POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi Magistrale

# Servizi di Rete in Linux e eBPF: Competizione o Cooperazione?

**Relatore**
prof. Fulvio Risso

**Candidato**
Francesco Paolo Messina

Anno Accademico 2018-2019

*Alla mia famiglia*

# Indice

**Elenco delle figure** 7

**Abstract** 8

**1 Introduzione** 9

**2 Background** 12
2.1 eBPF . . . 12
2.1.1 BPF . . . 12
2.1.2 eBPF . . . 13
2.2 Polycube . . . 18
2.2.1 Controllo Centralizzato . . . 18
2.2.2 Polycubed . . . 19
2.2.3 Polycubectl . . . 20
2.2.4 Cubo . . . 20
2.2.5 Servizio Polycube . . . 20
2.2.6 Interconnessione di Servizi . . . 22
2.2.7 Comunicazione tra Slow Path e Fast Path . . . 23

**3 Cooperazione tra Polycube e Linux** 25
3.1 Il Problema: La Visibilità . . . 25
3.1.1 Esempi pratici . . . 26
3.2 Re-Implementare Tutto . . . 27
3.2.1 Primo motivo per cooperare con Linux . . . 29
3.3 Command Line Interface . . . 29
3.3.1 Secondo motivo per cooperare con Linux . . . 30
3.4 Obiettivo della tesi . . . 30

**4 Shadow Service** 32
4.1 Visibilità delle Porte Polycube . . . 32
4.1.1 Problema . . . 32
4.1.2 Come Fare . . . 33

4.1.3 Interfacce virtuale . . . 33
4.1.4 Quando e Dove creare l'Interfaccia Virtuale . . . 35
4.1.5 Shadow Service o Shadow Port . . . 35
4.2 Traffico di Rete . . . 37
4.2.1 Problema . . . 37
4.2.2 Soluzione . . . 38
4.3 Due o più Shadow Service . . . 41
4.3.1 Problema . . . 41
4.3.2 Soluzione . . . 42
4.4 Implementazione di uno Shadow Service . . . 42
4.4.1 Proposta . . . 42
4.5 Esempi Pratici . . . 46
4.5.1 pcn_simplebridge . . . 47
4.5.2 pcn_router . . . 48
4.6 Vantaggi . . . 49
4.6.1 Resilienza . . . 50

**5 Netlink e CLI** 51
5.1 Netlink . . . 51
5.1.1 Perchè usare Netlink . . . 51
5.1.2 Socket Netlink . . . 53
5.1.3 Protocolli Netlink . . . 53
5.1.4 Indirizzamento . . . 55
5.1.5 Messaggi Netlink . . . 55
5.1.6 Esempi di utilizzo Netlink . . . 56
5.2 Netlink e Polycube . . . 57
5.2.1 Accedere alle Informazioni nel Kernel . . . 57
5.2.2 Due CLI un solo Comando . . . 58
5.2.3 Da Linux a Polycube . . . 58
5.2.4 Da Polycube a Linux . . . 60
5.3 Vantaggi . . . 61

**6 Validazione dei risultati** 62
6.1 Test Environment . . . 62
6.2 Test 1 . . . 63
6.2.1 Linux Router . . . 63
6.3 Test 2 . . . 63
6.3.1 Router Shadow con Quagga . . . 63
6.4 Test 3 . . . 66
6.4.1 Cattura del Traffico . . . 66
6.4.2 Modalità di Debug . . . 66
6.5 Prestazioni . . . 69

6.5.1 Valori numerici . . . . . 69

**7 Conclusioni** 73
7.1 Sviluppi Futuri . . . . . 73
7.1.1 Estensione CLI Polycube . . . . . 74
7.1.2 Collegamento tra Shadow Service e Interfacce Linux . . . . . 74

**Bibliografia** 76

# Elenco delle figure


2.1 Architettura di un Packet Filter . . . . . 14
2.2 Architettura eBPF . . . . . 15
2.3 Esempio di eBPF Service Chain . . . . . 18
2.4 Architettura Polycube . . . . . 19
2.5 Struttura di un Servizio Polycube . . . . . 21
2.6 Catena di servizi Polycube . . . . . 22
2.7 Encapsulator e Decapsulator . . . . . 23

4.1 Virtual Ethernet . . . . . 34
4.2 Codice YANG aggiunto ai Servizi Polycube . . . . . 35
4.3 Flow Chart di una Porta Shadow . . . . . 36
4.4 Porte Duplicate per Gestire il Traffico verso Linux . . . . . 39
4.5 Implementazione Pratica di un Router Shadow . . . . . 43
4.6 Visione Logica di un Router Shadow . . . . . 44
4.7 Struttura di uno Shadow Service . . . . . 45
4.8 Flow Chart del traffico da/verso il namespace del Servizio . . . . . 47

5.1 Esempi di Socket Netlink . . . . . 55
5.2 Struct nlmsghdr . . . . . 56

6.1 Test 1: Linux Router . . . . . 64
6.2 Test 2: Router Shadow con Quagga . . . . . 65
6.3 Test 3: Cattura del Traffico con Wireshark . . . . . 67
6.4 Ping tra due namespace collegati tramite router Shadow . . . . . 68
6.5 Confronto cattura del Traffico con Linux e con Shadow Service . . . 68
6.6 Latenza del comando Ping nei 3 casi di studio . . . . . 70
6.7 Bit-rate nei tre casi di studio . . . . . 71
6.8 Ping su una porta del Servizio/Interfaccia Linux . . . . . 72

7.1 Collegamento tra uno Shadow Service e una Interfaccia Linux . . . 75
7.2 Collegamento alternativo tra uno Shadow Service e una Interfaccia Linux . . . . . 75

# Sommario

Il mondo del Networking è sempre più orientato alla virtualizzazione. Funzioni di rete virtuali consentono una scalabilità e una flessibilità senza precedenti.

Polycube è un framework in grado di creare e distruggere funzioni di rete leggere e in modo veloce. Questa tecnologia si basa sulla macchina virtuale eBPF, che permette di iniettare codice verificato e compilato direttamente nel kernel Linux.

I programmi eBPF possono intercettare, accedere, modificare e reindirizzare i pacchetti prima che questi attraversino tutto lo stack di rete, garantendo ottime prestazioni e permettendo lo sviluppo di nuove funzioni di rete.

Gli stessi programmi eBPF però nel momento in cui vengono iniettati nel kernel, devono essere in grado di gestire tutta la complessità della nuova funzione di rete che intendono implementare.

Ad esempio, sviluppare un router eBPF richiede la gestione di una tabella di routing, gestire protocolli quali ARP, ICMP e soprattutto i protocolli di routing: OSPF, RIP, BGP ecc.

Linux sarebbe in grado di gestire quel traffico (direttamente o tramite l'utilizzo di tools) senza dover re-implementare tutti i protocolli e le funzionalità da zero.

Lo scopo di questo progetto di tesi è proprio quello di capire se e come, eBPF e Linux possano parlarsi, scambiarsi informazioni e come trarre vantaggi da questa nuova cooperazione.

# Capitolo 1

# Introduzione

La virtualizzazione è nata con la diffusione di macchine sempre più potenti in termini di CPU, RAM e storage, questo ha fatto riflettere molto sull'idea di collassare su un unico server quello che prima era distribuito su più server fisici. Controllare un'unica macchina contenente più unità logiche risulta essere molto più vantaggioso rispetto ad avere hardware dedicato. Si riducono i costi, si aumenta la flessibilità e si migliora lo sfruttamento delle risorse.

La stessa idea ha influenzato il mondo del Networking, dal punto di vista delle funzioni di rete dando vita al concetto di *Network Function Virtualization (NFV)*. La sigla NFV è una delle parole chiavi che riecheggiano nel mondo "Network" al giorno d'oggi, ed è sulla bocca di tutti i vendor tradizionali del networking ma anche dei nuovi players di estrazione più orientata all'IT.
Il concetto di NFV è esattamente la virtualizzazione applicata al mondo delle reti. Un router, uno switch, un firewall, un DNS, un NAT e le altre funzioni di rete non sono altro che delle macchine con un loro sistema operativo su cui viene eseguito del software che realizza le funzionalità di rete.
Allora la domanda sorge spontanea: Perchè continuare a comprare hardware dedicato (con tutti i vincoli che ci si porta dietro) quando si può sfruttare server general purpose (cosiddetti COTS - *Commercial On The Shelf*) ed installarci sopra tutte le virtual machine che servono?

Un framework capace di virtualizzare funzione di rete è *Polycube*, sviluppato come progetto di ricerca presso il Politecnico di Torino. Polycube consente la creazione e l'implementazione di funzioni di rete arbitrarie, leggere e veloci, che funzionano nel kernel di Linux e possono essere combinate tra di loro per costruire catene di servizi complesse. Polycube si basa sulla macchina virtuale *eBPF*.

eBPF è l'evoluzione del *Berkeley Packet Filter (BPF)*, che è il motore che alimenta strumenti come Wireshark e tcpdump. Con eBPF è possibile iniettare codice

nel kernel Linux in fase di runtime, codice che prima dell'esecuzione viene compilato e verificato per sicurezza. I programmi eBPF, che sono chiamati *cubi* nel contesto di Polycube, possono intercettare, accedere, modificare e reindirizzare i pacchetti prima che questi arrivino a Linux. eBPF consente ai cubi di comunicare tra loro e condividere aree di memoria, chiamate *mappe*.

L'utilizzo di funzioni di rete virtualizzate offre molti vantaggi in termini di flessibilità, scalabilità e non solo. Bisogna però considerare le tecnologie esistenti, e ricordarsi che anche se i servizi Polycube girano direttamente nel kernel (bypassando lo stack Linux) questo continua ad essere utile per molti altri servizi. In più potrebbe essere lo stesso Polycube ad avere bisogno di Linux nel momento in cui non si voglia re-implementare per ogni nuovo servizio tutte le funzionalità richieste. Si pensi al caso del *pcn_router*, il router sviluppato utilizzando Polycube, questo ha richiesto la gestione di una tabella di routing e di una tabella ARP, l'implementazione di protocolli come ARP, ICMP e cosa ancora più complicata di cui il servizio *pcn_router* ancora sente il bisogno in quanto non esiste ad oggi una soluzione, è la gestione dei protocolli di routing dinamico come ad esempio OSPF, RIP, BGP ecc. Il discorso non è limitato solo al router, altri esempi possono essere lo Spanning Tree sugli switch o più in generale tutte quelle funzioni di reti che richiedono funzionalità complesse per funzionare.

Linux potrebbe semplificare l'implementazione di queste funzionalità ma affinché ciò sia possibile è richiesta una cooperazione tra Polycube e il sistema operativo; in altre parole i due dovrebbero iniziare a parlarsi.
Coinvolgere Linux nell'elaborazione del traffico che è in grado di gestire direttamente o avvalendosi dell'utilizzo di tools senza dover re-implementare tutti i protocolli e le funzionalità da zero, sarebbe un vantaggio enorme per gli sviluppatori, senza contare il vantaggio degli utenti che si ritroverebbero a poter scegliere quale ambiente utilizzare per configurare e inserire comandi sui loro servizi Polycube.
Lo stack di rete Linux è li da decenni, funziona bene ed ha superato innumerevoli test. Questo per dire che Polycube non vuole avere la presunzione di sostituirlo o buttarlo, ma al contrario lo scopo è esattamente quello di integrare e far cooperare i due ambienti.

Creare sintonia tra il mondo Linux e il mondo Polycube è quindi l'idea che ha dato vita a questo lavoro di tesi.

Questo elaborato è strutturato come segue:

- **Capitolo 2:** introduce le tecnologie di base e il framework Polycube. Concetti essenziali per inquadrare il problema e capire tutto il lavoro che segue.

- **Capitolo 3:** espone in maniera esaustiva il problema affrontato in questa tesi, soffermandosi sul perchè è così sentito.
- **Capitolo 4:** descrive l'idea e l'implementazione di uno *Shadow Service*, vale a dire una modifica ai tradizionali servizi Polycube così da poter essere visti anche dal sistema operativo Linux.
- **Capitolo 5:** presenta *Linux Netlink* e come questo strumento si è rivelato essere la soluzione a molti dei problemi affrontati durante l'implementazione degli Shadow Service.
- **Capitolo 6:** fornisce una panoramica dei risultati ottenuti, basandosi su una serie di test eseguiti.
- **Capitolo 7:** espone le conclusioni e presenta alcune considerazioni utili per progetti futuri che avranno come base il lavoro svolto in questa tesi.

# Capitolo 2

# Background

In questo capitolo viene presentata la tecnologia *eBPF* e l'architettura del framework *Polycube*. Il capitolo è costituito da due sezioni apposite in modo da chiarirne concetti fondamentali e ambiti applicativi.
La comprensione di questi concetti è essenziale per poter affrontare il problema su cui si basa lo sviluppo di tutto il lavoro di tesi.

## 2.1 eBPF

### 2.1.1 BPF

BPF sta per *Berkeley Packet Filter* e può essere vista come una macchina virtuale eseguita nel kernel Linux in grado di filtrare i pacchetti.
Introdotto nel kernel Linux 2.1.75 nel 1997, inizialmente è stato usato come semplice filtro dallo strumento di cattura dei pacchetti *tcpdump*.
Essendo nel kernel di Linux, BPF non soffre di context switching o system call; questo significa che il processo di filtraggio può iniziare non appena il pacchetto raggiunge l'interfaccia, il che lo rende un vantaggio dal punto di vista delle prestazioni.

BPF funziona grazie a due componenti principali:

1. **Il Network Tap**, che preleva una copia dei pacchetti dall'interfaccia di rete e la invia all'applicazione in ascolto nello userspace.

2. **Il Filtro**, che decide se accettare o meno un pacchetto e quanti dati del pacchetto devono essere inviati all'applicazione in ascolto nello userspace.

Quando BPF è attaccato ad una interfaccia, il driver di rete prima chiama il codice BPF che processa il pacchetto e sulla base delle informazioni che ha, decide se inviare o meno il pacchetto all'applicazione in ascolto nello userspace.

Dopo aver fatto questo, il codice BPF ritorna il controllo al driver di rete che continua la sua esecuzione.
L'architettura generale di un packet filter, incluso BPF, è mostrata in Figura 2.1.

Alcune delle caratteristiche principali dei programmi BPF sono:

- **Runtime bytecode injection**: i programmi possono essere scritti, compilati e iniettati nel kernel in qualsiasi momento, modificando eventualmente il comportamento dei programmi BPF esistenti in base alle nuove condizioni.
- **Sicurezza**: i programmi BPF iniettati nel kernel sono sicuri, il che significa che un programma non può assumere il controllo della CPU o accedere ad aree di memoria non autorizzate.
  Il compilatore BPF utilizza un validatore per controllare e garantire la sicurezza del codice.

  Il validatore verifica che:

  – le istruzioni sono corrette;
  – non esistono loop infiniti nel codice;
  – il codice non acceda ad aree di memoria non valide;
  – la dimensione del programma non superi il limite consentito;
  – il numero di istruzioni sia finito: poiché i loop sono srotolati, il numero di istruzioni può aumentare notevolmente se il numero di iterazioni è elevato;
- **Portatilità**: Il bytecode BPF compilato è indipendente dall'hardware e può essere eseguito su qualsiasi architettura.
- **Efficienza**: il codice BPF quando è in esecuzione consuma pochissime risorse ed è molto vicino agli eventi del kernel. Il compilatore JIT ottimizza il codice immediatamente prima della compilazione.

### 2.1.2 eBPF

L'architettura originale BPF è nata con l'obiettivo di catturare e filtrare il traffico di rete. Un filtro di pacchetti è implementato come un programma basato su una macchina virtuale eseguita nel kernel.
Con il tempo però si è sentito il bisogno di aggiornare ed estendere l'architettura BPF esistente, con l'obiettivo di migliorarla.
Questo motivo portò Alexei Starovoitov nel 2013 a proporre eBPF *extended Berkeley Packet Filter* [1][2][3].
eBPF semplifica il mapping tra bytecode BPF e ISA (Instruction Set Architecture) e garantisce prestazioni migliori grazie ai cambiamenti introdotti.

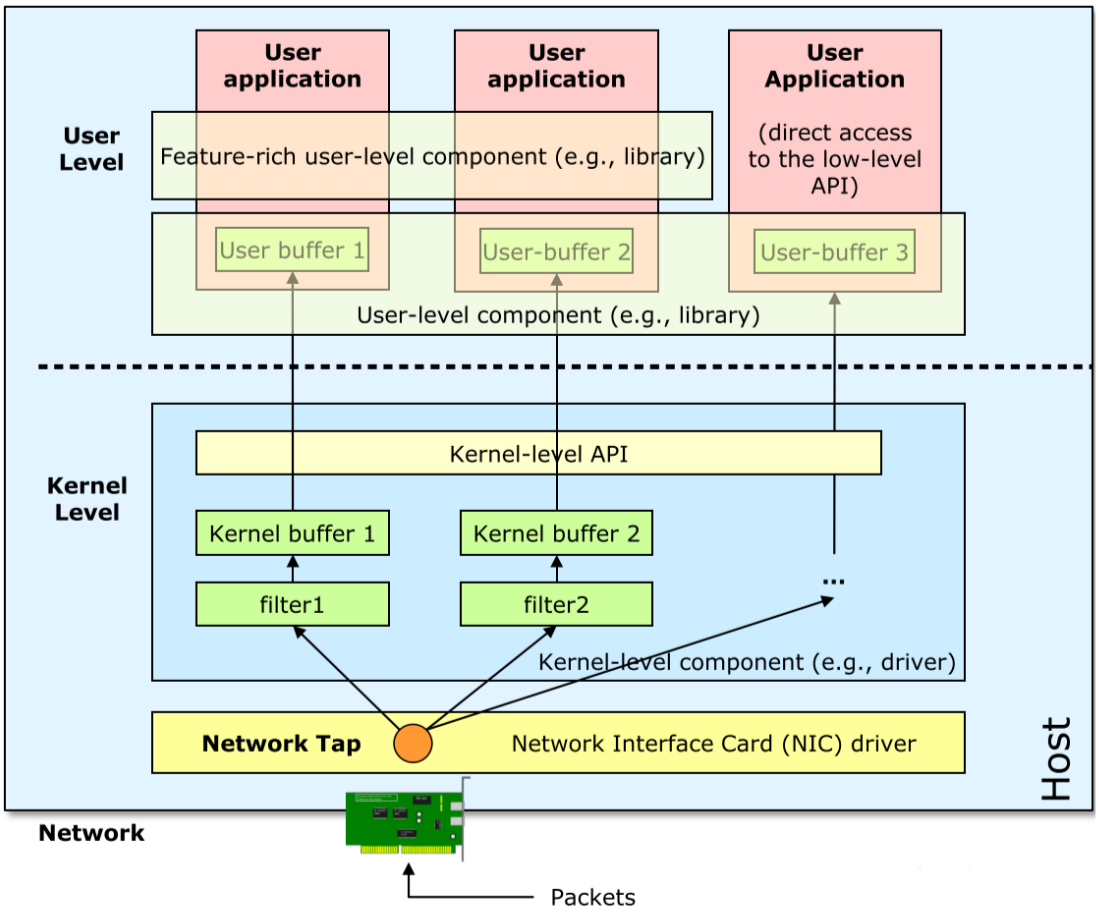


Figura 2.1: Architettura di un Packet Filter

eBPF ad oggi si è sostituito completamente al "classico" BPF.
Tra i cambiamenti più importanti introdotti da eBPF ci sono:

- Registri a 64 bit.
- 10 Registri (BPF usava solo 2 registri).
- Nuove istruzioni.

Negli ultimi anni, eBPF è stato regolarmente esteso con l'introduzione di nuove funzionalità come helpers e tail calls. Inoltre l'obiettivo originale del progetto BPF con il tempo è cambiato. Oggi eBPF è utilizzato anche per tracciare le attività del kernel, rilevare problemi di prestazioni e monitorare o modificare il traffico di rete. Una panoramica dell'architettura runtime di eBPF è riportata in Figura 2.2.

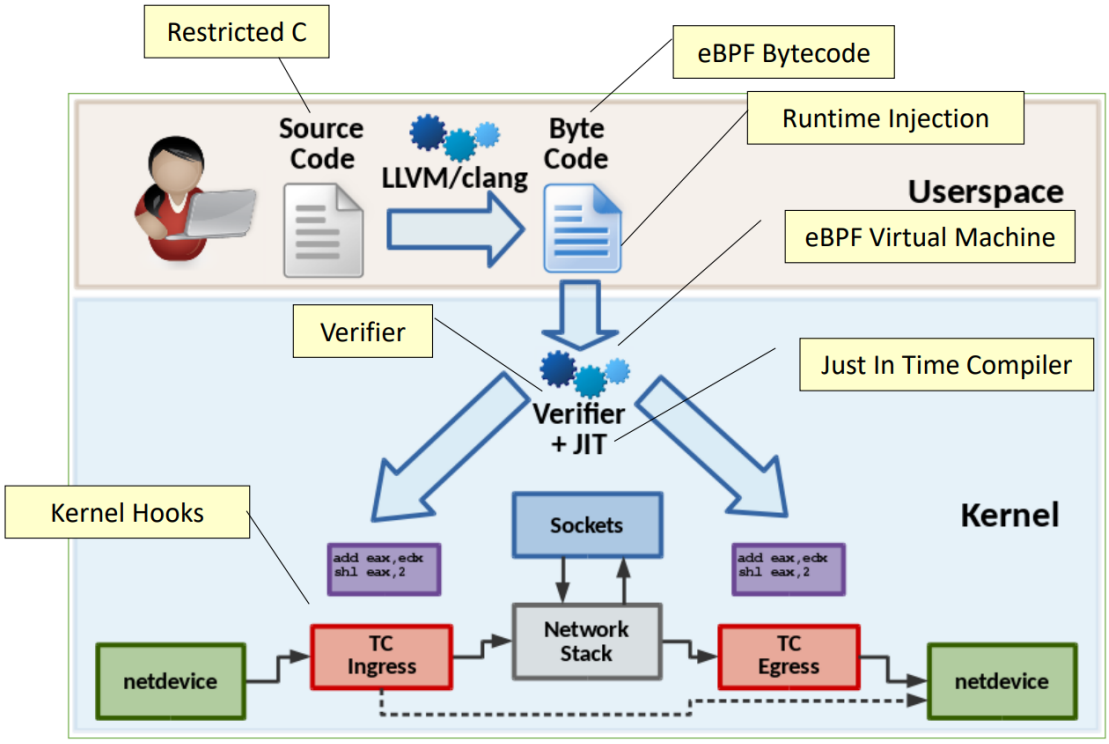


Figura 2.2: Architettura eBPF

### Architettura eBPF

Di seguito vengono spiegate alcune delle parti rilevanti dell'architettura eBPF.

