ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA - SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA INFORMATICA E DELLE TELECOMUNICAZIONI

COMPOSIZIONE DINAMICA DI FUNZIONI DI RETE VIRTUALI IN AMBIENTI CLOUD

Elaborato in

Applicazioni e Tecniche di Telecomunicazioni

Relatore

Prof. Ing. WALTER CERRONI

Presentato da

FRANCESCO FORESTA

Correlatori

Ing. CHIARA CONTOLI

Ing. GIULIANO SANTANDREA

SESSIONE III

ANNO ACCADEMICO 2013/2014

Alla mia famiglia,
che mi ha sempre sostenuto

Indice

Sommario

Questo documento si interroga sulle nuove possibilità offerte agli operatori del mondo delle Reti di Telecomunicazioni dai paradigmi di Network Functions Virtualization, Cloud Computing e Software Defined Networking: questi sono nuovi approcci che permettono la creazione di reti dinamiche e altamente programmabili, senza disdegnare troppo il lato prestazionale.

L'intento finale è valutare se con un approccio di questo genere si possano implementare dinamicamente delle concatenazioni di servizi di rete e se le prestazioni finali rispecchiano ciò che viene teorizzato dai suddetti paradigmi.

Tutto ciò viene valutato per cercare una soluzione efficace al problema dell'ossificazione di Internet: infatti le applicazioni di rete, dette middle-boxes, comportano costi elevati, situazioni di dipendenza dal vendor e staticità delle reti stesse, portando all'impossibilità per i providers di sviluppare nuovi servizi.

Il caso di studio si basa proprio su una rete che implementa questi nuovi paradigmi: si farà infatti riferimento a due diverse topologie, una relativa al Livello L2 del modello OSI (cioè lo strato di collegamento) e una al Livello L3 (strato di rete).

Le misure effettuate infine mostrano come le potenzialità teorizzate siano decisamente interessanti e innovative, aprendo un ventaglio di infinite possibilità per il futuro sviluppo di questo settore.

Capitolo 1

Introduzione

Negli ultimi anni si è iniziata a intravedere nel mondo delle Reti di Telecomunicazioni quella che sotto molti aspetti è stata definita una vera e propria rivoluzione: la struttura delle reti infatti è stata progressivamente modificata, portando le risorse delle stesse dal centro (core networks) ai margini (edge networks), sfruttando nuovi concetti e paradigmi, come quello del Cloud Computing, quello del Software Defined Networking e della Network Functions Virtualization.

Il Cloud Computing sta portando ad una graduale trasformazione dei servizi di rete, delle risorse di calcolo e di quelle di storage in un modello simile a quello che si ha per l'elettricità, il gas o il telefono, dove le risorse vengono offerte da un provider al client [1]; nel caso del Cloud, questo è svolto permettendo la memorizzazione, archiviazione e/o elaborazione dei dati grazie all'utilizzo di risorse hardware/software distribuite e virtualizzate in rete in un'architettura tipica client-server [2].

L'aspetto caratterizzante di questo fenomeno è connesso alla possibilità di utilizzare hardware e software ubicato geograficamente in qualsiasi zona del mondo, sfruttando la velocità della connessione con la banda larga.

La Sun Microsystems, nota azienda del settore elettronico e software, ha affermato, relativamente al fenomeno cloud: *"at a basic level, cloud computing is simply a means of delivering IT resources as services"* [3], che dà adito a quanto sopra, ossia al cloud come un servizio offerto da un provider ad un client. Una scelta di questo tipo naturalmente permette l'abbattimento dei costi delle infrastrutture tecniche, rendendola una tecnologia vantaggiosa sotto il profilo economico.

È inoltre importante tenere conto di come il paradigma su cui si è basato e si basa tuttora Internet, principalmente incentrato sul forwarding dei pacchetti basandosi sugli indirizzi IP, sia stato negli ultimi tempi modificato; infatti nelle moderne reti i pacchetti IP vengono ulteriormente processati in nodi intermedi delle stesse al fine di far svolgere funzioni aggiuntive, come l'implementazione di NAT, Firewall, load balancers e altri.

Ognuno di questi nodi intermedi è chiamato *middle-box*, dispositivo tipicamente hardware costruito su piattaforma dedicata e che supporta solo un set limitato di funzioni; risulta quindi semplice immaginare come un insieme di middle-boxes proprietarie, peraltro dal costo non indifferente, possano comportare il fenomeno del "vendor lock-in", ossia la situazione di dipendenza del client dal fornitore, ad esempio in attesa di un nuovo firmware per i dispositivi.

Questa è una delle cause che hanno portato all'ossificazione di Internet, comportando agli operatori grandi difficoltà nell'ambito dello sviluppo e della creazione di nuove funzionalità di rete e servizi e portando le reti di telecomunicazioni a perdere il loro carattere intrinseco di dinamicità.

Per uscire da questa situazione si sono quindi studiate nuove soluzioni e nuovi paradigmi atti a rivoluzionare la situazione statica in cui il mercato delle Reti di Telecomunicazioni al momento risulta essere: l'introduzione della *Network Functions Virtualization* [4] va proprio in questa direzione, permettendo all'operatore di creare virtualmente le proprie middle-boxes su hardware generico (economico, ma abbastanza potente), rompendo il legame con i vendor, con la diretta conseguenza di poterle successivamente modificare e riprogrammare dinamicamente a seconda dell'esigenza.

Questo concetto trova spazio nella teoria per cui nel futuro si immagina che tutte le risorse della rete verranno allocate quanto più possibili vicino all'utente (fino al limite in cui queste vengano direttamente implementate nel router del client), mentre le core networks siano solo un insieme di collegamenti ad alta velocità che permettono alle edge networks di comunicare tra di loro [5].

Al fine di far funzionare flessibilmente quanto sopra, si sfrutta un altro importante paradigma nato negli ultimi anni, il *Software Defined Networking* [6]. Questo è un nuovo approccio al computer networking che permette agli amministratori di rete di gestire i servizi della stessa tramite astrazioni di funzionalità a basso livello, disaccoppiando il piano di trasporto dei dati da quello di controllo, direttamente via software; infatti, il paradigma SDN permette all'amministratore di rete di programmare direttamente il piano di trasporto e di astrarre l'infrastruttura sottostante da applicazioni e servizi di rete.

Uno dei meccanismi nato per far comunicare il piano di trasporto con il piano di controllo è il protocollo di comunicazione *OpenFlow* [7] che permette ad un controller remoto di determinare il percorso che dovranno seguire i pacchetti attraverso gli switch di rete.

Risulta quindi immediato che questi nuovi paradigmi portano ad un'innovazione che il mondo delle reti di telecomunicazioni richiede, al fine di ritrovare la dinamicità che lo ha sempre contraddistinto.

In questo documento ci si è interrogati su come sia possibile sviluppare una rete cloud valutando sia il lato client che il lato provider, al fine di comprendere quanto questi nuovi approcci alla rete possano essere globalmente favorevoli al mercato delle reti di telecomunicazioni. È stata infatti implementata una sperimentazione quanto più simile al caso reale di un operatore di rete che dà la possibilità a due utenti, i quali hanno sottoscritto diversi tipi di contratti, di accedere alla rete internet, sfruttando alcuni meccanismi pratici di Software Defined Networking e Network Functions Virtualization. Viene infatti mostrato come un approccio di questo genere possa realmente portare a numerosi benefici e ad infinite possibilità in questo campo, integrando i nuovi paradigmi con i vecchi, permettendo la costruzione di reti dall'alto coefficiente dinamico.

Nel Capitolo 2 saranno esposti più a fondo i concetti teorici relativamente ai paradigmi SDN e NFV, necessari alla comprensione dei successivi casi pratici degli stessi, OpenFlow e OpenStack, che verranno studiati nei dettagli nel Capitolo 3.

Questi ultimi saranno a loro volta sfruttati ai fini del lavoro sperimentale svolto; verrà infatti presentata nel Capitolo 4 la sperimentazione: sarà mostrato come una rete di questo tipo venga creata lato provider, con le necessarie middle-boxes, nel caso in cui venga data la possibilità ai due diversi utenti, residenziale e business, di accedere alla rete internet, sfruttando i meccanismi di cui sopra.

Nei Capitoli 5 e 6 sono invece presenti le parti pratiche della sperimentazione sopra citata, con le relative misure effettuate durante il funzionamento.

Si sfrutterà infatti la Network Functions Virtualization per classificare il traffico ed elaborarlo al fine di dare ad ogni utente una adeguata Quality of Service, modificandola dinamicamente a seconda del contesto in cui la rete si trova e al Service Level Agreement dell'utente; questo traffico verrà inoltrato sfruttando invece i meccanismi del Software Defined Networking, il tutto chiaramente in ambiente cloud.

Infine nell'Appendice A si farà riferimento all'implementazione dell'edge

network di destinazione, presente in un host fisico esterno al cluster, mentre nell'Appendice B verrà mostrato il codice del controller POX utilizzato per la sperimentazione ed alcuni chiarimenti a riguardo.

Capitolo 2

Introduzione al NFV e SDN

Come presentato nel Capitolo 1, nel futuro prossimo il computing, lo storage e i servizi di connettività saranno forniti da infrastrutture software-defined costruite in accordo al paradigma cloud, dove le funzioni di rete possono essere virtualizzate ed eseguite sopra hardware generico. Questo paradigma con molta probabilità verrà reso effettivo ai bordi nelle reti, dove, al momento, la maggior parte delle funzioni richieste (analizzare, classificare e modificare il percorso del traffico) sono tradizionalmente implementate da middle-boxes vendor-dependant.

È quindi chiara la necessità di nuovi paradigmi di rete, atti a permettere agli operatori di rete di selezionare e applicare le specifiche funzioni di rete necessarie per una certa classe di utenti ad un certo tempo, il tutto dinamicamente e flessibilmente. Non deve dunque stupire il vasto eco mediatico generato dall'introduzione nel mondo delle telecomunicazioni dei paradigmi NFV e SDN.

Questi infatti hanno generato una possibilità di evoluzione per le reti così come lo fu per il computing: è infatti da molto tempo che i programmatori sono in grado di implementare sistemi complessi senza la necessità di gestire i singoli dispositivi coinvolti e senza dover interagire in linguaggio macchina, grazie all'introduzione di opportuni livelli di astrazione nell'architettura e di interfacce Open. [8]

Risulta quindi evidente come un nuovo approccio di questo genere possa essere tecnologicamente interessante dal punto di vista degli operatori, delle aziende e degli utenti; è necessario però valutare bene ogni singolo aspetto di questi concetti innovativi.

