POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Prototipazione di servizi di rete distribuiti con eBPF

Relatore
prof. Fulvio Risso

Supervisore:
ing. Sebastiano Miano

Candidato
Francesco Picciariello

Aprile 2018

a mia nonna

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

Negli ultimi anni il mondo del networking è stato interessato dall'utilizzo crescente di funzioni di rete virtualizzate (NFV, *Network Function Virtualization*), precedentemente implementate sotto forma di hardware dedicato. L'impiego di tecniche di virtualizzazione consente di rendere il servizio di rete, e di conseguenza l'intera infrastruttura, più flessibile e semplice da controllare. In questo scenario, le VNF (*Virtual Network Function*) possono essere eseguite all'interno di macchine virtuali o di container Linux, potendo essere aggiunte o rimosse *on-demand*, a seconda delle richieste del sistema e del sovraccarico della rete. In secondo luogo, le VNF possono essere distribuite, ossia singole istanze allocate su server fisici differenti e che eseguono la stessa funzione di rete, possono collaborare nel processo di computazione sfruttando le interconnessioni per condividere dati e il proprio stato.

Inoltre, le ultime *release* del kernel di Linux hanno introdotto alcune funzionalità per consentire l'esecuzione di programmi lato kernel, in modo da monitorare la rete, tracciare eventi all'interno del kernel stesso, o implementare funzioni di sicurezza. BPF (*Berkley Packet Filter*) - successivamente eBPF (*Enhanced BPF*) - permette ai programmatori di sviluppare VNF e, più in generale, software *general purpose* eseguito a basso livello, migliorandone le performance.

Questo lavoro di tesi pone la sua attenzione sullo sviluppo di VNF distribuite e su eBPF, utilizzato sia per la creazione di VNF che per il monitoraggio lato kernel. In particolare, l'elaborato cerca di sfruttare appieno le potenzialità di eBPF per monitorare le VNF distribuite, al fine di centralizzarne lo stato e distribuirlo a tutte le istanze. Per garantire piena flessibilità, infatti, è necessario che ciascuna istanza di VNF sia a conoscenza dello stato associato a tutti i flussi di rete che interessano il sistema, al fine di poter operare non solo in maniera efficiente, ma anche efficace.

Capitolo 2

Presentazione dello scenario

Questo capitolo descrive le principali caratteristiche dei sistemi in cui si utilizzano funzioni di rete virtualizzate (NFV, e.g. *Virtual Network Function*) in architetture progettate per ospitare centri di elaborazione dati. Sono quindi presentate le principali applicazioni pratiche, le relative criticità e, infine, l'obiettivo di questo lavoro di tesi.

2.1 NFV e SDN

Nell'ambito del networking, quando si parla di **NFV** (Network Function Virtualization), ci si riferisce alla virtualizzazione di servizi di rete astratti dall'hardware dedicato. Il deploy di VNF sfrutta tipicamente server *general purpose* per eseguire la versione software del servizio di rete; alcuni esempi tipici sono router, firewall, load balancer e NAT [1].

Differentemente, per **SDN** (Software Defined Network) si intende la separazione del *control plane* della funzone di rete dal relativo *data plane*. La finalità del disaccoppiamento è quella di ottenere una rete programmabile gestita da un controllore centralizzato. L'applicazione tipica di questa tecnologia è l'uso in data center aziendali, per clienti che richiedono una rete che possa adattarsi più facilmente alle esigenze dell'azienda rispetto alle tradizionali architetture [1]. Un esempio rilevante in questo caso è OpenFlow[1], che è considerato uno dei primi standard per le SDN [2]

I due approcci si basano sull'astrazione della rete: le SDN separano le funzionalità di controllo dalle funzionalità di *forwarding*, mentre le NFV astraggono le funzioni di rete dall'hardware sottostante. Inoltre, una infrastruttura basata su SDN può andare a gestire delle funzioni di rete virtualizzate (NFV): quindi, le

[1] https://www.opennetworking.org

NFV forniscono funzioni di networking standard, mentre le SDN offrono controllo e orchestrazione per usi specifici.

2.1.1 Vantaggi e svantaggi

L'implementazione di funzionalità di rete via software, precedentemente installate sotto forma di hardware dedicati, porta i benefici derivanti dall'esecuzione in ambienti virtualizzati (macchine virtuali o container Linux) in particolar modo all'interno di data center. L'impiego delle tecniche di virtualizzazione permette, infatti, di rendere il software indipendente dall'hardware sottostante, le cui specificità vengono mascherate dal sistema di virtualizzazione. Questo consente di [3]:

- ottimizzare l'uso delle risorse attivando sullo stesso server fisico più server virtuali che implementano diverse tipologie di servizio, in modo da sfruttare appieno la capacità disponibile e ridurre il consumo energetico;
- ampliare o ridurre in modo dinamico la capacità allocata in base al carico effettivo. Ciò può essere ottenuto incrementando o riducendo le risorse assegnate ad ogni container o variando il numero dei container che realizzano una specifica funzione;
- garantire alta affidabilità, in quanto a fronte di un malfunzionamento hardware le componenti software possono essere spostate da un server fisico all'altro;
- riconfigurare la topologia della rete in tempo quasi reale per ottimizzarne le prestazioni e/o estenderne la distribuzione locale.

Allo stesso tempo, è necessario sottolineare che ci sono alcuni punti di criticità a cui rivolgere particolare attenzione [3]:

- le funzioni di rete virtualizzate non possono essere un semplice porting del software oggi utilizzato su hardware proprietario, ma vanno progettate per funzionare in ambiente cloud e sfruttarne appieno le caratteristiche;
- l'operatore si deve dotare di una piattaforma di orchestrazione per automatizzare la gestione del ciclo di vita delle VNF, al fine di evitare interventi manuali sull'infrastruttura.

2.2 Computazione distribuita

Con il termine *calcolo distribuito* si indica un'applicazione dell'informatica in cui le componenti del sistema sono indipendenti e dialogano attraverso connessioni di

rete. L'interazione tra le varie componenti consente a quest'ultime di collaborare nella computazione in modo da raggiungere un obiettivo comune [4]. Un software eseguito in un sistema distribuito è detto *software distribuito*, e la programmazione distribuita è il processo con cui si realizzano i suddetti programmi. In particolare, una sfida perseguita da molti sviluppatori è stata quella di ottenere la *trasparenza della posizione*, ossia la possibilità di identificare risorse di rete a prescindere dalla loro posizione fisica, ma esclusivamente in base al loro nome [5]. Tuttavia, questo obiettivo non ha avuto applicazioni pratiche poiché i sistemi distribuiti sono differenti dai sistemi convenzionali, e le differenze come partizionamento della rete, guasti parziali del sistema e aggiornamenti parziali, non possono essere semplicemente cancellati dai tentativi di trasparenza [6].

2.2.1 Applicazioni

Ci sono due principali ragioni per usare sistemi distribuiti e il calcolo distribuito:

- La natura dell'applicazione può richiedere l'uso di un network di comunicazione che colleghi più computer. Ad esempio, i dati sono prodotti in una certa locazione fisica e sono richiesti in un'altra locazione;
- In molti casi l'uso di un singolo nodo computazionale è fattibile in linea di principio, ma l'uso di un sistema distribuito è vantaggioso per motivi pratici. Per esempio, può essere più efficiente ottenere il livello prestazionale desiderato utilizzando un cluster di diversi computer di fascia bassa, anziché utilizzare un singolo nodo più potente ma che a questo punto sarebbe uno SPoF (*Single Point of Failure*). Inoltre, un sistema distribuito può essere più facile da espandere in confronto a un sistema classico [7].

Di seguito, i principali sistemi ed applicazioni di calcolo distribuite [8]:

- Reti di telecomunicazioni
- Applicazioni di rete
- Controllo in tempo reale
- Calcolo parallelo

2.3 Distribuzione dello stato

L'applicazione di tecniche di virtualizzazione nell'ambito del networking permette di rendere il sistema più flessibile, offrendo la possibilità di spostare una VNF su una macchina fisica differente, oppure di lanciare una nuova istanza in base alle richieste del sistema. In aggiunta a tutto ciò, sfruttando le potenzialità offerte da SDN e

NFV, è possibile realizzare sistemi distribuiti, ossia sistemi in cui le singole istanze di un particolare servizio di rete collaborano in modo da ottenere un incremento globale delle performance.

Si consideri ad esempio uno scenario in cui un IPS (*intrusion prevention system*) sia altamente sovraccarico e necessiti di essere scalato orizzontalmente in modo da effettuare il processing in tempistiche accettabili. In questo caso, sarebbe possibile lanciare una nuova istanza dell'IPS in modo che possa partecipare alla gestione del traffico e attenuare l'incremento dei tempi di latenza causati dal sovraccarico. Tuttavia, per garantire che la nuova istanza blocchi il traffico sospetto in modo corretto, è necessario assicurarsi che quest'ultima sia a conoscenza dello stato associato ai flussi già esistenti [9].

Si deduce, quindi, che uno dei problemi principali nel deploy flessibile di VNF risiede nel fatto che la maggior parte delle funzioni di rete sono *stateful*, ossia necessitano di informazioni raccolte a runtime per poter svolgere il proprio compito [10].

In generale, è possibile suddividere la struttura che caratterizza lo stato delle VNF in due classi differenti. La prima classe contiene lo *stato globale*, a cui si accede indipendentemente dal traffico processato. La seconda classe contiene lo *stato partizionato*, che consiste in frammenti di stato direttamente correlati a uno o più flussi o sessioni processati dal servizio di rete. Questi frammenti possono essere identificati dalle stesse informazioni usate per identificare singoli flussi o sessioni. Per connessioni IP, ad esempio, si avrà la presenza della quintupla formata da IP sorgente, IP destinazione, porta sorgente, porta destinazione, protocollo. Tipicamente, la maggior parte dello stato è rappresentato da questa seconda classe [11], [12].

2.4 Obiettivo della tesi

Questa tesi propone una nuova architettura basata sull'integrazione, all'interno della VNF, di una componente software che esegua il monitoraggio in tempo reale dello stato della funzione di rete stessa. Questa componente sfrutterà la tecnologia eBPF (introdotta nella sezione 4.1) per garantire performance, flessibilità, e disaccoppiamento dalle operazioni svolte dalla VNF. A questa componente si aggiunge l'introduzione all'interno del sistema di una base dati efficiente ed affidabile, che possa centralizzare lo stato di tutte le istanze dello stesso servizio di rete. La caratteristica principale della base dati è la possibilità di essere distribuita; offre cioè la possibilità di definire un cluster e di avere una gestione automatizzata della consistenza. Oltre alla centralizzazione dello stato, l'architettura prevede che questo sia successivamente distribuito a tutte le VNF, in modo che ciascuna di esse abbia la visione complessiva del sistema.

Questo tipo di approccio consente di non modificare in alcun modo il *data plane* della VNF, ma di intervenire esclusivamente sul piano di controllo. Ciò è possibile grazie alla parallelizzazione del piano dati e del monitoraggio dello stato,

che possono operare indipendentemente e in modo non concorrenziale. Inoltre, per quanto concerne la distribuzione e centralizzazione dello stato tramite l'utilizzo di una base dati, si annoverano i seguenti vantaggi:

- Maggior flessibilità del sistema, con la possibilità di spegnere, accendere e rilocare VNF stateful (e.g. per ottenere fault tolerance, fast recovery o scalabilità orizzontale) che potranno iniziare subito a operare abbattendo i tempi di latenza dati da una eventuale fase di apprendimento;
- Assenza di un sistema di orchestrazione che gestisca i vecchi flussi e i nuovi flussi in modo da sopperire alla mancanza dello stato nella VNF appena avviata;
- L'istanza della VNF non dovrà implementare nessun protocollo di consenso distribuito, essendo a conoscenza della situazione globale durante tutta la sua esecuzione, ed essendo questo demandato alla base dati.

Dopo aver esposto i principali lavori correlati, nei prossimi capitoli saranno presentate le tecnologie necessarie per realizzare questo elaborato, l'architettura proposta e le principali caratteristiche. Infine, sarà presentato un prototipo con lo scopo di testare la soluzione in un caso d'uso reale e apprezzarne, in termini numerici, i vantaggi e i limiti.

Capitolo 3

Lavori correlati

In questa sezione saranno presentati i principali approcci per quanto concerne la gestione dello stato di VNF in sistemi distribuiti.

3.1 OpenNF

La soluzione proposta da OpenNF [13] consiste nella definizione di un piano di controllo che permette di migrare lo stato interno della VNF, per specifici flussi, verso una nuova istanza. In particolare, l'architettura proposta si fa carico delle seguenti problematiche:

- **Concorrenza**. Questo è il primo aspetto da considerare quando si effettua una riallocazione di flussi in corso: quando lo stato interno di una VNF sta subendo la migrazione, i pacchetti potrebbero arrivare all'istanza sorgente, oppure potrebbero arrivare all'istanza di destinazione prima che il processo sia giunto a termine.

  Per risolvere questa complicazione, OpenNF introduce due nuovi costrutti:

  - Astrazione degli eventi in modo da osservare esternamente e prevenire i cambiamenti locali all'interno delle VNF. Questo significa che il controller può assicurare che le operazioni di migrazione siano *loss-free* e che la copia dello stato sia *eventually consistent*.
  - Schema a due fasi per l'aggiornamento dello stato della rete. Realizzato bufferizzando gli eventi corrispondenti agli aggiornamenti e gestendoli uno alla volta in concomitanza con la copia progressiva dello stato. In questo modo il controller può garantire che le copie siano consistenti.

- **Limitazione del sovraccarico**. Consiste nell'assicurare che la rilocazione avvenga in maniera efficiente. Spostare e condividere lo stato ta VNF richiede sia CPU che risorse di rete. Inoltre, evitare perdite, riordinamento e stati

inconsistenti comporta la bufferizzazione dei pacchetti, che introduce latenza e consumo di memoria. Se questo consumo di risorse fosse illimitato, non sarebbe possibile garantire determinati livelli di performance (e.g. disponibilità, tempo di risposta ridotto) e rispetto dei vincoli sui consumi operativi. Per limitare il sovraccarico, OpenNF propone delle *northbound API* flessibili che consentono di specificare esattamente quale stato migrare, copiare o condividere, e con quali garanzie (e.g. loss-free).

- **Ospitare VNF introducendo modifiche minime**. Il problema finale consiste nel garantire che il framework sia in grado di adattarsi a una vasta gamma di VNF in maniera non invasiva. Per questo, è stata progettata una *southbound API* per le VNF che consente al controller di richiedere l'esportazione o l'importazione dello stato senza modificare il modo in cui le VNF lo gestiscono internamente.

3.1.1 Architettura

OpenNF è una nuova architettura per il piano di controllo (e.g. Figura 3.1) che soddisfa i requisiti e gli obiettivi sopra citati. Il framework permette alle applicazioni di controllo di gestire da vicino il comportamento e le performance delle VNF, in particolare:

- Determinare il preciso set di flussi che una specifica VNF dovrebbe processare;
- Dirigere il controller in modo che fornisca lo stato a ciascuna istanza, includendo sia lo stato relativo al flusso specifico che lo stato condiviso tra più flussi;
- Richiedere al controller di fornire determinate garanzie circa le operazioni relative allo stato.

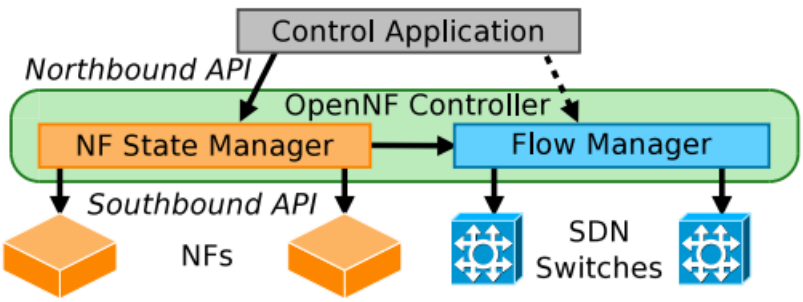

Figura 3.1. Architettura di OpenNF [13]

3.1.2 Southbound API

Il design della Southbound API è stato concepito per garantire che un vasto numero di VNF possano essere facilmente integrate in OpenNF. Per soddisfare questo requisito è stato necessario affrontare due sfide principali:

- Tenere in considerazione le differenze degli stati delle VNF
- Minimizzare le modifiche apportate alle VNF

Classificazione di stato

Il primo punto è stato affrontato identificando le similarità in come gli stati interni dei vari servizi di rete sono allocati e letti. Si è notato che lo stato creato o aggiornato da una NF durante l'elaborazione del traffico può essere applicato sia a un singolo flusso (e.g. connessione TCP) che a un insieme di flussi. Di conseguenza, è stata fatta una classificazione dello stato basata sullo *scope*, ossia a quanti flussi si applica lo stato creato dalla NF: *per-flow*, *multi-flow* o *all-flow*. In particolare:

- per-flow indica uno stato che fa riferimento a strutture/oggetti che sono lette o aggiornate solo quando si stanno processando pacchetti dello stesso flusso (e.g. connessione TCP);
- lo stato multi-flow è letto o aggiornato quando si processano pacchetti appartenenti a flussi multipli (ma non tutti);

Importare e esportare lo stato

Questa distinzione aiuta a ragionare su come l'applicazione dovrebbe muovere/-copiare/condividere lo stato. Ad esempio, il controller che instrada tutti i flussi destinati a un host H verso una specifica istanza di VNF, può assumere che l'istanza necessiterà dello stato per-flow per i flussi destinati a H, e di tutti gli stati multi-flow che conservano informazioni circa uno o più flussi destinati a H.

Avendo definito il concetto di scope (per-flow, multi-flow e all-flow), è stato possibile progettare API semplici per importare ed esportare parti di stato. In particolare, OpenNF fornisce tre funzioni per ogni scope: *get*, *put* e *delete*. In generale:

- La funzione get accetta come parametro il *filter*, ossia un valore specifico che uno o più campi dell'header devono assumere (e.g. IP sorgente/destinazione, protocollo di rete, porta sorgente/destinazione).
- La funzione put accetta *flowid* e *chunk*, che sono rispettivamente l'identificativo univoco del flusso e una o più strutture dati interne alla NF.
- La funzione delete accetta come parametro il *flowid*

L'unica eccezione è rappresentata dalle funzioni all-flows, che non accettano il filtro in quanto fanno riferimento allo stato relativo a tutti i flussi.

Osservare e prevenire aggiornamenti di stato

Le API descritte sino ad ora non intervengono né sulla creazione né sull'accesso dello stato. Tuttavia, ci sono situazioni in cui è necessario impedire a un'istanza di NF di aggiornare il proprio stato (e.g. durante la migrazione). Per questo motivo OpenNF usa due meccanismi per prevenire e osservare gli aggiornamenti:

- nel caso in cui una NF generi eventi del tipo *pacchetto ricevuto*, il controller comunica alla funzione di rete qual è il sottoinsieme di pacchetti che possono scatenare il suddetto evento;
- successivamente, controllando quale azione dovrebbe compiere la NF sui pacchetti che hanno generato l'evento (e.g. processarli, bufferizzarli o eliminarli).