### Programmazione C-based

Il codice eBPF è scritto utilizzando una versione limitata di C; il codice viene compilato e tradotto in assembler BPF attraverso un compilatore backend (esempio LLVM). A questo punto si ha una sorta di bytecode intermedio, e sarà compito del kernel verificare la sicurezza di questo codice (gli aspetti di sicurezza sono descritti tra le caratteristiche BPF) e rimappare queste istruzioni in codice eseguibile dal calcolatore, utilizzando un compilatore Just-In-Time (JIT).

### Mappe

Uno degli aspetti chiave introdotto da eBPF sono le *mappe* [4] [5]. Le mappe sono strutture dati chiave/valore che permettono di salvare e scambiare informazioni tra programmi eBPF diversi o tra programma eBPF e userspace. Questo permette di creare programmi eBPF stateful, in cui lo stato del programma è separato dal codice, o di scrivere applicazioni complesse dotate di un *fastpath*, nel kernel, e di

uno *slowpath* nello userspace, capaci di condividere e mantenere informazioni.
Le mappe non sono tecnicamente dei buffer, anche perchè se lo fossero richiederebbero di gestire problemi quali l'accesso concorrente; quindi le mappe non sono accessibili direttamente, ma si legge e si scrive sulle mappe attraverso opportune system call.

Ecco alcuni tipi di mappe disponibili nel kernel Linux:

- **BPF_MAP_TYPE_ARRAY**: i dati vengono memorizzati in modo sequenziale e sono accessibili attraverso un indice che varia da 0 alla dimensione massima -1. Questo tipo di mappa è ideale quando la chiave corrisponde alla posizione del valore nella mappa.

- **BPF_MAP_TYPE_HASH**: i dati sono memorizzati in una tabella hash. Questo tipo di mappa è molto efficiente per fare ricerche dirette: viene calcolato un hash della chiave e il valore viene usato come indice per accedere alla mappa.

- **BPF_MAP_TYPE_LRU_HASH**: è sempre una tabella hash, in cui però vengono salvati solo gli ultimi elementi o quelli più recenti. Questo tipo di mappa per esempio è utile a tenere traccia delle entries di una cache, forzandone l'aggiornamento.

- **BPF_MAP_TYPE_PROG_ARRAY**: questo tipo di mappa è utile per memorizzare i file descriptor dei programmi eBPF, usati per le *tail calls*.

Esistono altri tipi di mappe, usate in altri contesti. Per un riferimento completo, si rimanda alla lettura della documentazione del kernel [1].

### Helpers

Gli helpers sono set di funzioni precompilate e disponibili dentro il kernel Linux pronti per essere utilizzate. I programmi eBPF possono chiamare queste funzioni, che si trovano al di fuori dalla macchina virtuale eBPF.
Gli helper permettono di superare le restrizioni eBPF e di sfruttare alcune funzionalità avanzate del sistema operativo, delegando compiti complessi al sistema operativo stesso.
Uno svantaggio degli helpers è che devono essere disponibili nel kernel Linux, questo significa che il kernel deve essere ricompilato ogni volta che viene aggiunto un nuovo helper, il che non è molto veloce considerando i tempi di rilascio dei vari kernel.

[1] `https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html`

### Hook

I programmi eBPF possono essere chiamati al verificarsi di eventi kernel generici, e non solo quando si riceve un pacchetto. Il concetto base è quello di *hook*, cioè punto di intercettazione di un dato evento.
eBPF supporta diversi tipi di programmi che si differenziano a seconda di dove vengono collegati allo stack di rete Linux. Sebbene esistano circa diciotto tipi di programmi eBPF, solo due di questi sono usati per il Networking, ovvero i programmi *TC* e *XDP*, che sono quelli per l'appunto supportati anche da Polycube.

- **XDP**: i programmi XDP sono collegati alla prima fase del driver di rete e si attivano nel momento in cui viene ricevuto un pacchetto. Non esiste un punto software precedente in cui elaborare il pacchetto, quindi si intuisce facilmente che i programmi XDP sono quelli che garantiscono le prestazioni migliori. Tuttavia poiché l'elaborazione avviene così presto nello stack di rete, lo stack non ha ancora estratto i metadati del pacchetto.

- **TC**: i programmi TC invece vengono eseguiti in un punto successivo dello stack di rete, quindi non avranno le stesse prestazioni di quelli XDP ma potranno fare accesso a più metadati e funzionalità kernel.

### Service chain

Mentre in BPF non esisteva proprio il concetto di programmi cooperanti, ogni programma BPF riceveva una copia del pacchetto e la elaborava. Con eBPF si possono collegare più programmi per creare catene di servizi. Le catene di servizi possono essere create sfruttando collegamenti virtuali diretti tra due programmi eBPF o *tail call*.

Le tail call possono essere viste come chiamate di funzioni: un programma eBPF ad un certo punto nel codice decide di chiamare un secondo programma eBPF che inizierà la sua esecuzione.

Questo è utile anche a superare la limitazione sulla lunghezza dei programmi eBPF, era stato detto infatti che un programma eBPF ha un limite sul numero di istruzioni che può contenere; il limite può essere facilmente superato dividendo il codice in più programmi collegati tra di loro tramite tail call.

In Figura 2.3 è mostrato un esempio di service chain in cui due programmi eBPF sono collegati tra di loro attraverso una tail call e utilizzano una mappa per condividere informazioni.

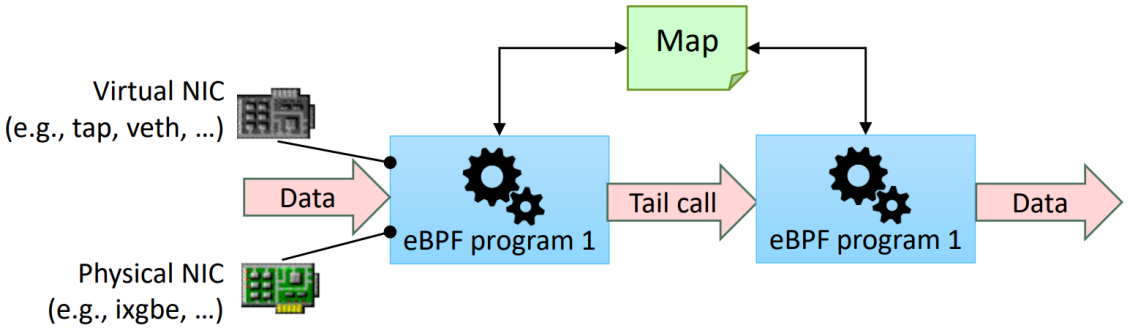


Figura 2.3: Esempio di eBPF Service Chain

## 2.2 Polycube

La creazione di funzioni di rete con eBPF può risultare agli occhi di uno sviluppatore complicata a causa di una mancanza di astrazioni utili e dalla presenza di problemi comuni o limitazioni note.

Questo ha portato ad immaginare una nuova architettura software, che prende il nome di *Polycube* [6], in cui i servizi di rete basati su eBPF possono essere gestiti in maniera più semplice e controllati da una logica centralizzata.

Gli utenti finali possono sfruttare i servizi Polycube già disponibili per creare, configurare e controllare complessi servizi di rete nei loro host. Gli sviluppatori di servizi invece possono creare nuovi servizi di rete sfruttando la potenza del framework Polycube, che si occupa della gestione della maggior parte della logica necessaria per il funzionamento del servizio.

Operazioni quali la gestione e la convalida dei dati e l'interfaccia REST utilizzata da Polycube, vengono generate automaticamente, mentre altre attività come le interazioni con il *data plane* in esecuzione nel kernel sono notevolmente semplificate. Ciò consente agli sviluppatori di concentrarsi sulla logica principale dei loro servizi, lasciando il resto a Polycube.

Una panoramica dell'architettura Polycube è riportata in Figura 2.4.

### 2.2.1 Controllo Centralizzato

In Polycube è stata fatta la scelta di adottare una logica centralizzata, questo perchè la centralizzazione semplifica la gestione e il controllo delle funzioni di rete.

Avere un unico punto di controllo facilita la gestione di più servizi e permette un controllo di livello superiore, fornendo uno sguardo alla topologia dei servizi e facilitando l'adozione di ottimizzazioni che non sarebbero possibili gestendo i servizi separatamente.

Il demone che utilizza Polycube per la gestione centralizzata si chiama *polycubed*, mentre l'interfaccia a riga di comando è *polycubectl*.

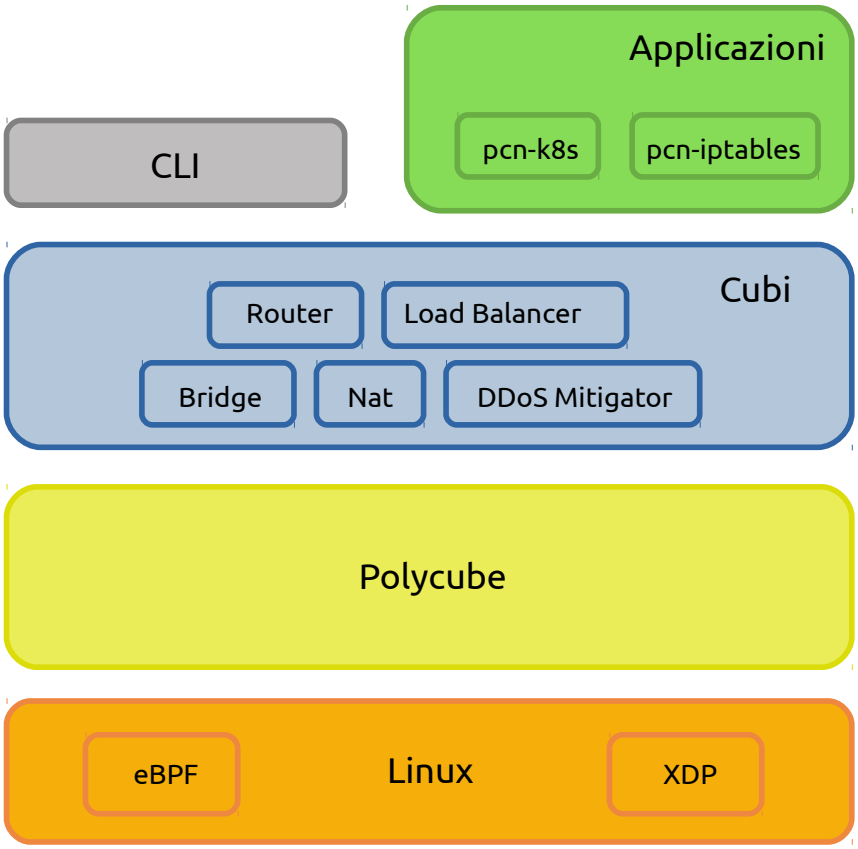


Figura 2.4: Architettura Polycube

### 2.2.2 Polycubed

Polycubed è un service-agnostic demone, che si trova nello spazio utente ed ha il compito di interagire con le diverse istanze di servizi, chiamati anche *cubi* nel contesto Polycube.
Si occupa dell'avvio, della configurazione e dell'arresto di tutte i cubi disponibili. Questo modulo agiste principalmente come Proxy: riceve una richiesta dall'interfaccia REST, la inoltra al servizio a cui è destinata, e ritorna indietro la risposta all'utente.

### 2.2.3 Polycubectl

L'interfaccia a riga di comando che si occupa dell'interazione con Polycubed e chiama Polycubectl. Polycubectl è una interfaccia unificata, semplice che permette all'utente di dare comandi a Polycube e di gestire i vari servizi; in più è dotata di un comando *help* che ne facilita notevolmente l'utilizzo e l'apprendimento.

### 2.2.4 Cubo

Ogni funzione di rete instanziata da Polycube prende il nome di cubo. I cubi sono simili a plug-ins che possono essere installati e lanciati in fase di esecuzione.
I cubi possono essere combinati per creare catene di servizi arbitrarie e fornire connettività di rete personalizzata a namespace, container, macchine virtuali e host fisici.

### 2.2.5 Servizio Polycube

Polycube è stato progettato per semplificare il concatenamento dei servizi: i cubi possono essere istanziati dinamicamente e collegati in modo trasparente tra loro tramite collegamenti virtuali, imitando le reti tradizionali in cui i middlebox dedicati sono collegati tra loro tramite fili fisici.

Ad esempio, diversi utenti mobili possono comunicare attraverso un bridge, che viene poi collegato a un router per fornire la connettività Internet (magari attraverso un NAT), mentre un firewall protegge l'intera infrastruttura.

Ogni servizio deve implementare una interfaccia specifica e deve essere riconosciuto e controllato da Polycubed.

Tutti i servizi implementano un *Control Plane* e un *Data Plane*, così come illustrato in Figura 2.5.

#### Data plane

Il Data Plane è caratterizzato da un *fast path*, ossia il codice eBPF che viene iniettato nel kernel, e da uno *slow path*, che gestisce i pacchetti che non possono essere processati nel kernel o che richiedono ulteriori informazioni e che potrebbero quindi rallentare l'elaborazione degli altri pacchetti.

- **Fast path**: è il codice eBPF eseguito per ogni pacchetto in arrivo, quindi è necessario mantenere il suo costo il più ridotto possibile. Le operazioni tipiche sono piuttosto veloci, e si concludono con l'inoltro del pacchetto o l'invio del pacchetto allo slow path per ulteriori elaborazioni.

- **Slow path**: a causa di alcune limitazioni note di eBPF le operazioni più complicate o che richiedono più tempo vengono eseguite nello slow path che si trova nello userspace.

Lo slow path una volta elaborato il pacchetto (ed aggiornate le informazioni in memoria utili al fast path) può decidere se iniettare il pacchetto nella coda d'ingresso del Servizio (il fast path elabora di nuovo il pacchetto) o direttamente nella coda di uscita.

### Control e Management Plane

Il Control e Management Plane si occupa di quei compiti che riguardano eventi complessi o processi che vanno gestiti ad un livello superiore rispetto al semplice cubo (es. Protocolli di Routing e Spanning Tree).

In più è il punto di ingresso per i players esterni (orchestratore di servizi, CLI degli utenti) che vorrebbero accedere alle risorse del servizio, modificarne i parametri o ricevere le notifiche inviate dal fast path o dallo slow path.

Polycube mette a disposizione un modulo di controllo e gestione che fornisce un set di *API REST* utilizzate per eseguire le tipiche operazioni *CRUD (Create-Read-Update-Delete)* sul servizio.

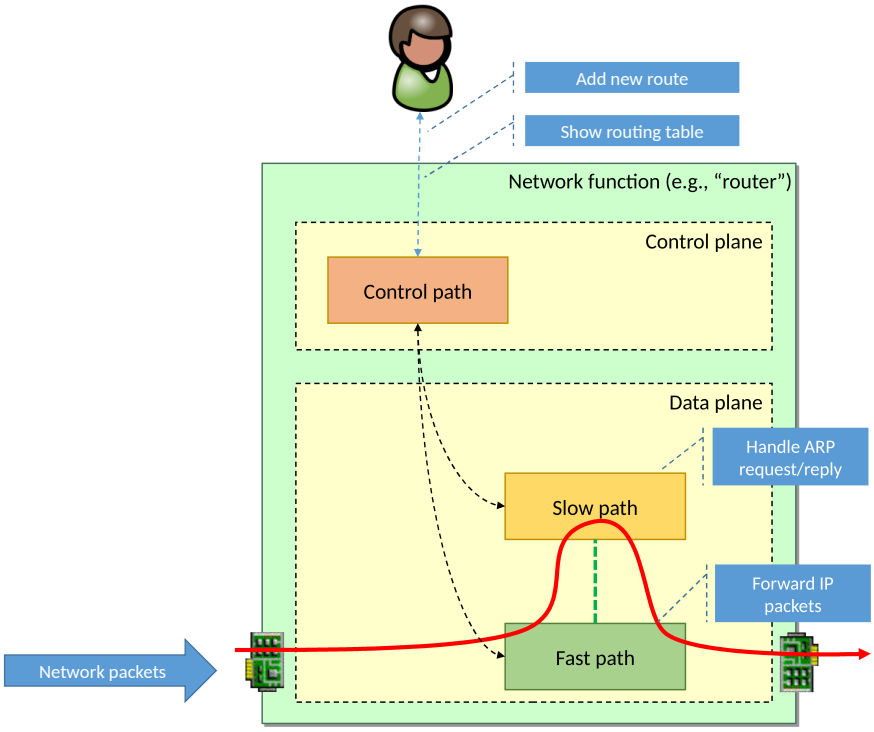


Figura 2.5: Struttura di un Servizio Polycube

### 2.2.6 Interconnessione di Servizi

Una catena di servizi Polycube è costituita da un insieme di cubi collegati tra di loro attraverso delle porte. Nel modello standard però, i programmi eBPF non hanno il concetto di porta su cui inviare o ricevere il traffico. Polycube fornisce questa possibilità, utilizzando una serie di componenti eBPF che sono nascosti agli sviluppatori, e che si possono osservare nella Figura 2.6.

Quando un pacchetto attraversa una catena di servizi in Polycube, trasporta con se alcuni metadati (es. porta virtuale di ingresso, indice del cubo) che devono essere scambiati tra i vari moduli. Queste informazioni sono salvate in strutture ausiliarie completamente nascoste agli sviluppatori.

Polycube quindi aumenta il codice eBPF della funzione di rete scritta dallo sviluppatore, con un set di programmi di supporto che vengono iniettati prima e dopo il servizio stesso.
Questi programmi di supporto vengono chiamati *pre-processore* e *post-processore* ed hanno lo scopo di semplificare l'interazione tra le varie componenti del framework.

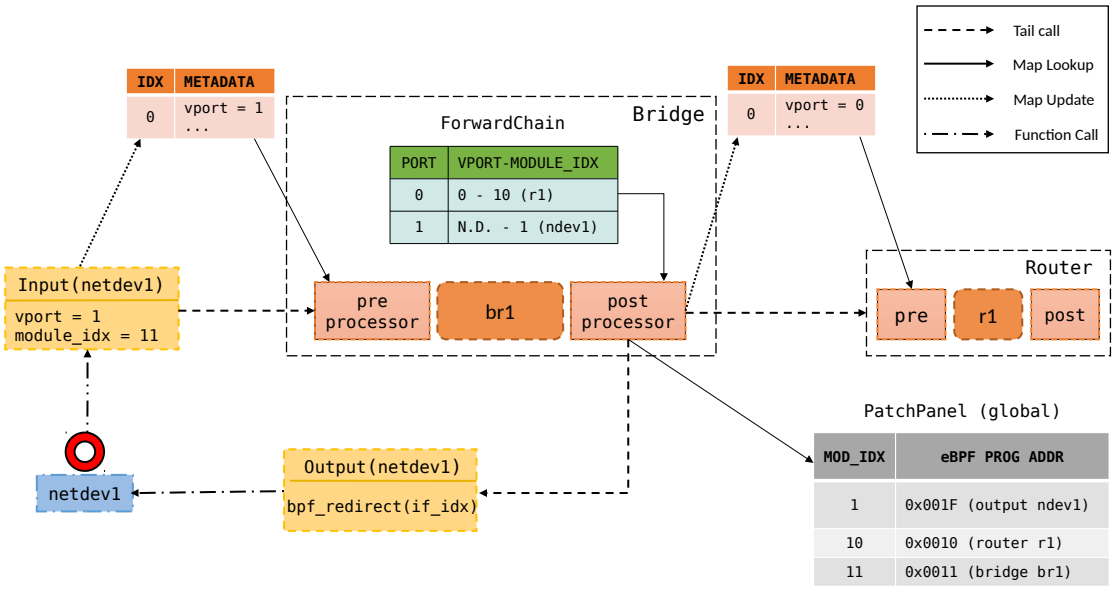


Figura 2.6: Catena di servizi Polycube

### 2.2.7 Comunicazione tra Slow Path e Fast Path

Come descritto precedentemente, il data plane è costituito dal fast path e dallo slow path. Ogni istanza di un servizio ha le sue copie private di fast path e slow path ed è compito di Polycube isolare le varie copie e consegnare i pacchetti che dal fast path devono raggiungere il corrispondente slow path e viceversa.

Per eseguire questo compito Polycube utilizza due programmi eBPF separati che prendono il nome di *Encapsulator* e *Decapsulator*.

In Figura 2.7 si può osservare uno schema logico di questi due componenti.

