IP64_ADDR(&addr,ip4_dot[0],ip4_dot[1],ip4_dot[2],ip4_dot[3]);
  }
```

```
  if(cfg->operation==RTM_NEWROUTE){
    printf("error add route lwipv6 %d\n", lwip_add_route((struct stack*)cfg
        ->stack, IP_ADDR_ANY, IP_ADDR_ANY, &addr, (struct netif*)(cfg->
        interface),0));
  }
  else if(cfg->operation==RTM_DELROUTE){
    printf("error del route lwipv6: %d\n", lwip_del_route((struct stack*)cfg
        ->stack, IP_ADDR_ANY, IP_ADDR_ANY, &addr, (structnetif*)(cfg->
        interface), 0));
  }
}

struct response* getroute(struct config* cfg){

}
```

Appendice C

Test

minlibnetlink.h

```
#ifndef _MINLIBNETLINK_H
#define _MINLIBNETLINK_H
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <lwipv6.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>

#include <asm/types.h>
#include <linux/netlink.h>
#include <linux/rtnetlink.h>

#include <vsnshared.h>

struct info{
  struct rtattr attr;
  unsigned char ip6_address[sizeof(struct in6_addr)];
};

struct addr_payload{
  struct nlmsghdr header;
  struct ifaddrmsg ifa;
  struct info attribute[2];
};
```

```
struct route_payload{
  struct nlmsghdr header;
  struct rtmsg rtm;
  struct info attribute;
};

struct link_payload{
  struct nlmsghdr header;
  struct ifinfomsg ifi;
};

void fill_buf_link(struct link_payload* buffer, int mode);
void fill_buf_route(struct route_payload* buffer, char* IPv6, int mode, int
    modeIP);
void fill_buf_addr(struct addr_payload* buffer, char* IP, int mask, int mode
    , int modeIP);

#endif
```

minlibnetlink.c

```
#include "minlibnetlink.h"

void fill_buf_link(struct link_payload* buffer, int mode){

  // nlmsghdr init
  buffer->header.nlmsg_len=32;
  buffer->header.nlmsg_type=RTM_NEWLINK;
  buffer->header.nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK;

  buffer->ifi.ifi_flags=mode;
  buffer->header.nlmsg_seq=102;
  buffer->header.nlmsg_pid=0;
  // ifaddr init
  buffer->ifi.ifi_family=AF_UNSPEC;
  buffer->ifi.ifi_type=AF_NETROM; //maschera
  buffer->ifi.ifi_index=0;
  buffer->ifi.ifi_change=0x1;
}

void fill_buf_route(struct route_payload* buffer, char* IPv6, int mode, int
    modeIP){
  inet_pton(modeIP, IPv6, buffer->attribute.ip6_address);

  // rtattr init
  buffer->attribute.attr.rta_len=20;
  buffer->attribute.attr.rta_type=RTA_GATEWAY;
  // nlmsghdr init
```

```
  buffer->header.nlmsg_len=64;
  buffer->header.nlmsg_type=mode;

  buffer->header.nlmsg_seq=101;
  buffer->header.nlmsg_pid=0;
  // rtmsg init
  if(modeIP==AF_INET6){
      buffer->rtm.rtm_family=AF_INET6;
  }
  else{
    buffer->rtm.rtm_family=AF_INET;
  }
  buffer->rtm.rtm_dst_len=0; //maschera
  buffer->rtm.rtm_src_len=0;
  buffer->rtm.rtm_tos=0;
  buffer->rtm.rtm_table=RT_TABLE_MAIN;
  buffer->rtm.rtm_protocol=RTPROT_UNSPEC;
  buffer->rtm.rtm_scope=RT_SCOPE_NOWHERE;
  buffer->rtm.rtm_type=RTN_UNSPEC;
  buffer->rtm.rtm_flags=0;
  if(mode==RTM_NEWROUTE){
    buffer->header.nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK|NLM_F_EXCL|
        NLM_F_CREATE;
  }
  else if(mode==RTM_DELROUTE){
    buffer->header.nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK;
  }
}