3.1.3 Northbound API

La Northbound API consente all'applicazione di controllo di muovere, copiare o condividere sottoinsiemi di stati in maniera flessibile tra istanze di NF, e di richiedere importanti garanzie come *loss-freedom*, preservazione dell'ordine e diversi livelli di consistenza. OpenFN mette a disposizione le operazioni di `move`, `copy` e `share`. Tutte e tre le operazioni sfruttano le chiamate messe a disposizione dalla Southbound API.

Move

Questa operazione trasferisce sia lo stato che l'input (e.g. il traffico) relativi a un insieme di flussi da una istanza di NF a un'altra. La funzione accetta i seguenti parametri:

- *srcInst*: istanza sorgente di cui si vuole migrare lo stato;
- *dstInst*: istanza destinazione su cui si vuole migrare lo stato;
- *filter*: definisce l'insieme di flussi che si vogliono migrare;
- *scope*: specifica quali classi di stato si vogliono migrare (e.g. per-flow e/o multi-flow);
- *properties*: indica se lo spostamento dello stato debba preservare l'ordine ed essere senza perdita.

Copy e Share

L'operazione di copia clona lo stato da una istanza di NF in un'altra, e accetta come parametri *srcInst*, *dstInst*, *filter* e *scope*. La copia è utile quando non è richiesta la consistenza dello stato, oppure è sufficiente la consistenza finale.

La funzione share è utilizzata quando sono richieste consistenza forte o stretta, più difficili da garantire in quanto le letture e gli aggiornamenti dovrebbero verificarsi, in ciascuna NF, nello stesso ordine globale. In entrambi i casi i parametri accettati dalla funzione sono *list<inst>*, *filter*, *scope* e *consistency*. Dove *list<inst>* rappresenta l'insieme delle istanze che devono sincronizzarsi su letture/aggiornamenti, mentre *consistency* può assumere i valori di `strong` o `strict`.

3.1.4 Valutazione

I principali vantaggi di OpenNF sono:

- Migrazione dello stato di VNF in maniera coordinata e con un basso impatto in termini di quantità di dati trasferiti;
- Scelta delle proprietà relative alla migrazione, in termini di assenza di perdite, assenza di reordering e livelli di consistenza;
- Possibilità di invocare le Northbound API a vari livelli di granularità;
- Richiesta modesta di modifiche al codice della VNF, di cui buona parte può essere generata automaticamente.

Tuttavia, si annoverano i seguenti svantaggi:

- Incremento dei tempi di latenza in caso di migrazione con assenza di perdite e un numero crescente di *packet rate*;
- Incremento del numero di pacchetti persi in caso di operazioni senza garanzie;
- Per quanto concerne la scalabilità, il tempo medio per migrazioni senza perdita incrementa linearmente sia in caso di `move` simultanee che all'aumentare dei flussi migrati. Il collo di bottiglia è dovuto sostanzialmente al controller che bufferizza pacchetti durante il trasferimento dello stato. Questo processo comporta alte latenze sui pacchetti che arrivano durante la migrazione e un significativo consumo di CPU e memoria da parte del controller.

3.2 CoGS

La proposta di CoGS (*Coordinator for Global States*) [14] consiste in un framework in grado di scalare lo stato globale su più *middlebox*[1]. Ciascuna middlebox mantiene gli stati per-flow necessari per la computazione locale. CoGS fornisce una gestione generalizzata in modo da conservare lo stato globale richiesto da più middlebox durante il processamento dei pacchetti. Allo stesso tempo, il framework consente alle middlebox di accedere agli stati globali e di processare pacchetti in base ad essi. L'implementazione mette a disposizione un set di API per l'accesso agli stati globali durante il processamento dei pacchetti.

3.2.1 Architettura

Il framework è costituito da un coordinatore, che gestisce lo gli stati globali, e un certo numero di middlebox che sfruttano il coordinatore per gestire il proprio stato. Gli stati globali possono essere organizzati in una tabella costituita da coppie chiave/valore:

- la chiave identifica lo stato globale;
- il valore è lo stato necessario alla middlebox durante il processamento del pacchetto.

A seconda di come è gestita la tabella degli stati globali, si parla di:

- *stati globali centralizzati*, ossia la tabella è mantenuta interamente all'interno del coordinatore. Ciascuna middlebox è in grado di leggere o aggiornare lo stato facendo una richiesta allo *state manager* (e.g. Figura 3.2);
- *stati globali distribuiti*, se la tabella è suddivisa su tutte le middlebox. Ciascuna middlebox contiene un sottoinsieme degli stati globali, mentre il coordinatore possiede una tabella con i puntatori alle middlebox che contengono un determinato stato (e.g.Figura 3.3).

CoGS suddivide inoltre gli stati globali nelle seguenti tre categorie:

- **Trigger state.** Per gli stati globali a cui si accede secondo il pattern *check-update* (e.g. analisi del traffico per monitoraggio o rilevamento di intrusioni), è essenziale riassumere tutte le operazioni di aggiornamento dello stato da più middlebox e rappresentarle negli stati globali in maniera consistente tra tutte le middlebox. Essendo l'aggiornamento un'operazione frequente, mantenere un certo numero di repliche è un'operazione costosa, di conseguenza questi tipi di stati globali sono mantenuti in maniera centralizzata.

[1] https://tools.ietf.org/html/rfc3234

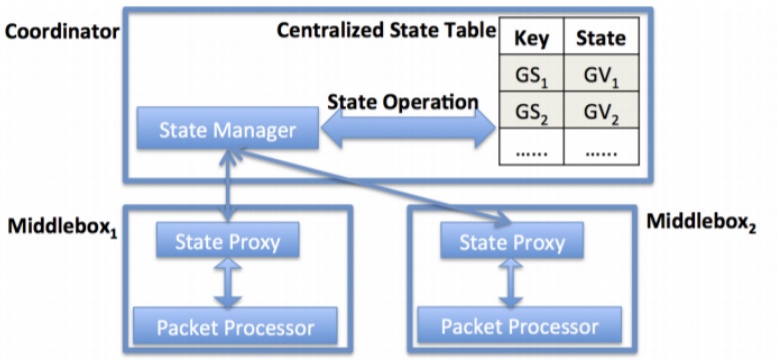

Figura 3.2. Stati globali distribuiti in CoGS [14]

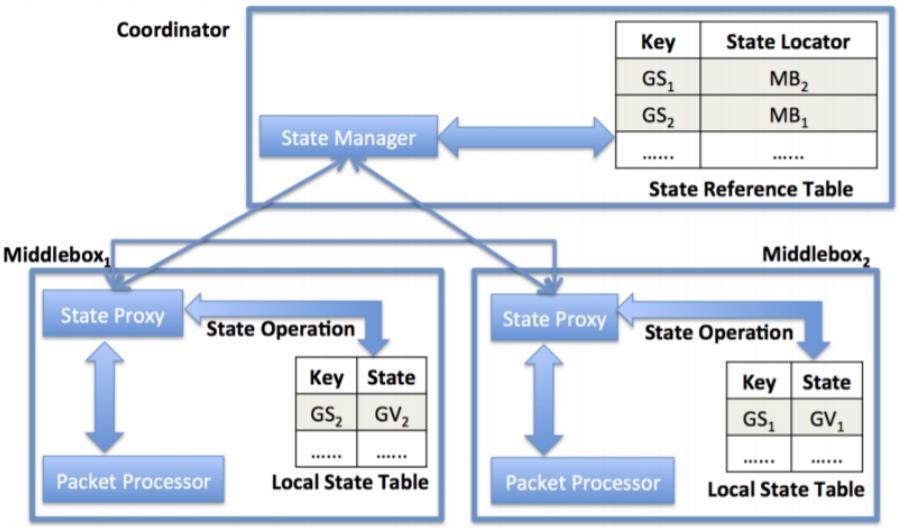

Figura 3.3. Stati globali distribuiti in CoGS [14]

- **Direct-retrieval state.** Gli stati globali a cui si accede secondo il pattern *access-modify* (e.g. funzioni di manipolazione del traffico) possono essere centralizzati. Ciascuna middlebox li recupera dal coordinatore per il packet processing.

- **Indirect-retrieval state.** Nel caso in cui gli stati globali siano troppo grandi per poter essere salvati all'interno delle middlebox, non è ragionevole usare un approccio centralizzato. Di conseguenza, questi tipi di stato sono distribuiti tra le middlebox. Per leggere lo stato, una middlebox prima legge il puntatore dal coordinatore, per poi richiedere lo stato globale alla locazione in cui è salvato.

Trigger state

Le middlebox aggiornano lo stato durante il processing del pacchetto. Al'interno del coordinatore, il gestore dello stato aggiorna o crea gli stati: se la middlebox aggiorna uno stato già esistente, il coordinatore aggiorna lo stato, altrimenti crea una nuova entry nella tabella degli stati.

Direct-retrieval state

Questo tipo di stato è aggiornato/letto da più middlebox per definire il processamento del pacchetto. Il coordinatore mantiene tutti gli stati di questo tipo ed esegue le operazioni di lettura e aggiornamento su di essi. Le operazioni di aggiornamento eseguono la modifica e ritornano il nuovo stato, mentre le operazioni di lettura ritornano lo stato senza nessun cambiamento. La middlebox esegue le suddette operazioni e attende il valore di ritorno per poter proseguire col processamento.

Indirect-retrieval state

Quando una NF possiede stati globali di grandi dimensioni, CoGS usa la strategia di recupero indiretto dello stato. Il coordinatore mantiene esclusivamente una tabella dei riferimenti, mentre gli stati sono distribuiti tra le varie middlebox. È tipico che uno stato di questo tipo sia letto e aggiornato in maniera remota, invece che sulla stessa middlebox su cui lo stato è memorizzato.

3.2.2 Valutazione

I test eseguiti su CoGS mostrano:

- un aumento del throughput in maniera lineare rispetto all'incremento delle middlebox;
- alta velocità di processing del coordinatore negli stress test;
- tuttavia, all'aumentare delle middlebox la velocità di processing non cresce in maniera lineare ma mostra un andamento logaritmico;
- il limite più evidente di CoGS è che gestisce gli stati globali della stessa NF, e non fornisce quindi un framework uniforme per gestire NF differenti.

3.3 Approccio stateless

Questa proposta prevede il disaccoppiamento della componente di processing della VNF e del relativo stato (e.g. Figura 3.4). Spezzando questo accoppiamento stretto è possibile ottenere una rete più elastica ed efficiente [15].

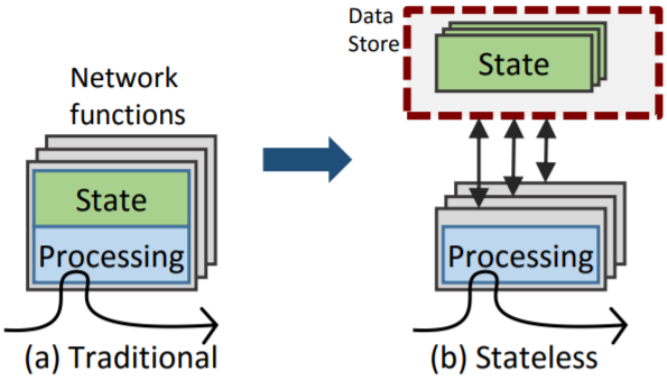

Figura 3.4. Approccio stateless [15]

Il disaccoppiamento permette di garantire determinate proprietà:

- **Resilienza**. È possibile infatti avviare istantaneamente una nuova istanza in seguito a un failure, poiché quest'ultima avrà accesso a tutto lo stato necessario.

- **Elasticità**. Quando si scala orizzontalmente, con funzioni stateless, può essere lanciata una nuova istanza di VNF e il traffico può essere immediatamente deviato verso di essa. Allo stesso modo, se si vuole ridurre il numero di istanze, è necessario redirigere altrove il traffico destinato alla VNF che dovrà essere spenta.

- **Routing asimmetrico**. In un approccio stateless ciascuna istanza condivide tutto lo stato, quindi la correttezza delle operazioni non dipende dalla specifica istanza che intercetta il traffico. Questo modello, infatti, assume che ogni pacchetto possa essere gestito da una qualsiasi VNF.

Nonostante i suddetti benefici, questo tipo di soluzione presenta alcuni svantaggi:

- Necessità di un data storage resiliente e a bassa latenza, essendo il centro di raccolta di tutto lo stato del sistema. La caratteristica della resilienza viene affrontata con la ridondanza dei dati, e un conseguente alto consumo di RAM.

- Necessità di interconnessioni ad alte performance, poiché queste sono utili per mettere in contatto le VNF con lo storage centrale, e quindi con lo stato.

- Latenza inevitabilmente introdotta dall'interazione con lo storage remoto.

3.4 Differenze con la soluzione proposta

Per un'analisi completa delle principali differenze tra le soluzioni sopra citate e la proposta di questa tesi, si rimanda alla sezione 5.3.

Capitolo 4

Tecnologie utilizzate

Questo capitolo presenta le principali tecnologie e gli strumenti utilizzati per definire l'architettura e perseguire gli obiettivi precedentemente illustrati.

4.1 Berkley Packet Filter (BPF)

BPF è una macchina virtuale altamente flessibile ed efficiente eseguita nel kernel di Linux, con la caratteristica principale di consentire l'esecuzione in modo sicuro di *bytecode* - appositamente iniettato e interpretato dalla macchina virtuale stessa - in vari punti di aggancio. È utilizzata in molti sottosistemi del kernel, maggiormente per networking, tracing e sicurezza. La prima implementazione risale al 1997, quando è stata introdotta nella versione 2.1.75 del kernel di Linux [16].

L'idea alla base di BPF è quella di spostare il filtro dallo user-space verso il kernel-space, in modo tale da eseguire il processing molto più efficientemente. Grazie a questo approccio è possibile scartare pacchetti irrilevanti già nel kernel senza doverli copiare nello user-space, con dei guadagni in termini di performance soprattutto in modalità promiscua.

Alcuni esempi concreti dell'uso di BPF sono tcpdump[1], libpcap[2], wireshark[3] e nmap[4]. Tutti questi tool usano la macchina virtuale BPF in modo da eseguire filtering di pacchetti nel kernel efficientemente.

```
user@linux$ tcpdump -d ip
(000) ldh      [12]
(001) jeq      #0x800       jt 2    jf 3
(002) ret      #96
```

[1] https://www.tcpdump.org/tcpdump_man.html

[2] http://www.tcpdump.org/manpages/pcap.3pcap.html

[3] https://www.wireshark.org/

[4] https://nmap.org/

```
(003) ret      #0
```

Questo esempio di BPF bytecode generato da tcpdump istanzia un filtro che invia allo spazio utente solamente pacchetti di tipo IP. La logica prevede il caricamento dell'ethertype all'interno di un registro e la verifica del match con 0x800, che è l'ethertype di un pacchetto IP. In caso di match positivo il pacchetto è inoltrato verso lo spazio utente, altrimenti è ignorato dal filtro.

4.1.1 eBPF

Negli ultimi anni BPF ha subito una profonda evoluzione, con l'introduzione di nuove strutture dati come *hash table* e *array* per conservare lo stato della computazione e poter eseguire operazioni addizionali sui pacchetti, come ad esempio manipolazione, forwarding e incapsulamento. Con eBPF (extended BPF) i programmatori possono sfruttare una macchina virtuale in grado di eseguire programmi *general purpose* nello spazio kernel in un ambiente isolato e totalmente sicuro.

Architettura

eBPF non definisce solamente il set di istruzioni, ma offre una serie di infrastrutture intorno ad essa come mappe che agiscono da *key/value store* efficiente, funzioni di helper per interagire con le funzionalità del kernel, *tail call* ad altri programmi eBPF, primitive di sicurezza, uno pseudo file system per il *pinning* di oggetti (mappe e programmi), e un'infrastruttura che consente a eBPF di essere scaricato sulla scheda di rete (e.g. Figura 4.1).

Occorre specificare che esistono vari tipi di programmi eBPF:

- `SOCKET_FILTER`. I programmi possono essere agganciati a un socket ed eseguire il classico processo di *packet filtering*: iniezione del programma nel kernel che ritornerà allo spazio utente i pacchetti filtrati, mentre scarterà tutto il resto.

- `TRACING`. Sono uno strumento molto potente, in quanto permettono di attivare il programma eBPF ogni qualvolta viene intercettato uno specifico evento all'interno del kernel.

- `SCHED_CLS` e `SCHED_ACT`. I programmi possono essere agganciati allo strato di traffic control dello stack di rete di Linux. Questi programmi possono eseguire manipolazione, redirezione e copia di pacchetti e, più in generale, implementare funzionalità di rete.

- `XDP`. Permettono l'implementazione di un *eXpress Data Path*, ossia di un percorso di rete performante e programmabile all'interno del kernel di Linux. XDP fornisce gli strumenti per eseguire processamento di pacchetti al livello

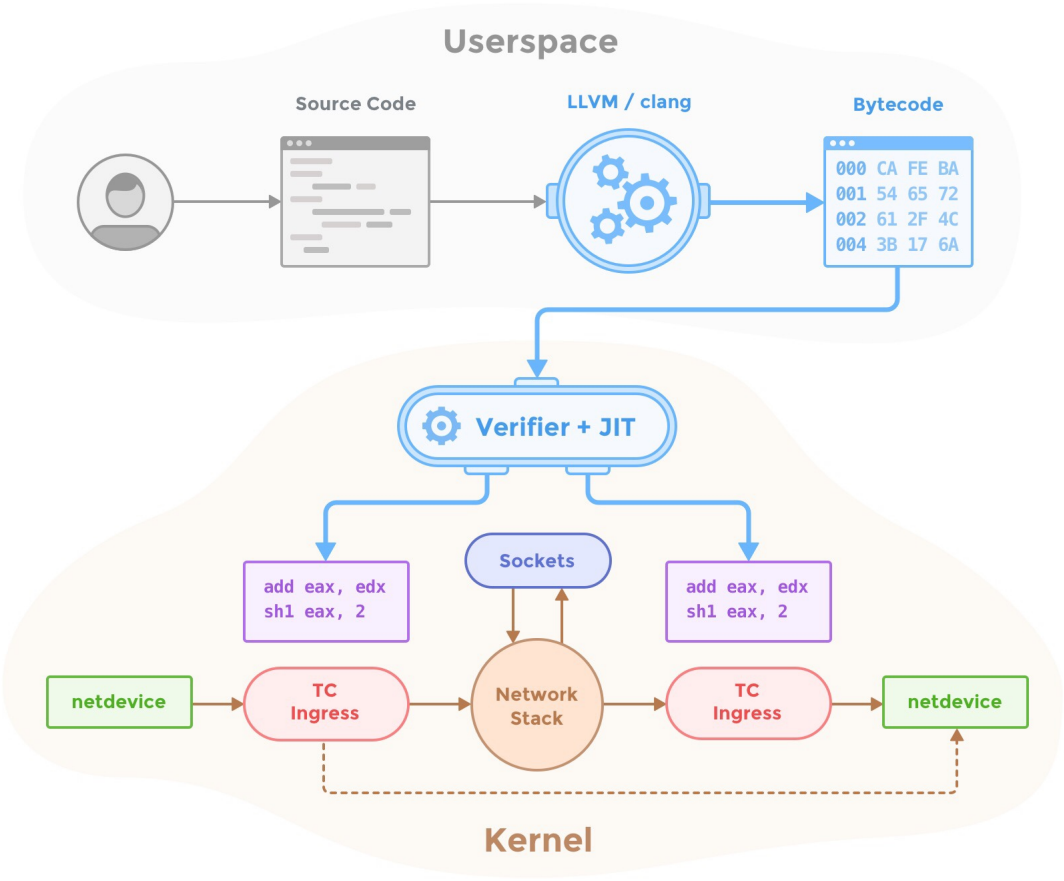

Figura 4.1. Panoramica di eBPF [17]

più basso dello stack, ad esempio direttamente sulla scheda di rete prima che avvenga l'allocazione delle strutture dati all'interno del kernel.