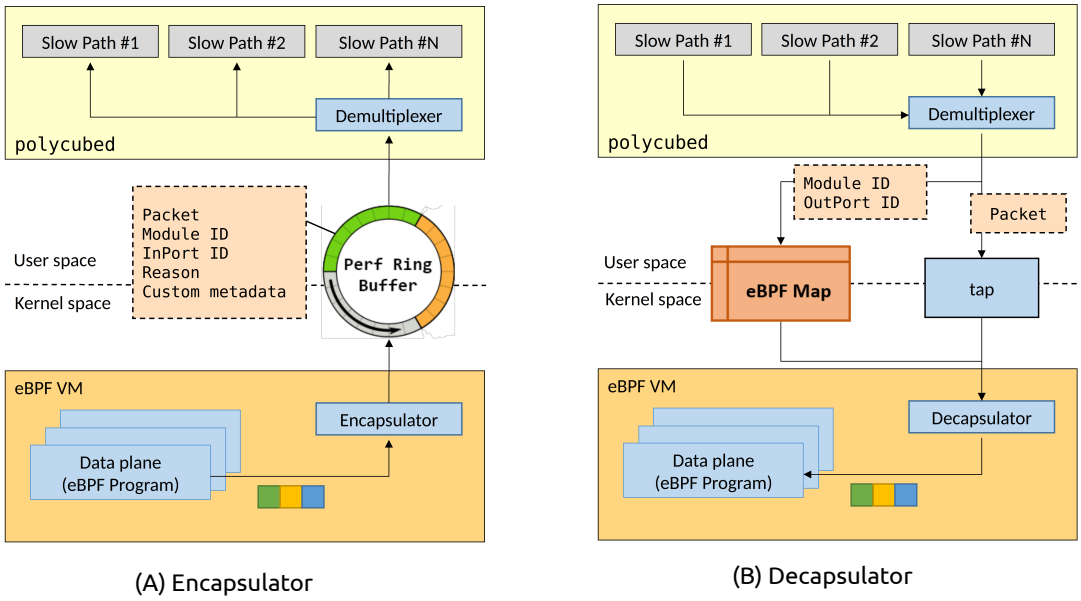


Figura 2.7: Encapsulator e Decapsulator

#### Encapsulator

L'encapsulator viene eseguito per inviare un pacchetto dal fast path eBPF allo slow path nello userspace. Quando il fast path decide di inviare un pacchetto allo slow path, il pacchetto viene passato al post-processore del servizio, specificando il motivo per il quale viene passato il pacchetto e i metadata aggiuntivi che dovranno essere passati allo slow path, utili per l'elaborazione.

Il post-processore copia queste informazioni in un'area di memoria condivisa (una mappa) e chiama l'encapsulator. L'encapsulator estrai i dati dalla memoria

condivisa, scrive il pacchetto con i suoi metadati in un buffer circolare, e aggiunge delle informazioni interne che serviranno a Polycubed per prelevare il pacchetto e consegnarlo allo slow path corretto.

Dall'altro lato del buffer c'è un thread, il *Demultiplexer* che ha il compito di estrarre i pacchetti dal buffer e consegnarli ai diversi slow path dei servizi.

### Decapsulator

Il decapsulator gestisce la comunicazione inversa, cioè il caso in cui lo slow path vuole iniettare un pacchetto nel fast path o farlo uscire da una porta del cubo. In questo caso il *Demultiplexer* cercherà l'indice del programma eBPF corrispondente al servizio a cui dovrà consegnare il pacchetto. Inserisce questa informazione in una mappa condivisa con il decapsulator e invia il pacchetto su un'interfaccia TAP creata appositamente quando viene avviato Polycubed. Diversamente dall'encapsulator, non viene utilizzato il buffer circolare, che è disponibile solo per le comunicazioni kernel-userspace e non viceversa.

Alla ricezione di un pacchetto sull'interfaccia TAP si attiva il programma eBPF decapsulator, che legge l'indice del programma eBPF scritto dal demultiplexer nella mappa e salta al programma corretto.

# Capitolo 3

# Cooperazione tra Polycube e Linux

In questo capitolo sono descritte alcune limitazioni eBPF ed i problemi che si riscontrano nel momento in cui si vuole utilizzare Polycube per implementare funzionalità di rete sempre più complicate.

Vengono poi spiegati i motivi per cui cooperare con Linux potrebbe risultare la soluzione migliore sia dal punto di vista dell'utente utilizzatore di Polycube che dal punto di vista dello sviluppatore di servizi che vuole utilizzare Polycube.

Infine viene delineato l'obiettivo della tesi in base alle considerazioni esposte.

## 3.1 Il Problema: La Visibilità

Uno dei vantaggi significativi di eBPF è quello di essere legato al kernel Linux e poter così fornire le stesse garanzie di sicurezza e l'accesso a primitive e funzionalità del kernel, senza dover re-implementare tutto da zero.

Una funzione di rete eBPF quindi potrebbe essere utile per personalizzare il comportamento dei pacchetti quando questi arrivano su un hook eBPF. Tuttavia quando il codice eBPF termina di elaborare il pacchetto dovrebbe ritornare il controllo al kernel per far si che il pacchetto possa continuare il suo percorso e raggiungere le applicazioni in esecuzione sull'host.

Polycube offre la possibilità di creare catene di servizi nel kernel collegando tra di loro più programmi eBPF. Questo può diventare un problema nel momento in cui il primo servizio della catena è collegato ad una interfaccia fisica (o virtuale) dell'host. Tutto il traffico ricevuto da quella interfaccia viene intercettato dal programma eBPF che lo elabora e lo passa agli altri servizi della catena, dove eventualmente sarà droppato, reindirizzato su un'altra interfaccia o inviato allo slow path per ulteriori elaborazioni.

Quindi, indipendentemente dal punto di hook usato (TC o XDP), il programma eBPF diventa il possessore dell'interfaccia. Sara il programma eBPF a ricevere tutto il traffico, e l'host non vedrà più nessun pacchetto.

Il sistema operativo quindi non ha più la stessa visibilità di prima sulla rete.

### 3.1.1 Esempi pratici

Alcuni esempi pratici possono aiutare ad inquadrare meglio il problema:

#### SSH

Si supponga di eseguire Polycube su un server remoto e di essere connessi a questo server tramite connessione SSH.

Quando si collega una porta Polycube all'interfaccia fisica, il sistema operativo del server perderà il controllo su quella interfaccia che passera al programma eBPF.

Se si considera il caso in cui l'interfaccia in questione è esattamente quella utilizzata per la connessione SSH si intuisce subito che non ci sarà più modo di comunicare con il server da remoto ne tanto meno di ricollegarsi al server finché Polycube non lascerà il controllo di quella interfaccia e quindi il traffico possa nuovamente passare nello stack di rete e raggiungere il sistema operativo.

#### ARP e ICMP

Molte funzioni di rete devono essere in grado di gestire il protocollo ARP e il protocollo ICMP; devono cioè essere in grado di riconoscere quando su una loro porta arriva traffico ARP o ICMP, capire se quel traffico è diretto a loro ed eventualmente generare un pacchetto ARP o ICMP di risposta.

Nel momento in cui si sceglie di utilizzare un programma eBPF per implementare una funzione di rete, tra le prime cose da fare bisogna gestire i protocolli ARP e ICMP. Il programma eBPF quando prenderà il controllo di una interfaccia fisica (o virtuale) sarà così in grado di rispondere a quel tipo di pacchetti.

Ma Linux è in grado di gestire il traffico ARP e ICMP, perchè non lasciarglielo fare? Perchè per l'appunto quel traffico non arriva più all'host e Linux non ha più modo di vederlo.

#### Protocolli di Routing

Il router Polycube (*pcn_router*) al momento è in grado di funzionare solo se vengono inserite le rotte a mano e cioè supporta solo il routing statico.

Se si volesse implementare uno qualsiasi dei protocolli di routing, sarebbe possibile farlo al livello di Control Plane, ma non sarebbe la strada più comoda, senza contare la mole di lavoro che richiederebbe implementare uno o più protocolli di routing da zero.

Perchè allora non sfruttare Linux che in maniere diretta o indiretta (attraverso l'utilizzo di tools, quali ad esempio Quagga [7] [8]) sarebbe in grado di occuparsi della complessità che sta dietro a tutto questo? Il motivo è sempre lo stesso, l'host non ha completa visibilità sul traffico e sulla rete costruita utilizzando Polycube, e non sarebbe quindi in grado di calcolare informazioni di routing corrette.

#### Spanning Tree

Analogo discorso fatto per il router sui protocolli di routing potrebbe essere ripetuto per il bridge e lo spanning tree.

In generale più si vuole implementare funzionalità complesse utilizzando Polycube più si capisce di avere bisogno della cooperazione con Linux per farlo; cooperare però non sarà possibile finché Linux non avrà completa visibilità della rete.

## 3.2 Re-Implementare Tutto

Per iniziare viene affrontato il problema dal punto di vista degli sviluppatori.

Pensare di risolvere il problema implementando tutto da zero potrebbe essere una soluzione, sicuramente non la migliore, ma pur sempre una soluzione.

Nessuno vieta agli sviluppatori di scrivere la loro versione di spanning tree, i loro protocolli di routing (OSPF, RIP, BGP ecc.), scrivere per ogni servizio la propria versione di ARP e ICMP e si potrebbe continuare elencando altri protocolli.

Perchè allora questa strada non è praticabile o è sconsigliata? Ecco un elenco di motivazioni a supporto di questa tesi:

1. **Versioni diverse dello stesso Protocollo**

   Polycube è un framework orientato sia agli utenti che vogliono utilizzarlo per sfruttare i vantaggi offerti da eBPF, ma anche agli sviluppatori che si vogliono scrivere il proprio servizio o la propria network function. Ogni sviluppatore quindi sarà libero di estendere le funzionalità e i servizi offerti dal framework.

   Chiedere però a due programmatori diversi di implementare lo stesso protocollo (quali ad esempio ARP o ICMP) porterà molto probabilmente a due soluzioni diverse dello stesso problema; in altre parole due versioni diverse dello stesso protocollo.

   Sicuramente il risultato finale dovrà essere quello, o quanto meno dovrà rispecchiare gli standard richiesti dagli RFC perchè le due versioni possano parlarsi, ma perchè allora non mettere gli sviluppatore in condizioni tali da concentrarsi solo sulle estensioni vere e proprie dei servizi che vogliono implementare senza costringerli a riscrivere ogni volta protocolli noti?

2. **Aggiornamenti**

   Rilasciare versioni nuove di un protocollo non è sicuramente qualcosa che avviene ogni giorno, ma nulla vieta che questo possa succedere. Possono benissimo essere inserite nuove features o essere risolti alcuni problemi che prima non esistevano.

   Sfruttare Linux per gestire protocolli noti vorrebbe dire non preoccuparsi di ciò, perchè quando sarà aggiornato il sistema operativo automaticamente anche la versione del protocollo sarà aggiornata e con essa tutti i servizi Polycube che cooperano con Linux.

   Al contrario aver implementato il protocollo più volte in diversi servizi costringe gli sviluppatori a rimettere mano al codice e dover risolvere loro stessi il fastidio degli aggiornamenti.

3. **Testing e Bug**

   Scrivere codice nuovo è un lavoro che richiedere tempo per l'implementazione ma non solo. Porta con se anche altre fasi, quali il testing del codice scritto, e l'analisi del codice per trovare e risolvere eventuali bug che ci si porta dietro, e non sempre i problemi escono fuori al primo tentativo.

   Considerando il sistema operativo Linux come un singolo blocco di codice, si può senza dubbio affermare che è uno dei programma più complessi e testati che è mai stato sviluppato dall'uomo.

   Avere quindi la presunzione di sostituirsi a Linux, ed implementare cose che lui sarebbe in grado di fare vorrebbe dire pensare di essere più bravi di un'intera comunità di sviluppatori che da decenni collaborano, testano e risolvono bug.

4. **Complessità**

   Ultimo punto, ma non per questo trascurabile, la complessità.
   Ogni servizio Polycube è dotato di control plane e data plane (che a sua volta si divide in slow path e fast path). Sicuramente l'implementazione delle funzionalità più complesse sarebbe fatta al livello più alto possibile, nel Control Plane e quindi nello userspace. Nessuno penserebbe mai di implementare un protocollo di Routing a livello kernel ne tanto meno di rallentare il fast path con elaborazioni complicate.
   Implementare queste funzionalità è comunque complesso a prescindere dal fatto di essere o meno nello userspace e di non soffrire delle limitazioni eBPF.

### 3.2.1 Primo motivo per cooperare con Linux

A questo punto sarà chiaro al lettore che la cooperazione con Linux è la strada migliore almeno per quanto riguarda il punto di vista degli sviluppatori di servizi che decidono di utilizzare Polycube.

Quindi il primo motivo per cooperare con Linux è quello di rendere la vita più semplice agli sviluppatori che vogliono evitare di riscrivere codice e funzionalità, delegando questa complessità a Linux ed evitando così di ritrovarsi in una delle situazioni descritte poco fa.

Va aggiunto anche il vantaggio che cooperare con Linux vorrebbe dire poter utilizzare tools di rete che prima non era possibile usare in Polycube quali ad esempio Wireshark per sniffare il traffico sulle interfacce o altri strumenti e tools utili per il debug ma non solo.

## 3.3 Command Line Interface

In questa Sezione viene presentato il secondo problema che la tesi si propone di risolvere e il secondo motivo per cui la cooperazione risulta essere vantaggiosa. Il punto di vista però questa volta cambia, non sarà più quello dello sviluppatore ma bensì quello dell'utente utilizzatore di Polycube.

Polycube è dotato della sua personale CLI (Polycubectl), che permette all'utente di controllare, modificare o eliminare le varie istanze di servizi e le relative interfacce. La semplicità offerta dalla CLI di Polycube potrebbe essere apprezzata o meno dai nuovi utenti, che inevitabilmente si ritroveranno nella situazione di dover imparare ad utilizzarla.

Linux però a sua volta ha già la sua CLI che permette di modificare le interfacce, settare i vari parametri (indirizzo IP, Netmask, indirizzo MAC ecc..). La CLI di Linux però non funziona su Polycube e ne tanto meno si può pensare di usare Polycubectl per gestire interfacce Linux.

Questo porta con se due problemi:

- Confusione degli utenti che si ritrovano due ambienti simili (Linux e Polycube) dove i parametri di rete si configurano in modo diverso, ed in più non si parlano tra di loro, obbligando l'utente che vuole tenere le informazioni allineate a dover impartire lo stesso comando due volte con due CLI diverse.

- Script di rete esistenti, utilizzati dai vari amministratori di sistema per configurare o gestire le proprie reti andrebbero riscritti per essere compatibili con Polycube. Questo potrebbe presentarsi come un ostacolo nel momento in cui si pone un amministratore di rete davanti alla scelta di adottare Polycube o farne a meno.

### 3.3.1 Secondo motivo per cooperare con Linux

Il secondo motivo che spinge verso la cooperazione è quindi quello di semplificare la vita dell'utente Polycube, che vuole poter utilizzare le funzioni di rete eBPF con il minimo sforzo.

Questo punto che potrebbe risultare meno importante è in realtà un fattore chiave per le nuove tecnologie e più in generale per qualunque cosa che debba scontrarsi con il mercato.

La storia ormai insegna che non è sempre la tecnologia migliore dal punto di vista tecnico o di prestazioni ad avere successo, ma al contrario è quella che più piace agli utenti e che sarà inevitabilmente adottata. Quindi rendere Polycube più intuitivo, facile e friendly possibile, potrebbe fare la differenza tra renderlo un framework di successo adottato da tutti o essere un ottimo framework ma conosciuto e apprezzato da poche persone.

## 3.4 Obiettivo della tesi

Sulla base delle considerazioni fatte fino a questo punto, si potrebbe pensare che Polycube e più nello specifico le funzioni di rete scritte in eBPF siano al momento in competizione con Linux; come se ci fosse una sorta di lotta tra Polycube e Linux per cercare di sostituire lo stack di rete.

Il lettore quindi potrebbe pensare che l'obiettivo di Polycube sia quello di sostituirsi allo stack di rete, imporsi come unico framework in grado di gestire la rete e che possa farlo senza cooperare con il resto del mondo, senza quindi l'aiuto di Linux o di altri programmi e tools disponibili.

Ma non è sicuramente questo l'intento di Polycube, anzi la cooperazione tra Polycube e Linux è sicuramente la strada migliore da intraprendere, per le diverse ragioni viste in questo capitolo.

Per concludere quindi lo scopo di questa tesi è esattamente quello di provare a far parlare tra di loro i due ambienti e trarre più benefici possibili dalla condivisione delle informazioni.

La cooperazione tra Linux e Polycube in altre parole porterebbe vantaggi da entrambi le parti:

- **Linux**: Lo stack di rete Linux si troverebbe solo più a gestire una parte del traffico di rete, la maggior parte dei pacchetti sarebbero elaborati e gestiti dal codice eBPF nel fast path e quindi nel kernel ottenendo prestazioni migliori del sistema e della macchina. Senza dimenticare che avere un framework ottimizzato come Polycube in grado di cooperare con Linux darebbe agli utenti del sistema operativo tutte la flessibilità e la dinamicità che vogliono nel gestire il traffico di rete della macchina.

- **Polycube**: Sarebbe in grado di sviluppare funzionalità complesse in modo più semplice delegando parte di queste al sistema operativo e avrebbe a disposizione tutti i tool offerti da Linux. Liberando l'utente finale e lo sviluppatore da problemi che ad oggi Polycube ancora si porta dietro.

# Capitolo 4

# Shadow Service

La prima parte di soluzione del progetto di tesi si basa sull'estendere i servizi già presenti in Polycube, ed in particolare ci soffermeremo su un nuovo concetto: gli *Shadow Service*.

L'obiettivo è quello di fare in modo che i servizi Polycube possano essere visti anche dal sistema operativo Linux, e non rimangano più confinati all'interno del framework stesso.

## 4.1 Visibilità delle Porte Polycube

In questa sezione viene analizzata una possibile soluzione al problema della visibilità delle porte create all'interno del framework. L'obiettivo delle scelte architetturali che verranno proposte è quello di estendere il concetto di *Porta* per renderlo più ampio e poterlo esportare al di fuori del solo contesto Polycube.

### 4.1.1 Problema

Si è parlato di come Polycube "inventi" il concetto di Porta che per sua natura non esiste nei programmi eBPF (si rimanda alla Sezione 2.2.6).

Questo si traduce in pratica nel dire che quella astrazione è limitata al solo contesto Polycube. Linux non ha minimamente idea del numero di porte che esistono in Polycube, ne tanto meno di come queste sono state configurate o collegate tra di loro.

L'idea di partenza allora è stata quella di trovare un modo che rendesse possibile esportare questa astrazione al di fuori di Polycube e rendere le porte visibili anche su Linux.

### 4.1.2 Come Fare

La soluzione pensata è stata quella di rendere una porta Polycube anche una interfaccia Linux. Per capire meglio si considerino i due casi possibili:

1. Esiste già una interfaccia (fisica o virtuale) in Linux e la si vuole anche come porta in Polycube.

2. Non esiste ancora una interfaccia in Linux che abbia il nome e i parametri della porta Polycube appena creata e la si vuole aggiungere sotto forma di interfaccia virtuale.

Il primo caso può sembrare relativamente semplice a parte dover risalire agli attributi dell'interfaccia esistente in Linux e utilizzarli per creare una nuova porta in Polycube. Il secondo caso invece ci pone davanti più domande, quali:

- Che tipo di interfaccia virtuale scegliere tra le possibilità interfacce offerte da Linux?

- Quando e dove creare questa nuova l'interfaccia?

Per rispondere a questi quesiti sono state provate e testate più soluzioni che con il tempo hanno portato cambiamenti nelle scelte implementative, con l'obiettivo di trovare la soluzione migliore. Prima di presentarla però proviamo a ripercorrere la strada che ci ha portato all'implementazione finale.

### 4.1.3 Interfacce virtuale

Linux è un sistema che fornisce una varietà di interfacce virtuali [9], solo conoscendole bene si può capire qual è la differenza tra le varie interfacce offerte, quando usarle e come crearle.

In questa tesi la scelta è ricaduta su due tipi di interfacce, le *interfacce TAP* e le *virtual Ethernet*.

#### Interfaccia TAP

*TUN* e *TAP* sono interfacce virtuali del kernel Linux [10].

- TUN (ovvero *network TUNnel*) simula un dispositivo di rete che funziona al livello 3, ovvero con i pacchetti IP.

- TAP (ovvero *network Tap*) simula un dispositivo di livello collegamento e funziona con i pacchetti di livello 2, ovvero frame Ethernet.

Le interfacce TUN/TAP sono interfacce software, nel senso che esistono solo nel kernel e a differenza delle normali interfacce di rete non hanno alcun componente

hardware fisico. Non esiste cioè un "filo" su cui collegarle. Vengono create da un programma in userspace, e tutto il traffico che arriva su queste interfacce viene reindirizzato dal kernel a quel programma.

Quando viene creata una interfaccia TUN/TAP si può decidere se farla:

- *Transitoria*: viene creata, utilizzata e distrutta dallo stesso programma, quando questo termina.

- *Permanente*: si crea attraverso una utility e i programmi userspace possono collegarsi se ne hanno necessità.

Una volta che l'interfaccia è stata creata, può essere utilizzata come qualsiasi altra interfaccia, il che significa poter assegnare indirizzi IP, analizzare il traffico che ci passa sopra, creare regole di firewall ecc.

### Virtual Ethernet

Le *virtual Ethernet* o meglio note come *veth* sono device Ethernet virtuali [11]. Si comportano come dei tunnel per il semplice fatto che vengono sempre create in coppia come mostrato in Figura 4.1. I pacchetti trasmessi da un lato della veth vengono immediatamente ricevuti dall'altro. Per questo motivo sono quelle che si prestano meglio per far comunicare dei namespace con il namespace principale o tra di loro.

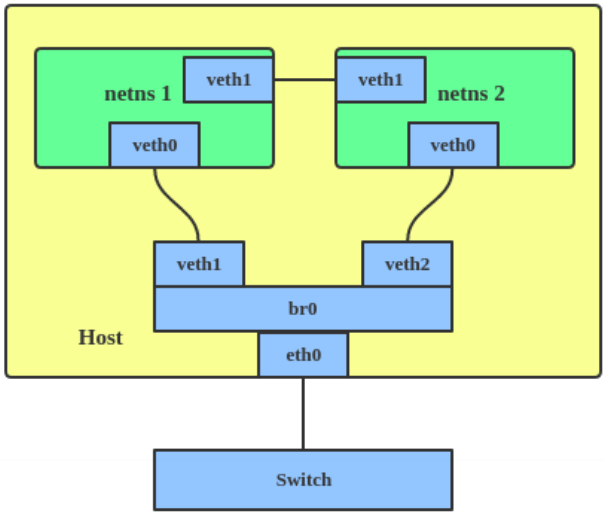


Figura 4.1: Virtual Ethernet

### 4.1.4 Quando e Dove creare l'Interfaccia Virtuale

Il punto in cui andava creata una interfaccia virtuale è stato più volte messo in discussione durante la stesura della tesi.