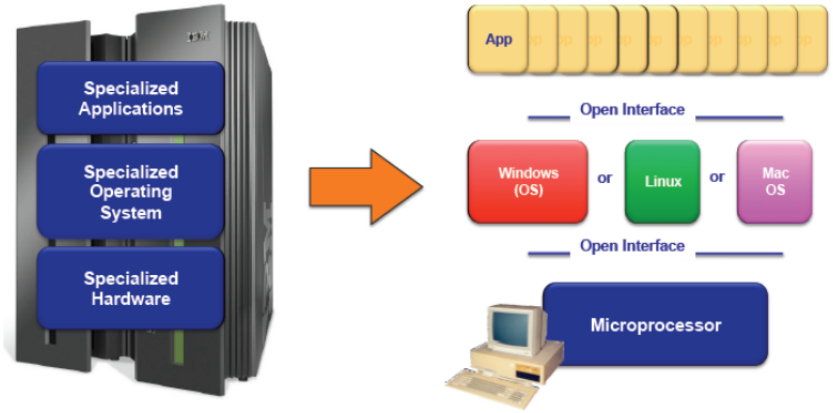

Figura 2.1: Evoluzione del Computing [8]

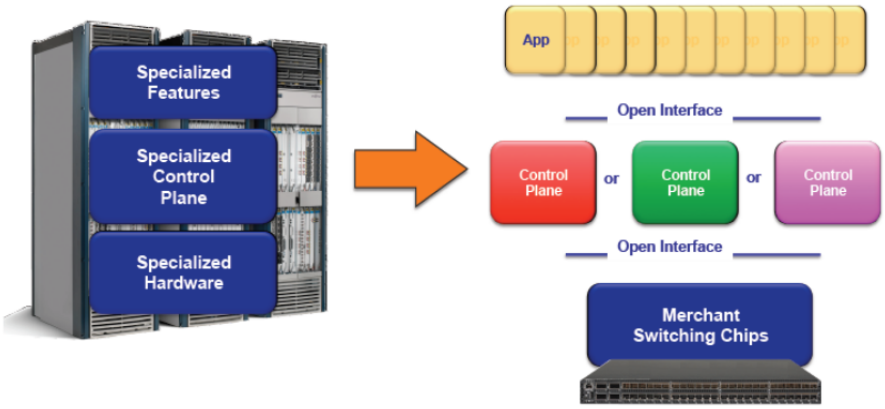

Figura 2.2: Possibile evoluzione del networking [8]

2.1 Network Function Virtualization

Nell'ottobre del 2012, ad una conferenza sul Software Defined Networking e OpenFlow, venne pubblicato un white paper da un gruppo di rappresentanti delle industrie delle telecomunicazioni appartenenti all'European Telecommunications Standards Institute (ETSI), incentrato sul primo concetto di Network Functions Virtualization. [9]

I problemi delle reti moderne riscontrati dagli autori del paper erano quelli presentati nel precedente capitolo: la presenza di una vasta gamma di middleboxes hardware proprietarie non permette di lanciare nuovi servizi di rete senza costi indifferenti e senza difficoltà nell'integrazione con altri componenti della rete; inoltre, la breve vita di questi dispositivi e la necessità di essere rinnovati spesso a causa del progresso sempre più veloce della tecnologia comportano ben pochi benefici.

La proposta presentata dagli autori era quindi quella di sostituire le middleboxes hardware e vendor-dependant sfruttando i meccanismi della virtualizzazione di sistemi operativi, arrivando dunque a implementare delle middleboxes riprogrammabili dinamicamente, poco costose e sfruttando hardware general-purpose.

Un approccio del genere comporta numerose opportunità che possono essere colte, come:

Diminuzione dei costi: gli operatori, che in questo periodo di crisi economica non hanno più grandi capitali da investire, trovano in questa soluzione un risparmio notevole potendo virtualizzare risorse che nelle core networks sarebbero state hardware e vendor-dependant, il tutto offrendo nuovi tipi di servizi a costi minori per l'utente.

Nuove opportunità di business: gli operatori potranno infatti creare nuove offerte, forti anche dei costi minori che la rete impone, al fine di attirare a sè più utenti e ottenere potenzialmente più profitti.

Possibilità di smarcarsi dal vendor: la virtualizzazione delle risorse precedentemente solo hardware comporta anche una possibile fine della dipendenza economica dell'operatore dal vendor.

Multi-tenancy: quanto sopra porta alla possibilità di utilizzare una singola piattaforma per applicazioni, utenti e tenant differenti, permettendo agli operatori di ridurre anche le risorse fisiche necessarie a concedere i servizi richiesti.

Eventuale facilità di gestione: nel caso in cui venisse sviluppata una Northbound API standard, per l'operatore sarebbe molto semplice gestire la rete una volta avviata, portando dunque ad un notevole risparmio non solo in termini economici e permettendo anche una possibile riprogram-

mazione, migrazione, cancellazione o duplicazione delle middle-boxes a seconda delle necessità.

In quest'ultimo caso sono già stati fatti progressi notevoli, sfruttando ad esempio la possibilità di gestire dinamicamente le funzioni di rete virtuali considerando il relativo stato del forwarding SDN [10] o con la creazione di framework che permettono di accelerare le stesse funzioni di rete virtuali [11].

La possibilità di poter gestire dinamicamente questi dispositivi permetterà agli operatori di combinare un aumento della Quality of Service (QoS) dell'utente con un efficiente utilizzo delle risorse di comunicazione.

Il paradigma NFV propone quindi soluzioni interessanti, portando la comunità scientifica a interrogarsi sulle effettive prestazioni di questo approccio. Una delle principali ricerche che si stanno svolgendo in questi ultimi tempi è relativa alle prestazioni di questo tipo di middle-boxes: risulta infatti evidente che la domanda che la comunità scientifica si stia ponendo sia "può una rete virtualizzata avere performances simili a quelle delle reti fisiche o comporterà delle limitazioni?".

Numerosi articoli sono stati pubblicati in merito alle effettive prestazioni relative all'implementazione effettiva del paradigma NFV [12] ed altri verranno studiati nel capitolo 3.

Al fine di migliorare le prestazioni delle Virtual Machines quanto più possibile sono state effettuate delle ricerche che hanno portato a risultati interessanti: è stato ad esempio creato ClickOS [13], un Sistema Operativo ad-hoc che sfrutta Xen (e quindi implicitamente il router modulare Click), risolvendo numerosi problemi relativi alle generiche Virtual Machines ed aumentando le prestazioni del sistema stesso; un altro caso è invece quello di NetVM [14], che sfrutta il DPDK e il KVM per ottenere risultati migliori delle strutture basate su SR-IOV.

2.2 Software Defined Networking

Nonostante numerosi casi pratici di SDN siano stati proposti negli anni da aziende come Sun, AT&T ed Ericsson, la reale documentazione standard relativa a questo paradigma nasce nelle Università di Berkeley e Stanford, intorno all'anno 2008. Successivamente, nel 2011, viene fondata la Open Networking Foundation, al fine di promuovere il paradigma SDN [15] e OpenFlow, protocollo di comunicazione che verrà presentato nel capitolo 3.

Per dare un'idea anche soltanto economica del fenomeno SDN, secondo [16], il mercato del SDN è destinato ad aumentare esponenzialmente, fino ad una cifra di 35 Miliardi di Dollari nel 2018, come si può vedere nell'infografica in Figura 2.3.

Figura 2.3: Valore atteso del mercato SDN [16]

Nell'architettura proposta dal paradigma SDN il piano di trasporto è disaccoppiato dal piano di controllo dei dati, l'intelligenza della rete e lo stato della stessa sono centralizzate in un dispositivo esterno denominato controller e le strutture sottostanti al piano di trasporto vengono astratte dalle applicazioni e dai servizi di rete. Questa nuova libertà proposta permette agli operatori e alle imprese un guadagno in programmabilità, automazione e controllo della rete, permettendo loro di costruire reti flessibili che si adattano velocemente ai cambi richiesti dalle necessità del business.

Un approccio di questo tipo comporta dei benefici notevoli a operatori e imprese, come:

- **controllo** centralizzato dei dispositivi di rete, anche di vendor diversi che si sono uniformati allo standard;

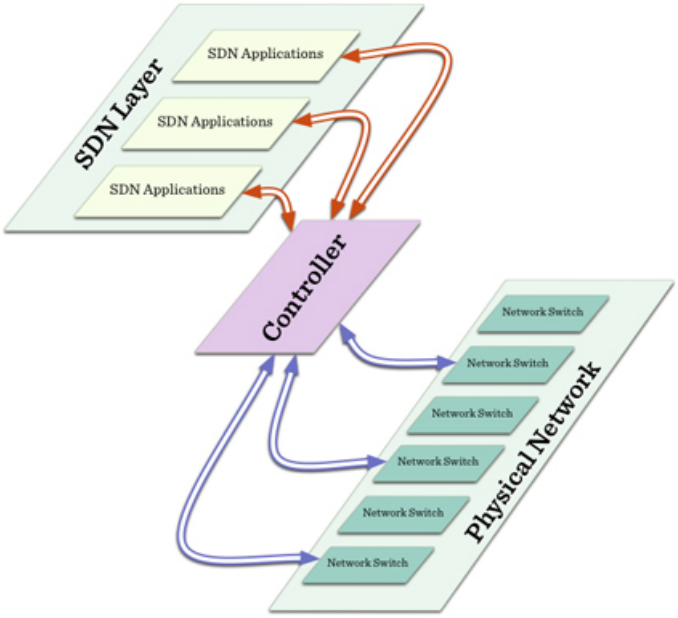

Figura 2.4: Architettura SDN [17]

- **automazione** e gestione delle reti migliorata sfruttando le APIs al fine di astrarre i dettagli degli strati sottostanti al piano di trasporto dalle applicazioni e dai servizi di rete;
- **implementazione** di nuovi servizi senza la necessità di configurare dispositivi singoli o attese di nuove releases da parte dei vendor;
- **programmabilità** semplificata per operatori, aziende, vendor indipendenti e utenti, semplicemente sfruttando ambienti di programmazione comuni, che riescono a dare ad ogni soggetto nuove opportunità, riportando un carattere di dinamicità nelle reti;
- **affidabilità**, grazie all'utilizzo di una gestione centralizzata e automatizzata di dispositivi di rete;
- **esperienza** lato utente migliorata, le applicazioni sfruttano le informazioni sullo stato della rete centralizzata per adattare al meglio il comportamento della rete ai bisogni degli utenti.

Risulta dunque evidente che SDN permette, grazie alla sua architettura dinamica e flessibile, di portare innovazione in un ambito statico come quello delle reti, dando la possibilità di trasformare le stesse in una piattaforma capace di adattarsi rapidamente alle necessità del business, degli utenti e del mercato.

Riconsiderando ciò che è stato affermato prima, viene da sè che la capacità di steering (indirizzare) del traffico dati attraverso le funzioni di rete virtuali sia considerato come un fattore chiave per poter raggiungere una struttura che implementa service chaining in modo del tutto dinamico.