// generea lo stesso pacchetto netlink generato da ip addr add <IPv6/mask>
    dev <device_name>

void fill_buf_addr(struct addr_payload* buffer, char* IP, int mask, int mode
    , int modeIP){
  inet_pton(modeIP, IP, buffer->attribute[0].ip6_address);
  inet_pton(modeIP, IP, buffer->attribute[1].ip6_address);
  if(modeIP == AF_INET6){
    // rtattr init
    buffer->attribute[0].attr.rta_len=20;
    buffer->attribute[1].attr.rta_len=20;
    buffer->header.nlmsg_len=64;
  }
  else{
    buffer->attribute[0].attr.rta_len=8;
    buffer->attribute[1].attr.rta_len=8;
    buffer->header.nlmsg_len=40;
  }
```

```
  buffer->attribute[0].attr.rta_type=IFA_LOCAL;
  buffer->attribute[1].attr.rta_type=IFA_ADDRESS;
  buffer->ifa.ifa_family=modeIP;
  buffer->header.nlmsg_seq=100;
  buffer->header.nlmsg_pid=0;
  // ifaddr init
  buffer->ifa.ifa_prefixlen=mask; //maschera
  buffer->ifa.ifa_flags=0;
  buffer->ifa.ifa_scope=RT_SCOPE_UNIVERSE;
  buffer->ifa.ifa_index = 0;
  buffer->header.nlmsg_type=mode;
  if(mode==RTM_NEWADDR){
    buffer->header.nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK|NLM_F_EXCL|
        NLM_F_CREATE;
  }
  else{
    buffer->header.nlmsg_flags=NLM_F_REQUEST|NLM_F_ACK;
  }
}
```

test4_vsnlib.c

```
#include "minlibnetlink.h"

#define BUFSIZE 1024
char buf[BUFSIZE];

int main(int argc, char** argv){

  struct sockaddr_in serv_addr;
  int fd;

  struct stack *stack;
  struct netif *nif;
  void* buf;
  void* buf_r;
  void* buf_l;
  struct response* ret;
  struct ip_addr addr_6;

  struct addr_payload buf_addr;
  struct route_payload buf_route;
  struct link_payload buf_link;

  struct ip_addr addr, mask;

  buf=&buf_addr.header;
  buf_r=&buf_route.header;
```

```
// call vsnlib
if(init_vsnlib(argv[1])==-1){
  perror("Cannot init vsnlib");
}

#ifdef LWIPV6DL
/* Run-time load the library (if requested) */
if ((handle=loadlwipv6dl()) == NULL) {
  perror("LWIP lib not loaded");
  exit(-1);
}
#endif
/* define a new stack */
if((stack=lwip_stack_new())==NULL){
  perror("Lwipstack not created");
  exit(-1);
}

/* add an interface */
if((nif=lwip_vdeif_add(stack,argv[2]))==NULL){
  perror("Interface not loaded");
  exit(-1);
}

fill_buf_link(&buf_link, 1);
handle_vsnlib(&buf_link, buf_link.header.nlmsg_len, (void*)nif, (void*)
    stack);

fill_buf_addr(&buf_addr, "192.168.250.20", 24, RTM_NEWADDR, AF_INET);
handle_vsnlib(buf, buf_addr.header.nlmsg_len, (void*)nif, (void*)stack);


fill_buf_route(&buf_route, "192.168.250.2", RTM_NEWROUTE, AF_INET);
handle_vsnlib(buf_r, buf_route.header.nlmsg_len, (void*)nif, (void*)stack)
    ;

memset((char *) &serv_addr,0,sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr("192.168.250.1");
serv_addr.sin_port        = htons(atoi("9999"));

  /* create a TCP lwipv6 socket */
  if((fd=lwip_msocket(stack,PF_INET,SOCK_STREAM,0))<0) {
    perror("Socket opening error");
    exit(-1);
  }
```