Le soluzioni pensare e testate sono state due, entrambe basate sull'attributo **Shadow**. L'attributo Shadow non è altro che un flag booleano che viene aggiunto al data model del servizio Polycube, modificando il codice *YANG* [12] del servizio.

In Figura 4.2 è mostrato il codice YANG che è stato aggiunto alla classe base del Servizio.

```
leaf shadow {
    type boolean;
    default "false";
    description "Defines if the service has
       the interfaces visible in Linux";
}
```

Figura 4.2: Codice YANG aggiunto ai Servizi Polycube

Le due soluzioni differiscono tra di loro sulla base del livello in cui viene aggiunto l'attributo *Shadow*:

1. *A livello di Porta*: Quando viene creata una porta si specifica se questa deve essere presente in Linux oppure se è solo una porta Polycube.
   Se Shadow vale *true* si controlla se esiste già una interfaccia con quel nome, in caso contrario verrà creata una interfaccia virtuale, che cesserà di esistere nel momento in cui l'istanza del servizio verrà distrutta.
   In questo caso si parla di **Shadow Port**. Il flow chart in Figura 4.3 mostra la logica a livello di porta.

2. *A livello di Cubo*: Quando viene creato un cubo si specifica se questo deve essere Shadow. Un cubo è Shadow quando tutte le sue porte sono porte di tipo Shadow. In questo caso si parla di **Shadow Service**.

### 4.1.5 Shadow Service o Shadow Port

Lavorare a livello di porta, e quindi avere più flessibilità è stata l'idea di partenza. L'utente in pratica era libero di decidere quando creava una porta se farla *Shadow* oppure no. Questa soluzione rende possibili tre casi distinti:

1. Tutte le porte create dall'utente sono Shadow. Si ricade nel caso di *Shadow Service*.

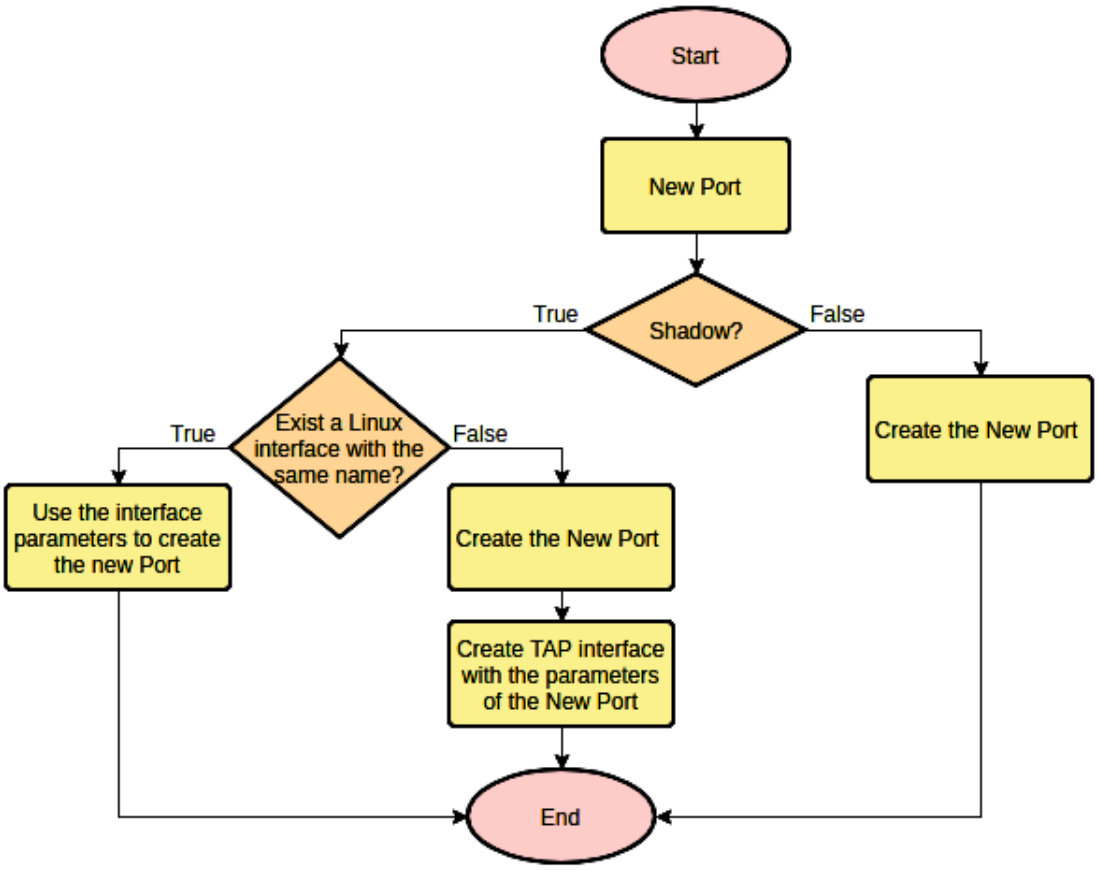


Figura 4.3: Flow Chart di una Porta Shadow

2. Nessuna porta creata dall'utente è Shadow. Si ricade nel caso di servizio Polycube tradizionale.

3. L'utente crea sia porte Shadow che porte non Shadow. In questo caso esistono contemporaneamente tutte e due i tipi di porta.

Il terzo e ultimo caso ha fatto riflettere sulla bontà di questa soluzione, perchè:

- Ha senso avere un servizio in cui sono presenti sia porte Shadow che porte non Shadow?

- Avere porte non Shadow implica nascondere l'esistenza di queste porte a Linux. Avrebbe ancora senso a questo punto interrogare Linux su decisioni che verrebbero prese sulla base di informazioni incomplete?

- Avere porte non Shadow obbliga lo sviluppatore a doverle gestire, e quindi doversi implementare protocolli quali ad esempio ARP e ICMP. Ma evitare

di re-implementare questi protocolli non era esattamente uno dei vantaggi derivanti dal cooperare con Linux?

Tutte queste domande sono state il motivo per cui si è scelto di andare verso la direzione degli *Shadow Service* e cioè ridurre i possibili casi a due.

1. Servizio Shadow → Tutte le porte sono Shadow.
2. Servizio non Shadow → Nessuna porta è Shadow.

## 4.2 Traffico di Rete

In questa sezione viene analizzata una possibile soluzione al problema del *traffico di rete*. Traffico che, nel caso di Shadow Service, sotto particolari condizioni deve arrivare a Linux.

### 4.2.1 Problema

Avere uno *Shadow Service* garantisce che tutte le porte create dal servizio saranno "duplicate" (se non già presenti) in Linux.

Le porte Polycube del servizio e le interfacce Linux avranno gli stessi parametri (indirizzo IP, Netmask, indirizzo MAC). Ma avere due porte con gli stessi parametri in due ambienti diversi non è garanzia del fatto che entrambe le porte vedano il traffico di rete.

Si è a conoscenza dalla Sezione 3.1, che i programmi eBPF eseguiti nel kernel elaborano il traffico di rete prima che questo attraversi lo *stack di rete* e arrivi a Linux. Solo in base alla decisione presa dal codice eBPF, Linux potrebbe vedere quel traffico o non vederlo mai.

Tutti i programmi eBPF che implementano i servizi Polycube, sulla base della logica che sta dietro al servizio, arrivano alla fine ad eseguire una di queste azioni:

- *RX_REDIRECT*: il pacchetto è rediretto su un'altra porta.
- *RX_DROP*: il pacchetto è droppato.
- *RX_CONTROLLER*: il pacchetto è inviato al controller.
- *RX_OK*: il pacchetto è lasciato libero di risalire lo Stack e arrivare a Linux.

Si capisce bene che solo nell'ultimo caso si ha la garanzia che il pacchetto arrivi a Linux. Nel primo caso, quando il pacchetto viene rediretto su un'altra porta si può ricadere in due situazioni distinte:

1. La porta su cui viene rediretto il pacchetto è collegata ad un'interfaccia Linux. Quel traffico arriva a Linux.

2. La porta su cui viene rediretto il pacchetto è collegata ad un'altra porta Polycube attraverso una *tail call*. Quel traffico non arriva a Linux.

### 4.2.2 Soluzione

Considerati i due comportamenti diversi della *Redirect* a seconda del collegamento presente sulla porta, è bene esaminare i casi separatamente.

#### Collegamento tra Porta e Interfaccia Linux

Collegare una porta ad un'interfaccia Linux è un'azione possibili su qualsiasi tipo di servizio, sia questo un servizio Shadow o meno.

Nel caso di Servizi Shadow però ha senso collegare una porta solo alla sua interfaccia Linux "gemella"; collegare una porta ad un'altra interfaccia che non sia quella che abbiamo chiamato gemella, sarebbe come prendere un filo e collegarlo su due porte dello stesso device (sia questo un router, un bridge o altro), non avrebbe senso.

Fatta questa premessa, nel momento in cui viene collegata una porta Polycube alla corrispondete interfaccia gemella Linux è come se le due interfacce diventassero un unico oggetto. Entrambe hanno gli stessi parametri, quindi eventuali risposte a protocolli ARP o ICMP e eventuali pacchetti IP possono essere generati indifferentemente da una o dall'altra, le informazioni trasportate sarebbero in ogni caso identiche e quindi giuste.

#### Collegamento tra due Porte Polycube

Discorso ben diverso si fa quando si ha un collegamento tra due porte Polycube. Queste anche se continuano ad avere sempre le corrispettive interfacce Linux (con lo stesso nome e gli stessi parametri) sono collegate attraverso una *tail call*, che abbiamo visto essere una sorta di chiamata a funzione. Le interfacce Linux quindi rimangono scollegate e non si vedono recapitare nessun pacchetto.

Anche in questo caso sono state implementate due soluzioni diverse per provare a risolvere questo problema.

1. **Duplicare le Porte**
   La prima soluzione è stata quella di duplicare ogni singola porta creata su un servizio Shadow. Secondo la seguente logica:

   (a) Quando si crea una porta P1 viene aggiunta una porta (nascosta all'utente) che chiameremo P1'.

(b) Se la porta P1 viene collegata alla sua interfaccia Linux, la porta P1' rimane inutilizzata.

(c) Se la porta P1 viene collegata ad una seconda porta P2, la porta P1' viene collegata automaticamente alla corrispondente interfaccia Linux di P1.

(d) Quando viene eliminata la porta P1, viene eliminata anche la porta P1'.

Con questa architettura si è in grado di gestire il traffico delle *tail call* che per ovvie ragioni deve essere recapitato a Linux grazie alla presenza della porta duplicata, che garantisce una via di comunicazione con il sistema operativo tutte le volte che serve. Sulla porta duplicata quindi può passare sia il traffico che dall'esterno deve essere recapitato a Linux, sia quello che da Linux deve uscire verso l'esterno.

La Figura 4.4 aiuta a capire meglio il concetto, in quanto chiaramente in entrambi i casi si riesce a vedere come esiste sempre una strada che da Polycube porta verso Linux e viceversa.

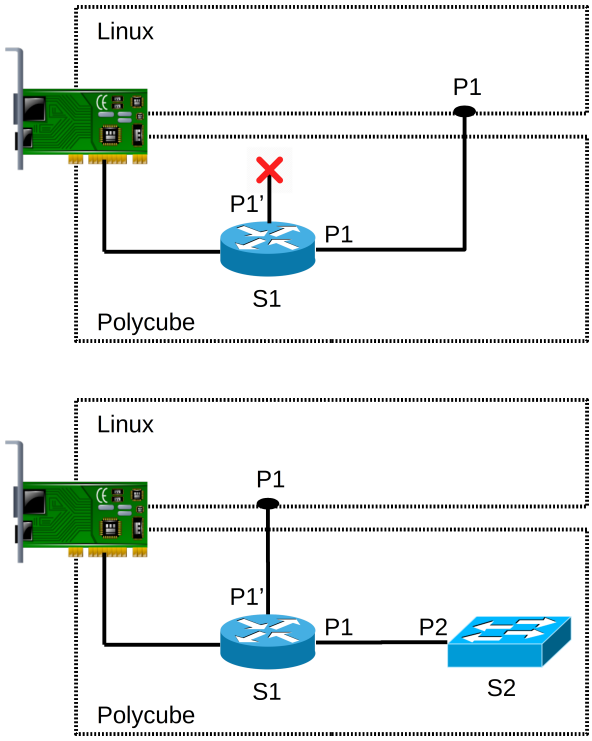


Figura 4.4: Porte Duplicate per Gestire il Traffico verso Linux

In questa situazione vale la pena far notare che le porte Polycube create su qualsiasi servizio sono identificate da un index, un intero crescente a partire

da zero. Considerando il fatto che in questa soluzione le porte vengono create a coppia, è facile distinguere le porte realmente usate da Polycube e visibili all'utente (tutte quelle con index pari) da quelle nascoste e dirette a Linux (tutte quelle con index dispari).
Passare quindi dalla porta *Px* alla *Px'* corrispondente o viceversa è una semplice operazione da non richiedere nessun tipo di ricerca. Basta fare un incremento o decremento di uno. È bene tenere a mente questa osservazione perchè si rivelerà utile in seguito quando verrà presentata la soluzione finale.

2. **Buffer Circolare**
La soluzione alternativa può sembrare più efficiente per certi versi, in quanto non raddoppia le porte effettivamente usate ma solleva altri tipi di problemi.
L'idea è stata quella di utilizzare un concetto già presente in Polycube e prendere ispirazione da quel caso d'uso che risolve un problema simile.
Il concetto in questione è la comunicazione tra Fast Path e Slow Path (si rimanda alla Sezione 2.2.7).
Così come fanno Fast Path e Slow Path a comunicare, attraverso l'utilizzo di un Buffer Circolare, anche Polycube e Linux possono farlo sfruttando la stessa idea. La logica è la seguente:

   (a) Ogni volta che viene creato un cubo Shadow, questo crea a sua volta un proprio Buffer Circolare.

   (b) Il traffico in ingresso e in uscita dal cubo che deve arrivare a Linux, viene copiato nel Buffer Circolare, insieme ai metadati necessari ad identificare la porta.

   (c) Un thread in ascolto sul Buffer, preleva i pacchetti e li invia alle rispettive interfacce TAP presenti in Linux.

   Questa soluzione che potrebbe sembrare interessante e sicuramente ottima per molti casi d'uso, ha dimostrato avere una limitazione.
   Il problema si presenta nel momento in cui il traffico da Linux deve uscire fuori e quindi essere re-inviato al servizio. Ricordiamo infatti che il buffer circolare andava bene come soluzione per far salire il traffico dal programma eBPF allo userspace ma non andava bene come soluzione per il caso inverso. Quando un pacchetto dallo userspace deve essere iniettato nel servizio si utilizzano infatti delle interfacce TAP appositamente create da Polycube al bootstrap.
   Il traffico che arriva dalle TAP non è però facilmente distinguibile dal traffico proveniente da qualsiasi altra porta del servizio. Non si ha modo di capire in maniera semplice e veloce se quel traffico deve essere processato o se deve semplicemente essere rediretto verso l'uscita.
   Non esiste in questo caso un'idea semplice ed efficiente come quella di differenziare il traffico sulla base dell'index pari o dispari della porta di provenienza del pacchetto.

Analizzando pro e contro delle due soluzioni e tenendo conto dell'impatto sulle prestazioni, si è scelto alla fine di implementare la prima soluzione presentata, e cioè duplicare le porte. La scelta tiene conto del maggior consumo di risorse (ogni porta viene duplicata) ma confrontata con il risparmio in termini di complessita computazionale e misurando le prestazioni si è rivelata essere la soluzione migliore.

## 4.3 Due o più Shadow Service

Questa sezione affronta il problema della coesistenza di più Shadow Service dal punto di vista di Linux. Dal punto di vista Polycube la garanzia dell'isolamento dei servizi viene data dal framework, ma cosa succede lato Linux?

### 4.3.1 Problema

Avere più *Shadow Service* in Polycube non è sicuramente un problema. Polycube stesso per come è costruito garantisce l'isolamento e permette all'utente di creare porte con lo stesso nome su servizi diversi.
Questo però non succede in Linux ed è per questo che bisogna prendere le dovute precauzioni. Per capire meglio vengono esaminati due esempi pratici che aiutano ad inquadrare il problema:

1. **Porte con lo stesso Nome**
   Ipotizziamo di avere due Shadow Service chiamati *S1* e *S2* e di voler creare una porta e chiamarla *P1* sia sul servizio S1 che sul servizio S2. Le due porte devono essere logicamente distinte essendo create su due servizi diversi.
   Per semplicità consideriamo il caso in cui P1 non è ancora presente come interfaccia Linux, ma nulla sarebbe cambiato a considerare il caso duale.
   Vediamo cosa succede su Linux nel momento in cui non viene presa nessuna contromisura:

   (a) Viene creata la porta P1 su S1 e contemporaneamente viene creata l'interfaccia virtuale P1 in Linux. Le due interfacce sono allineate.

   (b) S2 prova a creare P1 e si accorge che in Linux esiste già una interfaccia virtuale con quel nome e quindi prende i parametri da P1.

   (c) S1 e S2 si ritrovano ad avere due porte con gli stessi parametri e che dal punto di vista Linux sono viste come se fossero una sola porta.

   Questo non ha alcun senso considerando che l'obiettivo era solamente quello di avere due porte diverse su due servizi diversi che avessero solamente lo stesso nome. Pensare che un servizio debba controllare tutti i nomi delle porte degli altri servizi prima di crearne una nuova non è sicuramente la soluzione migliore.

2. **Rotte sbagliate**
   In questo secondo caso immaginiamo di trovarci nella situazione particolare in cui si hanno due Shadow Router chiamati *R1* e *R2*, di aver creato su R1 le porte P1 e P2 e su R2 le porte P3 e P4.
   Dal punto di vista di Linux le interfacce P1, P2, P3 e P4 sono viste come se fossero tutte sulla stessa macchina. Questo risulta essere un problema nel momento in cui bisogna calcolare le rotte da inserire nella tabella di Routing. Linux si ritroverebbe a vedere una topologia della rete che non è quella reale e di conseguenza sarebbero calcolate delle rotte sulla base di informazioni sbagliate.

### 4.3.2 Soluzione

È stato chiaro fin da subito che la strada da prendere era quella di "raggruppare" le interfacce Linux sulla base dello Shadow Service di appartenenza e di conseguenza isolarle dalle altre interfacce.

La soluzione a questo problema si ha utilizzando i *network namespaces* [13] [14].

Un namespace può essere visto come un modo per isolare alcune risorse del sistema in modo da renderle visibili solo ai processi che si trovano all'interno del namespace. Ogni namespace opera indipendentemente dagli altri e garantisce ai processi al suo interno una propria visione delle risorse del sistema.
Il kernel Linux fornisce 6 tipi di namespace: *pid, net, mnt, uts, ipc* e *user*. Quello di nostro interesse è il network namespace che dal punto di vista logico è una copia dello stack di rete Linux vista da tutti i processi che si trovano all'interno del namespace. Un network namespace avrà quindi le sue rotte, le sue regole di firewall, le sue tabelle di forwarding e le sue interfacce di rete.

## 4.4 Implementazione di uno Shadow Service

Dopo aver analizzato le diverse strade e aver descritto problemi e vantaggi incontrati nei singoli casi, in questa sezione viene presentata la soluzione finale che si è scelto di implementare e che fosse in grado di tenere conto dei problemi fin qui incontrati.

### 4.4.1 Proposta

La proposta di soluzione in grado di implementare uno Shadow Service che è stata scelta tiene conto dei concetti base su cui si basa Polycube, quali il concetto di *Porta*. L'obiettivo non è stato quello di sconvolgere l'architettura e i concetti di Polycube ma piuttosto cercare di adottarli per cooperare con Linux, riutilizzarli quindi per ottenere una soluzione interessante e creativa allo stesso tempo.

### Implementazione pratica

I seguenti punti spiegano passo passo come avviene la creazione di uno Shadow Service dal punto di vista pratico:

1. Quando viene dato a Polycube il comando per istanziare un nuovo Shadow Service, viene creato il cubo del servizio e subito dopo viene creato un network namespace in Linux con lo stesso nome del servizio appena creato.

2. Il namespace e il cubo non avranno nessuna interfaccia finché l'utente non decide di creare una porta. Quando viene lanciato il comando per creare una porta vengono eseguiti i seguenti passi:

   (a) Viene creata una porta sul cubo e vengono assegnati alla porta gli eventuali parametri passati dall'utente. La porta creata avrà index pari.

   (b) Viene creata una seconda porta sul cubo a cui non verrà assegnato nessun parametro. Questa porta sarà nascosta all'utente e avrà index dispari.

   (c) Viene creata una virtual Ethernet (veth) in Linux.

   (d) Una estremità della veth viene spostata nel namespace e avrà il nome e i parametri della porta visibile sul cubo.

   (e) L'altra estremità della veth non avrà nessun parametro, sarà accesa e collegata alla porta nascosta del cubo.

In Figura 4.5 si può vedere un esempio di router Shadow costruito secondo la procedura appena descritta.