La questione di un effettivo sviluppo di service chaining dinamico in edge networks cloud-based deve quindi essere affrontata da due lati. In prima istanza infatti la rete stessa deve essere programmata e configurata per indirizzare in modo corretto il traffico dati attraverso le funzioni richieste, considerando le risorse di comunicazione disponibili, il tutto dinamicamente. In seconda approssimazione, la consapevolezza del contesto sia del computing che delle risorse di comunicazione è di vitale importanza e va considerata la mutua dipendenza di funzioni di rete diverse e del loro chaining, sia nel tempo che nello spazio [18].

È proprio sotto questi aspetti che le numerose potenzialità del SDN, combinate a quelle del NFV, permettono ad operatori, aziende e utenti di rivoluzionare il mondo delle telecomunicazioni, portando ad infinite opportunità di crescita.

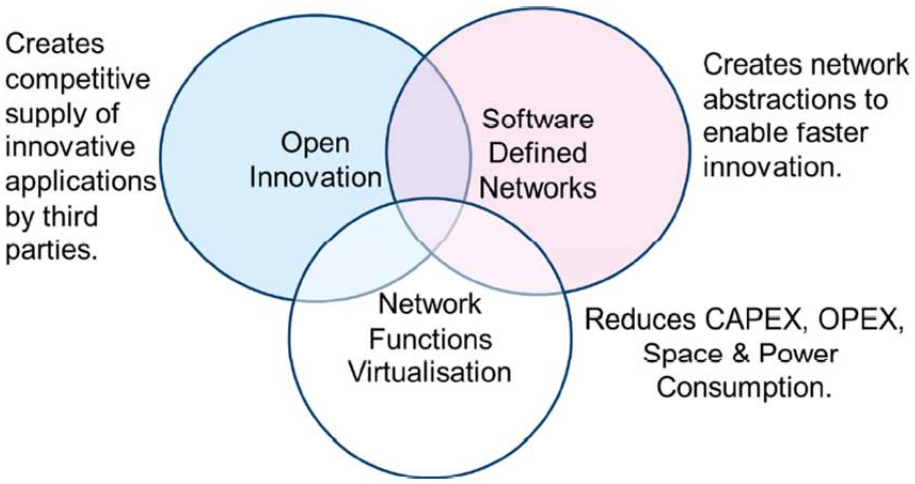

Figura 2.5: Vantaggi di SDN, NFV e Open Innovation [19]

Capitolo 3

SDN e NFV: implementazioni pratiche

Mentre nel Capitolo 2 sono stati introdotti i concetti di SDN e NFV, se ne vedranno qui due applicazioni pratiche di rilievo: il protocollo OpenFlow e il software di gestione di piattaforme cloud OpenStack.

Questi tool saranno essenziali per la sperimentazione pratica che verrà successivamente introdotta nel Capitolo 4 e poi sviluppata in quelli seguenti.

3.1 OpenFlow

Come considerato nel capitolo 2, il SDN disaccoppia il piano di trasporto dei dati da quello di controllo; questo permette dunque all'amministratore di rete di programmare direttamente il piano di trasporto e di astrarre l'infrastruttura sottostante da applicazioni e servizi di rete.

Per accedere al piano di trasporto di un dispositivo di rete, fisico o virtuale (hypervisor-based) attraverso la rete si sfrutta il protocollo di comunicazione *OpenFlow* [20], come è possibile vedere in Figura 3.1, la cui versione 1.0 è stata rilasciata nel 2009.

Questo standard viene gestito dalla Open Networking Foundation, organizzazione nata per la promozione e l'adozione di SDN; definisce OpenFlow come la prima interfaccia di comunicazione standard definita tra il piano di controllo e il piano di trasporto di un'architettura SDN.

Risulta quindi molto importante la presenza di un'interfaccia open al piano di trasporto; si ritiene infatti che sia stata l'assenza della suddetta a portare

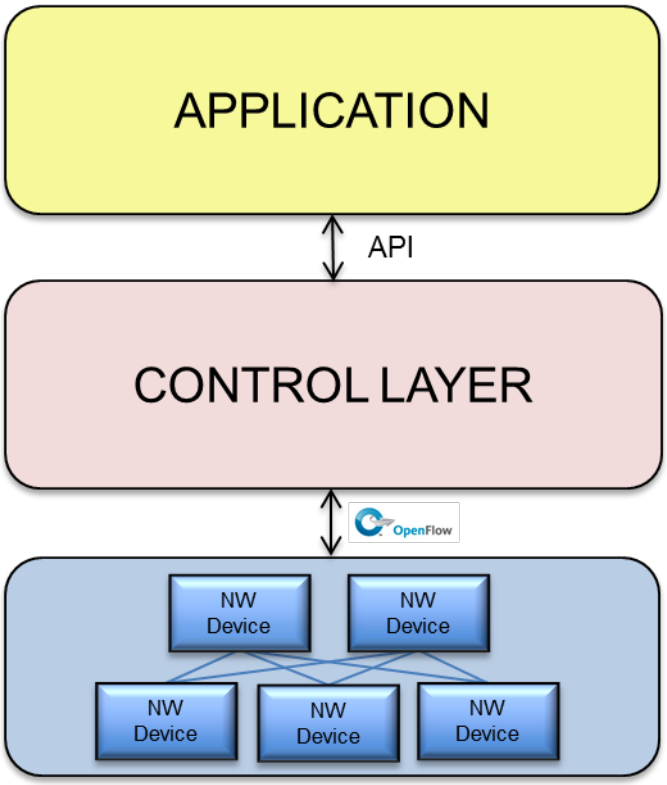

Figura 3.1: Stack di SDN [21]

alla staticità delle reti e dei dispositivi impiegati nelle stesse, con conseguenti fenomeni di "vendor lock-in" e altri precedentemente citati.

3.1.1 Funzionamento

Basandosi sulla specifica 1.0, uno switch OpenFlow può essere semplicemente visto come una *flow table*, che svolge la funzione di lookup dei pacchetti e il loro successivo inoltro e un *secure channel* ad un controller esterno, come mostrato in Figura 3.2 ; quest'ultimo gestisce lo switch attraverso il canale sfruttando il protocollo OpenFlow.

La flow table è un insieme di flow entries, contatori di attività e un insieme di azioni da applicare ai pacchetti che soddisfano una di queste regole. Se il pacchetto in arrivo soddisfa una entry, allora viene eseguita l'azione corri-

spondente (o più azioni, se previste); se invece non esiste una entry capace di soddisfare il pacchetto, allora questo viene inoltrato al controller attraverso il canale sicuro.

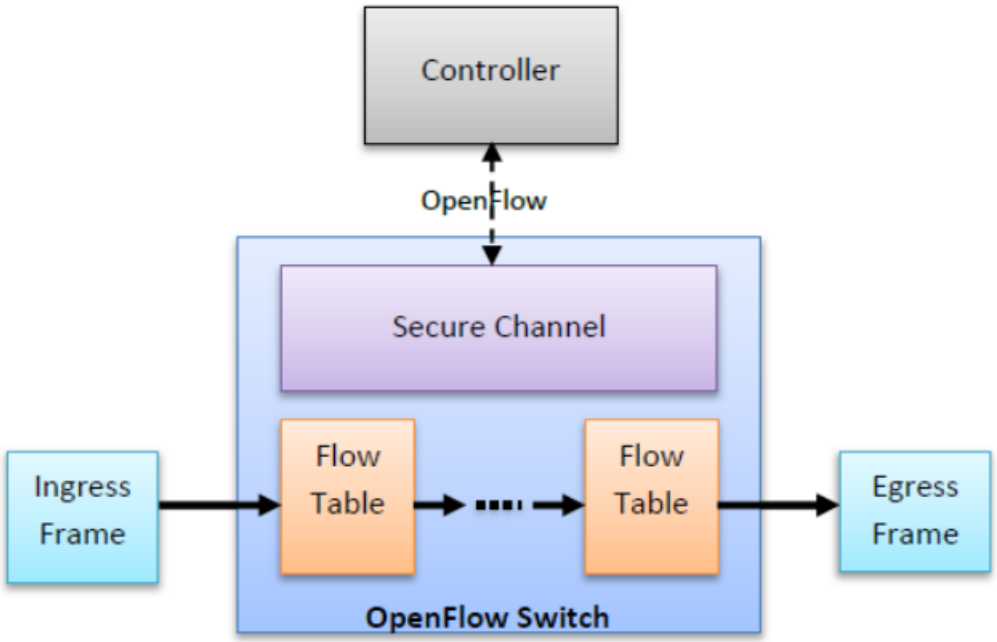

Figura 3.2: OpenFlow Switch e Controller [20]

Il compito principale del controller è dunque quello di gestire i pacchetti che non hanno flow entries valide, modificando la flow table dello switch aggiungendo e rimuovendo entries adeguate al caso in questione.

Una delle principali azioni svolte è quella di inoltro o forwarding a una o più porte OpenFlow; questo può essere fatto sia verso porte fisiche ma anche verso porte virtuali. In questo documento si farà sempre riferimento a queste ultime, dato l'utilizzo esclusivo che se ne è fatto nei successivi esperimenti; deve essere però chiaro al lettore che questo non preclude la possibilità di sfruttare il protocollo OpenFlow su dispositivi fisici che lo supportano e implementano, ad esempio inoltrando un pacchetto non più verso la porta virtuale, ma verso la porta fisica.

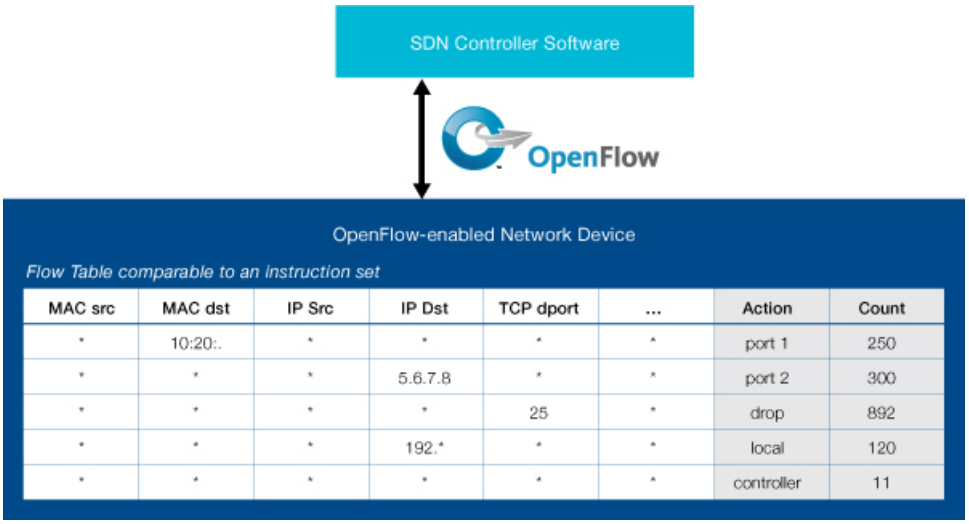

Figura 3.3: Esempio di Funzionamento [15]

Proprio relativamente alle porte virtuali OpenFlow esistono due azioni molto importanti che possono essere svolte da uno switch OpenFlow:

NORMAL: il pacchetto viene processato sfruttando il percorso tradizionale di forwarding supportato dallo switch (L2, L3, VLAN); in questo caso lo switch si comporta da Mac Learning Switch.