Set di istruzioni

L'interprete eBPF è dotato di un set di istruzioni di tipo RISC ed è stato progettato per accettare un sottoinsieme di comandi C, compilati da LLVM, in modo tale che il kernel possa mapparli attraverso un compilatore JIT (Just-in-time) in codici operativi che garantiscano un'esecuzione performante all'interno del kernel stesso. I vantaggi dell'iniezione del codice all'interno del kernel sono molteplici; di seguito i principali:

- Rendere il kernel programmabile senza dover attraversare le barriere tra il kernel-space e lo user-space. Ad esempio, i programmi eBPF che svolgono funzioni di networking possono implementare policy flessibili, load balancing e altri strumenti senza dover inoltrare i pacchetti verso lo user-space per poi riceverli nuovamente nel kernel.

- Data la flessibilità del *data path* programmabile, i programmi possono subire un notevole incremento di performance semplicemente omettendo la compilazione di moduli che non sono richiesti per un particolare caso d'uso. Per esempio, se un container non richiede IPv4, il programma eBPF potrà essere compilato in modo tale da gestire esclusivamente IPv6 e risparmiare risorse nel *fast-path*.

- Nel caso di networking, i programmi eBPF possono essere aggiornati atomicamente senza la necessità di dover riavviare il kernel, i servizi di rete, i container e senza interruzione del traffico. Inoltre, lo stato di qualsiasi programma può essere mantenuto attraverso aggiornamenti tramite le mappe eBPF.

- eBPF non richiede nessun modulo di terze parti all'interno del kernel, essendo parte integrante di quest'ultimo.

JIT

Le architetture 64 bit `x86_64`, `arm64`, `ppc64`, `mips64`, `sparc64` e 32 bit `arm` sono tutte fornite con un compilatore eBPF JIT presente nel kernel (disabilitato di default). Questo consente di convertire le istruzioni eBPF in codice macchina nativo in modo da migliorare le performance.

Verificatore

Considerando che un programma eBPF vive all'interno della macchina virtuale eseguita nel kernel, è necessario considerare eventuali problematiche correlate alla sicurezza e all'esecuzione di codice sicuro. In altre parole, prima di eseguire il codice, il verificatore - integrato nel kernel - deve accertarsi che l'esecuzione del programma terminerà dopo un numero limitato di istruzioni, e che non faccia accessi invalidi a memoria. Il primo passo consiste in una ricerca *depth-first* che verifica che il programma è un DAG (Dynamic Acyclic Graph). Di conseguenza, saranno rifiutati i programmi con:

- lunghezza maggiore di 4096 istruzioni

- presenza di cicli

- jump malformate o *out-of-bounds*

Il secondo passo consiste nell'analizzare tutti i possibili percorsi partendo dalla prima istruzione:

- valutazione di ogni percorso/istruzione tenendo traccia dello stato dei registri e dello stack
- verifica della validità degli argomenti nelle chiamate

Funzioni di helper

Le funzioni di helper abilitano i programmi eBPF a consultare un set di chiamate a funzione definite nel nucleo del kernel, in modo da ottenere / inviare dati verso / da il kernel. Le funzioni di helper disponibili possono differire per ciascun tipo di programma eBPF; ad esempio, un programma agganciato a un socket può accedere a un sottoinsieme di funzioni di helper rispetto a un programma eBPF agganciato allo strato di traffic control. Di seguito le principali funzioni di helper disponibili:

- `BPF_FUNC_csum_diff()`
- `BPF_FUNC_csum_update()`
- `BPF_FUNC_hash_recalc()`
- `BPF_FUNC_get_socket_uid()`
- `BPF_FUNC_getsockopt()`
- `BPF_FUNC_setsockopt()`
- `BPF_FUNC_ktime_get_ns()`
- `BPF_FUNC_l3_csum_replace()`
- `BPF_FUNC_l4_csum_replace()`
- `BPF_FUNC_map_delete_elem()`
- `BPF_FUNC_map_lookup_elem()`
- `BPF_FUNC_map_update_elem()`
- `BPF_FUNC_perf_event_output()`
- `BPF_FUNC_perf_event_read()`
- `BPF_FUNC_probe_read()`
- `BPF_FUNC_redirect()`

Mappe

Le mappe sono un key/value store efficiente che risiede nello spazio kernel. Sono una delle più importanti novità introdotte da eBPF, in quanto fungono da punto di contatto tra lo spazio kernel e lo spazio utente, potendo essere condivise da questi due spazi di indirizzamento attraverso l'utilizzo di file descriptor (e.g. Figura 4.2). L'implementazione delle mappe è fornita dal nucleo del kernel. Qui ci sono mappe generiche per-CPU e non-per-CPU, oltre che mappe non generiche che possono essere utilizzate assieme alle funzioni di helper.

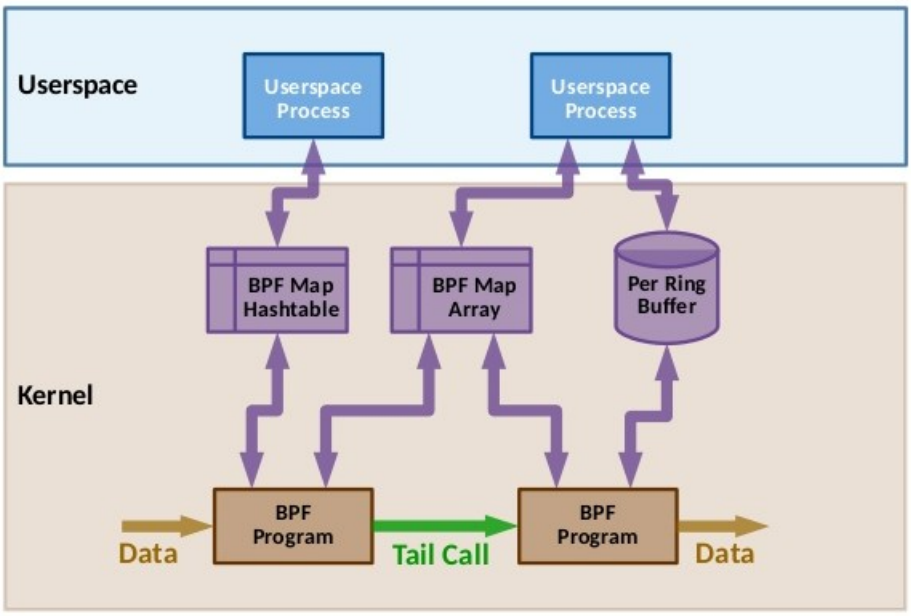

Figura 4.2. Mappe in eBPF

Le mappe generiche attualmente disponibili sono:

- `BPF_MAP_TYPE_HASH`
- `BPF_MAP_TYPE_ARRAY`
- `BPF_MAP_TYPE_PERCPU_HASH`
- `BPF_MAP_TYPE_PERCPU_ARRAY`
- `BPF_MAP_TYPE_LRU_HASH`
- `BPF_MAP_TYPE_LRU_PERCPU_HASH`
- `BPF_MAP_TYPE_LPM_TRIE`

Le mappe non generiche attualmente disponibili nel kernel sono:

- `BPF_MAP_TYPE_PROG_ARRAY`
- `PF_MAP_TYPE_PERF_EVENT_ARRAY`
- `BPF_MAP_TYPE_CGROUP_ARRAY`
- `PF_MAP_TYPE_STACK_TRACE`
- `BPF_MAP_TYPE_ARRAY_OF_MAPS`
- `PF_MAP_TYPE_HASH_OF_MAPS`

Pinning di oggetti

Le mappe e i programmi eBPF si comportano come una risorsa del kernel accessibile esclusivamente attraverso file desciptor, mappati come inode anonimi nel kernel. A questo seguono una serie di vantaggi e svantaggi:

- Le applicazioni user-space possono utilizzare le API relative a numerosi file descriptor che, tuttavia, sono limitati al ciclo di vita del processo a cui appartengono, il che rende la condivisione delle mappe molto più complicata da realizzare. Ciò porta a una serie di problematiche per specifici casi d'uso in cui il programma eBPF è caricato nel kernel per poi terminare in un tempo relativamente breve. In questo modo le mappe precedentemente create non saranno né disponibili per essere consultate da un programma utente né utilizzabili da un programma di terze parti che esegue monitoraggio o aggiornamento delle mappe durante l'esecuzione del programma eBPF.
- Per superare questo limite è stato implementato un file system eBPF dove poter agganciare mappe e programmi eBPF tramite un processo chiamato *object pinning*. Di conseguenza, le system call sono state estese con due ulteriori comandi, ossia `BPF_OBJ_PIN` e `BPF_OBJ_GET`.

Tail call

Questo meccanismo permette a un programma eBPF di chiamarne un altro senza ritornare al programma chiamante. Questa invocazione ha un sovraccarico minimo a differenza delle chiamate a funzione, essendo implementata come una *long jump* e riutilizzando lo stesso *stack frame*. Solo i programmi dello stesso tipo possono essere invocati in questo modo, e necessitano inoltre di corrispondere in termini di compilazione JIT. Di conseguenza, sia i programmi compilati JIT o solamente interpretati possono essere invocati, ma non possono essere utilizzati assieme.

Ci sono due componenti coinvolti nel meccanismo delle tail call: la prima parte necessita di impostare una mappa specializzata chiamata *program array* (`BPF_MAP_TYPE_PROG_ ARRAY`) che può essere popolata dallo spazio utente con chiave/valori, dove i valori sono i descrittori dei file dei programmi eBPF invocati; la seconda parte è una funzione di helper chiamata `bpf_tail_call()` a cui è passato un riferimento al program array e alla chiave utilizzata per eseguire il lookup.

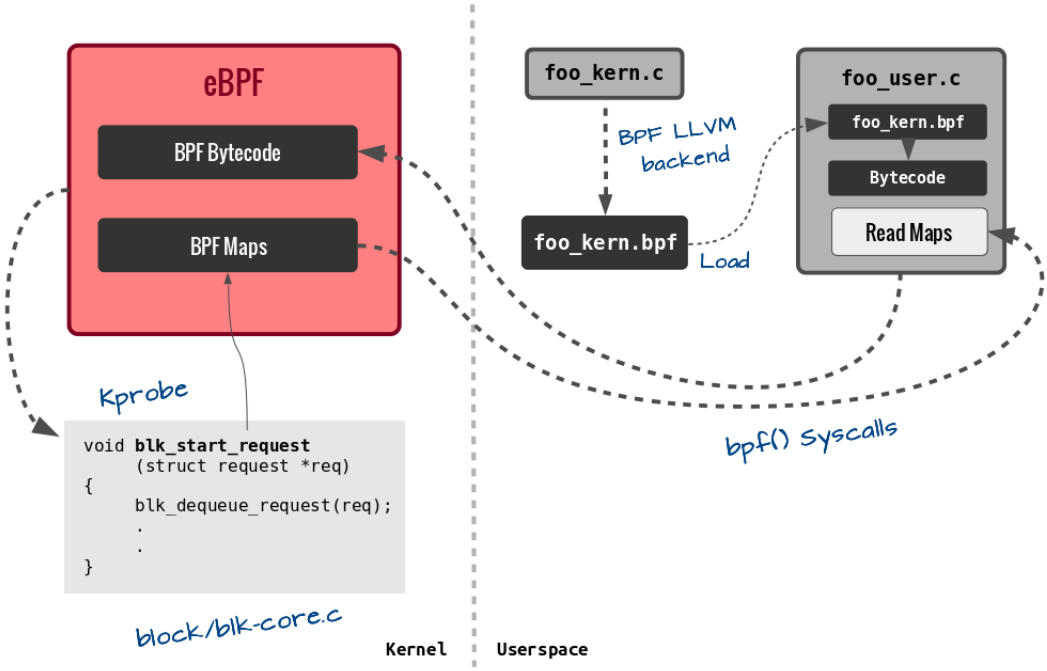

Figura 4.3. Esecuzione di un programma eBPF

4.2 IO Visor

IO Visor è un progetto open source che coinvolge una comunità di sviluppatori allo scopo di accelerarne innovazione, sviluppo e condivisione di servizi di input/output lato kernel per eseguire funzioni di tracing, analisi, monitoraggio, sicurezza e networking [18].

Il progetto fa riferimento a una serie di componenti open source che combinati insieme consentono lo sviluppo *IO Modules* (generici oggetti di input/output che possono essere dinamicamente iniettati nel kernel). IO Visor porta l'estensibilità dell'IO all'interno del kernel di Linux e fornisce agli sviluppatori i mezzi per poter creare e installare applicazioni in sistemi *live* senza dover ricompilare o riavviare l'intero data center.

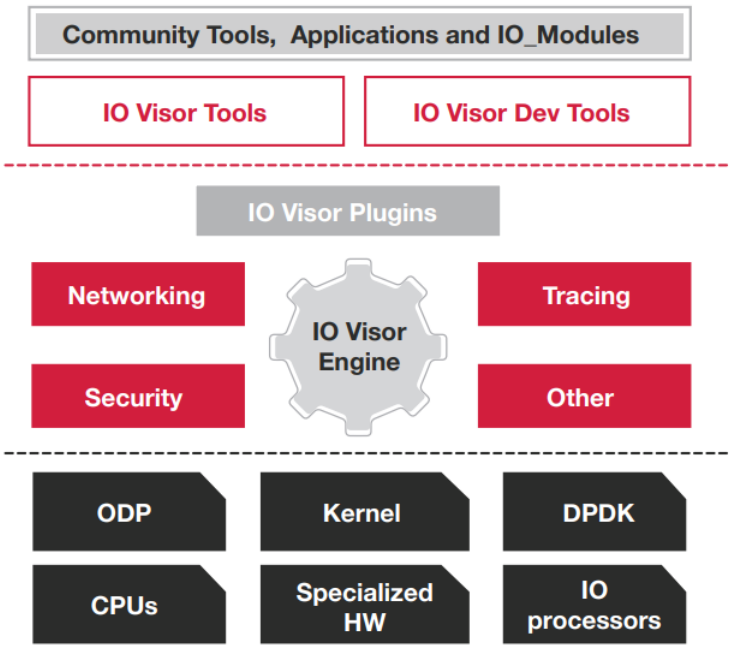

Figura 4.4. IO Visor

L'architettura di IO Visor consiste in uno o più *IO Visor Engine* che sfruttano la tecnologia eBPF. L'IO Visor Engine comprende un set di plugin che forniscono funzionalità in diverse aree, come networking, tracing e sicurezza. I due principali benefici derivanti da questo progetto sono:

- **Flessibilità**. Avere un'architettura programmabile ed estensibile consente agli sviluppatori di creare moduli dinamici eseguiti nel kernel, caricati e rimossi senza causare interruzioni del sistema. Inoltre, i moduli di IO sono *platform independent*, potendo quindi operare su qualsiasi hardware o chipset x86 su Linux.

- **Performance**. Lanciare funzioni di networking nel kernel consente di sfruttare appieno le performance dell'hardware, non dovendosi appoggiare ad alcuno strato software.

4.2.1 Networking

In IO Visor è possibile creare moduli software (e.g. Figura 4.5) che assolvano le funzioni di switch, router, load balancer, sicurezza, per poi essere combinati insieme in modo da formare una topologia completa all'interno del kernel del *compute node*. Il traffico può arrivare ai moduli da una VM o da un container, e attraversare

l'intera rete all'interno del kernel lasciando il compute node solo quando è necessario raggiungere la destinazione. Il beneficio maggiore di IO Visor è la possibilità di programmare qualsiasi logica di rete per ogni versione del protocollo, per poi dinamicamente arrestare l'implementazione esistente per farne posto a una nuova.

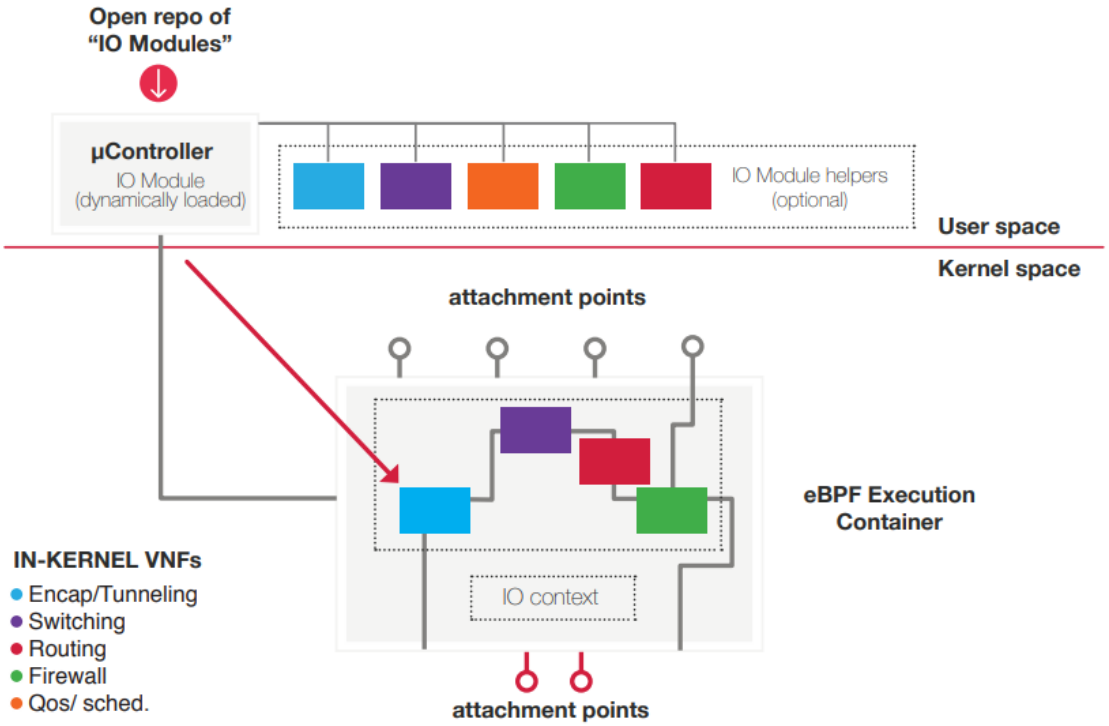

Figura 4.5. Costruzione di un'infrastruttura di rete virtuale in IO Visor

4.3 BPF Compiler Collection (BCC)

BCC rappresenta un set di strumenti per la creazione di programmi di tracing e manipolazione, basandosi sull'uso di eBPF [19]. BCC rende i programmi eBPF più facili da scrivere, con un'orchestrazione lato kernel in C e un fronted in python e lua (e.g. Figura 4.6).