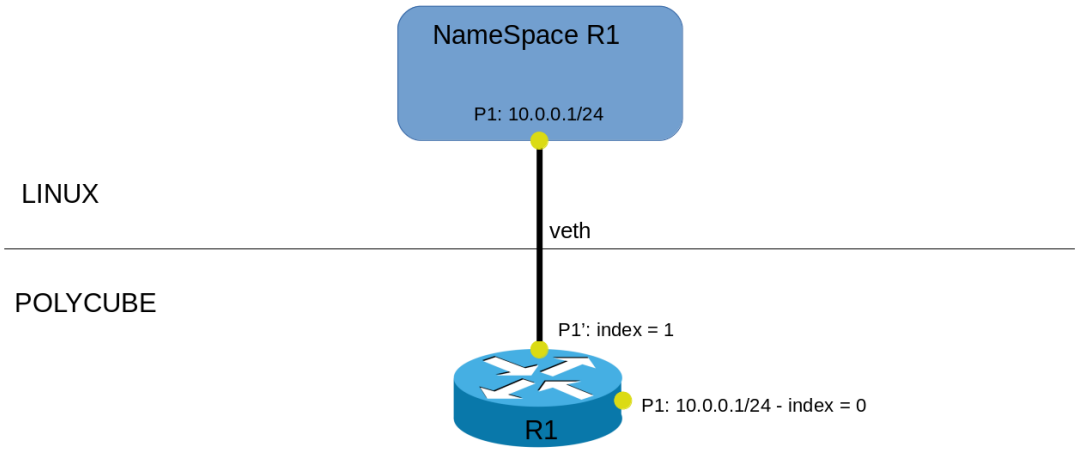


Figura 4.5: Implementazione Pratica di un Router Shadow

### Visione Logica

Dal punto di vista logico la soluzione implementata può essere vista come se lo Shadow Service fosse stato dotato di un sistema operativo.
Per capirci meglio, quando l'utente si posiziona all'interno del namespace ed esegue un tools o un comando quali ad esempio un ping, questo viene eseguito come se fosse stato lanciato sul servizio stesso.
Nell'esempio del router i pacchetti generati dentro il namespace e iniettati nella rete avranno l'indirizzo IP sorgente di una delle porte del router, saranno quindi visti come pacchetti che il router stesso ha generato a prescindere dal tools o programma usato. Mettersi dentro il namespace equivale ad essersi collegati fisicamente con un PC su un router fisico e aver lanciato un comando su quella macchina. La Figura 4.6 mostra una visione logica del router Shadow dotato di due porte P1 e P2, le stesse che vengono viste da dentro il namespace.

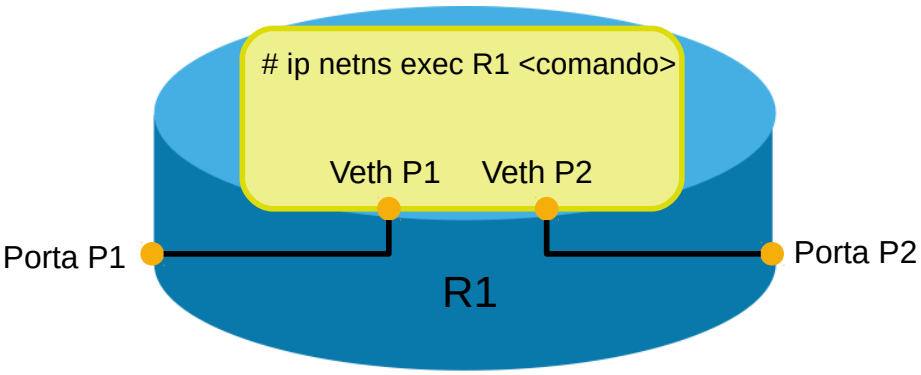


Figura 4.6: Visione Logica di un Router Shadow

### Cosa è Cambiato

Per capire cosa cambia in pratica con questa soluzione è bene disegnare la nuova struttura di un Servizio Shadow Polycube e metterla a confronto con quella descritta nella Sezione 2.2.5 e di cui si può vedere una rappresentazione grafica nella Figura 2.5. La nuova Struttura di uno Shadow Service si può vedere in Figura 4.7.

Si potrebbe pensare dopo aver visto la Figura 4.7 e averla confrontata con la Figura 2.5 che nei servizi Shadow il namespace sostituisca completamente sia lo Slow Path che il Control Path del servizio, ma non è esattamente così.

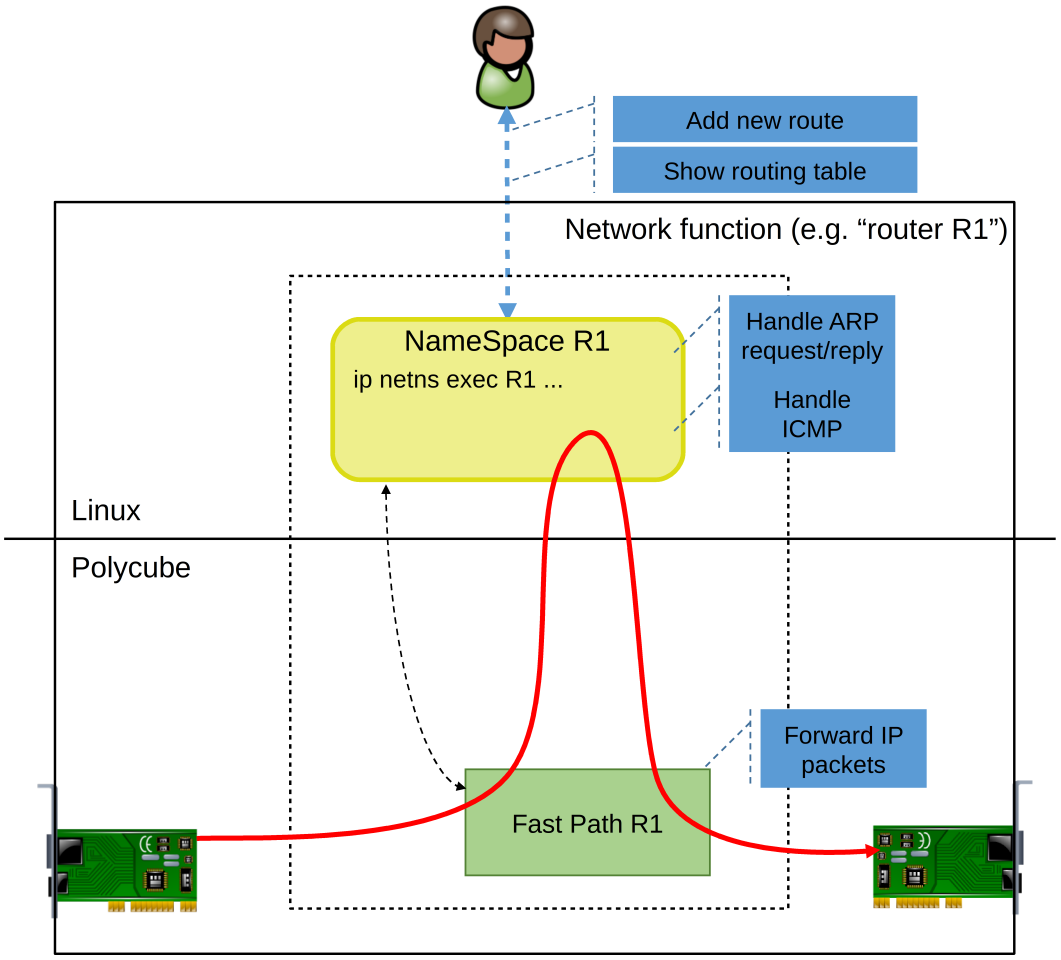


Figura 4.7: Struttura di uno Shadow Service

Il Control Path continua ad esistere e l'utente può continuare ad inserire comandi attraverso il Control Path di Polycube come ha sempre fatto. Comandi come "visualizzare la routing table" o "cambiare l'indirizzo IP della porta P1" possono essere inseriti dall'utente sia attraverso il namespace che attraverso il Control Path utilizzando la CLI di Linux o quella di Polycube. I due ambienti dal punto di vista dei comandi che gestiscono il servizio diventano quindi intercambiabili.
Quello che invece non può essere fatto attraverso il Control Path ma che il namespace aggiunge e permette di fare sugli Shadow Services è lanciare tools o eseguire comandi quali ad esempio il comando ping generato dal servizio stesso.

Per quanto riguarda lo Slow Path anche questo continua ad essere presente in uno Shadow Service, con la possibilità però da parte dello sviluppatore di scegliere se utilizzare il namespace e far gestire a Linux (o ad altri tools in esecuzione nel namespace) i pacchetti come ARP, ICMP e tutti gli altri protocolli che il Servizio deve saper gestire, lasciando di fatto lo Slow Path del servizio vuoto, oppure se lo sviluppatore lo ritiene utile scriversi una sua versione di codice nello Slow Path in grado di gestire quel traffico.

La possibilità di mettere lo sviluppatore davanti alla scelta di decidere se implementare o meno i protocolli che potrebbero essere gestiti da Linux e dare all'utente la possibilità di scegliere quale ambiente preferisce per impartire i comandi sono il valore aggiunto di questa soluzione.

#### Percorso del Traffico

A questo punto il lettore potrebbe chiedersi quale sia la strada che segue il traffico di rete per giungere al namespace e viceversa dal namespace per uscire sulla rete. È proprio in questo caso che entra in gioco l'index della porta. Si era fatto notare infatti che le porte nascoste che servivano a collegare il cubo con il namespace hanno tutte index dispari, mentre le porte visibili e utilizzate dal cubo hanno index pari.

Grazie a questa proprietà data alle porte è facile riconoscere il traffico che arriva dalla rete e differenziarlo da quello che arriva dal namespace e quindi da Linux, basta solamente controllare l'index della porta su cui si è ricevuto il pacchetto.
Sulla base del valore dell'index si ricade in due casi distinti:

- **Index Pari**: il pacchetto arriva dalla rete e va processato. Alla fine del processamento potrà essere instradato verso la destinazione, inviato allo Slow Path, inviato a Linux attraverso la porta dispari associata alla porta in ingresso o come ultima possibilità droppare il pacchetto.
- **Index Dispari**: il pacchetto arriva da Linux. È stato costruito e sono stati riempiti tutti i campi (MAC sorgente e MAC destinazione, IP sorgente e IP destinazione, ecc.). Il Fast Path che si vede arrivare il pacchetto dalla porta con index dispari deve semplicemente farlo uscire dalla corrispondente porta con index pari senza bisogno di processarlo.

In Figura 4.8 è illustrato il flow chart che spiega la logica utilizzata ad instradare il traffico da e verso il namespace.

## 4.5 Esempi Pratici

Fino a questo punto non ci siamo mai posti la domanda se avere uno *Shadow Service* ha senso con qualsiasi funzione di rete.

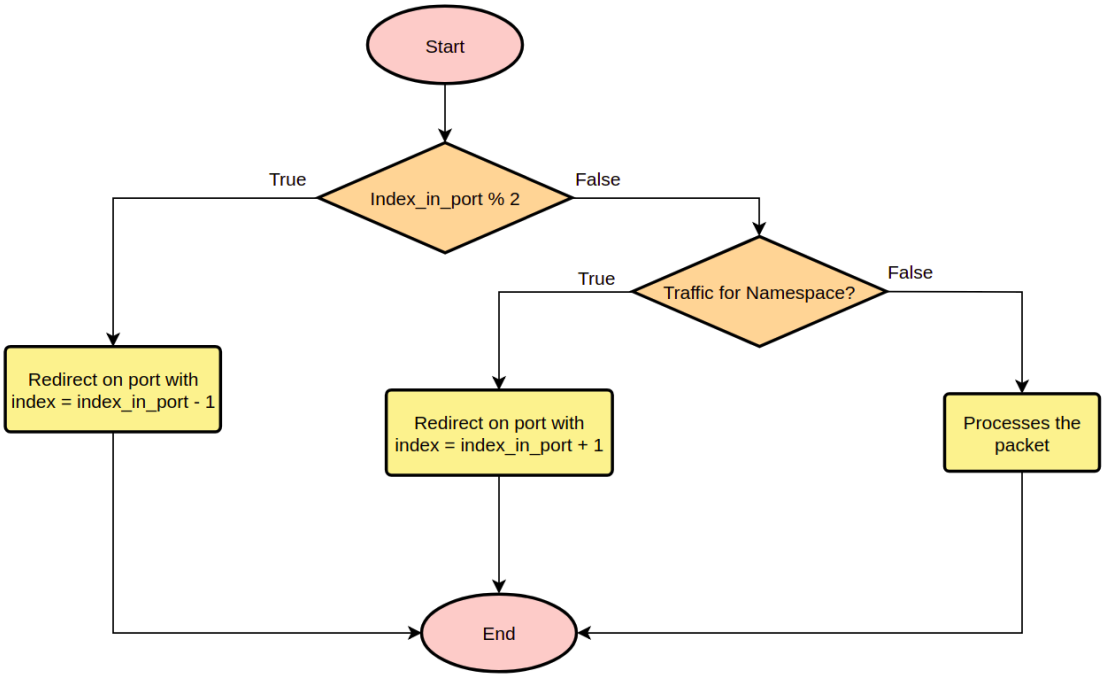


Figura 4.8: Flow Chart del traffico da/verso il namespace del Servizio

Sebbene l'idea di *Shadow Service* è generica, può cioè essere applicata a qualunque funzione di rete, visto che non ha vincoli particolari, è anche vero che funzioni di rete diverse possono trarre vantaggi diversi dall'essere Shadow.
Per questo motivo in questa sezione vengono analizzati due esempi pratici.

### 4.5.1 pcn_simplebridge

Il primo Servizio Shadow che viene analizzato è il *pcn_simplebridge*. Il bridge [15] si colloca al livello due del modello ISO/OSI (livello collegamento) e il suo compito è quello di decidere su quale porta inoltrare i frame, basandosi sulle sole due informazioni che sono in suo possesso, ossia l'indirizzo del nodo mittente e indirizzo del nodo destinatario.

Un simplebridge Shadow a parte poter sfruttare il vantaggio di essere configurato attraverso i due ambienti, Polycube o Linux in modo intercambiabile (vantaggio di cui godono tutti i servizi Shadow) ha l'interessante proprietà di non dover per forza implementare uno Spanning Tree "hardcoded" o a codifica fissa, basato cioè sui valori inseriti nel codice ma può essere usata un'implementazione di Spanning Tree esistente per Linux.

### 4.5.2 pcn_router

Il secondo Servizio Shadow che viene analizzato è il *pcn_router*, anche se già è stato più volte utilizzato come esempio nella tesi. Il router [16] si colloca al livello tre del modello ISO/OSI (livello rete) e il suo compito è quello di instradare i pacchetti tra reti IP diverse.
Le porte del router a differenza di quelle del simplebridge sono caratterizzate da più informazioni, quali indirizzo MAC, indirizzo IP e Netmask. Anche il numero di protocolli indispensabili aumenta, rendendolo sicuramente un servizio più complicato.
Il funzionamento del router si basa su una tabella ARP, dove vengono mappati gli indirizzi MAC e le porte, ma anche su una tabella di Routing. Questo rende il Router un ottimo candidato per essere un servizio Shadow, i motivi sono principalmente due:

1. **Tabella di Routing**
   La tabella di routing serve a salvare le informazioni necessarie ad instradare il traffico tra le varie reti IP.
   Un router Shadow rispetto ad un router classico Polycube ha la possibilità di inserire le informazioni nella tabella di routing senza che l'utente le inserisca manualmente oppure può scegliere semplicemente di fare a meno della tabella di routing e chiedere direttamente a Linux cosa fare del pacchetto. Il namespace è dotato di una tabella di routing ed ha tutte le interfacce e i parametri del router Shadow. Questo porta ad avere due strade possibili:

   (a) Utilizzare l'helper *fib_lookup* nel codice eBPF (si rimanda alla Sezione 2.1.2 per sapere cos'è un Helper), che consiste nel chiedere a Linux su quale interfaccia far uscire un pacchetto. Questa soluzione permette ad un router Shadow di fare completamente a meno della tabella di routing.

   (b) Leggere la tabella di routing in Linux e copiarsi le informazioni che interessano il router Shadow nella tabella di routing presente in Polycube. Questa soluzione permette di avere un router Shadow più simile al router non Shadow, evitando così di dover scrivere due servizi router completamente diversi.

2. **Routing Dinamico**
   L'attuale versione del router Polycube non è in grado di gestire il *routing dinamico*, tutto funziona solo se le rotte vengono inserite manualmente nella tabella di routing.
   Gestire reti più o meno complesse di livello 3 manualmente non è sicuramente la soluzione da adottare, ed è stato proprio per questo motivo che sono nati i protocolli di routing.
   Il router Shadow grazie alla cooperazione con Linux è in grado di gestire il

routing dinamico. Questo è stato il motivo principale che ha spinto più degli altri verso la cooperazione con Linux. È vero che è stato detto che cooperare porta tanti vantaggi ma il principale problema da risolvere era quello di rendere il router Polycube capace di gestire protocolli come OSPF [17].
Sono stati eseguiti dei test in cui utilizzando un software disponibile per Linux (Quagga [7]) un router Shadow è stato in grado di instradare pacchetti su rotte apprese in maniera automatica leggendo la tabella di routing del namespace.
L'unico compito che veniva assegnato al router Shadow era quello di riconoscere i pacchetti appartenenti ai protocolli di routing (per il test si è utilizzato il protocollo OSPF) così da far recapitare quel traffico al namespace e lasciarlo gestire a Quagga.

## 4.6 Vantaggi

Questa ultima sezione del capitolo prova a riassume i vantaggi generali che si ottengono adottando uno *Shadow Service* e apre la strada al capitolo 5 che segue dopo.

A questo punto dovrebbe essere chiaro che avere servizi Shadow risulta vantaggioso sia per l'utente che per lo sviluppatore, ecco l'elenco dei motivi:

- **Utente**:
  1. Poter scegliere con quale CLI configurare l'ambiente di rete.
  2. Poter riutilizzare gli script scritti per Linux.
  3. Avere a disposizione tools come ad esempio *ifconfig, ip, Wireshark* e molti altri.
- **Sviluppatore**:
  1. Essere libero di implementarsi un protocollo o lasciare che sia Linux a gestire quel traffico.
  2. Avere la possibilità di dotare le proprie network functions di funzionalità complesse senza richiedere sforzi eccessivi.
- **Entrambi**:
  1. Prestazioni del fast path che rimangono praticamente identiche, uno Shadow Service non influenza le prestazioni del servizio.
  2. Sotto alcune condizioni godere della proprietà di *Resilienza*.

### 4.6.1 Resilienza

Come ultimo vantaggio di uno Shadow Service è stato detto che può godere della proprietà di *Resilienza*. In informatica, la Resilienza è la capacità di un sistema di adattarsi alle condizioni d'uso e di resistere all'usura in modo da garantire la disponibilità dei servizi erogati [18].

Avere le stesse informazioni in Polycube e in Linux permette ad uno Shadow Service, date alcune condizioni, di godere di questa importantissima proprietà.

Si pensi al caso di un router Shadow che instrada i pacchetti in eBPF; avrà la sua tabella di routing allineata con quella del namespace Linux e funzionerà senza problemi finché Polycube sarà in esecuzione.

Si ipotizzi però che all'istante *Tx* il router Polycube inizi ad avere dei problemi e smetta di funzionare. Se Polycube fosse in grado di accorgersi del problema potrebbe evitare di distruggere il namespace e le interfacce virtuali in Linux e eliminare solo l'istanza del router Polycube, cioè il cubo.
Tutti i pacchetti ricevuti dall'istante *Tx+1* potrebbero allora essere rediretti al namespace, che se avrà il forwarding attivo potrà continuare ancora a fare routing, grazie al fatto di avere le stesse informazioni nella sua tabella di routing, garantendo la continuità del servizio finché Polycube non istanzi un nuovo router e lo colleghi al vecchio namespace.
Il nuovo router si ritroverebbe così tutte le interfacce già configurate e la tabella di routing già riempita grazie alla ridondanza delle informazioni presenti sia in Linux che in Polycube. Ciò garantisce una continuità ai servizi che risulta essere una delle proprietà più apprezzate oggigiorno.

La Resilienza così come gli altri vantaggi di uno Shadow Service derivano dal fatto di avere le informazioni duplicate in Linux e in Polycube. Questo però comporta saper mantenere queste informazioni allineate e reagire nel momento in cui uno dei due ambienti voglia aggiornare i dati che ha a disposizione.

La soluzione a questo problema è descritta nel prossimo capitolo della tesi.

# Capitolo 5

# Netlink e CLI

Nell'introdurre gli Shadow Service è capitato più volte di affermare *"vengono copiate le informazioni dall'interfaccia Linux"* o *"vengono prelevate le rotte dalla tabella di routing di Linux"* e altre frasi simili. Non si è spiegato bene come sono state implementate queste operazioni e soprattutto cosa è stato fatto per mantenere queste informazioni allineate nel momento in cui in uno dei due ambiente, sia questo Polycube o Linux le modifichi.

In questo capitolo viene presentato *Linux Netlink* [19], un meccanismo IPC basato su socket ed utilizzato per la comunicazione tra più processi dello userspace o tra processi e kernel. Questo strumento è utilizzato nella tesi come soluzione al problema appena descritto ma anche per fare in modo che le due *Command Line Interface*, quella di Linux e quella di Polycube possano essere intercambiabili.

## 5.1 Netlink

Possono il codice del kernel e il codice dello spazio utente comunicare tra loro?
Esistono diversi metodi per farlo, conosciuti come *inter-process communication (IPC)*, cioè quelle tecnologie software che hanno come scopo la comunicazione e lo scambio di dati e informazioni tra processi. Tra questi ci sono: *system call*, *ioctl* e *Netlink*.

### 5.1.1 Perchè usare Netlink

Perché si sceglie di utilizzare Netlink [20], invece di usare altri IPC quali ad esempio system call o ioctls per la comunicazione tra spazio utente e kernel?
La risposta a questa domanda include un insieme di motivi che vengono elencati di seguito:

1. Scrivere system call o ioctls ogni qual volta si deve aggiungere una nuova funzionalità al codice utente non è sicuramente una attività banale. Si rischia di inquinare il kernel e danneggiare la stabilità del sistema. I socket Netlink invece sono semplici, e permettono all'applicazione di parlare con il kernel in modo immediato.

2. Netlink è asincrono, così come qualsiasi altro socket possiede una coda in cui vanno a finire i messaggi e che ne facilita la gestione. L'invio di un messaggio Netlink prevede di mettere il messaggio nella coda del destinatario e avvertirlo della presenza di un nuovo messaggio. Il destinatario è libero di decidere se elaborare immediatamente il messaggio o lasciarlo nella coda ed elaborarlo successivamente in un contesto diverso. Le system call, a differenza di Netlink, richiedono l'elaborazione sincrona. L'uso di tecniche sincrone per gestire messaggi lunghi potrebbe influenzare le prestazioni dell'intero sistema.

3. Il codice che implementa una system call è collegato staticamente al kernel in fase di compilazione, pertanto non è possibile includere questo codice in un modulo caricabile. Con i Socket Netlink si evita ciò perchè non esiste alcuna dipendenza dal tempo di compilazione, le applicazioni Netlink sono *loadable kernel module*, cioè file oggetto in grado di estendere il kernel a runtime.