FLOOD: il pacchetto viene inviato a tutte le porte dello switch.

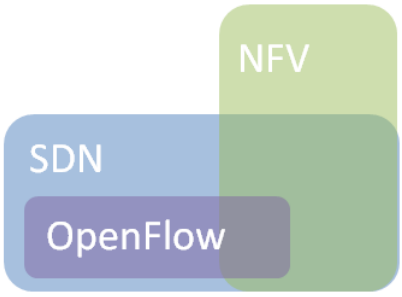

Figura 3.4: Interdipendenza tra SDN, NFV e OpenFlow [22]

OpenFlow offre dunque potenti funzionalità e tool interessanti e permette di implementare dal punto di vista pratico il concetto di SDN. Mediante

una programmazione mirata del controller con le relative regole OpenFlow, si può ottenere il funzionamento richiesto al verificarsi di determinati eventi, permettendo all'amministratore di rete di gestire il traffico della rete in modo dinamico.

3.1.2 Controller

Il controller è l'elemento intelligente della rete, riceve dallo switch i pacchetti che non hanno riscontri nella flow table e aggiunge o rimuove flow entries, in seguito alla ricezione; praticamente, il controller è un'entità programmabile e permette di gestire gli switch OpenFlow.

Negli anni sono stati creati numerosi controller, spesso profondamente diversi tra loro nel linguaggio e nella tipologia di programmazione, ma che portano, dal punto di vista pratico, a risultati simili: tra i più conosciuti ci sono POX (scritto in Python) e OpenDaylight (scritto in Java); entrambi sono finanziati da realtà del settore delle telecomunicazioni e hanno un grande seguito a livello di comunità di utenti che si scambiano informazioni, dubbi e problemi.

Altri controller meno noti ma altrettanto funzionali sono Ryu (Python), Floodlight (Java) e Beacon (Java); alcuni di questi, come OpenDaylight e Ryu, permettono anche l'integrazione automatica dell'ambiente OpenStack nel loro codice tramite plug-in forniti, mentre altri, come il POX, permettono comunque la gestione della suddetta piattaforma agendo però solamente sui bridge che compongono la struttura e che verranno successivamente descritti.

3.2 OpenStack

L'enorme aumento della domanda per applicazioni cloud-based ha comportato lo sviluppo di varie soluzioni per la gestione di infrastrutture cloud, tra cui alcune piattaforme software open-source che implementano il paradigma IaaS (Infrastructure-as-a-Service) in un contesto multi-tenant.

Una di queste piattaforme è OpenStack [23], nata da un'idea di Rackspace Cloud e NASA, successivamente coadiuvate da oltre 200 aziende del settore delle telecomunicazioni, dei sistemi operativi e dello sviluppo informatico.

OpenStack è quindi un software che permette di gestire piattaforme cloud; una piattaforma cloud è un cluster di macchine fisiche che ospitano alcuni server (denominati istances): questi server sono offerti all'utente come un servizio (service). L'utente sarà quindi capace di creare una infrastruttura vir-

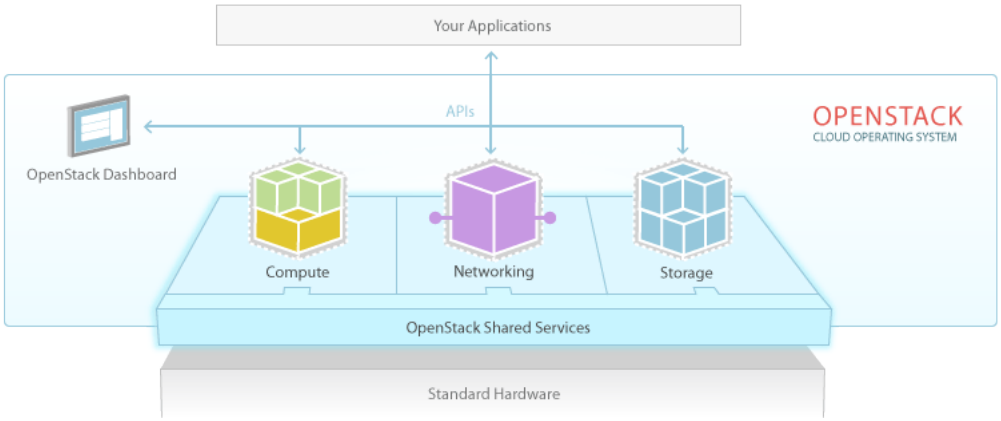

Figura 3.5: Struttura logica di OpenStack [23]

tuale composta da server e applicazioni di rete, come router, firewalls, WAN accelerator, eccetera.

Questi server possono essere implementati come Virtual Machines (attraverso KVM, VMWare, ecc), light container o bare metal.

OpenStack fornisce dei cloud managers mediante una potente e flessibile dashboard, visibile in figura 3.6, che permette di controllare un cluster di hosting servers eseguendo differenti tipi di Hypervisors e di gestire lo storage e le infrastrutture virtuali di rete. La dashboard permette inoltre ai client di istanziare velocemente e in modo trasparente risorse di networking e computing mediante una infrastruttura virtuale di data center.

Parallelamente è possibile utilizzare anche la CLI (Command Line Interface), che permette una maggiore libertà di sperimentazione e la possibilità di gestire ogni singolo dettaglio dell'implementazione che verrà creata.

Per avere un'effettiva stima dell'interesse crescente manifestato nei confronti di OpenStack nel corso degli anni è stato sfruttato Google Trends [24]; il risultato è mostrato in Figura 3.7.

3.2.1 Componenti

OpenStack, facendo riferimento alle versioni a partire da Havana, è costituito da numerosi componenti, visibili anche in Figura 3.8:

Horizon: Web dashboard;

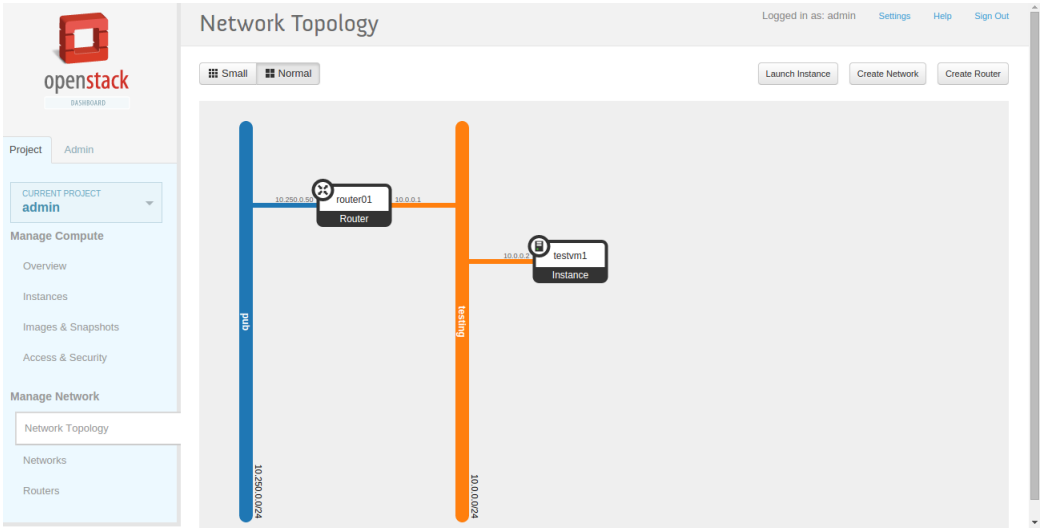

Figura 3.6: Web Dashboard di OpenStack

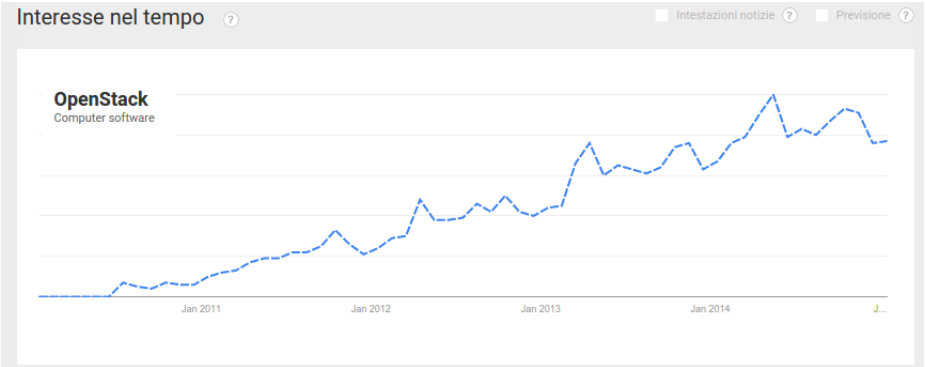

Figura 3.7: Trend di ricerche relative a OpenStack su Google

Nova: si occupa del computing, gestisce il ciclo di vita delle istanze;

Neutron: gestisce il networking;

Keystone: si occupa delle credenziali e della gestione di tutti i servizi OpenStack (contiene infatti la lista dei REST service endpoints che dopo verranno definiti);

Glance: permette di gestire le immagini contenenti una Virtual Machine "pre-confenzionata", viene infatti sfruttato dagli hypervisors per fare il boot di nuove istanze;

Cinder: si occupa dello storage;

Swift: storage distribuito per dati non strutturati.

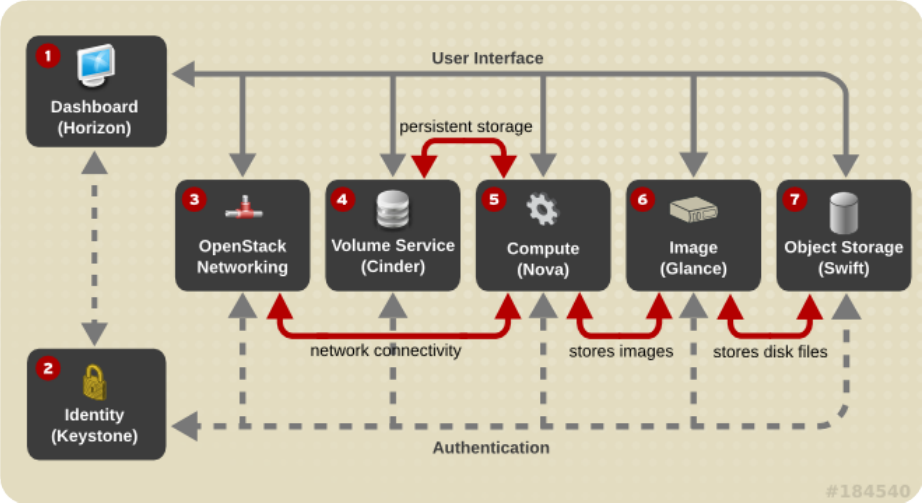

Figura 3.8: Componenti di OpenStack (da Havana in su) [23]

È importante sapere inoltre che nelle versioni precedenti ad Havana il networking era svolto da un sotto-componente di Nova, chiamato Nova-network. Questo era però limitante, in quanto la gestione della rete era disponibile solo all'amministratore del cloud e l'implementazione era troppo accoppiata con le astrazioni di rete; andando a visualizzare le regole iptables delle Virtual Machines infatti si troveranno ancora delle entries relative a nova-network.