```
    /* connect it to the address specified as argv[1] port argv[2] */
    if (lwip_connect(fd,(struct sockaddr *)(&serv_addr),sizeof(serv_addr)) <
         0) {
      perror("Socket connecting error");
      exit(-1);
    }

    while(1) {
      fd_set rfds;
      int n;
      FD_ZERO(&rfds);
      FD_SET(STDIN_FILENO,&rfds);
      FD_SET(fd,&rfds);
      /* wait for input both from stdin and from the socket */
      lwip_select(fd+1,&rfds,NULL,NULL,NULL);
      /* copy data from the socket to stdout */
      if(FD_ISSET(fd,&rfds)) {
        if((n=lwip_read(fd,buf,BUFSIZE)) == 0)
          exit(0);
        write(STDOUT_FILENO,buf,n);
      }
      /* copy data from stdin to the socket */
      if(FD_ISSET(STDIN_FILENO,&rfds)) {
        if((n=read(STDIN_FILENO,buf,BUFSIZE)) == 0)
          exit(0);
        lwip_write(fd,buf,n);
      }
    }
    fill_buf_route(&buf_route, "192.168.250.1", RTM_DELROUTE, AF_INET);
    handle_vsnlib(buf_r, buf_route.header.nlmsg_len, (void*)nif, (void*)
        stack);

    fill_buf_addr(&buf_addr, "192.168.250.20", 64, RTM_DELADDR, AF_INET);
    handle_vsnlib(buf, buf_addr.header.nlmsg_len, (void*)nif, (void*)stack);

    fill_buf_link(&buf_link, 0);
    handle_vsnlib(&buf_link, buf_link.header.nlmsg_len, (void*)nif, (void*)
        stack);
}
```

test6_vsnlib.c

```
#include "minlibnetlink.h"

#define BUFSIZE 1024
char buf[BUFSIZE];

int main(int argc, char** argv){
```

```
struct sockaddr_in6 serv_addr;
int fd;

struct stack *stack;
struct netif *nif;
void* buf;
void* buf_r;
void* buf_l;
struct response* ret;
struct ip_addr addr_6;

struct addr_payload buf_addr;
struct route_payload buf_route;
struct link_payload buf_link;

buf=&buf_addr.header;
buf_r=&buf_route.header;

// call vsnlib
if(init_vsnlib(argv[1])==-1){
  perror("Cannot init vsnlib");
}

#ifdef LWIPV6DL
/* Run-time load the library (if requested) */
if ((handle=loadlwipv6dl()) == NULL) {
  perror("LWIP lib not loaded");
  exit(-1);
}
#endif
/* define a new stack */
if((stack=lwip_stack_new())==NULL){
  perror("Lwipstack not created");
  exit(-1);
}

/* add an interface */
if((nif=lwip_vdeif_add(stack,argv[2]))==NULL){
  perror("Interface not loaded");
  exit(-1);
}

fill_buf_link(&buf_link, 1);
handle_vsnlib(&buf_link, buf_link.header.nlmsg_len, (void*)nif, (void*)
    stack);
```

```
fill_buf_addr(&buf_addr, "2001:db8:0:f101::3", 64, RTM_NEWADDR, AF_INET6);
handle_vsnlib(buf, buf_addr.header.nlmsg_len, (void*)nif, (void*)stack);

fill_buf_route(&buf_route, "2001:db8:0:f101::2", RTM_NEWROUTE, AF_INET6);
handle_vsnlib(buf_r, buf_route.header.nlmsg_len, (void*)nif, (void*)stack)
    ;

memset((char *) &serv_addr,0,sizeof(serv_addr));
  serv_addr.sin6_family = PF_INET6;
  char str[INET6_ADDRSTRLEN];
  int reto = inet_pton(PF_INET6, "2001:db8:0:f101::1", serv_addr.sin6_addr
      .s6_addr);
  inet_ntop(PF_INET6, serv_addr.sin6_addr.s6_addr, str, INET6_ADDRSTRLEN);
  serv_addr.sin6_port = htons(atoi("9999"));

  /* create a TCP lwipv6 socket */
  if((fd=lwip_msocket(stack,PF_INET6,SOCK_STREAM,0))<0) {
    perror("Socket opening error");
    exit(-1);
  }
  /* connect it to the address specified as argv[1] port argv[2] */
  if (lwip_connect(fd,(struct sockaddr *)(&serv_addr),sizeof(serv_addr)) <
       0) {
    perror("Socket connecting error");
    exit(-1);
  }