4. I socket Netlink supportano il multicast, che è un altro vantaggio rispetto a system call e ioctls. Un processo può inviare un messaggio multicast a un indirizzo di gruppo Netlink, e tutti i processi che sono in ascolto su quell'indirizzo di gruppo riceveranno il messaggio. Questo è un ottimo meccanismo di segnalazione degli eventi kernel, in quanto un processo user si registra ai gruppi Netlink di suo interesse e il kernel con un unico messaggio multicast avvisa tutti i processi interessati a quell'evento.

5. System call e ioctl permettono l'avvio della comunicazione solo da parte delle applicazioni nello userspace, ma cosa succede se un processo kernel ha un messaggio urgente da consegnare ad una applicazione dello userspace?
   Non c'è modo di comunicarlo finché l'applicazione non verrà a chiederlo, per questo normalmente le applicazioni richiedono periodicamente il polling del kernel per ottenere eventuali modifiche dello stato, sebbene il polling intensivo sia costoso. Netlink risolve questo problema permettendo anche al kernel di avviare la comunicazione. Per questo i socket netlink sono detti *full duplex*.

6. I socket netlink forniscono una API socket-style BSD che è ben conosciuta dagli sviluppatori di software, ciò riduce i tempi di apprendimento e ne facilita la comprensione.

### 5.1.2 Socket Netlink

Un *socket* - descritto nel RFC 147 [21] - è un oggetto software che permette l'invio e la ricezione di dati tra due endpoint, questi possono essere due host remoti (collegati tramite una rete) o due processi locali (Inter-Process Communication).

NETLINK - descritto nel RFC 3549 [22] - è una funzionalità del sistema operativo Linux che permette alle applicazioni dello spazio utente di comunicare con il kernel. Netlink quindi è un'estensione dell'implementazione standard di socket, cioè uno speciale IPC utilizzato per trasferire informazioni tra i processi kernel e quelli userspace. A differenza dei classici socket però, è bene ricordare che le comunicazioni Netlink non possono attraversare i confini dell'host.

La creazione di un socket netlink avviene nello stesso modo in cui avviene la creazione di un qualsiasi socket di rete [23]:

```
int socket (int domain, int type, int protocol);
```

- *domain*: si riferisce alla famiglia di socket. La famiglia **AF_NETLINK** è quella utilizzata da netlink.
- *type*: si riferisce al tipo di socket. Netlink può usare socket raw (**SOCK_RAW**) o socket datagram (**SOCK_DGRAM**), ma in genere si usano socket raw.
- *protocol*: si riferisce al numero di protocollo. La lista dei protocolli disponibili cambia nel tempo e si espande così che possano essere estese le caratteristiche e gli aspetti dello stack di rete Linux configurabili attraverso Netlink.

### 5.1.3 Protocolli Netlink

Di seguito è riportato un sottoinsieme (non esaustivo) di protocollo attualmente supportati dai socket netlink.

#### NETLINK_ROUTE

Il protocollo NETLINK_ROUTE è quello con il più vasto campo di applicazione ed è sicuramente il più maturo della famiglia dei protocolli Netlink. Contiene un proprio sottoinsieme di messaggi, che possono essere divisi in famiglie, ognuna delle quali controlla uno specifico aspetto del sottosistema di rete del kernel.

Alcune delle famiglie, con i rispettivi messaggi, sono:

- *Links*: RTM_NEWLINK, RTM_DELLINK, RTM_GETLINK
- *Addressed*: RTM_NEWADDR, RTM_DELADDR, RTM_GETADDR

- *Routes*: RTM_NEWROUTE, RTM_DELROUTE, RTM_GETROUTE
- *Neighbors*: RTM_NEWNEIGH , RTM_DELNEIGH , RTM_GETNEIGH
- *Rules*: RTM_NEWRULE, RTM_DELRULE, RTM_GETRULE

Ogni famiglia è composta da tre metodi; un metodo NEW, un metodo DEL e un metodo GET.
Poiché ogni famiglia controlla un insieme di data object che possono essere rappresentati come una tabella, ognuno di questi metodi permette all'utente di creare entries nella tabella, eliminare entries dalla tabella e ottenere uno o più oggetti dalla tabella.

### NETLINK_FIREWALL

Il protocollo NETLINK_FIREWALL è molto utilizzato. Nasce come strumento per facilitare la costruzione e il debugging di moduli iptables nello spazio utente, ma in combinazione con altri packet sockets e/o raw sockets può essere usato per pilotare traffico di rete di diverse applicazioni secondo specifici patterns.

### NETLINK_ARPD

Il protocollo NETLINK_ARPD fornisce una interfaccia per gestire la tabella *ARP* dallo spazio utente.

### NETLINK_NFLOG

Il protocollo NETLINK_NFLOG fornisce una interfaccia utile per le comunicazioni tra spazio utente e *Netfilter* o tra spazio utente e *iptables*.

### NETLINK_XFRM

Il protocollo NETLINK_XFRM riguarda invece la sicurezza e si occupa della gestione di *security policy databases* e *IPsec Security Association (AS)*. Utilizzato principalmente dai demoni key manager con il protocollo *Internet Key Exchange*.

### Aggiungere Protocolli Netlink nel Kernel

Così come i programmi nello spazio utente possono usare Netlink per comunicare con i vari sottosistemi del kernel attraverso l'uso dell'API standard dei socket, esiste una API lato kernel che consente agli sviluppatori di estendere o creare nuovi protocolli Netlink da aggiungere a quelli già esistenti. Questo nell'ottica di fornire ulteriore accesso ad altre funzionalità del kernel da parte dei programmi che si trovano nello userspace.

### 5.1.4 Indirizzamento

L'indirizzamento Netlink si basa su un numero di porta a 32 bit. La porta 0 (zero) è riservata per il kernel, quindi indica i socket lato kernel di ciascuna famiglia di protocolli, gli altri numeri sono usati nello userspace. Inizialmente ogni programma utilizzava *l'identificatore di processo* (PID) come numero di porta lato user. Questo si è scoperto essere una limitazione nel momento in cui una applicazione richiede l'utilizzo di più socket. Pertanto il sistema genera numeri di porta univoci basati sul PID (a cui aggiunge un offset) ogni volta che un processo apre un socket netlink. Il primo socket continua ad avere numero di porta uguale al PID per ragioni di compatibilità.

La Figura 5.1 mostra un esempio di socket netlink, con i casi d'uso comuni:

- Comunicazione tra userspace e kernel
- Comunicazione tra userspace e userspace
- Applicazione che si mette in ascolto sulle notifiche multicast del kernel

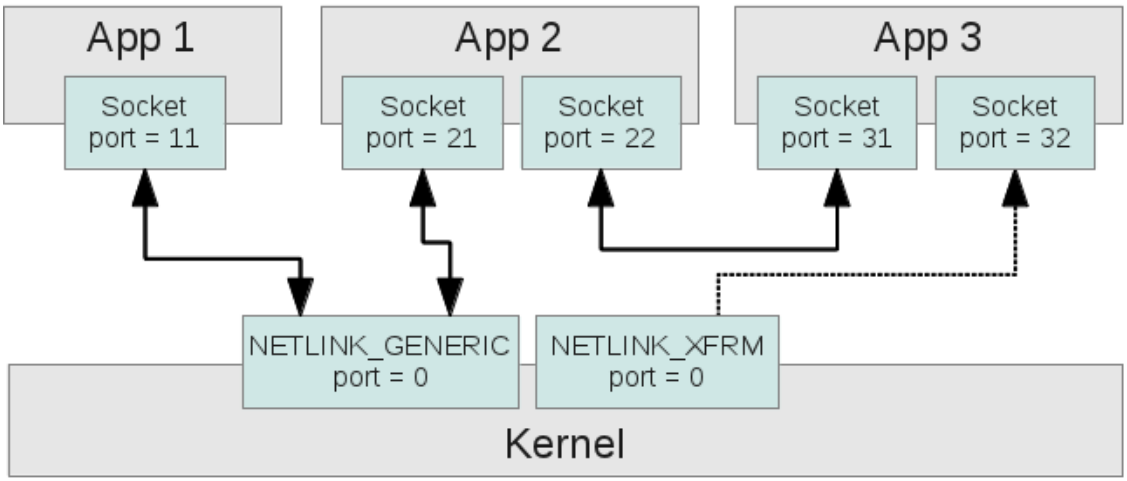


Figura 5.1: Esempi di Socket Netlink

### 5.1.5 Messaggi Netlink

Un protocollo Netlink si basa su dei messaggi. Un messaggio è costituito da una intestazione (*struct nlmsghdr*) comune a tutti i protocolli e dal payload.

La Figura 5.2 mostra la struttura utilizzata per l'intestazione.

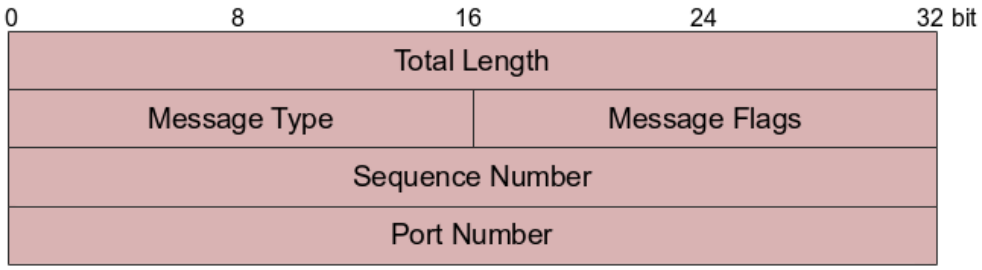


Figura 5.2: Struct nlmsghdr

#### Tipo di messaggio

Netlink supporta tre tipi di messaggi:

- *Richieste*: utilizzate quando si vuole chiedere al ricevente di compiere una azione. Una richiesta viene in genere inviata da un processo userspace al kernel.
- *Risposte*: utilizzate come risposta alla richiesta ricevuta. La risposta può trasportare un errore o una notifica di successo.
- *Notifiche*: sono di natura informale (non prevedono risposte) e vengono utilizzate dal kernel per notificare ai processi dello userspace interessati il verificarsi di un evento.

### 5.1.6 Esempi di utilizzo Netlink

Netlink è un modo semplice ma versatile di manipolare l'ambiente di rete di un host Linux. Chi gestisce i protocolli di rete dallo userspace è invitato ad utilizzare Netlink. Tanti protocolli standard rilasciati dall'*Internet Engineering Task Force* (IETF) sono implementati nello userspace, e molte di queste implementazioni richiedono la manipolazione del routing e la conoscenza degli eventi kernel o quello che succede in altri processi. Rientrano tra queste categorie di protocolli, e quindi possibili esempi di utilizzo di Netlink:

- **Protocolli di routing dinamico**: *Routing Information Protocol* (RIP), *Open Shortest Path First* (OSPF) e l'*Exterior Gateway Protocol* (EGP) sono solo alcuni dei protocolli di routing dinamico che hanno la necessità di gestire attivamente l'ambiente di routing di un host.
- **Protocolli di mobilità**: gli host mobili sono connessi a reti diverse in momenti diversi, utilizzando protocolli quali *Mobile IP* (MIP), *Session Initiation*

*Protocol* (SIP) e *Network Mobility* (NEMO) per gestire il routing e mantenere la connettività e la continuità delle comunicazioni.

- **Protocolli di rete ad hoc**: gli host mobili e ubicati in luoghi in cui non vi è alcuna infrastruttura di rete, come router o access point WLAN, richiedono comunicazioni peer-to-peer con host configurati in modo diverso. I computer mobili di soccorritori in un'area colpita da terremoto o altre emergenze di questo tipo possono utilizzare protocolli di rete ad hoc. Questi protocolli, come per esempio *Ad hoc On-demand Distance Vector* (AODV) e *Optimized Link State Routing* (OLSR), richiedono la gestione del routing per poter funzionare

Netlink aiuta a ridurre la complessità del codice kernel, permettendo di implementare questi protocolli nello userspace, semplificandone lo sviluppo e il testing e riducendone i rischi.
Si ricorda infine che anche la suite avanzata di comandi di routing IP, denominata *IPROUTE2* [24], si basa su Netlink.

## 5.2 Netlink e Polycube

Nello sviluppo della tesi Netlink è risultato essere la naturale soluzione ad alcuni dei problemi che sorgono nel momento in cui si implementa uno Shadow Service. Basti pensare al caso in cui è necessario risalire ad informazioni situate nel kernel Linux e non si sa come fare.

In questa sezione viene spiegato cosa è stato fatto e perchè Netlink è stato così utile.

### 5.2.1 Accedere alle Informazioni nel Kernel

Nel momento in cui un utente crea una porta su uno Shadow Service, potrebbe volerla creare con i parametri di una interfaccia già esistente in Linux, senza il bisogno di crearne altre virtuali. In questo caso si era detto che l'utente può fare a meno di dare parametri aggiuntivi a parte il nome dell'interfaccia, perchè sarà il Servizio stesso ad andare dal kernel e chiedere tutte le informazioni necessarie per creare quella porta. Questo è possibile perchè il Servizio apre un Socket Netlink con il kernel in cui invia una richiesta e attende di ricevere una risposta che conterrà tutti i dati che lo interessano. Analogamente questo meccanismo viene usato ogni qual volta un Servizio Shadow voglia accedere ad una delle tabelle di Linux, siano queste tabelle di Routing o tabelle ARP.

### 5.2.2 Due CLI un solo Comando

Avere due ambienti significa avere il doppio delle informazioni. Avere il doppio delle informazioni può essere un vantaggio come si è visto, ma significa anche gestire il doppio degli oggetti (tabelle, interfacce, porte, parametri).

Come se non bastasse ogni ambiente ha la proprio CLI, non basta quindi dare lo stesso comando due volte in due ambienti separati. Il risultato sarebbe quello di vedere in uno dei due ambienti il comando avere esito positivo e andare a buon fine mentre nell'altro si vedrebbe comparire un messaggio di errore perchè quello stesso comando non significherebbe nulla.

Solo questo basta a creare una tale confusione da rendere altamente probabile la situazione in cui si hanno i due ambienti, che dovrebbero cooperare, ad essere invece in contrasto tra di loro.
Per capire meglio ecco un esempio:

- Si consideri il caso in cui uno Shadow Service abbia una interfaccia chiamata *veth1*. Questa sarà presente anche su Linux e quindi è lecito chiedere a Linux di mostrare l'indirizzo IP di questa interfaccia.
  Linux quando interrogato risponde dicendo di avere *veth1 - IP=10.0.0.1 - Netmask=255.255.255.0*.
  Un istante dopo viene fatta la stessa domanda a Polycube, che però risponde dicendo di avere *veth1 - IP=10.0.0.2 - Netmask=255.0.0.0*.

A questo punto sorgono dei dubbi:

Chi dei due ha ragione? Linux o Polycube? Quale informazione è più recente e quindi da considerare valida?

Finché ci saranno due CLI e sarà lasciato all'utente il compito di mantenere allineate le informazioni ogni volta che c'è da modificare qualcosa, inserendo un comando per ogni ambiente, non si troverà mai una risposta a queste domande.

Si è quindi pensato di usare Netlink per risolvere questo problema e rendere le due CLI intercambiabili. Poter quindi inserire un solo comando o in Linux o in Polycube senza distinzione, ottenendo lo stesso risultato ma avendo le informazioni allineate.

### 5.2.3 Da Linux a Polycube

Se viene scelto di utilizzare la CLI di Linux, o un processo Linux aggiorna le informazioni presenti nel kernel, Polycube è in grado di reagire ai cambiamenti grazie alle seguenti due azioni:

1. **Registrarsi agli Eventi**

   Quando viene creato uno Shadow Service si sa in partenza che questo sarà interessato ad un determinato numero di eventi. Eventi che dovranno essere notificati al Servizio e sarà il Servizio stesso a reagire nel momento in cui si accorge che è successo qualcosa.
   Ogni Shadow Service è interessato a eventi diversi che dipendono dalla sua stessa natura. Consideriamo per esempio il caso del Router, per come è fatto si registrerà ad eventi diversi da quelli a cui si registrerà un SimpleBridge.

   Ecco la lista degli eventi Netlink che interessano un Router:

   - *RTM_NEWLINK*: Avvisa lo Shadow Service che è stata creata una nuova interfaccia su Linux, o che lo stato di un interfaccia è cambiato.
   - *RTM_DELLINK*: Avvisa lo Shadow Service che è stata rimossa una interfaccia su Linux.
   - *RTM_NEWADDR*: Avvisa lo Shadow Service che una interfaccia Linux ha modificato l'indirizzo IP o la Netmask.
   - *RTM_NEWROUTE*: Avvisa lo Shadow Service che una nuova rotta è stata inserita nella tabella di Routing.
   - *RTM_DELROUTE*: Avvisa lo Shadow Service che è stata eliminata una rotta dalla tabella di Routing.

   Da questa lista di eventi si deduce facilmente che ad un SimpleBridge non interessa sapere che è stata aggiunta una rotta nella tabella di Routing, ne tanto meno di essere notificato dei cambiamenti che riguardano gli indirizzi IP o le Netmask. Un SimpleBridge per esempio si registrerà agli eventi che lo notificano del cambiamento di un indirizzo MAC su una porta.
   Questo a conclusione del fatto che ogni Shadow Service nel suo costruttore si registrerà solo agli eventi di suo interesse che cambiano per l'appunto da un Servizio ad un altro.

2. **Reagire agli Eventi**

   Registrarsi agli eventi è il primo passo ma da solo non è sufficiente a risolvere il problema.
   Dopo che il Servizio Shadow si registra a tutti gli eventi che lo interessano deve anche essere in grado di reagire quando riceve una notifica. Per questo motivo ogni evento Netlink a cui un Servizio Shadow si registra richiede che venga implementata una funzione che sarà eseguita ogni volta che si riceve un

messaggio di notifica da parte del kernel.

Le funzioni in genere seguono i seguenti passi:

(a) Controlla se il messaggio di notifica faccia riferimento ad una delle interfacce del Servizio. Altrimenti viene semplicemente ignorato.

(b) Se il messaggio effettivamente interessa una delle interfacce di quel Servizio, verifica cosa è cambiato e risale ai nuovi parametri.

(c) Reagisce cancellando le vecchie informazioni e aggiornando i dati con i nuovi parametri.

### 5.2.4 Da Polycube a Linux

Se l'utente invece preferisce utilizzare la CLI di Polycube per cambiare i parametri di una porta o per aggiungere nuove informazioni (quali ad esempio nuove rotte statiche nella tabella di Routing). Sarà lo Shadow Service ad inviare un messaggio Netlink al kernel per chiedere che quella informazione venga aggiornata anche su Linux. Il messaggio questa volta sarà una richiesta Netlink, a cui seguirà una risposta da parte del kernel che notifica l'avvenuto successo della richiesta o eventualmente un messaggio di errore.

Vale la pena far notare a questo punto che data questa architettura è facile ricadere nella spiacevole situazione di un loop infinito.
Si consideri questo esempio:

1. L'utente decide di utilizzare la CLI di Polycube per modificare un parametro di una porta, per esempio l'indirizzo IP.

2. Polycube modifica l'indirizzo IP della porta e crea una richiesta Netlink da inviare al kernel per avvisarlo che su quella interfaccia è cambiato l'IP.

3. Il kernel ricevuto il messaggio Netlink modifica l'IP dell'interfaccia, risponde con una risposta positiva per dire che è andato tutto bene e subito dopo invia una notifica Netlink al gruppo multicast per avvertire tutti i processi userspace interessati del verificarsi di quell'evento.

4. La notifica multicast però viene anche ricevuta dallo Shadow Service che non ha modo di differenziarla dalle altre notifiche e quindi reagisce a quell'evento modificando di nuovo l'indirizzo IP e avvisando di nuovo il kernel (si ritorna al punto 2)

Il problema è facilmente superabile evitando di mandare una richiesta Netlink al kernel quando la modifica del parametro avviene come causa di una notifica che arriva dal kernel stesso. La soluzione è stata quindi quella di implementare due

funzioni distinte per aggiornare gli stessi parametri. Per rendere meglio l'idea si pensi nuovamente alla funzione che aggiorna l'indirizzo IP, esiste in due versioni diverse:

1. La prima aggiorna l'indirizzo IP della porta sullo Shadow Service e ritorna.
2. La seconda aggiorna l'indirizzo IP della porta sullo Shadow Service e invia il messaggio Netlink al kernel per avvertirlo della modifica.

Sulla base del fatto che sia l'utente tramite CLI Polycube o che sia una notifica Netlink del kernel a chiedere che venga aggiornato il parametro verrà eseguita l'una o l'altra versione della funzione.

## 5.3 Vantaggi

Questa ultima sezione del capitolo riassume i vantaggi che si ottengono nell'adottare Netlink come strumento per accedere alle informazioni del kernel e per mantenere i due ambienti allineati.

Accedere alle informazioni delle interfacce fisiche o virtuali o alle tabelle Linux contenute nel kernel è essenziale se si vogliono ricopiare questi dati anche in Polycube, e Netlink è il metodo migliore per chiedere queste informazioni al kernel per tutti i motivi spiegati all'inizio di questo capitolo (Sezione 5.1.1).

Poter utilizzare due CLI distinte per poter configurare l'ambiente di lavoro è una cosa molto gradita dagli utenti, specie se si pensa al caso frequente in cui molti sistemi sono gestiti e configurati attraverso l'utilizzo di *script*. Pensare di dover riscrivere tutti gli *script* esistenti non è sicuramente lavoro da poco.

Ultimo vantaggio nell'usare Netlink ma forse il più importante è la sincronizzazioni, e cioè poter avere i due ambienti sempre allineati ed evitare di ritrovarsi nella spiacevole situazione in cui le informazioni collidono tra di loro. Questa situazione sarebbe difficilmente gestibile e per questo motivo evitarla è l'unica maniera di risolvere il problema.

# Capitolo 6

# Validazione dei risultati

In questo capitolo vengono descritti i risultati ottenuti e presentati i test effettuati con la nuova architettura.