3.2.2 Struttura e gestione multi-tenancy

Un cluster OpenStack è solitamente composto da:

- un *controller node*, che gestisce la piattaforma cloud;
- un *network node*, che ospita i servizi di rete cloud;

- un numero di *compute nodes*, che eseguono le virtual machines;
- un numero di *storage nodes*, per le immagini delle virtual machines e i dati.

Questi nodi sono connessi tra di loro mediante due reti, una di Management e una di Data; il controller e il network node inoltre sono connessi alla rete esterna. La rete dati è usata per la comunicazione tra Virtual Machines, mentre la rete di management permette all'amministratore di accedere ai nodi del cluster ed è sfruttata per le comunicazioni di servizio; infine la rete esterna permette alle Virtual Machines di accedere a Internet, così come agli utenti di accedere alle stesse.

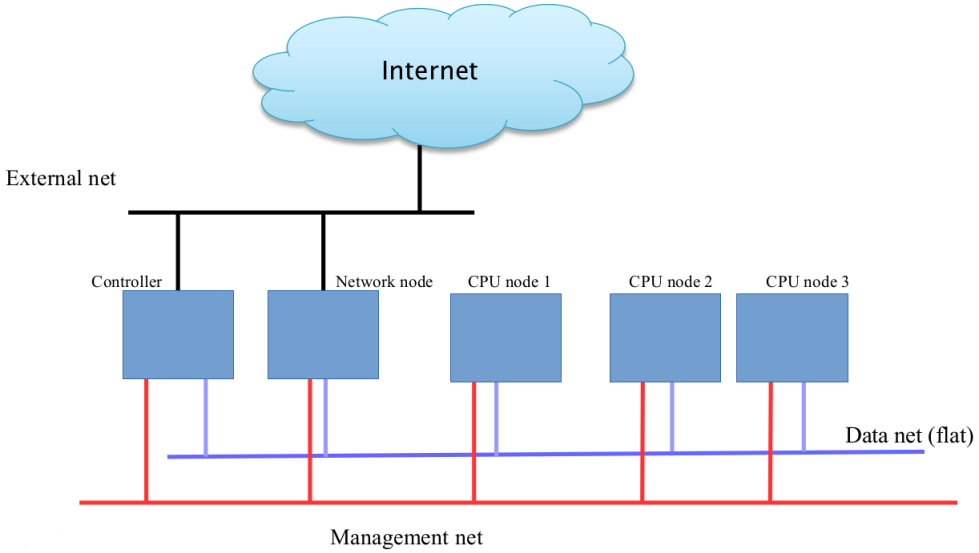

Figura 3.9: Topologia del cluster OpenStack [25]

OpenStack come già affermato permette il multi-tenancy (e quindi la possibilità di avere più utenti nello stesso cluster); questi tenant vanno naturalmente isolati tra di loro e questo è permesso sfruttando l'utilizzo di VLAN (o di VXLAN o di GRE) e namespaces, mentre i security groups proteggono le Virtual Machines da attacchi esterni o accessi non autorizzati.

I network namespaces permettono infatti di separare e isolare più domini di rete all'interno di un singolo host semplicemente replicando lo stack software della rete. Un processo eseguito all'interno di un namespace vede solo specifiche

interfacce, tabelle e regole. I namespaces garantiscono l'isolamento L3: in questo modo le interfacce relative a tenant diversi possono avere indirizzi IP sovrapposti.

Un security group è un insieme di regole che implementano un firewall configurabile dall'utente: durante la creazione delle Virtual Machines si possono associare a queste ultime determinati security groups; questi vengono implementati mediante regole iptables sul compute node.

Un Server Neutron centralizzato immagazzina tutte le informazioni relative alla rete, mentre gli agenti di Neutron vengono eseguiti nel network node e in ogni compute node che implementa l'infrastruttura virtuale di rete in modo coordinato; questo permette dunque a Neutron di gestire reti multi-tenant anche con numerosi compute nodes.

L'infrastruttura di rete virtuale implementata da OpenStack è composta da numerosi bridge virtuali che connettono interfacce virtuali o fisiche e che possono anche risiedere in namespaces differenti.

Sfruttando il tool grafico Show My Network State [26] si possono comprendere bene la struttura e gli elementi di rete usati da OpenStack; in Figura 3.10 e Figura 3.11 vengono mostrati rispettivamente un compute node e un network node che, in questo caso, eseguono tre Virtual Machines, poste su due subnets.

Come si può vedere ogni nodo contiene al suo interno un bridge del tipo OpenVSwitch [27] chiamato *br-int* e un bridge OVS aggiuntivo ad esso connesso per ogni network data center fisico connesso al nodo.

Inoltre il network node include un bridge chiamato *br-data* per la rete dati, dove le reti multi-tenant virtuali vengono fatte funzionare e *br-ex* per la rete esterna, come quella che connette il data center a Internet. Un compute node invece include, oltre a br-int, solo *br-data*, considerando che tipicamente è solo connesso alla rete dati.

All'interno di questi bridge andranno implementate le regole OpenFlow che permettono ai pacchetti di trovarsi modificato il proprio VLAN ID, in merito all'isolamento necessario a mantenere il Multi-tenancy; per esempio, in un compute node il br-data fa la conversione VLAN ID da interno ad esterno per i pacchetti che devono uscire dal nodo, mentre il br-int fa la conversione inversa per i pacchetti diretti alle Virtual Machines.

Le interconnessioni tra virtual bridge sono implementate grazie a una coppia di interfacce virtuali Ethernet, i veth pairs; ognuna di queste coppie consiste

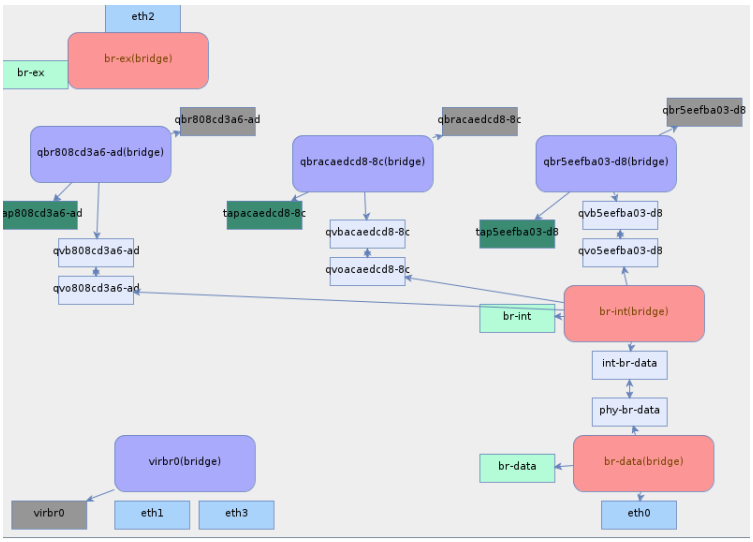

Figura 3.10: Esempio di struttura di un Compute Node

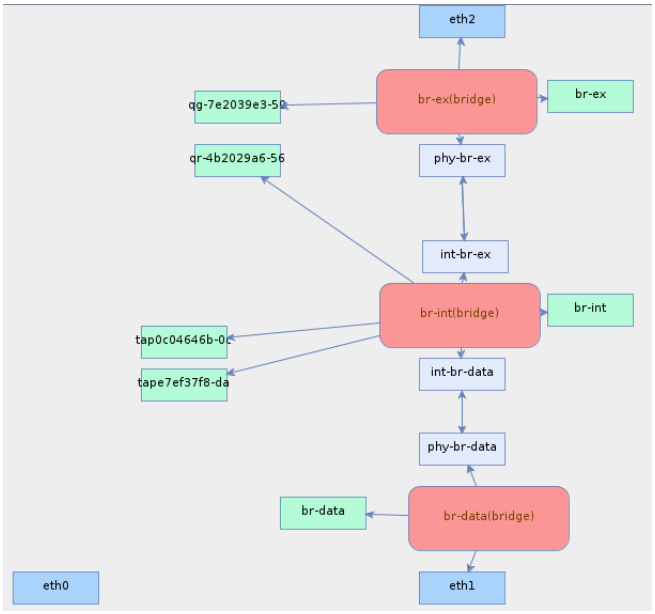

Figura 3.11: Esempio di struttura di un Network Node

in due interfacce che funzionano da endpoints di una pipe: tutto ciò che entra in una deve uscire dall'altra e viceversa.

Quando un utente crea una nuova istanza di Virtual Machine il componente di OpenStack che permette la pianificazione e che viene eseguito nel controller node, sceglie il miglior compute node che ospiterà la Virtual Machine.

Successivamente, per ogni sottorete e rete a cui la Virtual Machine sarà connessa, un insieme di elementi virtuali di rete vengono creati: un *Linux Bridge* [28] (e la sua interfaccia di management), un veth pair che connette questo Linux Bridge al bridge br-int e un'interfaccia di *tap* connessa al Linux Bridge. L'interfaccia di tap è un'entità software che permette di connettere il bridge livello-kernel ad una porta Ethernet della Virtual Machine che viene eseguita nello user-space. In teoria infatti, l'interfaccia di tap potrebbe essere connessa direttamente al bridge br-int, ma i security groups vengono implementati da Neutron applicando le funzioni native di filtering del kernel (netfilter) sulle interfacce di tap e questo funziona quindi solo con i Linux Bridge.

Proprio per questo motivo infatti è necessario un elemento intermedio per interconnettere l'interfaccia di tap al bridge br-int. Le regole iptables (relative al namespace globale) sono dunque implementate sulla porta del Linux Bridge.

3.2.3 Astrazioni di Neutron

Data l'impossibilità di continuare a lavorare con Nova sul networking, la creazione di *n*eutron è stata praticamente obbligata ed ha permesso di disaccoppiare le astrazioni di rete dalle attuali implementazioni e permette sia agli utenti che agli amministratori di gestire la rete virtuale mediante un'interfaccia flessibile.