4.3.1 Motivazioni

eBPF garantisce che il programma caricato nel kernel non possa essere soggetto a crash, e che abbia vita limitata, comunque permettendo a eBPF di essere general

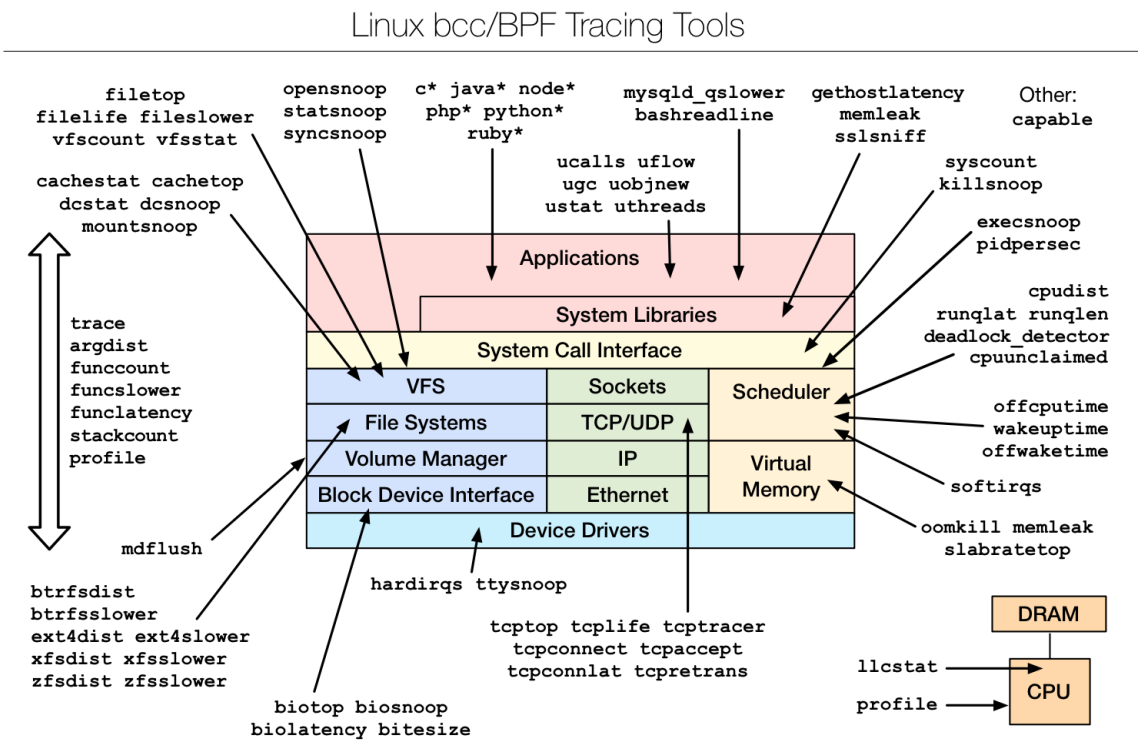

Figura 4.6. Architettura di riferimento in BCC

purpose abbastanza da eseguire molteplici operazioni di computazione. Correntemente, è possibile scivere un programma in C che sarà compilato in un programma eBPF valido, anche se è molto più semplice scrivere un programma in C che sarà compilato in un programma eBPF non valido. L'utente, di conseguenza, scoprirà che un programma è valido solo nel momento in cui proverà ad eseguirlo.

Con un frontend specifico per eBPF è possibile scrivere codice e ricevere un feedback dal compilatore circa la validità di un backend eBPF. BCC ha lo scopo di fornire un'interfaccia che possa generare esclusivamente programmi eBPF validi, continuando comunque a sfruttare la sua flessibilità.

4.4 Iovnet

Iovnet è un framework progettato per creare e distribuire funzioni di rete virtualizzate basate su IO Visor come bridge, router, NAT, firewall, e consente la creazione di catene complesse di sevizi, grazie alla connessione dei moduli sopra citati. Il framework sfrutta alcuni componenti già esistenti come la macchina virtuale eBPF e BCC, includendo questi strumenti in un software potente che consente la

creazione di VNF e veloci, eseguite nel kernel di Linux. Di seguito, le principali caratteristiche:

- supporto di generici servizi di rete attraverso la definizione di un *fast path* (nel kenel) e di uno *slow path* (nello spazio utente), superando quindi i limiti di eBPF in termini di supporto all'elaborazione arbitraria;
- integrazione di una metodologia generica per la configurazione di VNF;
- presenza di un'interfaccia di controllo e di configurazione indipendente dal servizio, che consente di interagire con tutti le VNF in esecuzione attraverso la stessa interfaccia, vale a dire sia REST (per la comunicazione machine-to-machine) che CLI (per l'interazione con l'uomo);
- possibilità di definire catene arbitrarie di servizi, semplificando quindi la creazione di servizi complessi attraverso la composizione di molte componenti elementari (i.e. i singoli servizi iovnet).

Iovnet si rivolge sia agli utenti finali, che possono sfruttare i servizi elementari già disponibili per creare delle catene, sia agli sviluppatori, che possono creare i servizi di cui hanno bisogno tramite un'API semplice ed elegante che si occupa della gestione e della generazione automatica della logica di base necessaria per il funzionamento del servizio stesso; in questo modo gli sviluppatori possono concentrarsi esclusivamente sulla logica principale del loro programma.

4.4.1 Architettura

L'architettura di Iovnet include quattro componenti principali (e.g. Figura 4.7):

- **iovnetd**, un demone che controlla il servizio iovnet;
- **iovnet services**, che implementa i servizi di rete;
- **iovnetctl**, la CLI che consente di interagire con iovnet e con tutte le funzioni di rete disponibili;
- **libiovnet**, una libreria che mantiene il codice condiviso, in modo da poter essere riutilizzato da più servizi di rete.

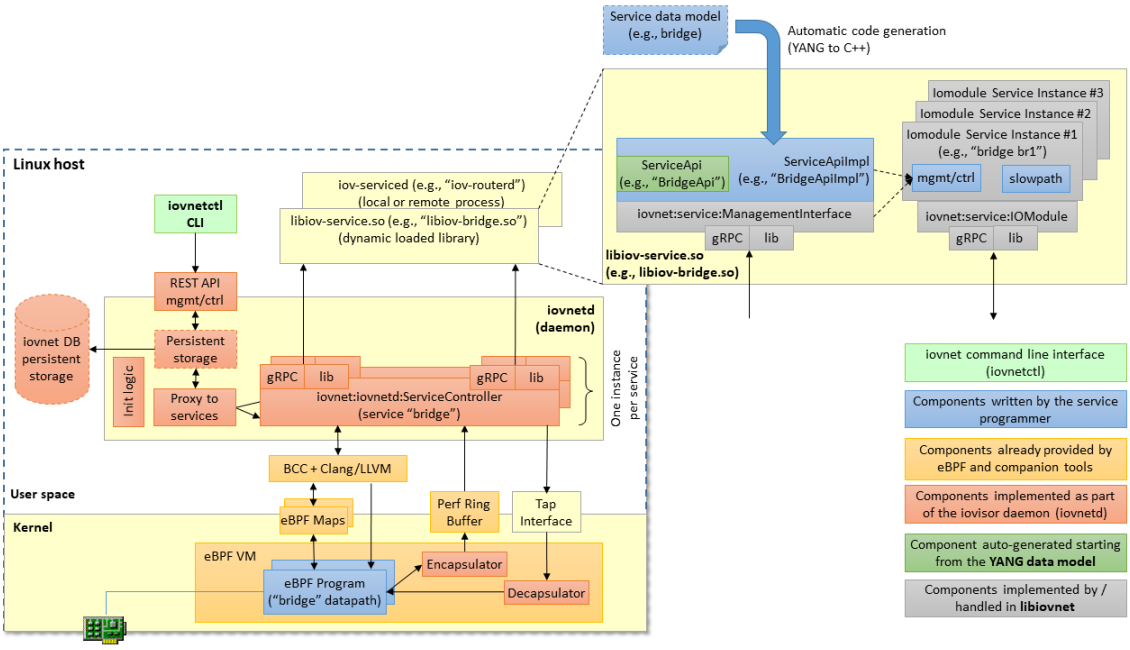

Figura 4.7. Architettura di Iovnet

4.5 Linux tracepoints

I tracepoint (letteralmente *punti di tracciamento*) in Linux sono delle *static probe*, ossia interrogazioni statiche localizzate in punti strategici all'interno del kernel [20]. Le *probe* registrano/rimuovono i tracepoint tramite il meccanismo delle *callback*. Le probe sono funzioni fortemente tipizzate che accettano un unico set di parametri definito da ciascun tracepoint. Tramite questo semplice meccanismo, queste funzioni possono essere utilizzate per profilare, analizzare e comprendere il comportamento del kernel.

4.5.1 Tracciare eventi

Le API che permettono di tracciare eventi sono basate sui tracepoint, e possono essere viste come API di livello più alto rispetto a questi ultimi. Inoltre, possono essere invocate dallo spazio utente, evitando la scrittura di moduli kernel appositi, con la conseguenza che in questo modo è possibile accedere ai dati in sola lettura [21].

```
TRACE_EVENT(bpf_map_create,

    TP_PROTO(const struct bpf_map *map, int ufd),
```

```
    TP_ARGS(map, ufd),

    TP_STRUCT__entry(
        __field(u32, type)
        __field(u32, size_key)
        __field(u32, size_value)
        __field(u32, max_entries)
        __field(u32, flags)
        __field(int, ufd)
    ),

    TP_fast_assign(
        __entry->type        = map->map_type;
        __entry->size_key    = map->key_size;
        __entry->size_value  = map->value_size;
        __entry->max_entries = map->max_entries;
        __entry->flags       = map->map_flags;
        __entry->ufd         = ufd;
    ),

    TP_printk("map type=%s ufd=%d key=%u val=%u max=%u flags=%x",
          __print_symbolic(__entry->type, __MAP_TYPE_SYM_TAB),
          __entry->ufd, __entry->size_key, __entry->size_value,
          __entry->max_entries, __entry->flags)
);
```

Listing 4.1. Dichiarazione di un tracepoint

L'esempio mostra la dichiarazione del tracepoint agganciato all'evento `bpf_map_create`, estrapolato dell'header definito del kernel di Linux. In questo modo si rende disponibile la funzione con il prototipo dichiarato in `TP_PROTO` e con i nomi degli argomenti in `TP_ARGS`. Il tracciamento di eventi corrisponde al caricamento di un modulo che copia i dati dagli argomenti della funzione nella struttura `TP_STRUCT__entry` che può essere consultata dallo spazio utente, mentre `TP_fast_assign` è il codice C che copia i dati nella suddetta struttura.

4.6 Etcd

Etcd (distributed etcd) è una base dati distribuita del tipo chiave/valore che garantisce un meccanismo di salvataggio affidabile all'interno di un cluster di macchine [22]. Etcd gestisce elegantemente l'elezione del leader in caso di partizionamento della rete e tollera il guasto della macchina, compreso il leader. L'applicazione può leggere e scrivere dati in etcd, ad esempio per salvare la configurazione di un server

sotto forma di coppie chiave/valore, con la possibilità di *osservare* una specifica chiave in modo da ricevere notifiche in caso di aggiornamento/rimozione della stessa.

Il software è scritto in Go[5], che è caratterizzato da un ottimo supporto multipiattaforma, piccoli file binari e una grande comunità dietro di esso. La comunicazione tra le macchine su cui etcd è eseguito viene gestita dall'algoritmo di consenso Raft[6]. Più nello specifico, Raft gestisce il consenso, ossia il consolidamento dei dati, distribuito, cioè in presenza di tanti nodi concorrenti (e.g. in un cluster). La latenza per il leader è un fattore molto importante di cui tenere traccia, che potrebbe portare instabilità del cluster in quanto Raft è veloce quanto la macchina più lenta della maggioranza.

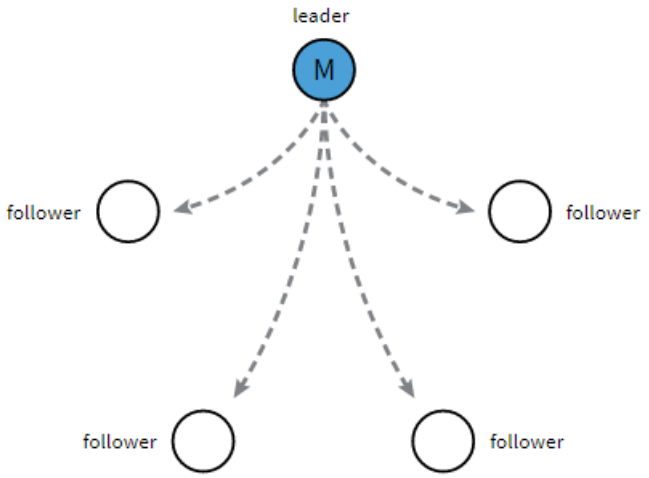

Figura 4.8. Replicazione dei dati su ogni nodo del cluster

[5] https://golang.org/

[6] https://raft.github.io/

Capitolo 5

Centralizzazione e propagazione dello stato

Questo capitolo presenta l'idea di base e le successive scelte implementative messe in atto per distribuire lo stato di singole istanze appartenenti alla stessa VNF in maniera efficiente.

5.1 Soluzione proposta

L'approccio seguito in questo lavoro di tesi vede la centralizzazione dello stato delle VNF del sistema con l'ausilio di una base dati affidabile ed efficiente. Per centralizzare le informazioni, tuttavia, è necessario leggere le variazioni di stato in tempo reale, possibilmente in maniera asincrona, e in seguito aggiornare la base dati. Allo stesso tempo, ogni singola VNF dovrà integrare un modulo che sincronizzi localmente le informazioni opportunamente centralizzate. Inoltre, nuove istanze di VNF devono essere in grado di leggere lo stato del sistema al momento della loro accensione, per poi poter iniziare ad operare.

5.1.1 Primo scenario

Si consideri un semplice scenario con due server, ciascuno con una catena di VNF (i.e. firewall, load balancer, nat), dove ognuna di esse è attraversata da un flusso dati. In generale, l'algoritmo si sviluppa nei seguenti passi:

1. Le VNF aggiornano il proprio stato dopo aver gestito la richiesta (e.g. Figura 5.1);

2. A questo punto la VNF rileva il cambiamento di stato e lo comunica alla base dati (e.g. Figura 5.2);

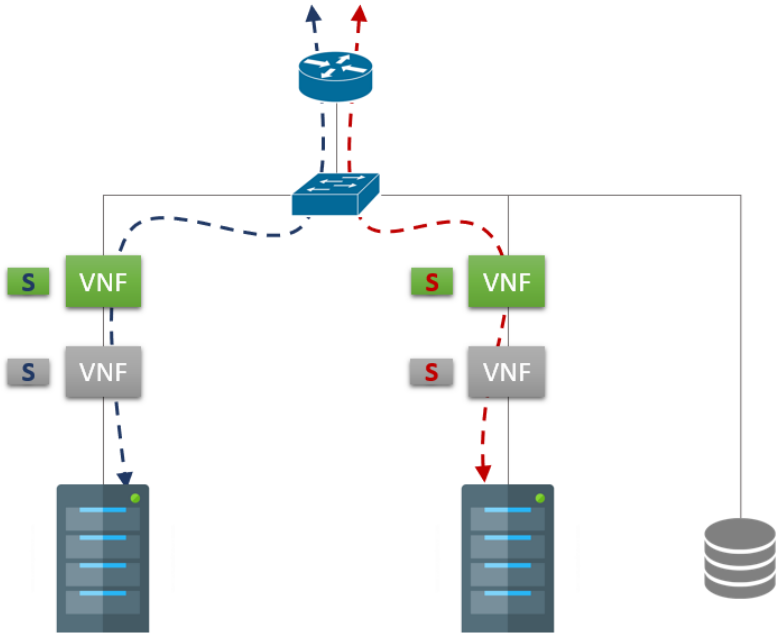

Figura 5.1. Entrambe le catene sono interessate da un flusso dati

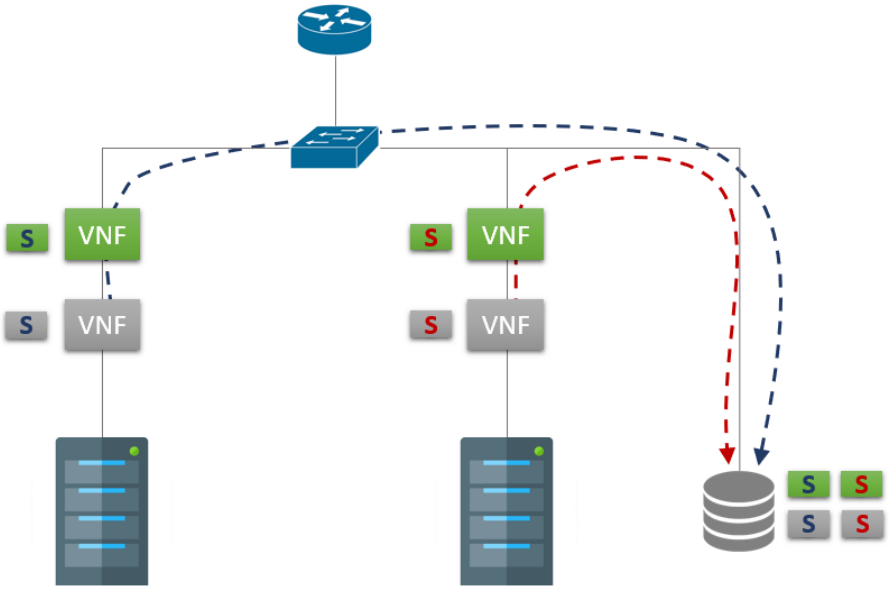

Figura 5.2. Ciascuna funzione di rete comunica la variazione di stato alla base dati

3. In ultimo, la base dati propaga le informazioni verso tutte le istanze che

necessitano del nuovo stato (e.g. Figura 5.3).

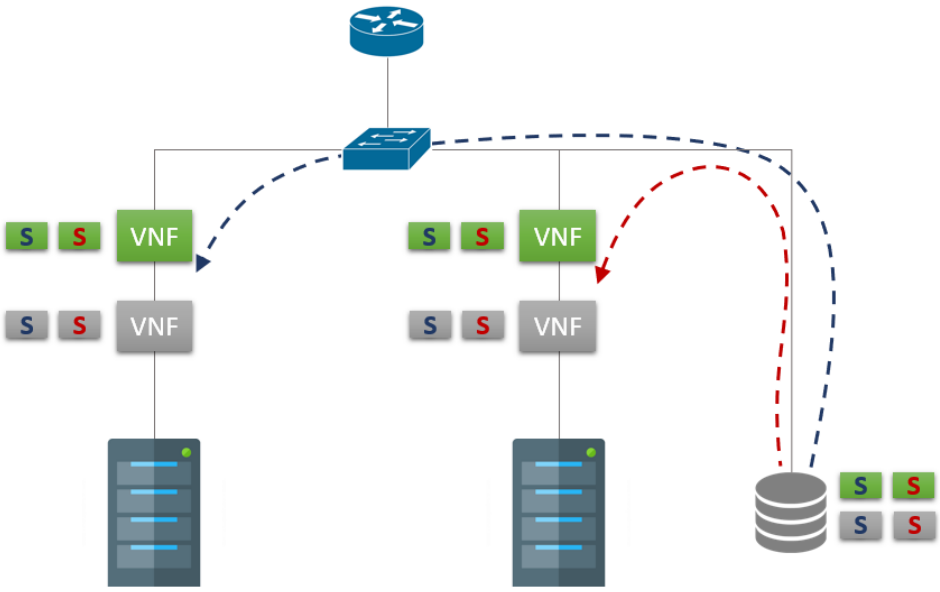

Figura 5.3. La base dati propaga le informazioni a tutte le istanze

In questo modo tutte le istanze della stessa funzione di rete possiedono lo stato globale nel loro storage locale. Ad ad esempio, tutte le istanze del load balancer conterranno lo stato globale utile al processo di load balancing, tutte le istanze del NAT conterrano la tabella di traduzione globale, e via discorrendo per tutte le VNF.