Nei test viene dato risalto alle nuove funzionalità che il framework Polycube è in grado di supportare grazie agli Shadow Service e vengono misurate e confrontate le prestazioni ottenute con la nuova soluzione.

## 6.1 Test Environment

Gli ambienti su cui sono stati eseguiti i test sono stati realizzati utilizzando:

- *pcn_router*: è stato il Servizio su cui sono stati eseguiti più test in assoluto. Questo perchè si presta molto bene ad essere Shadow e può trarre molti vantaggi dalla cooperazione con Linux.

- *pcn_simplebridge*: è stato utilizzato sia nella sua versione Shadow che non Shadow, per permetterci di ricreare topologie di rete più articolate. Vale la pena ricordare che nulla vieta di avere in Polycube contemporaneamente servizi Shadow e non Shadow e collegarli tra di loro.

- *Linux Network NameSpace* [13]: utilizzati per simulare gli host. Collegati ai servizi Polycube ci permettono di iniettare pacchetti nella rete eseguire comandi o lanciare tools.

- *Macchina virtuale*: Utilizzata per simulare un servizio esterno che non appartenesse a Polycube

- *Software Quagga*: la network routing suite Quagga [7] è stato il software che ha permesso a Linux di gestire il traffico OSPF e aggiornare la tabella di routing in maniera dinamica.

## 6.2 Test 1

### 6.2.1 Linux Router

Questo primo test a cui è stato dato il nome di *Linux Router*, non è altro che un router Polycube dotato di tante porte quante sono le interfacce del host Linux.

In questo test non è esattamente corretto parlare ancora di router Shadow. Il motivo è semplicemente perchè non viene creato un network namespace per il router ma si utilizza il *root namespace*. Questo è stato possibile perchè durante il test il router è l'unico servizio istanziato da Polycube e sul router vengono rese visibili tutte le interfacce del host.

Esternamente quindi può essere visto come se fosse Linux stesso a fare routing, con la differenza ovviamente rispetto al caso in cui si abilita semplicemente il forwarding dei pacchetti sul sistema operativo, che questa volta il routing viene fatto da Polycube e quindi si possono sfruttare tutti i vantaggi di eBPF.

L'ambiente per eseguire il test è stato preparato come segue:

1. Creare due namespace che simulano il comportamento di due host o più in generale di due rete IP.

2. Collegare i namespace alle interfacce Linux.

3. Le interfacce Linux sono sotto il controllo del programma eBPF del router che ne ha il possesso

4. Il fast path riceve tutti i pacchetti e decide se è traffico da instradare o eventualmente da reindirizzare a Linux.

Per questo test si è inoltre scelto di usare la versione del *pcn_router* senza tabella di routing, viene infatti utilizzato l'helper *fib_lookup* per instradare il traffico.

In Figura 6.1 viene mostrato lo schema del test 1.

## 6.3 Test 2

### 6.3.1 Router Shadow con Quagga

Il secondo test serve a verificare che un router Shadow è in grado di instradare il traffico su rotte imparate utilizzando un protocollo di routing dinamico gestito da Linux.

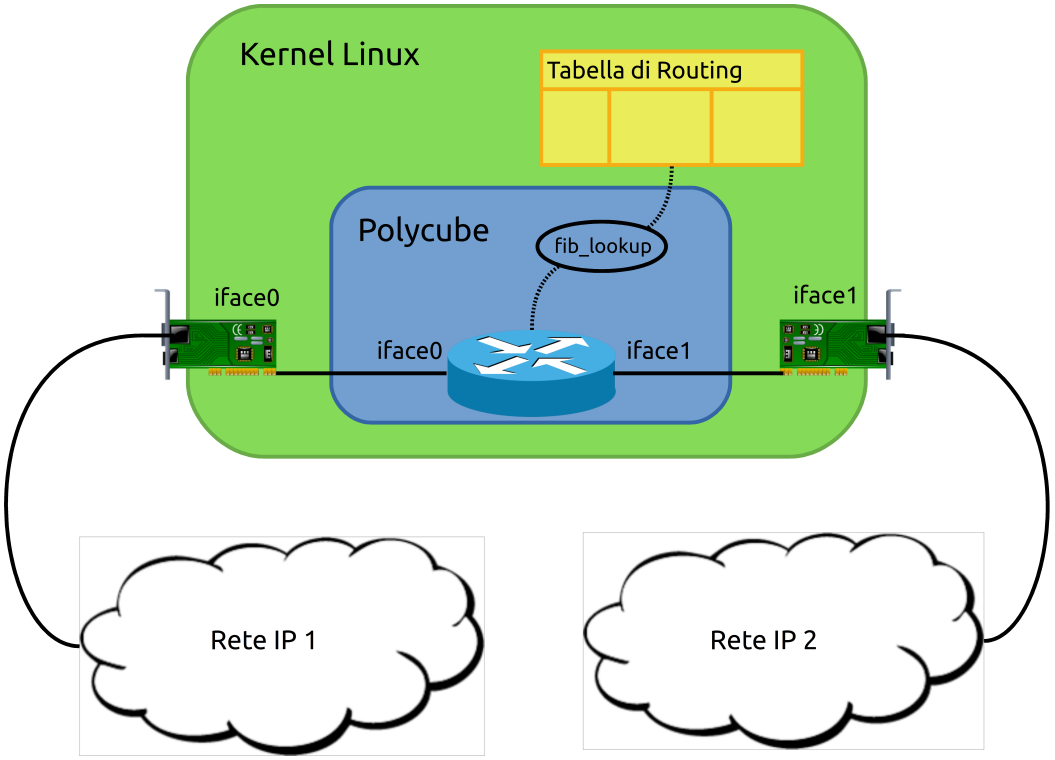


Figura 6.1: Test 1: Linux Router

Per questo test non si utilizza più l'helper *fib_lookup* in quanto non è stata ancora rilasciata una versione in grado di funzionare al di fuori del root namespace. Viene quindi usata come tabella di routing la mappa eBPF del router Polycube.

L'ambiente per eseguire il test è stato preparato come segue:

1. Creazione di un namespace sulla macchina fisica in grado di simulare il comportamento di un host o più in generale di una rete IP, che chiameremo *Rete IP 1*.

2. Esecuzione di una macchina virtuale su cui verrà lanciata una istanza del software Quagga con OSPF attivo.

3. Creazione di un secondo namespace sulla macchina virtuale in grado di simulare il comportamento di un secondo host o più in generale di una seconda rete IP che chiameremo *Rete IP 2*.

4. Creazione di un router Shadow dotato di due collegamenti, uno verso la macchina virtuale e l'altro verso il namespace della macchina fisica.

5. Esecuzione di una seconda istanza del software Quagga con OSPF attivo dentro il namespace del router Shadow.

Il processo Quagga in esecuzione sulla macchina virtuale annuncia attraverso OSPF ai suoi vicini di essere in grado di raggiungere la *Rete IP 2*. I pacchetti OSPF arrivano al router Shadow, il quale riconoscendoli nel fast path li reindirizza a Linux e quindi al namespace del servizio.
Nel namespace del servizio la seconda istanza di Quagga elabora i pacchetti OSPF, aggiorna la tabella di routing e annuncia a sua volta di essere in grado di raggiungere la *Rete IP 1*.
Qualche secondo dopo aver collegato il router Shadow alla macchina virtuale i due software Quagga avranno aggiornato le tabella di routing e in rete si vedranno solo più passare i classici pacchetti *Hello* del protocollo OSPF.

A questo punto posizionandosi nel namespace della *Rete IP 1* ed eseguendo un ping della *Rete IP 2* i pacchetti ICMP echo request e ICMP echo reply vengono instradati correttamente dal router Shadow.

In Figura 6.2 viene mostrato lo schema del test 2.

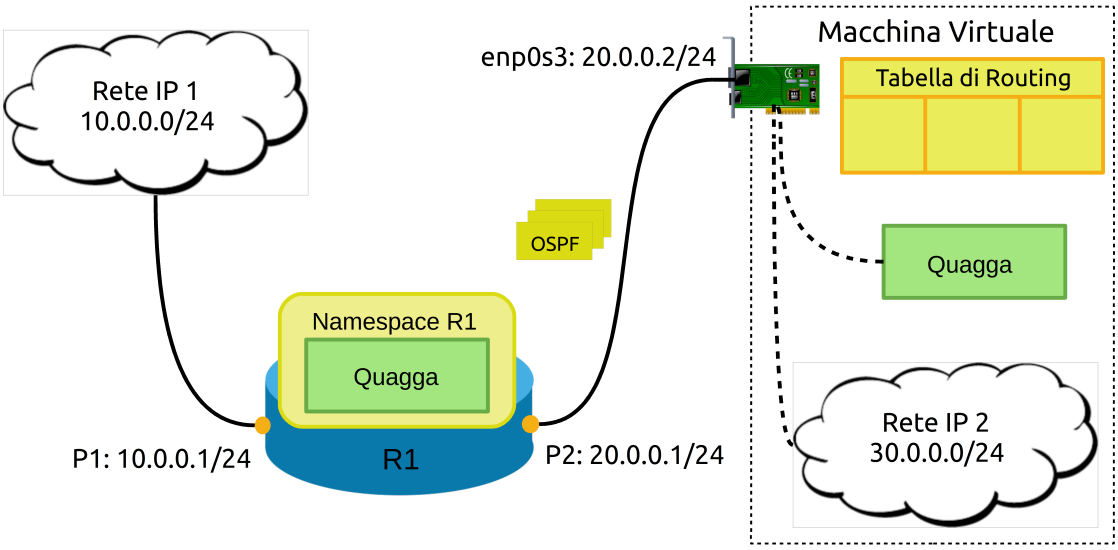


Figura 6.2: Test 2: Router Shadow con Quagga

## 6.4 Test 3

### 6.4.1 Cattura del Traffico

Nel terzo test è stato dimostrato che su uno Shadow Service è possibile utilizzare tools disponibili per Linux. Nello specifico si è provato a catturare il traffico sulle interfacce del Servizio utilizzando uno sniffer di pacchetti. I programmi testati nel test sono stati *Wireshark* [25] e *tcpdump*.

A differenza degli altri test questo ha richiesto una modifica al codice degli Shadow Service. Uno Shadow Service invia al suo namespace e quindi a Linux solo il traffico che non sa gestire nel fast path. Il traffico gestito dal codice eBPF del fast path si presume non debba mai arrivare a Linux per una questione di prestazioni. Nel caso di cattura del traffico da parte di uno sniffer tutto il traffico deve arrivare al namespace altrimenti lo sniffer catturerebbe solo parte dei pacchetti.

Duplicare i pacchetti che il Servizio è in grado di elaborare nel fast path per farli arrivare al namespace è una operazione sicuramente costosa, considerando che si tratterebbe della quasi totalità del traffico. Questa operazione va a pesare sulle prestazioni del Servizio ed è per questo che si è scelto di inserire un ulteriore flag all'interno del data model degli Shadow Service con lo scopo di lasciare decidere all'utente quando e su quale Servizio renderlo attivo.

Se l'utente intende (come in questo caso) attivare la cattura dei pacchetti per esempio utilizzando Wireshark, attiverà il flag e sarà consapevole che in quella modalità che chiameremo *Modalità di Debug* le prestazioni del Servizio non saranno ottimali.
Quando il flag della Modalità di Debug è attivo, posizionandosi nel namespace del Servizio e lanciando uno sniffer di pacchetti su una delle interfacce virtuali si è in grado di catturare tutto il traffico.

In Figura 6.3 vengono mostrati due casi d'uso in cui è stato possibile catturare il traffico su una qualsiasi delle porte P1, P2, P3 o P4.

### 6.4.2 Modalità di Debug

La *Modalità di Debug* è sicuramente comoda ed utile in molte situazioni, ma come è stato detto influisce sulle performance del servizio, per questo è importante attivarla solo quando serve e ricordarsi di spegnerla in caso contrario.

Per avere un'idea sull'impatto che ha sulle prestazioni sono stati eseguiti dei test. L'ambiente per i test si basa su due namespace, namespace1 e namespace2 collegati tra di loro tramite router Shadow o semplicemente attivando il forwarding IP su Linux.

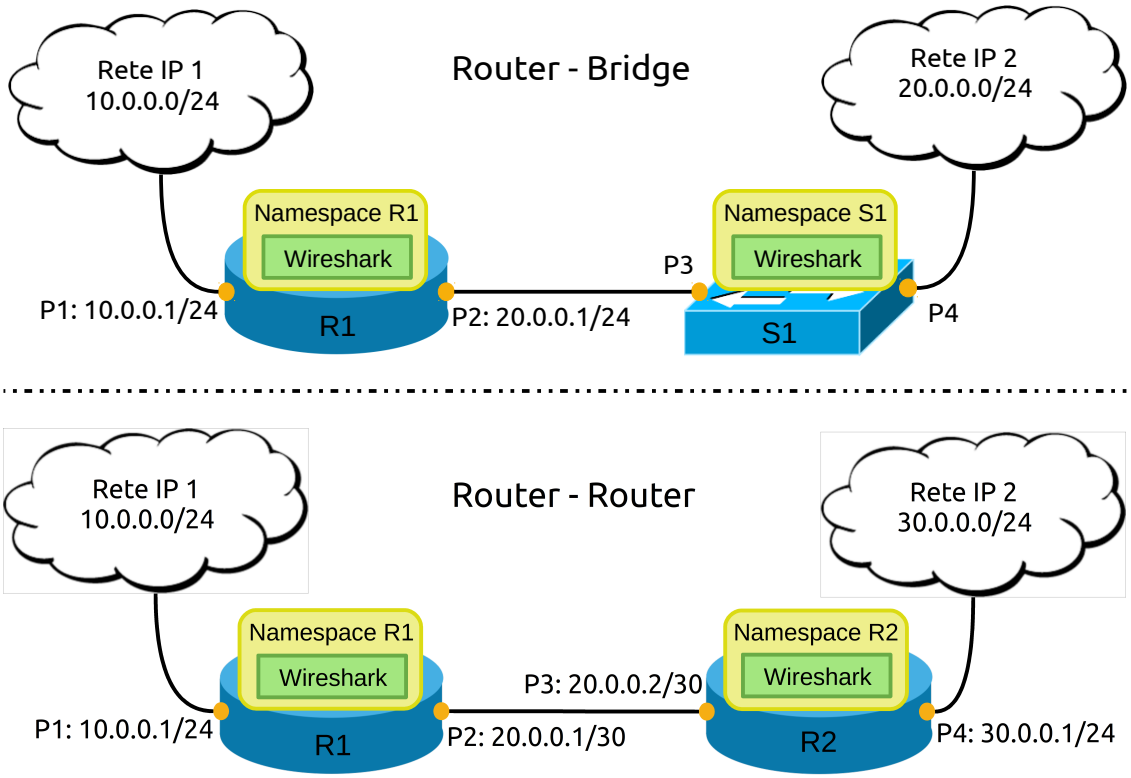


Figura 6.3: Test 3: Cattura del Traffico con Wireshark

1. La Figura 6.4 confronta un router Shadow in modalità di debug con un router Shadow avente la modalità di debug spenta. Il test esegue una sequenza icmp tra i due namespace.
   Si nota chiaramente come la modalità di debug influire sulle prestazioni dato che tutti i pacchetti icmp che potrebbero essere instradati in eBPF devono comunque passare dal namespace.

2. La Figura 6.5 basata sempre sul test che esegue una sequenza icmp tra i due namespace, confronta il caso di cattura del traffico (usando tcpdump) sulla porta chiamata per semplicità P1 del router Shadow collegata al namespace1 e la cattura del traffico sull'estremità della veth corrispondente in teoria alla porta P1 per il forwarding Linux.
   Anche in questo caso si nota come la cattura in modalità di debug sia più lenta rispetto alla cattura tradizionale.

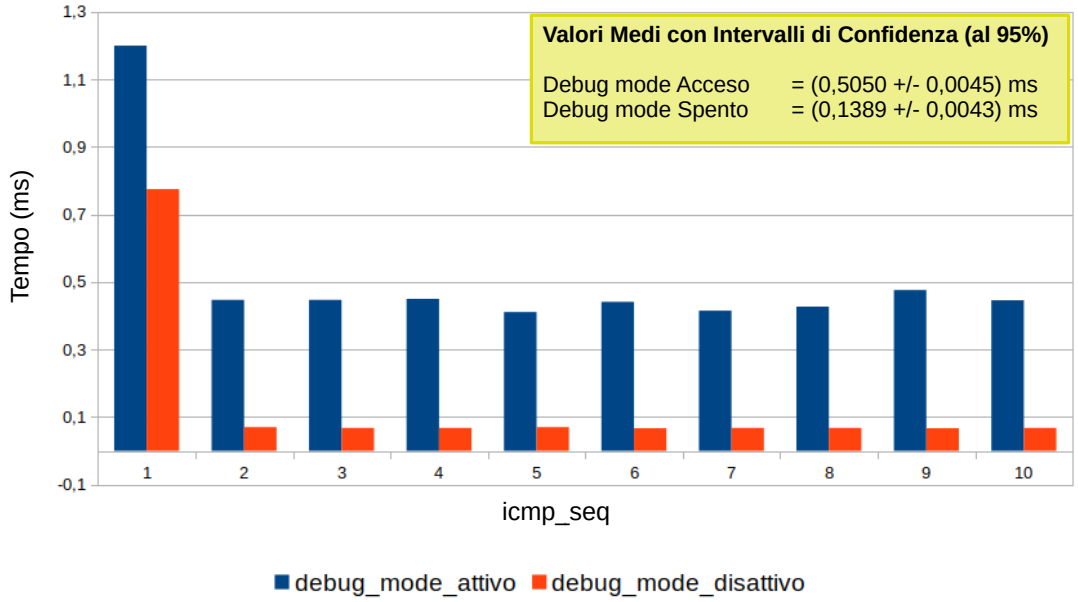


Figura 6.4: Ping tra due namespace collegati tramite router Shadow

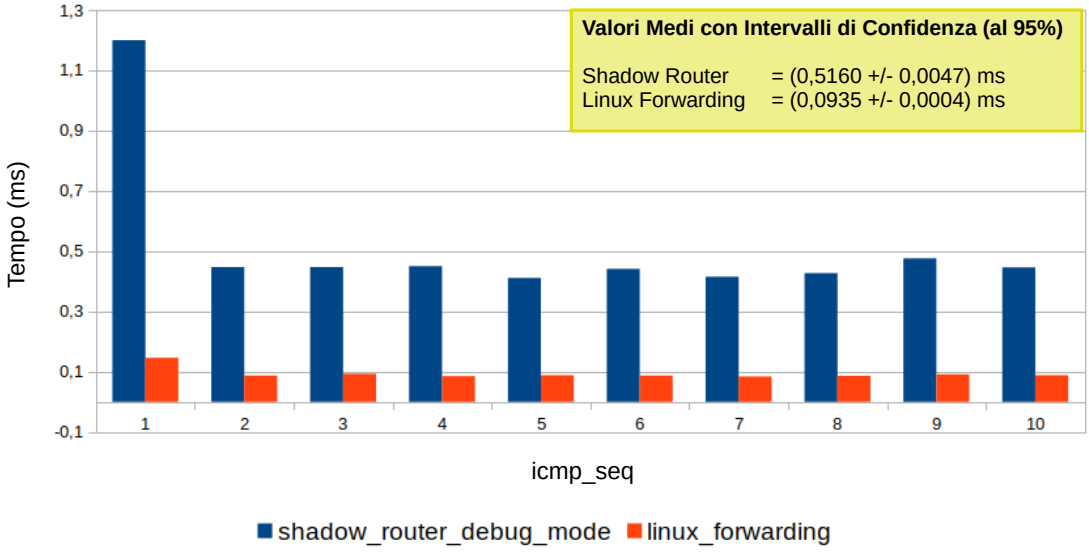


Figura 6.5: Confronto cattura del Traffico con Linux e con Shadow Service

Dai grafici salta fuori che la *Modalità di Debug* influisce sulle prestazioni di circa un fattore *5*, per questo è consigliato usarla solo quando strettamente necessario e tenerla spenta in caso contrario.

## 6.5 Prestazioni

Sebbene nel presentare il problema e nel trovare una soluzione non sia mai stata messa enfasi sulle prestazioni, a parte parlando della modalità di debug, si è sempre parlato di aggiungere funzionalità ai servizi Polycube che non fossero presenti fino ad oggi e di difficile implementazione.
Nell'aggiungere queste funzionalità non si è mai sottovalutato l'impatto che la soluzione potesse avere sui servizi e sull'architettura di Polycube in generale.

Soffermandosi sul fast path di un servizio e sapendo che è proprio il fast path a gestire la maggior parte dei pacchetti, e quindi di gran lunga il responsabile delle prestazioni del servizio stesso; si può analizzare il fast path di un servizio Shadow e fare un confronto con il fast path di un servizio non Shadow. I due sono praticamente identici cambiano solamente due cose:

1. I pacchetti provenienti da una porta dispari su uno Shadow Service vengono rediretti su quella pari senza essere processati.

2. I pacchetti che prima potevano essere solo diretti allo slow path adesso possono essere gestiti dallo slow path o rediretti a Linux.

Si può allora affermare che le modifiche apportate al fast path di uno Shadow Service non vanno ad influenzare le prestazioni del servizio, che continua a garantire le ottime performance derivanti dall'utilizzo della tecnologia eBPF.

### 6.5.1 Valori numerici

A supportare quanto detto sono state eseguite delle misurazioni sulla *latenza* e sul *bit-rate*, per fare il confronto tra servizi Shadow, servizi non Shadow ed eventuali alternative che non fossero basate su Polycube.

La macchina usata per eseguire tutti i test di performance è un Ubuntu 18.04.2 LTS (64-bit) con 32 GiB di memoria RAM, processore Intel Core i7-6700 e SSD di 256 GB.