Le astrazioni definite da Neutron sono numerose:

- una rete viene vista come un segmento virtuale di livello L2;
- una sottorete viene vista come un insieme di indirizzi IP usato in una network di livello L3;
- una porta è vista come un punto di connessione ad una rete e ad una o più sottoreti su quella stessa rete;
- un router è visto come una applicazione virtuale che compie meccanismi di routing mediante subnets e traduzioni di indirizzi;
- un server DHCP è visto come una applicazione virtuale in grado di distribuire indirizzi IP;

- un security group è visto come un set di regole usate da filtro che implementano un firewall al livello della piattaforma cloud.

3.2.4 Funzionamento

Un utente OpenStack, che rappresenta un dato tenant, può creare una nuova rete e definire una o più sottoreti sopra di essa, sfruttando, nel caso, i relativi server DHCP. Successivamente l'utente può far partire una nuova istanza di Virtual Machine, specificando la sottorete o le sottoreti alle quali deve connettersi: una porta su quella sottorete e sulla relativa rete viene creata, la Virtual Machine viene connessa a quella porta e un indirizzo IP viene assegnato ad essa mediante DHCP.

Altre applicazioni virtuali, come il virtual router, possono essere implementate direttamente nella piattaforma cloud sfruttando meccanismi come i containers e i namespaces, oppure realizzati dall'utente nelle Virtual Machines create, adibendole alle applicazioni che più gli sono comode. Questo può essere fatto anche con altri tipi di middle-boxes, come il Firewall, il NAT, il WAN Accelerator, ecc.

La comunicazione in OpenStack avviene attraverso REST API: l'utente invia le REST API calls ai service endpoints (usando la web dashboard o i clients da command line); i componenti di OpenStack comunicano tra di loro usando sempre le REST API calls o il message passing.

Un esempio semplice di quanto sopra può essere visto con il procedimento di creazione di una Virtual Machine: infatti, un utente che desidera creare una Virtual Machine deve inviare una REST API (da CLI client o dashboard) al Keystone REST endpoint e al Nova endpoint; "nova-scheduler" internamente sceglie allora il miglior compute node disponibile a ospitare la Virtual Machine.

In questo compute node, il componente "nova-compute" svolge le seguenti azioni: preparazione dell'hypervisor, richiesta a Glance dell'immagine della Virtual Machine, richiesta ai componenti di Neutron di allocare l'infrastruttura virtuale di rete, richiesta a Cinder di allocare lo storage della Virtual Machine.

3.2.5 Prestazioni di una rete OpenStack

Tenuta presente dunque la struttura di rete OpenStack, sono stati svolti alcuni esperimenti [29] al fine di valutare le relative prestazioni di un'architettura di

rete di questo tipo, cercandone i colli di bottiglia e valutando possibili soluzioni per migliorare il tutto.

Si è valutato che il vero collo di bottiglia è la presenza del Linux Bridge, che ha portato ad un significante calo delle prestazioni rispetto al caso ideale e rispetto a casi simili. Il Linux Bridge dunque, che si ricorda essere nato per implementare i security groups tramite neutron, comporta un problema in termini di prestazioni.

In modo simile sono stati svolti altri numerosi esperimenti volti a misurare le prestazioni nel caso multi-tenant, sfruttando una semplice architettura di rete ispirata al NFV [30].

Ciò che è stato scoperto è che l'architettura cloud pone alcune limitazioni alle performance delle reti, non solo a causa delle prestazioni dell'hardware che ospita il cloud, ma anche a causa dell'architettura implementata dalla piattaforma stessa OpenStack.

Da questi esperimenti ne consegue che una architettura di rete implementata da OpenStack non è in realtà ancora ottimale dal punto di vista prestazionale; questo è in parte comprensibile dato il continuo sviluppo che c'è dietro al software. Inoltre, considerando la cadenza circa semestrale di uscite di nuove releases, non è da escludere che nelle prossime versioni vengano sfruttati meccanismi diversi per risolvere questi problemi e permettere alla rete cloud creata con OpenStack di essere prestazionalmente valida.

Consolidati dunque i vari concetti di cui sopra, sarà di grande l'utilità la Figura 3.12, che riassume tutti i concetti teorici affrontati finora in questo documento in modo chiaro ed intuitivo.

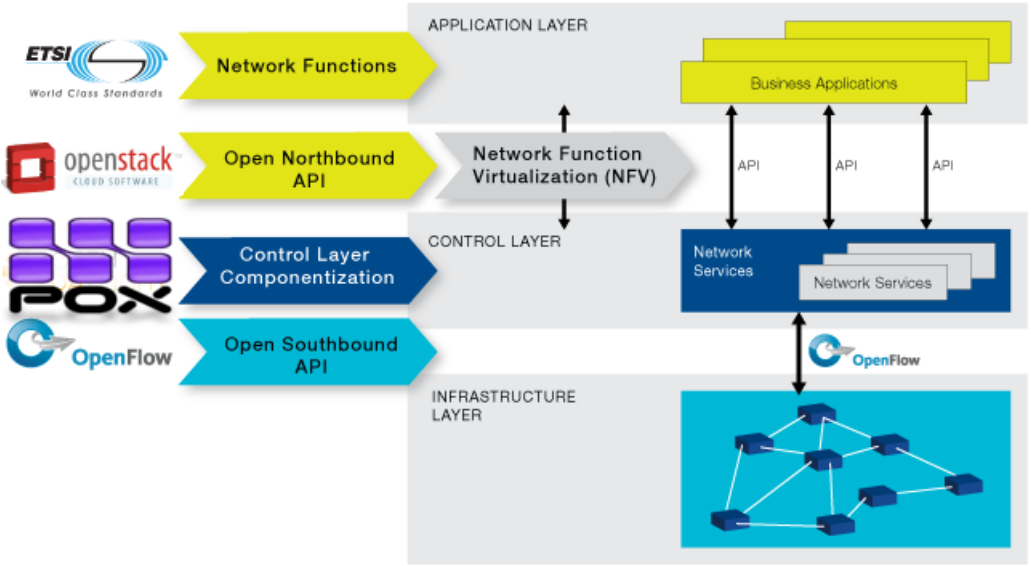

Figura 3.12: SDN e NFV implementati mediante OpenFlow, POX, OpenStack

Capitolo 4

Service Chaining

Nel Capitolo 3 sono stati introdotti OpenFlow e OpenStack come meccanismi che mettono in pratica i paradigmi SDN e NFV rispettivamente. In questo capitolo verrà invece introdotto il concetto del Service Chaining e al contempo spiegato come questo possa essere realizzato nel particolare caso di studio, sia da un punto di vista prevalentemente teorico che da un punto di vista funzionale, applicando già i primi concetti pratici, che verranno ripresi e poi approfonditi nel dettaglio nei capitoli 5 e 6.

Faranno inoltre da cornice al capitolo l'Appendice A e l'Appendice B contenenti rispettivamente l'implementazione del nodo ricevente e cioè dell'edge network di destinazione e il codice Python necessario a programmare il controller POX per gestire il traffic steering in modo corretto ed efficiente.

4.1 Service Chaining Dinamico in Edge Networks

Come asserito nel Capitolo 1, in futuro si presume che nelle infrastrutture di telecomunicazioni tutte le risorse e i servizi della rete vengano allocati quanto più possibili vicino all'utente, mentre le core networks siano solo un insieme di collegamenti ad alta velocità che permettono alle edge networks di comunicare tra di loro. Inoltre, sapendo che le funzioni di rete relative alle edge networks non sono implementate da middle-boxes hardware vendor-dependant, ma sotto forma di applicazioni software comprensive di isolamento multi-tenancy, eseguite sopra hardware general-purpose, si pensa che le future edge networks

andranno a prendere la forma di data centers dove le funzioni di rete virtuali possano essere facilmente sviluppate, copiate, migrate e distrutte, come una sorta di servizio on-demand in accordo con il paradigma del cloud computing.

Queste edge networks cloud hanno però la peculiarità di offrire qualcosa di diverso dai tipici servizi implementati nei data centers; infatti, uno spazio di storage condiviso o un'applicazione web multi-livello rappresentano sorgenti o pozzi di data flows dell'utente, mentre molte funzioni di rete virtuali devono essere applicate al traffico incrociato che si origina e termina fuori dal data center. Al fine di adattarsi alle condizioni di rete e data la flessibilità degli applicativi software, il tipo, il numero e la locazione delle funzioni di rete virtuali attraversate da un data flow di un certo utente può cambiare nel tempo.

In prima istanza, la scelta delle più appropriate funzioni di rete virtuali può essere il risultato dell'esecuzione di altre funzioni di rete virtuali; per esempio, la funzione di classificazione del traffico può essere usata per determinare quale elaborazione del traffico dovrà essere applicata ad un certo data flow, in un certo momento, al fine di migliorare la QoS.

Consequenzialmente, l'infrastruttura a data center dell'edge network deve essere abbastanza flessibile da permettere un chaining dinamico e condizionato, sia nel tempo che nello spazio. Un approccio per abilitare la flessibilità in entrambe le dimensioni è quello di usare OpenFlow per direzionare i flussi di traffico.

Servirà dunque un controller SDN centralizzato al fine di programmare l'edge network e far sì che ogni flusso di traffico dell'utente attraversi le corrette funzioni di rete virtuali nell'ordine richiesto. Il controller deve essere capace di coordinare dunque il comportamento degli elementi che compongono le edge networks anche quando questi sono molti, al fine di fornire data flows mediante un corretto service chaining edge-to-edge.

Infine è necessario che ogni modifica e cambiamento compiuti da chain di funzioni di rete virtuale debbano essere applicati in modo trasparente all'end user, il cui requisito di QoS deve essere soddisfatto; quindi, ogni implementazione specifica del paradigma NFV nel data center dell'egde network del provider deve essere in linea con i protocolli di accesso standard per l'utente.

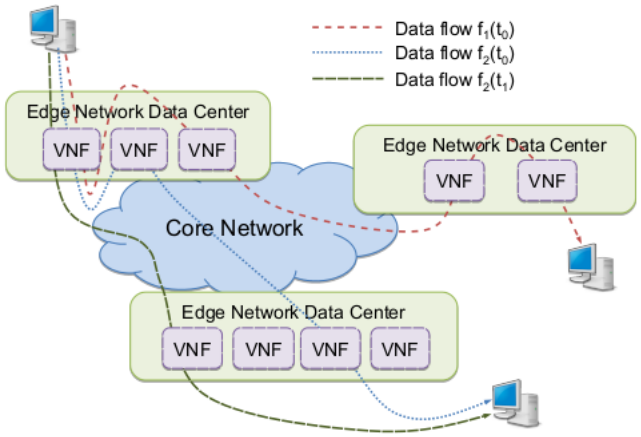

Figura 4.1: NFV Chaining nelle edge networks, con traffic steering dinamico [31]

4.2 Caso di studio: Descrizione pratica

Riprendendo i concetti sopra esposti, il caso di studio è quindi quello di un operatore di rete che deve fornire la connessione a due utenti che scambiano traffico con un utente remoto attraverso un router edge. I due end-points utente sono implementati come Virtual Machines (VMU1 e VMU2) che eseguono applicazioni all'interno di una piattaforma di cloud computing direttamente collegata al data center; quest'approccio segue l'idea di un'infrastruttura di data center condivisa che integra i concetti di cloud e NFV services per client, in modalità multi-tenant.