Vantaggi

Il vantaggio principale offerto da questa soluzione è la possibilità, per tutte le VNF, di poter prendere decisioni consapevoli, ossia basate sulla conoscenza dell'intero stato del sistema. Il NAT, ad esempio, potrà effettuare nuove traduzioni allocando porte sicuramente disponibili, evitando conflitti in fase di smistamento del traffico da parte del router. Inoltre, un pacchetto dati potrà viaggiare indipendentemente in entrambe le catene di VNF, conferendo all'intero sistema un'estrema flessibilità.

5.1.2 Secondo scenario

Si consideri ora uno scenario differente, in cui si decide di aggiungere a *runtime* un nuovo server, e di accendere le VNF che intercettano il traffico diretto ad esso. In questo caso le nuove istanze devono essere in grado di leggere l'intero stato

(precedentemente distribuito) e salvarlo localmente. Di seguito i passi principali dell'algoritmo:

1. Nel sistema c'è un solo server operativo mentre la seconda macchina è spenta. In questo caso lo stato globale salvato sulla base dati coincide con lo stato delle singole VNF (e.g. Figura 5.1);

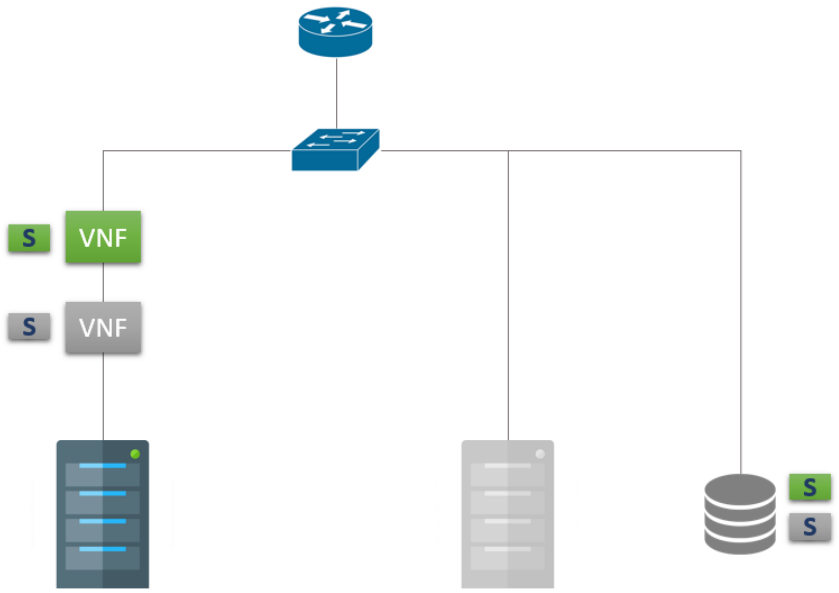

Figura 5.4. Nel sistema è presente un unico server acceso e funzionante

2. All'accensione del secondo server con le relative VNF per il processamento del traffico, c'è una lettura completa della base dati (e.g. Figura 5.2);

3. A questo punto, le VNF appena istanziate possiedono lo stato locale che è sincronizzato con lo stato delle funzioni di rete precedentemente funzionanti (e.g. Figura 5.3).

In questo secondo scenario si sfrutta la possibilità di istanziare nuove funzioni di rete a richiesta (i.e. per questioni di scalabilità), che potranno iniziare ad operare in maniera consapevole, ossia conoscendo lo stato del sistema consolidato fino a quel momento. Da questo punto in poi, l'algoritmo si comporta esattamente come descritto nello scenario precedente.

Vantaggi

Si ipotizzi che il primo server o le VNF subiscano un guasto, oppure che siano semplicemente sovraccarichi e non riescano più a servire le richieste in tempistiche

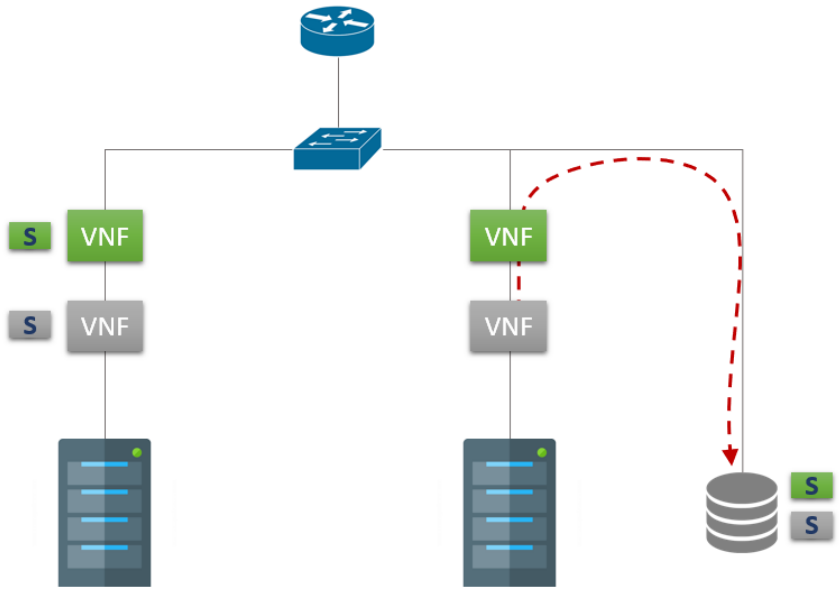

Figura 5.5. All'accensione le nuove VNF si sincronizzano con la base dati

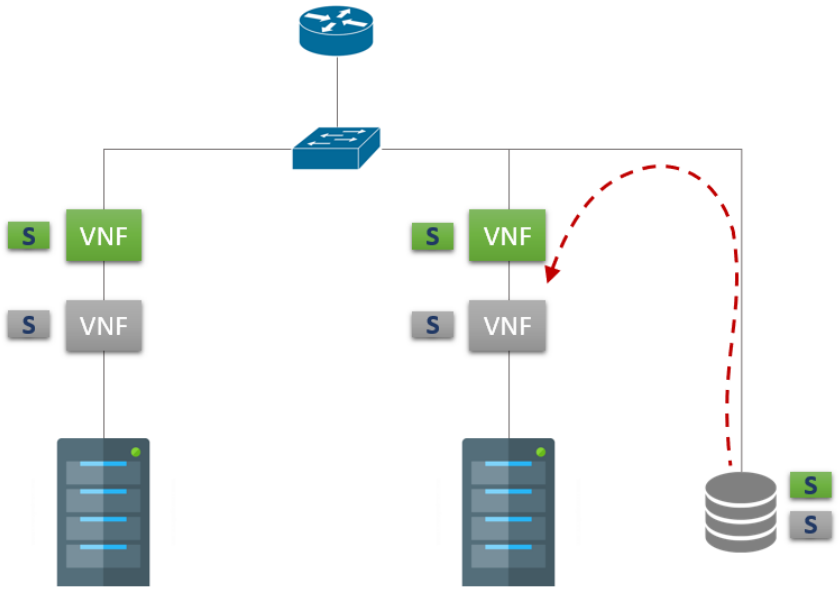

Figura 5.6. Tutte le VNF sono sincronizzate

accettabili. Sarà possibile accendere nuovi server e istanziare VNF senza perdere la computazione già effettuata. Nel caso di un NAT, di un load balancer, di un firewall e di molte altre funzioni di rete, questo è un punto fondamentale, essendo tutti

servizi *stateful*. Con un'opportuna redirezione del traffico, infatti, si garantirebbero *availability* e continuità del servizio, altrimenti irrealizzabili in una implementazione *standard*.

5.1.3 Architettura finale

Entrambi gli scenari fanno parte di una visione più generale, che ha la finalità ultima di rendere il sistema flessibile, disponibile, e con la possibilità di effettuare computazione distribuita in modo da migliorare le performance globali dal punto di vista della gestione del networking. Ne consegue, quindi, che le architetture e gli algoritmi proposti sono due facce della stessa medaglia, necessitando di scelte implementative comuni che offrano la possibilità di ottenere tutti i benefici che ne conseguono.

In quest'ottica, si è scelto di implementare la nuova architettura in Iovnet, un framework che offre di per sè il deploy di funzioni di rete altamente performanti, e che integra un sistema di controllo per le stesse. Grazie all'utilizzo di questo framework è possibile astrarre la distribuzione dello stato dalla specifica VNF, ottenendo una funzionalità aggiuntiva comune a tutti i servizi di rete.

5.2 Requisiti

L'effettiva messa in opera dell'idea proposta necessita dell'implementazione o della modifica di alcune componenti software. In particolare:

- Implementazione di un monitor che rilevi le modifiche relative allo stato della VNF;
- Implementazione di un client che invii le suddette notifiche alla base dati;
- Implementazione di un *watcher* che riceva informazioni dalla base dati in caso di aggiornamenti relativi allo stato della VNF a cui il watcher fa riferimento. Il watcher avrà anche la funzione di richiedere l'intero stato alla base dati al momento dell'accesione della VNF;
- Modifica di Iovnet in modo che possa integrare i moduli sopra citati.

Nella fattispecie, sono state fatte alcune **scelte architetturali e implementative**.

- Deploy dell'infrastruttura: Iovnet, con la conseguenza che lo stato della VNF risiederà nelle mappe eBPF;

- Programma di monitoraggio: sviluppato sfruttando le tencologie eBPF e BCC, in aggiunta ai Linux tracepoint: in questo modo sarà rilevato un evento ogni qualvolta le mappe eBPF afferenti al servizio di rete saranno interessate da modifica o cancellazione;

- Base dati: etcd, caratterizzata dall'elezione del leader e dall'utilizzo di Raft per la realizzazione del consenso distribuito;

- Client e watcher per interfacciarsi con la base dati: sviluppati in Go per sfruttare il supporto alle API offerto da etcd.

5.3 Differenze con i lavori correlati

Dopo aver esposto nel dettaglio l'architettura proposta e i punti cardini su cui è stata progettata, è possibile fare un'analisi specifica delle principali differenze tra la soluzione di questa tesi e le principali proposte descritte nella sezione 3.4.

5.3.1 OpenNF

OpenNF è un framework pensato principalmente per migrare flussi di rete da un'istanza di VNF verso un'altra. La migrazione di flussi si traduce in migrazione dello stato correlato al flusso da migrare. Questo è realizzato grazie all'utilizzo di API di northbound e southbound, che consentono rispettivamente di muovere lo stato e integrare un vasto numero di VNF in OpenNF.

La differenza principale è, dunque, l'obiettivo del lavoro. Questa tesi non ha la finalità di migrare specifici flussi, ma l'obiettivo è di rendere la rete più flessibile centralizzando e distribuendo lo stato tra tutte le istanze. Questo si traduce in una semplificazione in termini di programmazione del controller e di gestione dello stato, in quanto l'architettura proposta prevede una gestione dello stato (e.g. centralizzazione e distribuzione) completamente automatizzata.

5.3.2 CoGS

CoGS propone la divisione dello stato globale su più middlebox, dove ciascuna di esse conserva in locale lo stato necessario alla propria computazione. Allo stesso tempo, le middlebox possono accedere agli stati globali.

Il framework fa una distinzione per-flow, che è un concetto assente nella soluzione proposta. Inoltre, sebbene mostri risultati interessanti in termini di throughput, CoGS non fornisce un'astrazione che possa gestire uniformemente NF differenti, in contrasto con l'obiettivo di questa tesi.

5.3.3 Approccio Stateless

Questo approccio consiste nel disaccoppiare lo stato della VNF dallo strato di processing. In questo modo si vanno a realizzare funzioni di rete altamente flessibili, con l'ausilio di un data storage centralizzato. La soluzione stateless porta sicuramente dei vantaggi in termini di resilienza ed elasticità, ma introduce dei tempi di latenza in parte attenuati dall'utilizzo di un data storage efficiente.

La soluzione proposta in questa tesi ha il vantaggio di offrire tutte le proprietà derivanti dall'approccio stateless, con la differenza sostanziale che la VNF può eseguire il processing in qualsiasi istante, a latenza zero: l'unica latenza introdotta è relativa alla propagazione degli aggiornamenti di stato. Questo è dovuto alla distribuzione dello stato che avviene in maniera automatizzata e in tempo reale, esonerando quindi la VNF da fare richieste alla base dati quando si vuole processare uno specifico flusso. La distribuzione dello stato proposta, tuttavia, può comportare un alto consumo di storage locale all'interno della VNF.

Capitolo 6

Implementazione

Questa sezione descrive i punti salienti relativi all'implementazione, le cui scelte sono state illustrate nel capitolo precedente.

6.1 Software di monitoraggio

Questa componente sfrutta le potenzialità di eBPF, BCC e dei tracepoint in Linux per poter eseguire il monitoraggio in tempo reale e in modalità asincrona di eventi all'interno del kernel. Più nello specifico, eBPF e BCC permettono la scrittura e il deploy di un programma eseguito all'interno del kernel; i tracepoint consentono l'aggancio del programma eBPF a determinati eventi, che in questo caso sono modifica e cancellazione di una o più entry all'interno di mappe eBPF. Il software, inoltre, esegue un filtraggio in modo da segnalare all'applicazione user-space esclusivamente eventi relativi alle mappe eBPF di una specifica VNF. In questo modo ne trae giovamento l'efficienza, in quanto alla velocità offerta da eBPF si aggiunge un filtraggio lato kernel che, altrimenti, necessiterebbe di essere eseguito dalla funzione di rete, peggiorandone le performance.

Il programma viene interamente scritto sotto forma di stringa, per poi poter essere compilato e iniettato nel kernel con l'ausilio di BCC. In primo luogo, è stato necessario definire delle strutture dati in cui salvare le informazioni ottenute in seguito al verificarsi di un evento:

```
/* Tracepoint format:
   /sys/kernel/debug/tracing/events/bpf/bpf_map_update_elem/format
 * Code in:                kernel/include/trace/events/bpf.h
 */
struct bpf_map_update_elem {
     u64 __pad;          // First 8 bytes are not accessible by
        bpf code
     u32 type;       //   offset:8;  size:4; signed:0;
     u32 key_len;        //   offset:12  size:4; signed:0;
```

```
    u32 key;        //   offset:16  size:4; signed:0;
    bool key_trunc;          //   offset:20  size:1; signed:0;
    u32 val_len;        //   offset:24  size:4; signed:0;
    u32 val;        //   offset:28  size:4; signed:0;
    bool val_trunc;          //   offset:32  size:1; signed:0;
    int ufd;        //   offset:36  size:4; signed:1;
};
/* Tracepoint format:
   /sys/kernel/debug/tracing/events/bpf/bpf_map_delete_elem/format
 * Code in:                kernel/include/trace/events/bpf.h
 */
struct bpf_map_delete_elem {
    u64 __pad;          // First 8 bytes are not accessible by
       bpf code
    u32 type;       //   offset:8;  size:4; signed:0;
    u32 key_len;        //   offset:12  size:4; signed:0;
    u32 key;        //   offset:16  size:4; signed:0;
    bool key_trunc;          //   offset:20  size:1; signed:0;
    int ufd;        //   offset:24  size:4; signed:1;
};
```

Listing 6.1. Dichiarazione delle strutture dati utili per gestire gli eventi

Il formato delle strutture dati mostrate nel listato è specifico per ciascun evento, ed è specificato dal kernel. Queste informazioni sono utili per eseguire il processamento, e le più rilevanti in questo contesto sono:

- il file descriptor associato alla mappa eBPF,
- la chiave e il valore relativi alla entry modificata/eliminata.

In secondo luogo, sono state definite altre strutture dati per poter comunicare con l'applicazione user space. Le prime due strutture rappresentano le informazioni che saranno inviate allo spazio utente, mentre le successive sono a loro volta delle mappe eBPF: la prima è un *perf buffer* in cui scrivere gli eventi (già filtrati), la seconda è una struttura appositamente inizializzata che conserva i file descriptor relativi alle mappe che si vogliono monitorare:

```
typedef enum {
    U,   // update
    D    // delete
}enum_event;
struct data_def {
    enum_event event;
    int ufd;
    int name;
    u32 key[10];
    u32 key_len;
```

```
    u32 val[10];
    u32 val_len;
};

BPF_PERF_OUTPUT(events);
BPF_TABLE("hash", u32, int, fds, MAX_FDS);
```

Listing 6.2. Dichiarazione delle strutture dati utili per comunicare con lo spazio utente

Come è possibile osservare dal listato, nel buffer *event* sarà copiata la struttura dati che contiene:

- il tipo di evento (*delete* o *update*)
- il file descriptor associato alla mappa eBPF
- il nome della mappa espresso sotto forma di indice numerico
- la chiave e il valore relativi alla entry modificata/eliminata
- la lunghezza in byte della chiave e del valore

In ultimo, sono state implementate le funzioni che processano le informazioni, eseguono il filtraggio, e scrivono sul perf buffer. Di seguito, la gestione della modifica:

```
int on_bpf_map_update_elem(struct bpf_map_update_elem *p_elem)
{
    struct bpf_map_update_elem elem;
    bpf_probe_read(&elem, sizeof(elem), p_elem);

    int *name = fds.lookup(&elem.ufd);
    if (!name) return 0;

    struct data_def data;

    u16 offset;
    u16 len;

    data.event = U;
    data.ufd = elem.ufd;
    data.name = *name;

    offset = elem.key & 0xFFFF;
    len = elem.key >> 16;
    if (len <= 0) return -1;
```

```
    bpf_probe_read(&data.key, sizeof(data.key), (char *)p_elem +
        offset);
    data.key_len = len;

    offset = elem.val & 0xFFFF;
    len = elem.val >> 16;
    if (len <= 0) return -1;

    bpf_probe_read(&data.val, sizeof(data.val), (char *)p_elem +
        offset);
    data.val_len = len;

    events.perf_submit(p_elem, &data, sizeof(data));

    return 0;
}
```

Listing 6.3. Funzione che gestisce l'aggiornamento di una entry in una mappa eBPF

La funzione esegue un *lookup* nella mappa che contiene i file descriptor, ritornando se il file descriptor corrente non vi è presente. In caso contrario, il valore di ritorno conterrà il nome della mappa che ha generato l'evento (sotto forma di indice numerico). A questo punto è possibile impostare correttamente i campi della struttura dati che sarà inviata al programma utente tramite il perf buffer. In particolare, sia la chiave che il valore sono espressi su 32 bit, dove i 16 bit meno significativi rappresentano l'*indirizzo di offset* in cui reperire il dato in sé, mentre i 16 bit più significativi contengono la lunghezza in byte del dato.