  while(1) {
    fd_set rfds;
    int n;
    FD_ZERO(&rfds);
    FD_SET(STDIN_FILENO,&rfds);
    FD_SET(fd,&rfds);
    /* wait for input both from stdin and from the socket */
    lwip_select(fd+1,&rfds,NULL,NULL,NULL);
    /* copy data from the socket to stdout */
    if(FD_ISSET(fd,&rfds)) {
      if((n=lwip_read(fd,buf,BUFSIZE)) == 0)
        exit(0);
      write(STDOUT_FILENO,buf,n);
    }
    /* copy data from stdin to the socket */
    if(FD_ISSET(STDIN_FILENO,&rfds)) {
      if((n=read(STDIN_FILENO,buf,BUFSIZE)) == 0)
        exit(0);
      lwip_write(fd,buf,n);
    }
```

```
    }

    fill_buf_route(&buf_route, "2001:db8:0:f101::2", RTM_DELROUTE, AF_INET6)
        ;
    handle_vsnlib(buf_r, buf_route.header.nlmsg_len, (void*)nif, (void*)
        stack);

    fill_buf_addr(&buf_addr, "2001:db8:0:f101::3", 64, RTM_DELADDR, AF_INET6
        );
    handle_vsnlib(buf, buf_addr.header.nlmsg_len, (void*)nif, (void*)stack);

    fill_buf_link(&buf_link, 0);
    handle_vsnlib(&buf_link, buf_link.header.nlmsg_len, (void*)nif, (void*)
        stack);

}
```

Makefile

```
CC=cc
FLAG_CC=-ldl -lpthread -llwipv6 -lvsn -g
BUILDDIR=./build/
OBJ=$(BUILDDIR)minlibnetlink.o
FLAG_PREP=-c -o

all: test4 test6

test4: buildlib $(OBJ)
  $(CC) $(FLAG_PREP) $(BUILDDIR)test4.o test4_vsnlib.c
  $(CC) -o $(BUILDDIR)test4.exe $(BUILDDIR)test4.o $(OBJ) $(FLAG_CC)

test6: buildlib $(OBJ)
  $(CC) $(FLAG_PREP) $(BUILDDIR)test6.o test6_vsnlib.c
  $(CC) -o $(BUILDDIR)test6.exe $(BUILDDIR)test6.o $(OBJ) $(FLAG_CC)

buildlib:
  mkdir -p $(BUILDDIR)
  $(CC) $(FLAG_PREP) $(BUILDDIR)minlibnetlink.o minlibnetlink.c

clean:
  rm -rf build
```

Bibliografia

[1] Renzo Davoli. Internet of Threads. http://www.cs.unibo.it/~renzo/papers/2013.iciw.pdf, 2013.

[2] Renzo Davoli. Internet of Threads: Processes as Internet Nodes. http://www.cs.unibo.it/~renzo/papers/2014.IntTechIoTh.pdf, 2014.

[3] Renzo Davoli. Internet of Threads. http://www.cs.unibo.it/~renzo/papers/ConfGARR11_SelectedPapers_Davoli.pdf.

[4] Altran. picoTCP. https://github.com/tass-belgium/picotcp.

[5] Virtualsquare Team. vuos. https://github.com/virtualsquare/vuos.

[6] Virtualsquare Team. LWIPv6. http://wiki.v2.cs.unibo.it/wiki/index.php/LWIPV6.

[7] Virtualsquare Team. UMview. http://wiki.v2.cs.unibo.it/wiki/index.php/UMview.

[8] Virtualsquare Team. purelibc. http://wiki.virtualsquare.org/wiki/index.php/PureLibc.

[9] NetLink associazione LUGMan (Linux Users Group Mantova). http://lugman.org/images/4/43/NetLink.pdf, 2009.

[10] Thomas Graf. https://www.infradead.org/~RFC4291RFC4291tgr/libnl/.

[11] Thomas Graf. https://github.com/tgraf/libnl.

[12] Thomas Haller. https://github.com/thom311/libnl.

[13] LWIP. https://savannah.nongnu.org/projects/lwip/.

[14] Reti mesh picoTCP. http://www.picotcp.com/mesh-design-guide.

[15] Virtual Square. http://www.virtualsquare.org.

[16] Renzo Davoli. vdens. https://github.com/rd235/vdens.

[17] RFC 4291, https://tools.ietf.org/html/rfc4291.