Si ipotizza che gli utenti abbiano sottoscritto due tipi diversi di contratti, che portano a due diversi tipi di Service Level Agreement: VMU1 è un utente prioritario e avrà bisogno di un servizio virtuale di WAN Acceleration quando necessario; VMU2 è invece un utente di tipo best effort, quindi il suo traffico inizialmente seguirà il percorso tradizionale, ma dopo la classificazione il suo traffico deve attraversare un Traffic Shaper, con l'effettivo risultato di una limitazione di banda utilizzata.

Al fine di valutare a quale utente appartenga un certo data flow, si sfrutta un analizzatore di traffico basato sulla Deep Packet Inspection (DPI), implementato mediante la libreria software nDPI [32]; le altre funzioni virtuali di rete sfruttate sono implementate sfruttando un Wide Area Network Accele-

rator (WANA) implementato tramite Trafficsqueezer [33] e un Traffic Shaper implementato sfruttando il Traffic Controll (TC) del kernel Linux. DPI, WANA e TC sono naturalmente funzioni di rete offerte dal provider di servizi internet come Virtual Machines dedicate.

Inoltre il router che inoltra i pacchetti all'host remoto (H1) è un virtual router (VR), in accordo con il paradigma NFV.

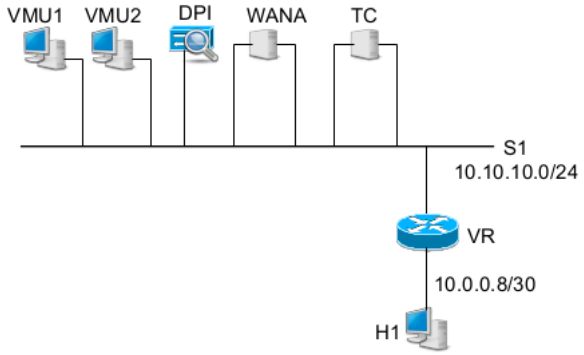

Figura 4.2: Topologia L2 [31]

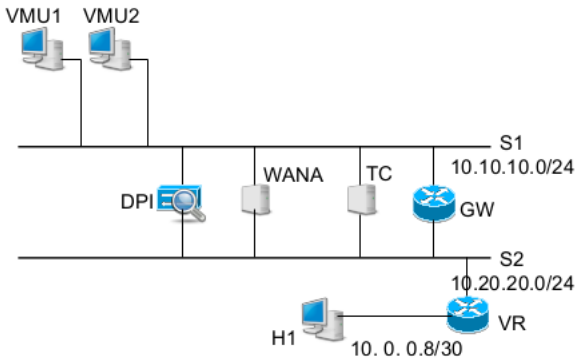

Figura 4.3: Topologia L3 [31]

I meccanismi di traffic steering vengono implementati in modo dinamico, sfruttando l'approccio SDN e programmando dunque lo strato di trasporto. In questo documento si farà riferimento a due topologie, una di livello L2 (Figura 4.2) e una di livello L3 (Figura 4.3).

Mentre le topologie relative a OpenStack sono state realizzate come visualizzato in figura, il nodo ricevente H1 ha richiesto un'implementazione più sofisticata che può essere vista nell'Appendice A e che cambia a seconda del livello implementato.

4.3 Caso di studio: Descrizione funzionale delle Chains

Il service chaining è dunque la concatenazione di un insieme di funzioni che elaborano il traffico e lo indirizzano verso determinati dispositivi.

Nel caso di studio affrontato in questo documento, si vuole testare il Service Chaining della rete in questione; si vuole cioè studiare come, in base agli eventi e ai flussi che vengono generati nella rete, il traffico dei due utenti venga direzionato attraversando funzioni di rete virtuali che implementeranno diversi tipi di servizi; un esempio di approccio al service chaining può essere quello di classificare il traffico dell'utente e successivamente indirizzarlo ad altre Virtual Machines in cui le funzioni di rete implementate sono relative al SLA dell'utente stesso.

Si sfrutterà un approccio per cui la rete stessa conosce già il Service Level Agreement dell'utente, utilizzerà un dispositivo che funziona da sniffer del traffico e deciderà il trattamento da applicare ai flussi di traffico.

Questo esperimento viene realizzato su due livelli, ossia il livello L2 e il livello L3: nel caso L2 la funzione di rete sarà implementata come un bridge e nel campo destinatario dell'intestazione della trama Ethernet ci sarà l'indirizzo di destinazione finale; nel caso L3 nel suddetto campo vi sarà invece l'indirizzo relativo alla funzione. Il pacchetto IP invece non viene modificato, in quanto nella sua intestazione avrà sempre l'indirizzo del destinatario finale.

La realizzazione avviene su due livelli perchè si ipotizza che la funzione di rete potrebbe essere già fornita (come peraltro avviene per queste sperimentazioni) e la rete quindi debba saper gestire entrambi i casi.

Inoltre, è possibile nelle Figure 4.2 e 4.3 farsi un'idea di come possano essere costruite le due topologie di rete per i livelli L2 ed L3; nonostante le differenze implementative tra le due, la politica di steering rimane comune a entrambe così come le funzioni di rete virtuali che verranno attraversate dai due data flows.

4.3.1 Livello L2

Relativamente al livello L2 si suppone inizialmente di avere solo traffico proveniente da una delle due Virtual Machine, ad esempio il BusUser. Facendo quindi riferimento alle Figure 4.4 e 4.5, nella prima fase, denominata Fase di Mirroring, i pacchetti da essa generati vengono inviati sia al DPI (che dopo un certo intervallo di tempo identificherà il traffico, scoprendo essere proveniente dalla BusUser) sia all'esterno della edge network 1, verso il nodo H1 (e quindi l'edge network 2), sfruttando il Virtual Router VR.

Nella seconda fase invece, essendo stato il traffico già categorizzato e non essendoci altre Virtual Machines a contendere la banda, i pacchetti provenienti dal BusUser continueranno ad andare verso H1 passando da VR, senza implementazione di servizi aggiuntivi. Un'altra ipotesi possibile, come poi è stato testato successivamente per motivi di semplificazione, è quella di inviare, dopo la classificazione, il traffico della BusUser direttamente al WANA.

È nella terza fase che entra in gioco la Virtual Machine ResUser che svolgerà la sua fase di Mirroring, contendendo la banda al BusUser per il tempo necessario alla categorizzazione del suo traffico da parte del DPI.

Nella fase finale infine, vengono applicati i servizi relativi al SLA dell'utente: il traffico del BusUser verrà quindi indirizzato all'interno del WANA, mentre quello del ResUser verrà mandato verso il TC; entrambi usciranno poi dalla rete per andare verso il nodo ricevente.

Come già fatto presente, l'implementazione del nodo ricevente in questo caso è stata schematizzata ad un singolo host; in realtà come si può vedere nell'Appendice A tutto ciò ha richiesto più di un singolo host; infatti un dispositivo come il WANA, sfruttando internamente il software Trafficsqueezer, ha bisogno anche di un dispositivo complementare alla destinazione in grado di "decomprimere" i pacchetti; anche in questo nodo infatti sono state implementate le regole di steering adeguate al fine di consentire al traffico relativo alla BusUser di essere "decompresso" attraverso un WANA Decompressor e viceversa, permettendo il corretto funzionamento dell'esperimento.

4.3.2 Livello L3

Nella topologia L3, illustrata in Figura 4.3, si ragiona sempre in termini di fasi: parallelamente a quanto succede nella topologia di rete L2, in quella L3 il traffico si suppone inizialmente solo generato dalla BusUser. Questo, come è possibile vedere in figura 4.6 sarà indirizzato sempre verso il DPI, che

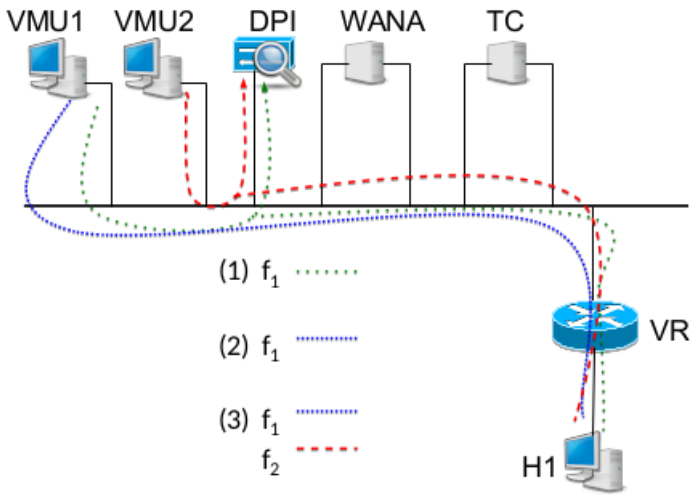

Figura 4.4: Traffic Steering L2: Fasi di transizione [31]

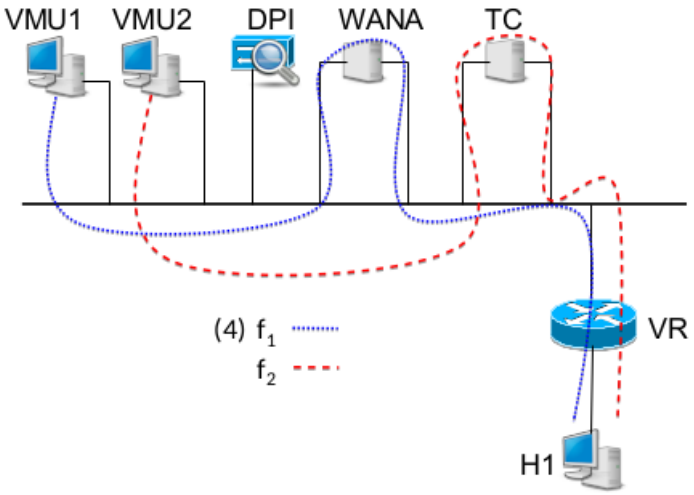

Figura 4.5: Traffic Steering L2: Fase finale [31]

provvederà alla classificazione del traffico che, una volta uscito dalla seconda interfaccia, andrà verso il VR al fine di uscire dall'edge network e, idealmente, attraversare la core network per raggiungere l'edge network di destinazione.