Allo stesso modo, il programma gestisce la cancellazione di una entry, con l'unica differenza che in questo caso il valore e la relativa lunghezza sono impostati a zero:

```
int on_bpf_map_delete_elem(struct bpf_map_delete_elem *p_elem)
{
    struct bpf_map_delete_elem elem;
    bpf_probe_read(&elem, sizeof(elem), p_elem);

    int *name = fds.lookup(&elem.ufd);
    if (!name) return 0;

    struct data_def data;

    u16 offset;
    u16 len;

    data.event = D;
    data.ufd = elem.ufd;
    data.name = *name;

    offset = elem.key & 0xFFFF;
```

```
    len = elem.key >> 16;
    if (len <= 0) return -1;

    bpf_probe_read(&data.key, sizeof(data.key), (char *)p_elem +
        offset);
    data.key_len = 0;

    bpf_probe_read(&data.val, sizeof(data.val), (char *)p_elem +
        offset);
    data.val_len = 0;

    events.perf_submit(p_elem, &data, sizeof(data));

    return 0;
}
```

Listing 6.4. Funzione che gestisce l'eliminazione di una entry in una mappa eBPF

6.2 Interazione con etcd

L'interazione con la base dati è strutturata in due componenti differenti:

- Un client integrato che che esegue la scrittura degli eventi su etcd;
- Un watcher connesso alla base dati impostato per ricevere gli eventi relativi a tutte le mappe della funzione di rete, oltre che a richiedere lo stato dell'intera funzione di rete.

6.2.1 Client

La componente client è implementata come due funzioni che saranno poi esportate - sotto forma di *shared object* - in modo da poter essere invocate dal servizio di rete:

```
func Write(endpoint, key, value string) {
    cli, err := clientv3.New(clientv3.Config{
        Endpoints:   []string{endpoint + ":2379"},
        DialTimeout: 3 * time.Second,
    })
    if err != nil {
        log.Fatal(err)
    }
    // set key with value
    log.Printf("Setting '%s' key with '%s' value\n", key, value)
    ctx, cancel := context.WithTimeout(context.Background(), 3 *
        time.Second)
```

```
    _, err = cli.Put(ctx, key, value)
    defer cancel()
    if err != nil {
        log.Fatal(err)
    } else {
        log.Printf("Set is done\n")
    }
    defer cli.Close()
}
```

Listing 6.5. Funzione che scrive la modifica sulla base dati

La funzione riceve come parametri l'*endpoint*, che insieme alla porta 2379 individua il servizio etcd a cui connettersi, la chiave e il valore che rappresentano rispettivamente il nome della mappa eBPF concatenato alla chiave relativa alla entry modificata, e il valore della entry.

La cancellazione di una entry è gestita seguendo la stessa logica:

```
func Delete(endpoint, key string) {     //pass empty string "" in
   order to delete all keys
    cli, err := clientv3.New(clientv3.Config{
        Endpoints:   []string{endpoint + ":2379"},
        DialTimeout: 3 * time.Second,
    })
    if err != nil {
        log.Fatal(err)
    }
    ctx, cancel := context.WithTimeout(context.Background(), 3 *
       time.Second)
    defer cancel()

    // count keys about to be deleted
    resp, err := cli.Get(ctx, key, clientv3.WithPrefix())
    if err != nil {
        log.Fatal(err)
    }

    for _, item := range resp.Kvs {
        _, err := cli.Delete(ctx, string(item.Key),
           clientv3.WithPrefix())
        log.Printf("Deleted %s %s \n", string(item.Key),
           string(item.Value))
        if err != nil {
            log.Fatal(err)
        }
    }
}
```

Listing 6.6. Funzione che segnala la cancellazione sulla base dati

6.2.2 Watcher

Questo componente, a differenza del client, è implementato sotto forma di programma indipendente che da un lato si connette a etcd per ricevere notifiche, dall'altro segnala le suddette notifiche alla VNF tramite una connessione instaurata in *localhost* sulla porta 12000:

```
func WatchRange(endpoint, key_start, key_end string) {
    log.Printf("Waiting for connection")
    service := ":12000"
    addr, err := net.ResolveTCPAddr("tcp4", service)
    listener, err := net.ListenTCP("tcp", addr)
    defer listener.Close()

    cli, err := clientv3.New(clientv3.Config{
        Endpoints:   []string{endpoint + ":2379"},
        DialTimeout: 3 * time.Second,
    })
    if err != nil {
        log.Fatal(err)
    }
    rch := cli.Watch(context.Background(), key_start,
       clientv3.WithRange(key_end))

    for {
        conn, err := listener.Accept()
        log.Printf("Connected with client")
        if err != nil {
            continue
        }
        for wresp := range rch {
                for _, ev := range wresp.Events {
                data := fmt.Sprintf("+%s+%s+%s", ev.Type,
                   ev.Kv.Key, ev.Kv.Value)
                data_len := len(data)
                conn.Write([]byte(fmt.Sprint(data_len)))
                conn.Write([]byte(data))
                log.Printf(data)

                // packet format:
                   "length+event_type+table_name+key;value"
                }
        }

        defer cli.Close()
        defer conn.Close();
    }
}
```

Listing 6.7. Applicazione per la ricezione degli eventi da etcd

La funzione permette di specificare un range di chiavi - ossia di mappe eBPF - di cui monitorare lo stato sulla base dati. Dopo aver instaurato la connessione con etcd ed essersi messo in ascolto sugli eventi, il *watcher* instaura una connessione col servizio di rete, tramite la quale invierà i seguenti dati concatenati in una stringa:

- length
- event type
- key
- value

Dove *event type* può assumere i valori *PUT* o *DELETE*, *key* è il nome della mappa eBPF che ha generato l'evento e *value* è la concatenazione della coppia chiave/valore relativa all'elemento modificato/eliminato.

In aggiunta, il watcher integra la funzione che esegue la lettura dell'intero stato relativo al servizio di rete, utile a sincronizzare la VNF appena istanziata con lo stato globale della computazione:

```
func Read(endpoint, key string) {       //pass empty string "" in
    order to read all keys
    cli, err := clientv3.New(clientv3.Config{
        Endpoints:   []string{endpoint + ":2379"},
        DialTimeout: 3 * time.Second,
    })
    if err != nil {
        log.Fatal(err)
    }
    // get key's value
    ctx, cancel := context.WithTimeout(context.Background(), 3 *
       time.Second)
    resp, err := cli.Get(ctx, key, clientv3.WithPrefix())
    defer cancel()
    if err != nil {
        log.Fatal(err)
    } else {
        // print value
        for _, item := range resp.Kvs {
            log.Printf("Read %s %s \n", item.Key, item.Value)
        }
    }
    defer cli.Close()
}
```

Listing 6.8. Funzione per la lettura dello stato globale

La funzione accetta come parametro, oltre all'endpoint, la chiave di cui si vuole leggere il valore. Invocando la funzione passando una stringa vuota, si andranno a leggere tutte le chiavi presenti sul database.

6.3 Integrazione in Iovnet

L'integrazione all'interno del framework di tutte le componenti sopra citate si sviluppa nei seguenti passi:

- Inizializzazione di due *thread* che si occuperanno rispettivamente di iniettare il programma di monitoraggio, con conseguente scrittura su etcd, e di leggere sul socket i dati inviati dal watcher;
- Scrittura in una mappa eBPF dei file descriptor relativi alle mappe che si vogliono monitorare (le mappe eBPF conterranno lo stato delle VNF);
- Iniezione del codice eBPF che esegue, sfruttando i tracepoint:
  - monitoraggio lato kernel dello stato delle VNF;
  - filtraggio basato sulla lettura dei file descriptor precedentemente impostati;
  - scrittura sul perf buffer.
- Definizione della *callback* user-space che sarà invocata ogni qualvolta saranno disponibili dei dati sul perf buffer;
- Lettura dei dati provenienti dal watcher e sincronizzazione - tramite scrittura su mappe eBPF - dello stato della VNF.

6.3.1 Iniezione del codice eBPF

L'iniezione del codice eBPF è effettuata dal thread che effettua la scrittura su etcd sfruttando il client precedentemente descritto:

```
bpf = new ebpf::BPF();
auto init_res = bpf->init(BPF_MONITOR_CODE);
if (init_res.code() != 0) {
  throw std::runtime_error(init_res.msg());
}
```

Listing 6.9. Iniezione del codice eBPF

In questa fase il verificatore controlla che il codice sia ben formato, non presenti cicli e abbia una dimensione inferiore a 4096 istruzioni.

6.3.2 Definizione dei tracepoint

In questa fase si vanno ad associare i tracepoint alle relative funzioni eseguite dal programma eBPF:

```
auto attach_res =
    bpf->attach_tracepoint("bpf:bpf_map_update_elem",
    "on_bpf_map_update_elem");
if (attach_res.code() != 0) {
  throw std::runtime_error(attach_res.msg());
}

attach_res = bpf->attach_tracepoint("bpf:bpf_map_delete_elem",
    "on_bpf_map_delete_elem");
if (attach_res.code() != 0) {
  throw std::runtime_error(attach_res.msg());
}
```

Listing 6.10. Associazione dei tracepoint alle funzioni eseguite nel kernel

Come mostrato dal listato, vengono associati i tracepoint `bpf:bpf_map_update_elem` e `bpf:bpf_map_update_elem` a due differenti funzioni, che si occuperanno di gestire la modifica e la cancellazione di un elemento all'interno di una mappa. Le funzioni `on_bpf_map_update_elem` e `on_bpf_map_delete_elem` sono state precedentemente descritte nella sezione 6.1.

6.3.3 Perf buffer

Il perf buffer è la struttura dati che consente di ricevere informazioni dal programma eseguito nel kernel. La lettura dal buffer avviene definendo una callback, che sarà delegata a ricevere i dati e a processarli:

```
auto open_res = bpf->open_perf_buffer("events", &handle_output);
if (open_res.code() != 0) {
  throw std::runtime_error(open_res.msg());
}

bpf->poll_perf_buffer("events");
```

Listing 6.11. Definizione della funzione delegata alla lettura dal perf buffer

Le informazioni presenti nel buffer *events* - descritto nella sezione 6.1 - saranno passati alla funzione `handle_output`. La chiamata `poll_perf_buffer` inizializza questo processo.

6.3.4 Lettura dal perf buffer

Dopo aver registrato la callback, la funzione `handle_output` sarà delegata alla lettura dei dati ricevuti dal perf buffer. In particolare, la funzione ha il compito principale di leggere il nome della mappa eBPF che ha generato l'evento e convertire

i dati nella struttura opportuna. L'ultimo passo è quello di generare delle stringhe compatibili col linguaggio Go e invocare la funzione descritta nella sezione 6.2.1, che si occuperà della scrittura su etcd.

```
void handle_output(void *cb_cookie, void *p_data, int data_size) {
  struct data_def *data = static_cast<struct data_def*>(p_data);

  bool del = data->event == D;
  std::string key = std::to_string(data->name);

  // ...

  void *raw_key = malloc(data->key_len);
  std::memcpy(raw_key, data->key, data->key_len);

  void *raw_val;
  if (!del) {
    raw_val = malloc(data->val_len);
    std::memcpy(raw_val, data->val, data->val_len);
  }

  switch (data->name) {

  // ...

  }

  GoString endpoint = {etcd_endpoint.c_str(), (long
      int)strlen(etcd_endpoint.c_str())};
  GoString go_key = {key.c_str(), (long int)strlen(key.c_str())};
  GoString go_value = {value.c_str(), (long
      int)strlen(value.c_str())};

  Write(endpoint, go_key, go_value);

  free(raw_key);
  free(raw_val);
}
```

Listing 6.12. Funzione che esegue la lettura dal perf buffer e converte i dati nel formato opportuno

Il primo passo osservabile nel listato è l'utilizzo della struttura `data_def` - la cui esplicazione è stata fatta nella sezione 6.1 - da cui si legge il tipo di evento (*create* o *delete*) e il nome della mappa eBPF interessata. A questo punto la funzione legge i restanti campi presenti nella struttura, ossia chiave e valore e rispettiva lunghezza in byte, il cui utilizzo è anch'esso spiegato nella sezione 6.1. Successivamente la funzione va a costruire quelle che saranno le stringhe da inviare a etcd: innanzitutto

si vanno a tradurre i dati letti dal perf buffer nel formato opportuno (deducibile dal campo `data_name`), per poi invocare la funzione che scrive su etcd le due stringhe `go_key` e `go_value`.

6.3.5 Interazione col watcher

La lettura dei dati inviati dal watcher è assegnata a un thread, che ha lo scopo di rimanere in ascolto su una connessione instaurata in *localhost* sulla porta 12000. Il watcher (descritto nella sezione 6.2.2) scriverà dati sul socket nei seguenti casi:

- ricezione di una notifica di aggiornamento/eliminazione su uno specifico range di chiavi da parte di etcd;
- richiesta da parte del watcher a etcd del *dump* dell'intera base dati al momento dell'accensione della VNF.

In entrambi i casi la lettura dei dati rimane identica all'interno di Iovnet, essendo i dati inviati dal watcher indipendenti rispetto alle due casistiche. Il procedimento può essere visto come l'inverso descritto nelle sezioni 6.1 prima e 6.3.4 poi:

- ricezione di informazioni da etcd;
- individuazione della mappa eBPF soggetta a modifica e conversione delle informazioni in strutture dati specifiche;
- scrittura dei dati nella mappa eBPF.

Capitolo 7

Validazione

In questo capitolo sarà presentato l'ambiente di test e le configurazioni utilizzate per valutare numericamente la bontà dell'implementazione. In particolare, seguirà prima una descrizione del load balancer, ossia la VNF impiegata per testare la soluzione, seguita dalla descrizione dell'architettura di test e, infine, dalla presentazione dei risultati.

7.1 Load Balancer

Il Load Balancer è un componente che si occupa della distribuzione del traffico di rete all'interno di un cluster, migliorando la responsività e incrementando la disponibilità delle applicazioni. Nell'approccio tradizionale il dispositivo è utilizzato come *reverse proxy*[1] collocato tra il client e la *server farm*, e accetta connessioni in ingresso distribuendole tra i vari *server di backend* con un comportamento che può variare a seconda della specifica implementazione e delle policy applicate (e.g. Figura 7.1). In questo modo è possibile ridurre il carico di lavoro di un singolo server, oltre che evitare che un server applicativo diventi uno SPoF (Single Point of Failure) [23].

Il load balancing è il metodo più semplice per scalare orizzontalmente un'infrastruttura di server applicativi: all'aumentare di richieste, nuovi server possono essere facilmente aggiunti al pool di risorse disponibili, e il Load Balancer inizierà immediatamente a inoltrare il traffico ai nuovi server [23]. Le applicazioni web, inoltre, non necessitano di essere a conoscenza di ogni singolo servizio in quanto devono inoltrare tutte le richieste verso un unico indirizzo IP, quello del Load Balancer.

[1] https://www.nginx.com/resources/glossary/reverse-proxy-server/

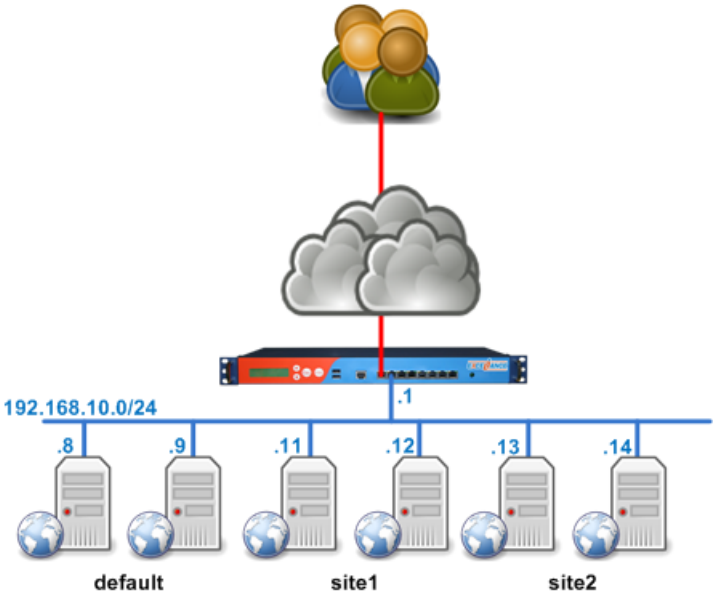

Figura 7.1. Reverse-proxy Load Balancer [24]

7.1.1 iov-lbrp

L'implementazione del load balancer utilizzata come punto di partenza è IOV-lbrp - ossia è integrata all'interno del framework Iovnet - che integra le componenti descritte nella sezione 5.1. In particolare, si tratta di un reverse-proxy load balancer, dove:

- **Load balancer** indica un servizio il cui compito è bilanciare il carico di lavoro dato dal traffico del sistema (e.g. in un data center), distribuendolo su più percorsi di rete differenti. Il prototipo utilizzato implementa un load balancer di livello 4 (livello di trasporto);
- **Reverse-proxy** indica un server posizionato frontalmente al *backend* con il compito di distribuire le richieste provenienti dai client verso più server. Il reverse-proxy intercetta tutte le richieste destinate a uno specifico *Virtual IP* e le invia a un server di backend situato in una subnet differente.

In accordo con la sua logica, il load balancer sceglie il server di backend a cui inviare la richiesta e sostituisce l'indirizzo IP di destinazione (*Virtual IP*) con l'indirizzo del server reale. Il processo di sostituzione dell'indirizzo IP è eseguito in entrambe le direzioni, sia per il traffico in ingresso che per il traffico in uscita. Inoltre, questa funzione di rete supporta tutti i tipi di traffico, ma gestisce esclusivamente TCP, UDP e ICMP; tutti gli altri pacchetti, detti *pacchetti sconosciuti* (e.g. ARP, IPv6), sono semplicemente reindirizzati sulla porta opposta così come sono.

7.1.2 Architettura

Il load balancer ha due porte con le quali può individuare il tipo di traffico: *incoming* e *outcoming* (e.g. Figura 7.2).

- **Porta di frontend**: è l'interfaccia del load balancer a cui arrivano tutte le richieste. Essa connette il load balancer con i client che inviano le richieste al server virtuale;
- **Porta di backend**: è l'interfaccia che connette i load balancer ai server di backend.

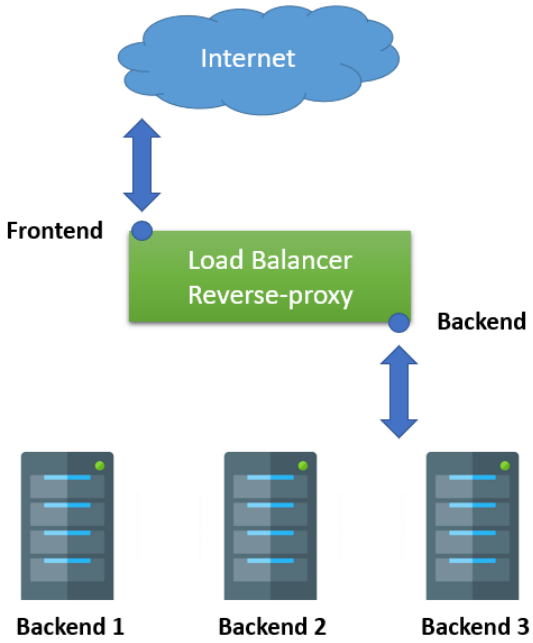

Figura 7.2. Schema delle porte del load balancer reverse-proxy

Data la sua implementazione (reverse-proxy), l'architettura del load balancer può essere suddivisa in due sotto architetture a seconda che il traffico sia in ingresso o in uscita. Di fatto, si ha:

- **Ingress**, il cui compito principale è quello di distribuire i pacchetti IP che hanno come destinazione il Virtual IP, mentre tutti gli altri pacchetti vengono lasciati passare così come sono;
- **Egress**, il cui compito principale è quello di fare da reverse proxy per tutti i pacchetti precedentemente processati, inserendo il Virtual IP come indirizzo sorgente.