#### Latenza

Il primo test sulle prestazioni ha come obiettivo misurare la latenza tra due namespace collegati tra di loro in tre modi diversi:

1. Namespace collegati attraverso un *pcn_router Shadow* .

2. Namespace collegati attraverso un *pcn_router non Shadow*.

3. Namespace collegati attraverso Linux.

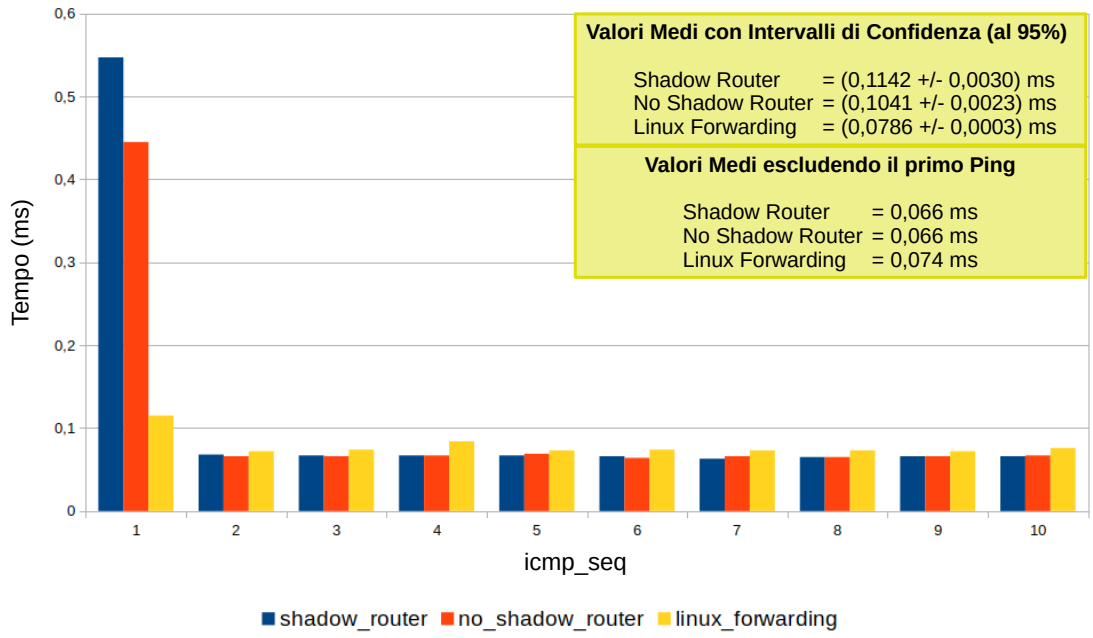


Figura 6.6: Latenza del comando Ping nei 3 casi di studio

In Figura 6.6 viene riportato un grafico con i risultati ottenuti eseguendo una sequenza di 10 richieste icmp tra i due namespace.

Da un primo sguardo alla Figura 6.6 si potrebbe pensare che in media il router Shadow abbia prestazioni peggiori del router non Shadow e che entrambi abbiano latenze più alte addirittura del semplice forwarding di Linux.

In realtà non è così, perché i valori sono falsati dalla prima sequenza icmp; si nota bene sul grafico che nel primo caso i due router sono più lenti rispetto al Linux forwarding.

Il perchè è semplice, il primo pacchetto *icmp request* necessita di essere preceduto da un *ARP request* con lo scopo di riempire la tabella ARP. L'ARP reply però viene generata in modo diverso nei tre casi:

- *Router Shadow* invia l'ARP request al suo namespace, il namespace genera l'ARP reply che passa dal fast path eBPF e viene inviata sulla porta d'uscita.
- *Router non Shadow* invia l'ARP request allo slow path nello userspace, lo slow path genera l'ARP reply e la invia direttamente sulla porta d'uscita.
- *Linux forwarding*, l'ARP reply e generata direttamente dall'altra estremità della veth garantendo in questo caso tempi minori.

Dopo aver risposto alle richieste ARP tutti i pacchetti che seguono, sia nel caso di router Shadow che nel caso di router non Shadow non dovranno più salire nel

namespace/slow path e saranno elaborati direttamente dal codice eBPF garantendo prestazioni migliori rispetto al Linux forwarding.

Soffermandosi sui valori della seconda tabella in Figura 6.6 dove vengono riportate le medie escludendo la prima sequenza icmp, si nota come il router Shadow e il router non Shadow abbiano le stesse performance e come entrambi riescano ad essere migliori del forwarding Linux grazie alla tecnologia eBPF.

### Bit-rate

Nel secondo test sulle prestazioni l'obiettivo è quello di confrontare i bit-rate raggiunti nei tre casi di studio presentati prima.

Per il test si è usato il tools *iperf3* che permette di lanciare un server nel namespace 1 e un client nel namespace 2; client e server si scambiano traffico IP (TCP o UDP) cercando di massimizzare l'uso delle risorse.

Più che i valori assoluti, i quali possono dipendere dalla macchina usata e da altri parametri che entrano in gioco, quello su cui si vuole porre l'attenzione è il confronto tra i vari servizi a parità di condizioni iniziali.

Il grafico in Figura 6.7 mostra chiaramente come il router Shadow e il router non Shadow si comportano esattamente allo stesso modo e come invece il forwarding Linux risulta essere meno performante.

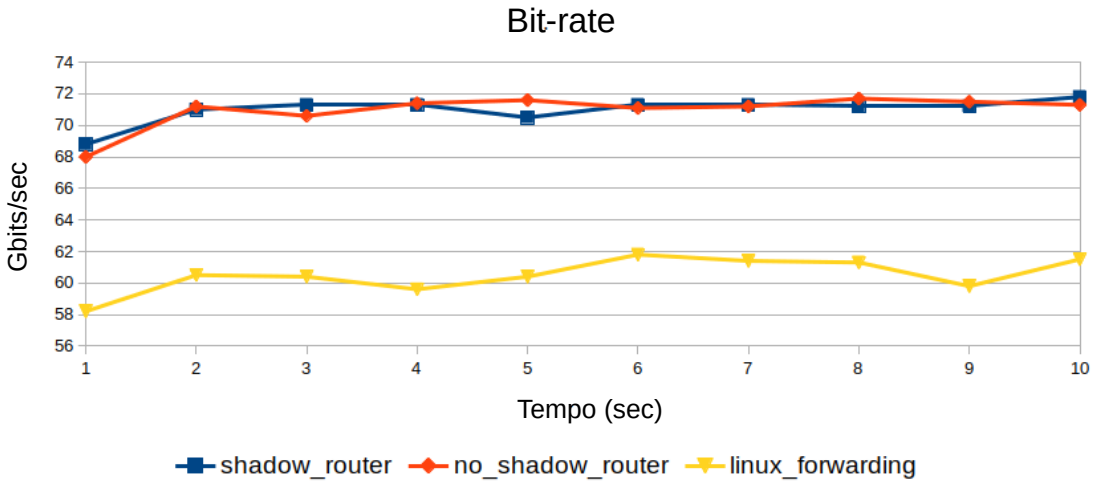


Figura 6.7: Bit-rate nei tre casi di studio

### Ping sull'Interfaccia del Servizio

Come ultimo test è stato eseguito un ping su una interfaccia del servizio; il confronto è stato fatto tra router Shadow, router non Shadow e ping su una interfaccia Linux. In questo caso il risultato ottenuto fa vedere come il router Shadow abbia le stesse prestazioni di Linux nel rispondere alle richieste icmp e come si ottengono prestazioni migliori rispetto al router non Shadow; questo è valido solo se il ping viene effettuato su una delle interfacce del servizio.

In altre parole il pacchetto di icmp reply generato da Linux è molto più veloce dello stesso pacchetto icmp reply generato dallo slow path del router non Shadow.

Il grafico in Figura 6.8 riporta i risultati ottenuti.

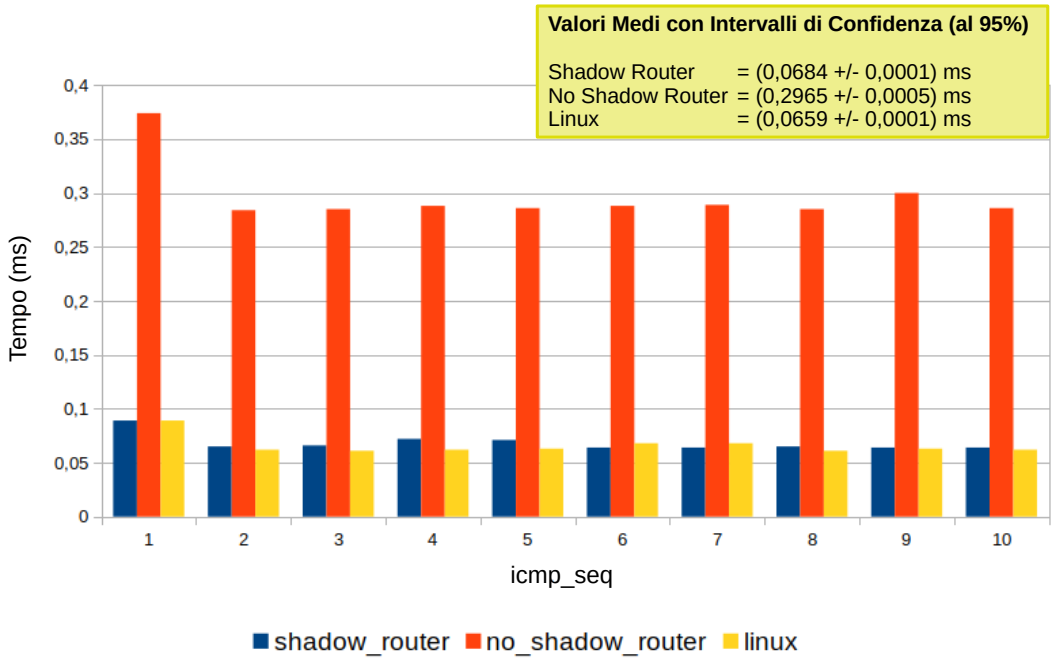


Figura 6.8: Ping su una porta del Servizio/Interfaccia Linux

# Capitolo 7

# Conclusioni

In questa tesi è stata definita una architettura che fosse in grado di estendere i concetti base di Polycube, esportarli al di fuori del framework stesso e renderli visibili a Linux.

È stato definito un nuovo tipo di Servizi, chiamati **Shadow Service**, in cui le porte e i rispettivi parametri sono accessibili e configurabili attraverso Linux usando dei network namespace.

L'architettura è stata pensata per trarre più vantaggi possibili dalla cooperazione tra Polycube e Linux. I vantaggi riguardano sia l'utente inteso come utilizzatore del framework Polycube che lo sviluppatore inteso come colui che usa il framework per scriversi una propria funzione di rete.

In generale è stato dimostrato che cooperare con Linux è possibile ma soprattutto risulta essere la soluzione a molti dei problemi che Polycube si portava dietro da tempo. La cooperazione apre quindi la strada a nuove funzionalità e crea la possibilità in futuro di essere utilizzata come base per estendere il framework in maniera significativa.

Una dimostrazione è stata ottenuta durante la validazione dell'architettura, dove è stato mostrato come i tools di Linux fossero in grado di gestire ed elaborare il traffico dei Servizi Polycube. Quelli sono stati solo alcuni esempi delle possibilità offerte dai nuovi Shadow Service, che in generale aprono la strada all'integrazione tra funzioni di rete basate su eBPF e il mondo Linux.

## 7.1 Sviluppi Futuri

In questa sezione vengono presentati dei possibili esempi di lavori futuri che basandosi sul lavoro di questa tesi potrebbero migliorare ed estendere alcuni aspetti lasciati ad oggi in sospeso.

### 7.1.1 Estensione CLI Polycube

Al momento si sa che il traffico che lo Shadow Service lascia gestire a Linux viene deciso dallo sviluppatore del servizio in fase di implementazione, che decide i protocolli da implementare nello Slow Path e quelli che invece lascia gestire al Sistema Operativo e ai vari tools. Non c'è un modo per l'utente di scegliere quali protocolli vuole che siano gestiti dal namespace del Servizio, e quindi ricevere quei pacchetti su Linux e quali invece no.

Fa eccezione il flag della Modalità di Debug, che viene usato per rendere visibile tutto il traffico del Servizio anche al namespace.

Questo fa pensare ad una probabile futura soluzione in cui invece di adottare un flag che serve a dire "o tutto o niente" si possa estendere il concetto e di conseguenza la CLI di Polycube per rendere il sistema più flessibile. Si pensi al caso in cui l'utente o un processo in esecuzione nel namespace possa utilizzare un comando e chiedere al Servizio solo il tipo di traffico che intende ricevere su Linux.

### 7.1.2 Collegamento tra Shadow Service e Interfacce Linux

Un aspetto che invece potrebbe essere migliorato in futuro riguarda il collegamento tra Shadow Service e interfacce Linux, siano queste fisiche o virtuali.

Al momento il collegamento tra uno Shadow Service e una interfaccia Linux avviene nello stesso modo in cui vengono collegate due porte, Figura 7.1.

Si può facilmente notare che questa soluzione porta ad avere non più due ambienti da mantenere allineati (il Servizio Polycube e il namespace del Servizio), ma si aggiunge anche il root namespace, l'ambiente in cui l'interfaccia Linux si trova.

Avere tre ambienti vuol dire avere tre punti diversi dove le informazioni possono cambiare, di conseguenza tre ambienti da monitorare e modificare.

Una soluzione alternativa potrebbe essere quella di chiedere a Polycube di spostare l'interfaccia Linux nel namespace del Servizio. Polycube potrebbe accorgersi quando una delle porte dello Shadow Service viene collegata ad una interfaccia del root namespace, di conseguenza sarebbe in grado di distruggere la veth interna tra namespace e porta nascosta e spostare l'interfaccia Linux nel namespace del Servizio. La Figura 7.2 aiuta a capire meglio i passaggi.

Questa soluzione alternativa crea un problema al gestore della macchina che vedrebbe le interfacce muoversi tra i namespace, senza che lui possa averne il controllo; il che rende questa soluzione non proprio perfetta.

In chiusura quindi si può pensare che esistono ancora margini di miglioramento prima di poter affermare che l'integrazione tra Polycube e Linux possa ritenersi conclusa.

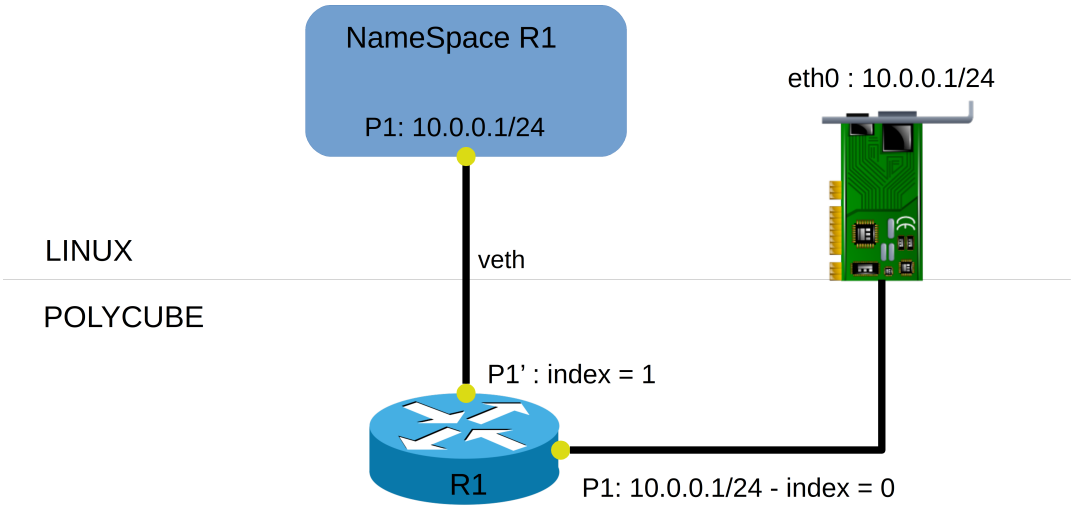


Figura 7.1: Collegamento tra uno Shadow Service e una Interfaccia Linux

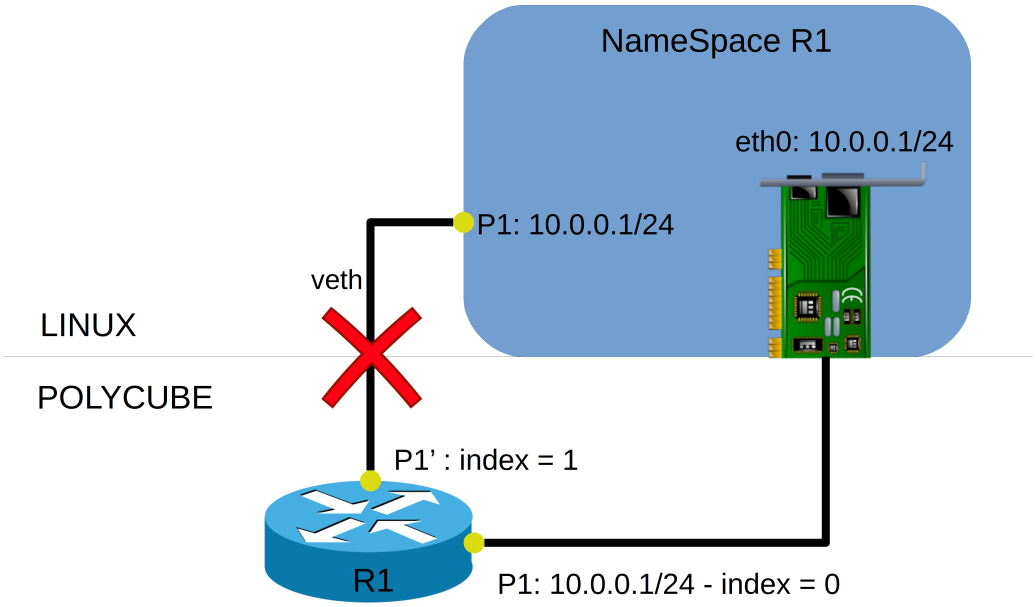


Figura 7.2: Collegamento alternativo tra uno Shadow Service e una Interfaccia Linux

# Bibliografia


[1] eBPF - extended Berkeley Packet Filter
https://prototype-kernel.readthedocs.io/en/latest/bpf/

[2] Exploring eBPF, IO Visor and Beyond
https://www.iovisor.org/blog/2016/04/12/exploring-ebpf-io-visor-and-beyond

[3] eBPF, part 1: Passato, Presente e Futuro
https://ferrisellis.com/posts/ebpf_past_present_future/

[4] Mappe eBPF
https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html

[5] Tipi di mappe eBPF
https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps_types.html

[6] Polycube
https://github.com/polycube-network/polycube

[7] Quagga Routing Suite
https://www.quagga.net/

[8] Dynamic Linux Routing with Quagga
https://www.linux.com/learn/intro-to-linux/2018/3/dynamic-linux-routing-quagga

[9] Introduction to Linux interfaces for virtual networking
https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/

[10] Tun/Tap interface tutorial
https://backreference.org/2010/03/26/tuntap-interface-tutorial/

[11] veth - Virtual Ethernet Device
http://man7.org/linux/man-pages/man4/veth.4.html

[12] YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
https://tools.ietf.org/html/rfc6020

[13] Introducing Linux Network Namespaces
https://blogs.igalia.com/dpino/2016/04/10/network-namespaces/

[14] namespaces - overview of Linux namespaces
http://man7.org/linux/man-pages/man7/namespaces.7.html

[15] Materiale corso di Reti Locali e Data Center
https://landc.frisso.net/syllabus
[16] Materiale corso Software Networking
https://swnet.frisso.net/syllabus
[17] OSPF Version 2
https://tools.ietf.org/html/rfc2328
[18] Resilienza
https://it.wikipedia.org/wiki/Resilienza
[19] Netlink Library (libnl)
https://www.infradead.org/ tgr/libnl/doc/core.html
[20] Why and How to Use Netlink Socket
https://www.linuxjournal.com/article/7356
[21] The Definition of a Socket
https://tools.ietf.org/html/rfc147
[22] Linux Netlink as an IP Services Protocol
https://tools.ietf.org/html/rfc3549
[23] Manipulating the Networking Environment Using RTNETLINK
https://www.linuxjournal.com/article/8498
[24] iproute2 wikipedia
https://en.wikipedia.org/wiki/Iproute2
[25] Wireshark website
https://www.wireshark.org/

# Ringraziamenti

Eccomi giunto ai ringraziamenti che inevitabilmente mi fanno ripensare, dove sono arrivato, come ci sono arrivato, ma soprattutto grazie a chi, ed è per questo che voglio ringraziare tutte le persone che mi hanno aiutato a raggiungere questo importante traguardo.

Inizierò con il ringraziare il mio professore Fulvio Risso, per aver accettato l'incarico di relatore, per l'aiuto, la disponibilità, il supporto e gli stimoli che insieme al mio correlatore Sebastiano mi hanno dato. Con la loro passione e la loro instancabile assistenza mi hanno permesso di ampliare le mie conoscenze e le mie vedute e ispirato a non accontentarmi mai, ma a migliorarmi sempre.

Un ringraziamento particolare va alla mia famiglia, a mamma e papà che hanno sempre creduto in me, per l'incoraggiamento e il supporto, per essere un punto di riferimento costante su cui poter sempre contare; ringrazio la mia sorellina Natascia che mi ha sempre supportato e sopportato ed è stata sempre al mio fianco tutto questo tempo non facendomi mai sentire solo; ringrazio il suo fidanzato Giovanni, che considero il fratello che non ho avuto; e ringrazio la mia fidanzata Sara per aver condiviso con me le esperienze più importanti, per avermi donato la spensieratezza necessaria e per essere stata al mio fianco tutte le volte che ne ho avuto bisogno.

Ringrazio i miei nonni, nonno Gabriele e nonna Lina che mi hanno sostenuto durante questi anni universitari e mi sono stati sempre vicini, permettendomi di affrontare quest'esperienza lontano da casa nel migliore dei modi.

Ringrazio i miei zii, in particolare zia Livia e zio Massimo per esserci stati sempre in qualsiasi momento e per qualsiasi cosa, e per avermi considerato come un loro figlio.

Infine, non per importanza, ci tengo a ringraziare i miei amici per aver condiviso con me le tante esperienze e per il tempo trascorso insieme.

Grazie a tutti.

*Francesco*