Ingress

In questo scenario, il load balancer riceve e intercetta tutto il traffico la cui destinazione è il Virtual IP. Successivamente, il servizio di rete attua il processo di *load balancing*, scegliendo un backend relativo al Virtual IP e cambiando indirizzo IP di destinazione e porta (se necessario) (e.g. Figura 7.3).

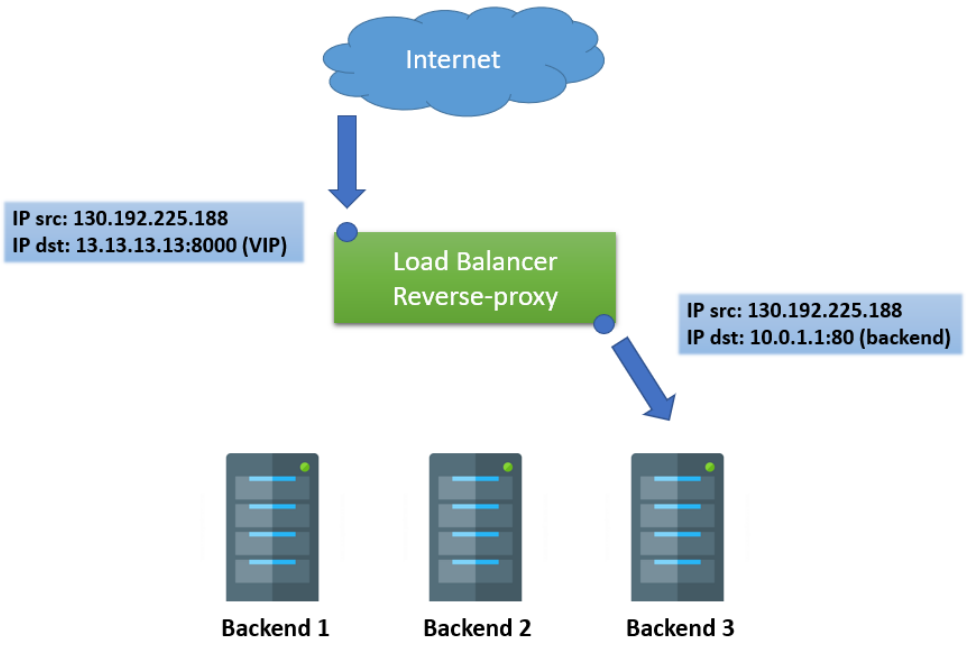

Figura 7.3. Architettura ingress del load balancer

Come è possibile vedere in figura, il load balancer controlla che l'IP di destinazione sia un Virtual IP (13.13.13.13:8000 in questo esempio) precedentemente configurato, per poi scegliere l'indirizzo IP del backend reale a cui inviare il pacchetto (10.0.1.1:80 in questo esempio).

Egress

Questa architettura fa riferimento allo scenario in cui i pacchetti provengono dalla porta di backend del load balancer, che deve quindi fare da reverse-proxy solo per il traffico che è stato precedentemente processato (il traffico non gestito dal load balancer viene semplicemente inoltrato). L'algoritmo prevede che la funzione di rete calcoli l'*hash* sui parametri che identificano la sessione, tramite il quale controlla se lo stato è stato precedentemente memorizzato (e.g. Figura 7.4):

- Se il riscontro è positivo, allora il load balancer fa da reverse-proxy e sostituisce l'indirizzo IP sorgente del backend reale col Virtual IP;

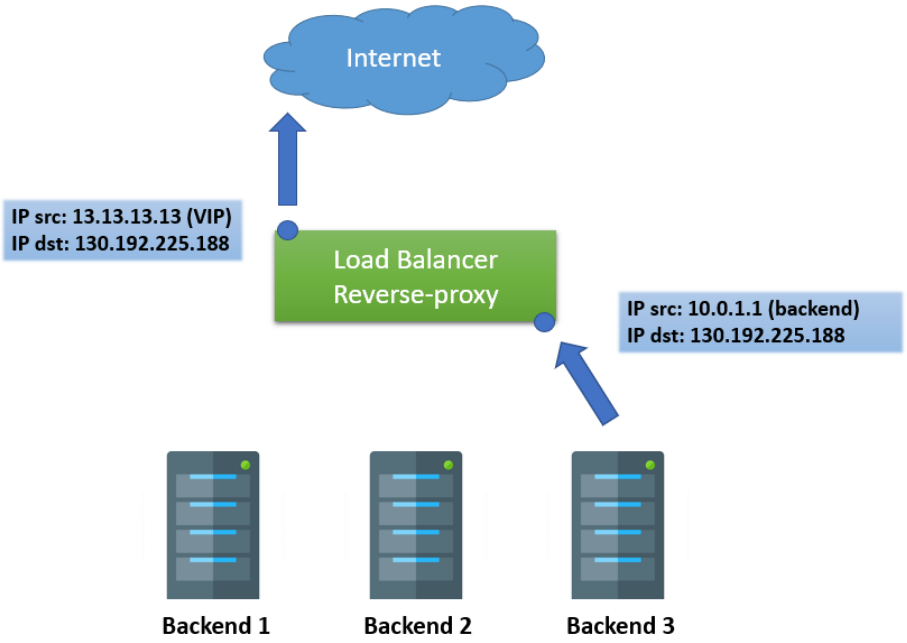

Figura 7.4. Architettura egress del load balancer

- In caso contrario, il pacchetto è inoltrato in maniera trasparente.

7.2 Ambiente di test

I test sono stati condotti in un ambiente composto da tre macchine (e.g. Figura 7.5):

- **Server 1**, dotato di CPU Intel i7-4770 @ 3.4 GHz x 8, RAM 32 GB, SO Ubuntu 16.04 LTS, Kernel 4.15;
- **Server 2**, dotato di CPU Intel i7-3770 @ 3.4 GHz x 8, RAM 32 GB, SO Ubuntu 16.04 LTS, Kernel 4.15;
- **Client**, dotato di CPU Intel i7-7500U @ 2.7 GHz x 2, RAM 16 GB, SO Ubuntu 16.04 LTS, Kernel 4.15;

In particolare, i Server ospiteranno Iovnet e i load balancer, mentre il Client, oltre a inoltrare le richieste ai server, farà anche da nodo etcd.

La Figura 7.6 mostra più nel dettaglio la configurazione dell'infrastruttura di rete e le interconnessioni tra le componenti.

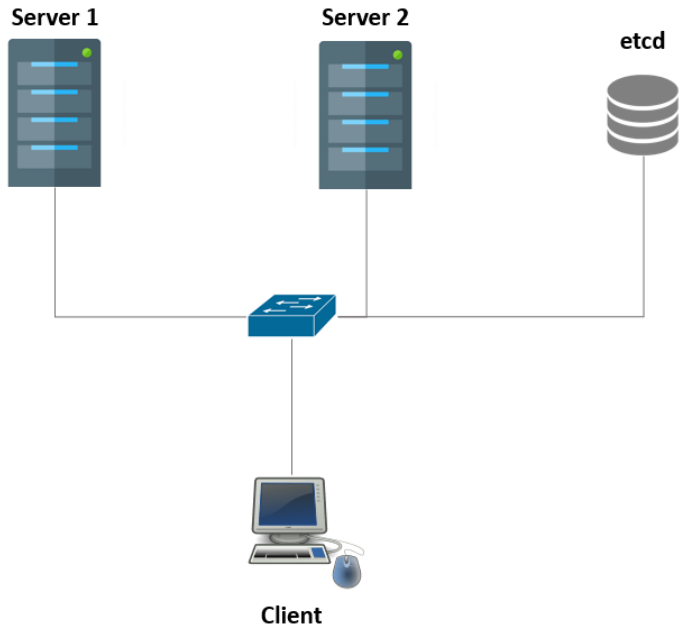

Figura 7.5. Schema delle componenti utilizzate per la validazione

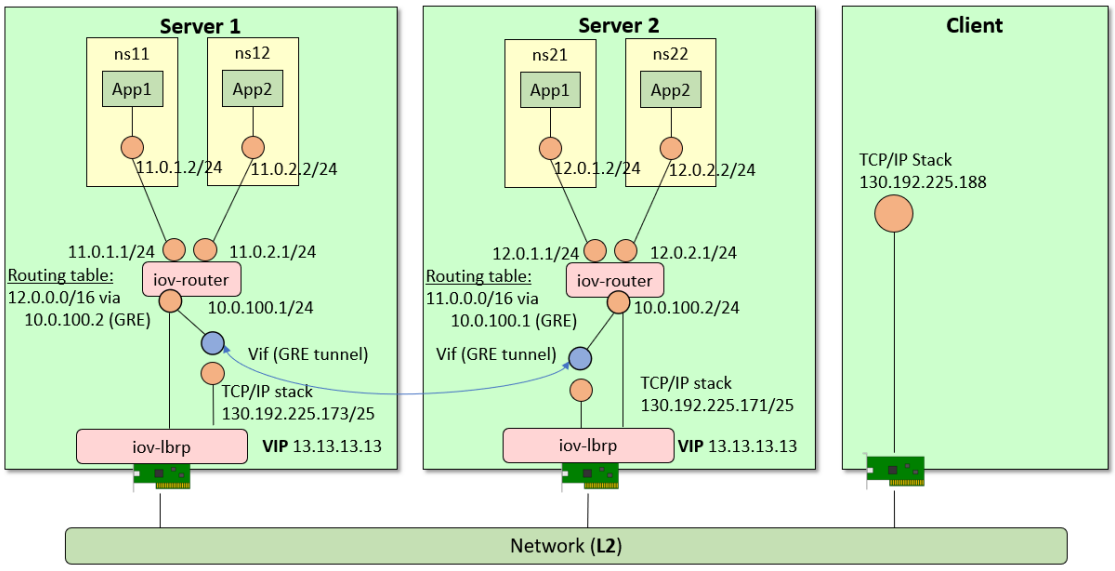

Figura 7.6. Topologia della rete nell'ambiente di test

La rete è stata configurata in modo da consentire ai *namespace* dei due server di parlarsi tramite un *tunnel GRE*[2], instaurato tra le due macchine. Inoltre, per quando riguarda la configurazione degli indirizzi, tutti i computer sono nella stessa sottorete, mentre il VIP dei load balancer è 13.13.13.13. Il funzionamento del tunnel è garantito dalla configurazione delle rotte del router virtuale (iov-router), e dalla configurazione dei due load balancer. Questi ultimi, infatti, sono stati impostati per catturare tutto il traffico intercettato sulla NIC; di conseguenza, devono passare allo stack TCP/IP di Linux tutti i pacchetti con ethertype GRE, in modo che siano correttamente inoltrati verso l'interfaccia virtuale utilizzata dal tunnel. In ultimo, sono state utilizzate connessioni Gigabit Ethernet.

7.3 Scenario

Per una questione di semplicità nel deploy della topologia, tutte e tre le macchine si parlano al livello 2 dello stack ISO/OSI. Tuttavia lo scenario di riferimento, riproducibile fedelmente tramite un'opportuna configurazione, è rappresentato in Figura 7.7.

La situazione presentata in figura, presa come riferimento nella pianificazione del test, rispecchia uno scenario molto comune all'interno di infrastrutture dedicate ad ospitare data center. In particolare, si osserva la presenza di un primo load balancer che sceglie un server di backend a cui inoltrare le richieste. Successivamente, il secondo load balancer decide a quale namespace inoltrare il traffico (si noti che potrebbe inserire come indirizzo IP di destinazione anche quello di un namespace allocato su una macchina fisica differente). I due load balancer interessati dal traffico seguono spesso logiche diverse e, in molti casi, il primo load balancer non è sotto il controllo del gestore del data center. Questo si traduce in attuazione di politiche, da parte dei load balancer differenti, non coerenti tra loro. Si potrebbe verificare, ad esempio, che il traffico sia prima indirizzato verso il Server 1 e, successivamente, verso un namespace sul Server 2. Questa, come già accennato, è una situazione molto comune che comporta un percorso non ottimizzato del pacchetto, in quanto questo deve poi tornare indietro seguendo la stessa strada percorsa all'andata. La Figura 7.8 spiega meglio questo concetto.

[2] https://tools.ietf.org/html/rfc2784

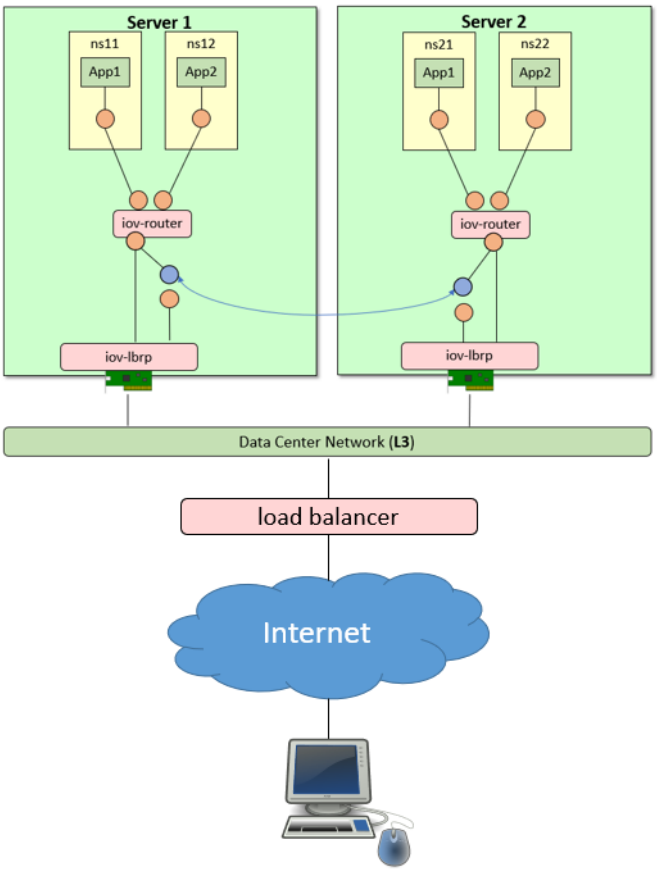

Figura 7.7. Scenario di riferimento

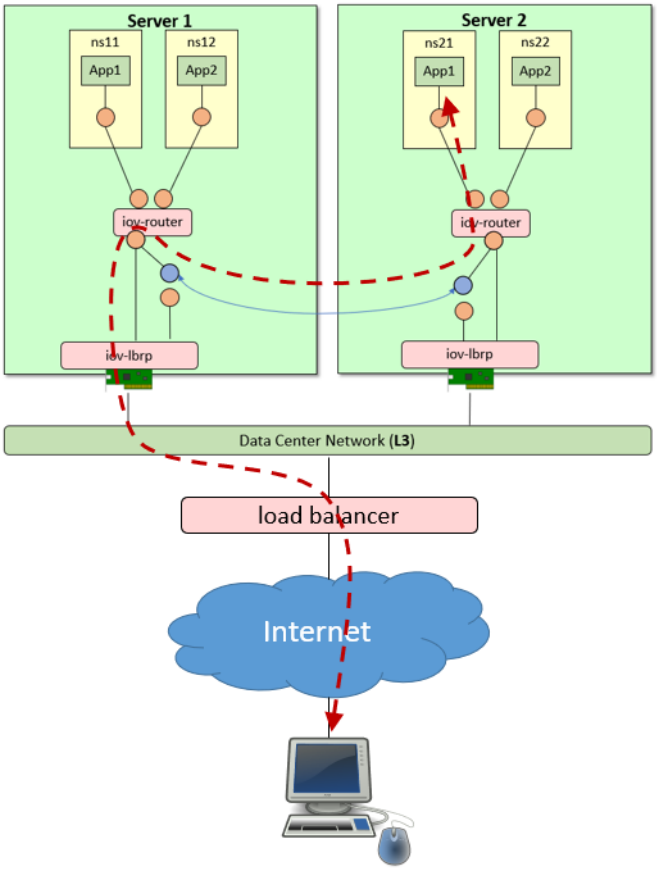

Figura 7.8. Percorso non ottimizzato del traffico

7.3.1 Ottimizzazione

Una delle cause della problematica sopra citata risiede nel fatto che in un'implementazione standard il load balancer presente sul Server 2 non è in grado processare il traffico in uscita che, necessariamente, deve attraversare il tunnel per poi poter essere instradato verso il Client. Una possibile applicazione del lavoro svolto in questa tesi è proprio la risoluzione della suddetta problematica (e.g. **routing asimmetrico**): la distribuzione dello stato dei load balancer consentirebbe alla VNF presente sul Server 2 di processare il traffico in uscita (e.g. rimpiazzare l'indirizzo IP sorgente del namespace con il Virtual IP). Il risultato dell'ottimizzazione è rappresentato graficamente in Figura 7.9.

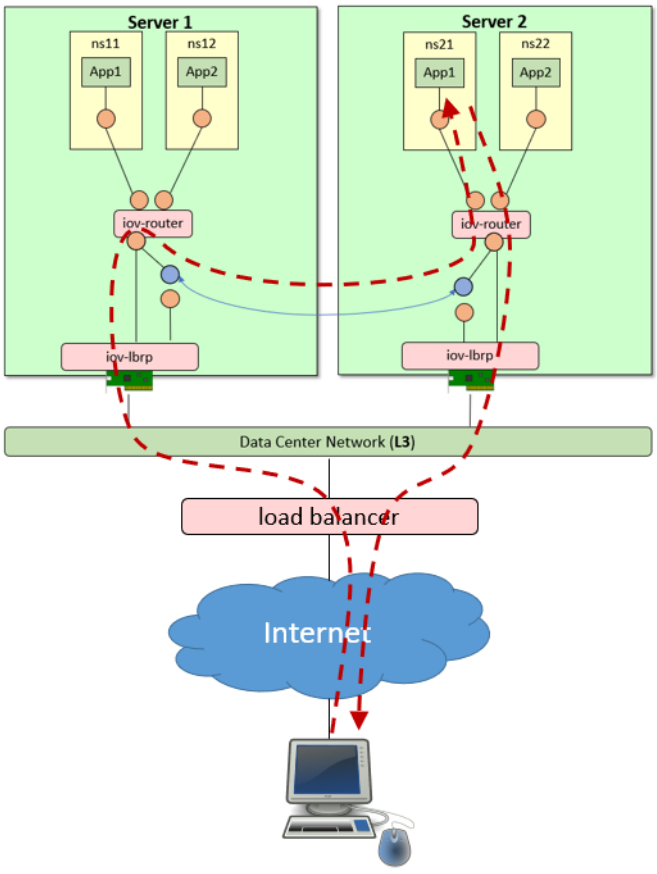

Figura 7.9. Percorso del traffico ottimizzato

7.4 Test

La validazione consiste, come già anticipato, nel testare l'effettiva efficacia dell'implementazione proposta; in particolare, valutare in termini numerici i benefici derivanti dall'ottimizzazione del percorso compiuto da un pacchetto all'interno di un cluster. Questa procedura si divide sostanzialmente in due fasi:

1. **Studio dello scenario in assenza di ottimizzazioni**
    - Utilizzo di Iovnet e delle componenti implementate.
    - Deploy della topologia illustrata in Figura 7.6, a simulare la situazione descritta in Figura 7.7. Ciò è stato possibile configurando opportunamente il client: il suo *default gateway* non è altro che il router virtuale posto sul Server 1. Di conseguenza, le richieste rivolte al Virtual IP saranno automaticamente dirette a questo server (essendo il VIP 13.13.13.13, ossia appartenente a una sottorete diversa), simulando appunto la presenza di un primo livello di load balancing che ha scelto il Server 1 come backend per le richieste provenienti dal client.
    - Configurazione del load balancer in modo da forzare la scelta del backend, prediligendo il namespace presente sul Server 2.
2. **Studio dello scenario con path ottimizzato**
    - Riconfigurazione del router virtuale del Server 2 in modo che inoltri il traffico di ritorno non più verso l'interfaccia virtuale del tunnel, ma attraverso il load balancer.

In entrambi gli scenari, il load balancer relativo al Server 1 è configurato in modo da avere un servizio all'indirizzo 13.13.13.13 e porta 8000, per traffico di tipo TCP, UDP e ICMP, e una serie di backend agli indirizzi dei namespace. Il secondo load balancer è privo di configurazione, in quanto avrà a disposizione tutte queste informazioni al momento dell'accensione, leggendo da etcd. Inoltre, a runtime saranno generate nuove informazioni relative alle sessioni create dal load balancer, anch'esse distribuite tra tutte le istanze.

7.5 Risultati

Le valutazioni numeriche si sono focalizzate sul confronto dei tempi di latenza e delle performance di connessioni TCP nei due scenari appena presentati. Inoltre, è stata testata l'efficienza della soluzione implementata in termini di latenza del monitor lato kernel e dell'interazione con etcd. In particolare, sono stati utilizzati i seguenti tool:

- **ping** per calcolare l'*RTT*, ossia il tempo impiegato dal pacchetto per compiere il tragitto in entrambe le direzioni;
- **iperf3**[3] per misurare la bontà della connessione TCP in termini di throughput e ristrasmissioni TCP effettuate;
- **ftrace**[4], per leggere gli eventi che si verificano nel kernel e analizzarne i timestamp.

7.5.1 Tempi di latenza

Per calcolare la latenza sono stati effettuati cento tentativi di ping da parte del client - successivamente mediati - tutti con lo 0% di *packet-loss*. Il client ha eseguito il comando **ping 13.13.13.13 -c 100**.

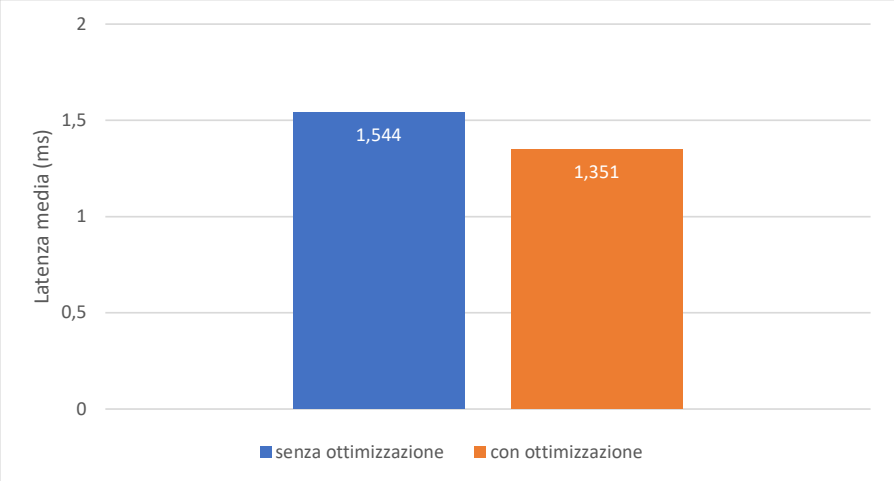

Figura 7.10. Latenza media

Il valore del RTT medio in presenza dell'ottimizzazione, ossia con la distribuzione dello stato del load balancer e successivo instradamento intelligente del pacchetto, risulta essere pari all'87,5% dell'RTT medio misurato nello scenario standard.

[3] https://iperf.fr/

[4] https://www.kernel.org/doc/Documentation/trace/ftrace.txt

In altre parole, il guadagno si attesta sul 12,5%. Questo risultato è dovuto sostanzialmente al *path* ottimizzato: il traffico, infatti, può compiere un percorso più intelligente grazie alla maggior flessibilità dei load balancer, e arrivare prima a destinazione.

7.5.2 Performance della connessione TCP

Le misurazioni circa le performance della connessione TCP sono state eseguite con iperf3 (), lanciando dieci test da dieci secondi ciascuno, e calcolando le medie dei valori ottenuti. Sul namespace impostato come backend è stato eseguito il comando **sudo ip netns exec ns21 iperf3 -s -p 800**, mentre sul client **sudo iperf3 -p 8000 -c 13.13.13.13**.

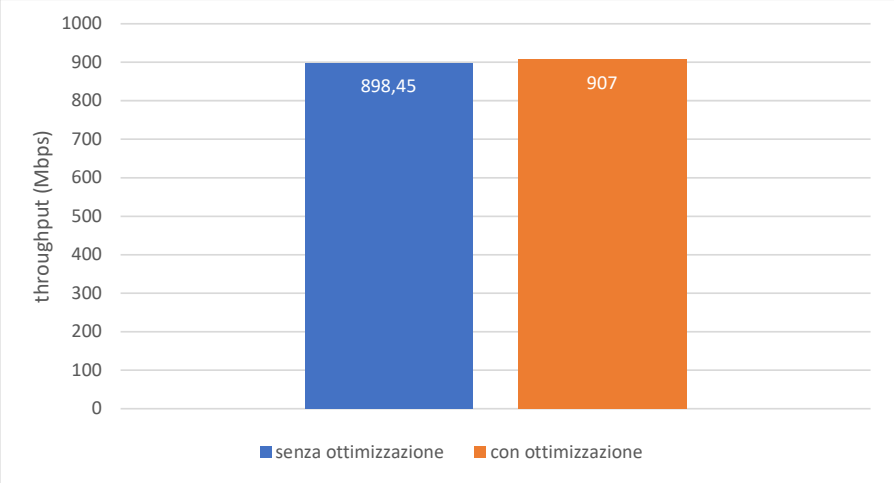

Figura 7.11. Throughput medio

Il grafico 7.11 mostra che le performance, in termini di throughput, beneficiano di un incremento dell'1% nella soluzione ottimizzata. Analizzando il throughput si nota subito che i maggiori benefici dati da questa implementazione sono apprezzabili soprattutto in termini di latenza. Questa, infatti, influisce solo in minima parte sulle performance in una trasmissione TCP, in cui la latenza è attenuata da una trasmissione ad alta velocità.

Il grafico 7.12, invece, evidenzia come il numero medio di ritrasmissioni TCP sia radicalmente calato quando è stata sfruttata la centralizzazione dello stato per ottimizzare il traffico. Più precisamente, il calo è stato circa del 96%. Questo dato

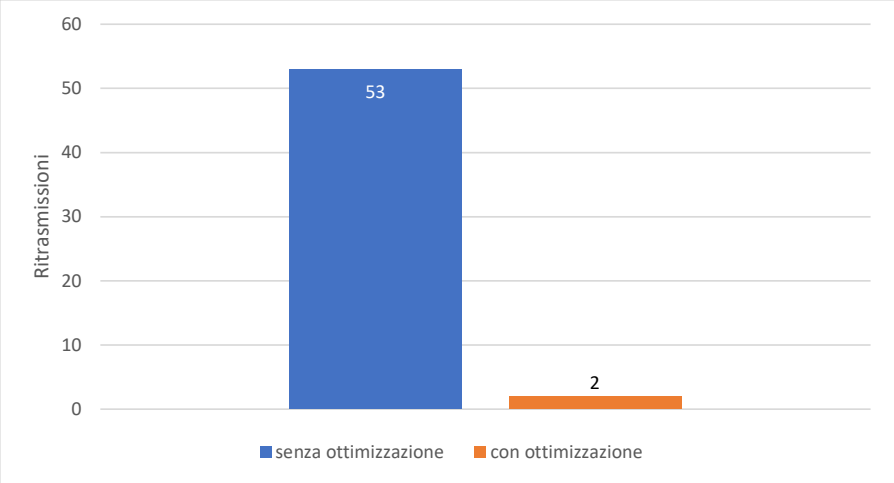

Figura 7.12. Ritrasmissioni medie

dimostra nuovamente come la riduzione della latenza vada a influenzare specifici parametri di una connessione TCP. Il calo così netto delle ritrasmissioni, infatti, si spiega con una minore attesa degli ACK nella soluzione ottimizzata.

7.5.3 Efficienza dell'implementazione

In ultima analisi, sono state eseguite delle misurazioni utilizzando ftrace, un tool di linux per tracciare gli eventi del kernel. Nello specifico, sono stati osservati i timestamp relativi agli eventi `bpf_map_update_elem` e `bpf_map_lookup_elem`.

Durata dell'intero processo

Per misurare l'efficienza dell'intero processo, che consiste in:

- aggiornamento/eliminazione di una mappa eBPF nel load balancer;
- scrittura delle informazioni su etcd;
- ricezione dei dati da parte del watcher;
- sincronizzazione della mappa eBPF interessata da modifica;

sono state spostate entrambe le istanze del load balancer sulla stessa macchina fisica, in modo da ottenere valori di timestamp affidabili. L'interazione con etcd e col client è stata lasciata invariata.

Il test quindi, è stato eseguito provocando una modifica dello stato di un load balancer, in modo che questa venisse replicata sull'altra istanza. In seguito, sono state confrontati timestamp relativi agli eventi dei due load balancer (usando ftrace) ed è stato calcolato il tempo medio necessario per far sì che lo stato fosse sincronizzato.

Il risultato mostra una latenza media di **96 millisecondi**.

Programma di monitor

Questa misurazione, ottenuta effettuando il test precedente, consiste nel calcolare il tempo impiegato dal software eBPF, che esegue monitoraggio a basso livello degli eventi nelle mappe eBPF (utilizzate da load balancer), per rilevare l'evento e comunicarlo allo user-space.

Il test ha prodotto un tempo medio di **62 millisecondi**. Questo è il tempo necessario per:

- rilevare l'evento su una mappa eBPF;
- controllare che il file descriptor sia relativo a una mappa del load balancer;
- costruire la struttura dati da inviare allo spazio utente e scriverla sul perf buffer;
- ricevere i dati nello spazio utente e convertirli nella struttura dati opportuna;
- scrivere l'evento su etcd.

Etcd

La determinazione della latenza introdotta dalla rete e da etcd (scrittura dei dati e segnalazione al watcher), è stata calcolata indirettamente sottraendo i due valori estrapolati nelle misurazioni precedenti.

Ne consegue che la latenza è pari a **34 millisecondi**.

Valutazione

L'analisi dei dati relativi alle componenti implementate fornisce un'indicazione importante circa le tempistiche impiegate dalle singole componenti per poter mettere in atto la distribuzione dello stato delle VNF.

Il primo valore presentato, ossia il tempo totale richiesto dall'intero processo, è forse il più significativo in quanto fornisce una valutazione complessiva della soluzione proposta. Questo, però, è un valore che va interpretato. Lo stato infatti,

non è composto da una singola entry in una mappa eBPF, ma da centinaia o addirittura migliaia di elementi. Questo significa che quando una VNF modifica il proprio stato intervenendo su più elementi nell'arco di un tempo breve, il protocollo impiega in media 96 millisecondi per distribuire le modifiche a tutte le istanze. C'è un altro dato però: se si considera il load balancer, nel test eseguito ha modificato trenta entry di una particolare mappa in meno di un millisecondo. Sebbene ci sia una latenza, quindi, questa è stata attenuata dall'alto rate di modifiche: tutto il processo, infatti, è durato **97 millisecondi**. Ciò significa che la latenza introdotta ha influito solo inizialmente, ma a regime i tempi sono gli stessi che si avrebbero operando in locale.

Capitolo 8

Conclusioni

In questo lavoro di tesi è stato proposto un nuovo approccio per la distribuzione dello stato delle VNF in modo da migliorare la flessibilità e le performance del networking all'interno di un cluster di macchine.

È stato visto come l'utilizzo di eBPF per monitorare lo stato delle VNF consenta di ridurre le modifiche da apportare al servizio di rete e di effettuare il deploy di un modulo unico fruibile da tutte le VNF in Iovnet.

L'architettura e l'implementazione si discostano da altri lavori correlati soprattutto in termini di gestione dello stato: in questa tesi, infatti, si propone prima una centralizzazione dello stato, poi una distribuzione dello stesso a tutte le istanze dello stesso servizio di rete. Questo si traduce in un'alta flessibilità delle VNF, col rischio di un potenziale sovraccarico dello storage locale in caso di uno stato globale notevolmente esteso.

La centralizzazione dello stato, inoltre, consente di accendere una VNF e di reperire tutte le informazioni necessarie al processamento del traffico, abbattendo di fatto i tempi di latenza causati dalla fase di apprendimento della VNF, e azzerando possibili errori nelle decisioni determinati dalla mancata conoscenza dello stato globale.

I test hanno dimostrato come in un caso d'uso reale una soluzione di questo tipo sia applicabile, portando benefici tangibili in termini di performance del networking. Inoltre, è stato dimostrato come le latenze introdotte dall'implementazione si vadano notevolmente ad attenuare in caso di intensa attività di modifica dello stato da parte della VNF.

8.1 Sviluppi futuri

I test svolti nel precedente capitolo, atti a validare l'implementazione proposta, mostrano solo una delle innumerevoli problematiche che questo elaborato può aiutare a risolvere.

Si pensi, infatti, al caso in cui una VNF subisca un *crash*, oppure la macchina fisica che la ospita sia soggetta a un guasto. Attivando un servizio di *fault detecton* all'interno del data center sarebbe possibile avviare una nuova istanza della VNF e, di lì in poi, demandare tutte le operazioni al protocollo: il processo automatizzato, infatti, prevede che la VNF in primo luogo recuperi lo stato dalla base dati, e poi inizi a processare il traffico. In questo modo è necessario semplicemente ridirezionare il traffico verso nuova istanza.

Allo stesso modo, sarebbe possibile *fare scale-out* della VNF avviando una nuova istanza e inviando parte del traffico ad essa.

Di conseguenza, l'elaborato fornisce sicuramente un valido punto di partenza per lo sviluppo di ulteriori componenti software che facciano fault detection, oppure che gestiscano il sovraccarico replicando opportunamente le funzioni di rete. Processi automatizzati di questo tipo, in connubio con la soluzione proposta, permetterebbero di incrementare ulteriormente le performance e la flessibilità dal punto di vista del controllo del networking.

Bibliografia

[1] Brandon Butler. *What's the difference between SDN and NFV?* 2017. URL: https://www.networkworld.com/article/3206709/lan-wan/what-s-the-difference-between-sdn-and-nfv.html.

[2] *What is OpenFlow? Definition and How it Relates to SDN*. URL: https://www.sdxcentral.com/sdn/definitions/what-is-openflow/.

[3] Giancarlo Marasso Simone Ruffino Luigi Grossi Eugenio Maffione. *SDN e NFV: quali sinergie?* URL: http://www.telecomitalia.com/content/dam/telecomitalia/it/archivio/documenti/Innovazione/MnisitoNotiziario/2014/2-2014/capitolo-5/SDN%20e%20NFV%20quali%20sinergie.pdf.

[4] George Coulouris. *Distributed systems : concepts and design*. Boston: Addison-Wesley, 2012. ISBN: 978-0132143011.

[5] *What Is Location Transparency?* URL: http://www.wisegeek.com/what-is-location-transparency.htm.

[6] Seth Gilbert e Nancy Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (feb. 2012), pp. 30–36. ISSN: 0018-9162. DOI: 10.1109/MC.2011.389. URL: http://dx.doi.org/10.1109/MC.2011.389.

[7] Ramez Elmasri. *Fundamentals of database systems*. Reading, Mass: Addison-Wesley, 2000. ISBN: 0-201-54263-3.

[8] Gregory Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Reading, Mass: Addison-Wesley, 2000. ISBN: 0-201-35752-6.

[9] Aaron Gember-Jacobson e Aditya Akella. "Improving the Safety, Scalability, and Efficiency of Network Function State Transfers". In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox '15. London, United Kingdom: ACM, 2015, pp. 43–48. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785997. URL: http://doi.acm.org/10.1145/2785989.2785997.

[10] M. Peuster e H. Karl. "E-State: Distributed state management in elastic network function deployments". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. 2016, pp. 6–10. DOI: 10.1109/NETSOFT.2016.7502432.

[11] Aaron Gember et al. “Toward software-defined middlebox networking”. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM. 2012, pp. 7–12.

[12] Shriram Rajagopalan et al. “Split/Merge: System Support for Elastic Execution in Virtual Middleboxes.” In: *NSDI*. Vol. 13. 2013, pp. 227–240.

[13] Aaron Gember-Jacobson et al. “OpenNF: Enabling Innovation in Network Function Control”. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (ago. 2014), pp. 163–174. ISSN: 0146-4833. DOI: 10.1145/2740070.2626313. URL: http://doi.acm.org/10.1145/2740070.2626313.

[14] X. Shao, L. Gao e H. Zhang. “CoGS: Enabling distributed network functions with global states”. In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. 2017, pp. 1–9. DOI: 10.1109/NETSOFT.2017.8004112.

[15] Murad Kablan et al. “Stateless Network Functions: Breaking the Tight Coupling of State and Processing”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 97–112. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan.

[16] *BPF and XDP Reference Guide*. URL: http://cilium.readthedocs.io/en/stable/bpf/.

[17] *What is eBPF and XDP?* URL: https://github.com/cilium/cilium/.

[18] *IO Visor Project*. URL: https://www.iovisor.org/.

[19] *BPF Compiler Collection (BCC)*. URL: https://github.com/iovisor/bcc.

[20] William Cohen Jason Baron. *The Linux Kernel Tracepoint API*. URL: https://www.kernel.org/doc/html/v4.15/core-api/tracepoint.html.

[21] Scott James Remnant. *Tracing on Linux*. 2011. URL: http://netsplit.com/tracing-on-linux.

[22] *etcd*. URL: https://coreos.com/etcd/.

[23] *What is load balancing?* URL: https://www.citrix.it/glossary/load-balancing.html.

[24] Baptiste Assmann. *Aloha Load-balancer as a reverse proxy*. 2011. URL: https://www.haproxy.com/blog/aloha-load-balancer-as-a-reverse-proxy/.