Una volta trascorso il tempo necessario si entrerà nella seconda fase dove il

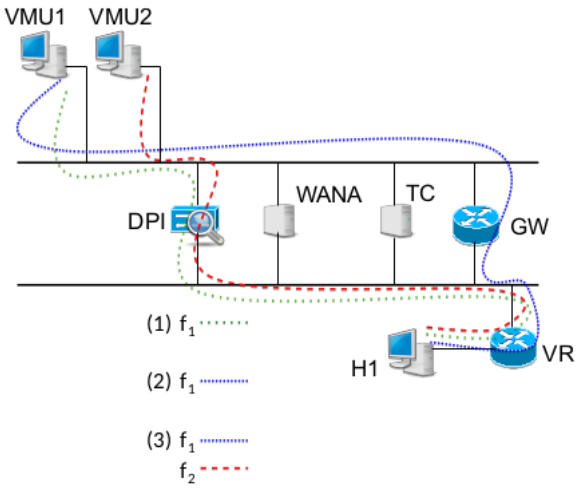

Figura 4.6: Traffic Steering L3: Fasi di transizione [31]

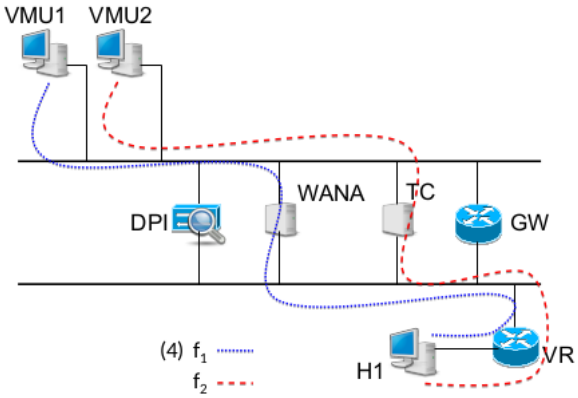

Figura 4.7: Traffic Steering L3: Fase finale [31]

traffico verrà reindirizzato verso il gateway GW per poi terminare nell'host di destinazione, sfruttando tutta la banda disponibile; questo non verrebbe naturalmente più rispettato nel caso in cui una seconda Virtual Machine dovesse entrare in gioco; è infatti in questa fase, la terza, che la Virtual Machine ResUser, di tipo best effort, inizia a inviare traffico, il quale naturalmente passerà sempre dal DPI al fine di una classificazione.

In questa fase la banda è contesa tra le due Virtual Machine; ma alla fine

dell'intervallo di tempo necessario al DPI per la classificazione, in virtù del diverso tipo di contratto stipulato dai due utenti, verranno dunque implementate le manovre necessarie per concedere al BusUser quanta più banda disponibile, come è possibile vedere in Figura 4.7. Questo viene effettuato mediante l'indirizzamento del suo traffico attraverso il WANA, mentre quello del ResUser verrà inviato al TC.

In questo modo si avrà dunque una diminuizione della banda per il ResUser, mentre il BusUser otterrà anche un'ottimizzazione del traffico grazie al WAN Optimizer.

4.4 Dettagli realizzativi

Il lavoro di seguito riportato è stato implementato nel cluster OpenStack di Cesena, sfruttando un server con Ubuntu 14.04 e OpenStack Havana.

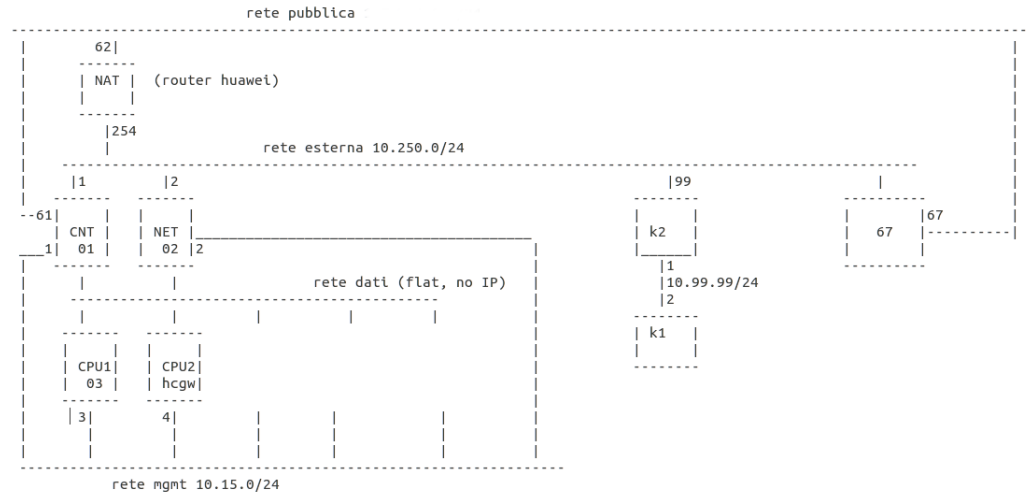

Figura 4.8: Cluster OpenStack in uso

Il fine di questo lavoro è riuscire a ricreare in ambiente OpenStack la sperimentazione implementata dapprima su Mininet [31], portando dunque l'emulazione delle service chain implementate ad un caso reale, testandone il corretto funzionamento ed, eventualmente, le prestazioni.

Il controller utilizzato per gestire i bridge OVS è stato il POX; nonostante esistano controller più adeguati al coordinamento con OpenStack, si è deciso

di sfruttare il POX per mantenere una dualità col caso di Mininet [34], dove il controller utilizzato era sempre lo stesso, in modo da poter evidenziare anche le differenze tra i codici nei due casi.

Il nodo ricevente è stato implementato esternamente ad OpenStack in un altro host fisico tenendo conto di svariate necessità legate allo stesso; per completezza di informazione questo è stato trattato nell'Appendice A.

Capitolo 5

Caso di studio: Livello L2

Come descritto nel capitolo 4, il livello L2 è uno dei due sui quali verte il caso di studio e in questo capitolo ne verrà data un'idea di come è stato realizzato fisicamente su OpenStack e quali altre tematiche sono state affrontate al fine di renderlo funzionante.

Un'importante premessa da fare è quella per cui deve essere chiaro al lettore, dopo le precisazioni fatte nel Capitolo 3, che la topologia fisica che verrà implementata nel cluster di OpenStack non rispecchia in tutto e per tutto la topologia virtuale che si va a realizzare: questo avviene infatti a causa delle astrazioni di rete effettuate dal componente neutron.

Al fine di semplificare tutto ciò, sono state eliminate alcune fonti di complessità, come è possibile vedere nei seguenti paragrafi: una di queste è l'utilizzo di un singolo compute node sul quale installare le Virtual Machines, permettendo dunque di avere una topologia fisica già più simile a quella virtuale.

5.1 Realizzazione della Topologia OpenStack

Come è possibile vedere in Figura 4.2, la topologia di rete del livello L2 è così composta:

- due Virtual Machines, rispettivamente utente Business e utente Residenziale (d'ora in poi BusUser e ResUser) con una interfaccia sulla rete net1;
- un Deep Packet Inspector (d'ora in avanti DPI) anch'esso con un'interfaccia sulla rete net1;

- un Wan Accelerator (WANA) e un Traffic Shaper (TC), con due interfacce a testa sulla stessa rete;
- un Virtual Router per uscire sulla rete pubblica pub.

La realizzazione di questa topologia è stata effettuata mediante CLI di OpenStack; al fine di lavorare sul cluster, la connessione utilizzata è mediante Secure Shell (SSH) ad uno dei due indirizzi del compute node sulla rete pubblica di ateneo, i cui valori sono stati parzialmente oscurati per motivi precauzionali.

```
ssh -X stack@x.y.z.63 -p 22001
```

Si noti la presenza del *-X*, che indica l'abilitazione del parametro del X forwarding, necessario per il tunneling grafico (ad esempio l'apertura del virt-manager o l'utilizzo di Show My Network State).

Il primo passo effettuato è stato quello della creazione di una rete interna net1 e della relativa sottorete subnet1_net1, sempre tenendo conto delle astrazioni implementate da OpenStack presentate nel Capitolo 3.

```
neutron net-create net1
neutron subnet-create subnet1_net1 10.0.0.0/24
```

Sono state aggiunte poi le Virtual Machines necessarie: prima di tutto è stato recuperato l'id dell'immagine di *Ubuntu 14.04 - TS3* in formato *.qcow2, successivamente si è recuperato l'id del flavor *small* (che è risultato essere il 2) e il net-id della rete net1 a cui si sono andate collegare queste Virtual Machines; la creazione di queste è stato dunque l'ultimo passo.

```
nova image-list
nova flavor-list
neutron net-list
nova --flavor 2 --image fd23f971-582a-4d6b-a9be-
  dd6d6487bda9 --nic net-id=f9c3d24e8-2b0c-4a48-
  93c4-d108a1252435 bususer
nova --flavor 2 --image fd23f971-582a-4d6b-a9be-
  dd6d6487bda9 --nic net-id=9c3d24e8-2b0c-4a48-
  93c4-d108a1252435 resuser
nova --flavor 2 --image fd23f971-582a-4d6b-a9be-
  dd6d6487bda9 --nic net-id=9c3d24e8-2b0c-4a48-
  93c4-d108a1252435 dpi
```

```
nova --flavor 2 --image fd23f971-582a-4d6b-a9be-
  dd6d6487bda9 --nic net-id=9c3d24e8-2b0c-4a48-
  93c4-d108a1252435 wana
nova --flavor 2 --image fd23f971-582a-4d6b-a9be-
  dd6d6487bda9 --nic net-id=9c3d24e8-2b0c-4a48-
  93c4-d108a1252435 tc
```

È stato successivamente possibile creare il Virtual Router vr, trovando prima l'id della subnet subnet1_net1; a questo router è stata aggiunta un'interfaccia eth0 che è andata connessa alla rete net1 e ne è stato impostato il gateway sulla rete pubblica pub.

```
neutron router-create vr
neutron subnet-list
neutron router-interface-add vr 16df9249-37e8-
  457d-95b0-c6b98b0f6eb0 eth0
neutron router-gateway-set vr pub
```

È stata dunque eseguita una parte in cui si controllano le porte viste da OpenStack prima e dopo ogni istruzione di creazione di una porta sulla sottorete della net1, per poi farne un "attachment" alla relativa Virtual Machine, come il WANA e il TC. Il raggiungimento di questo traguardo è importante, perchè si è riusciti effettivamente a creare una Virtual Machine con due interfacce presenti contemporaneamente sulla stessa rete e che, tra l'altro, sono presenti anche come astrazioni in OpenStack.

```
neutron port-list
neutron port-create subnet1_net1
neutron port-list
nova interface-attach --port-id 32f06ff6-83b5-
  45b9-87dc-5839a73a2b20 wana
neutron port-create subnet1_net1
neutron port-list
nova interface-attach --port-id 8b96c835-9fd8-
  4d0b-b5eb-79ce0a6152a5 tc
```

La topologia OpenStack è stata dunque costruita mediante l'applicazione di questi comandi, con un risultato non dissimile da quello presentato in Figura 5.1, che rappresenta la topologia di rete vista dalla Web Dashboard di OpenStack, accessibile mediante browser.

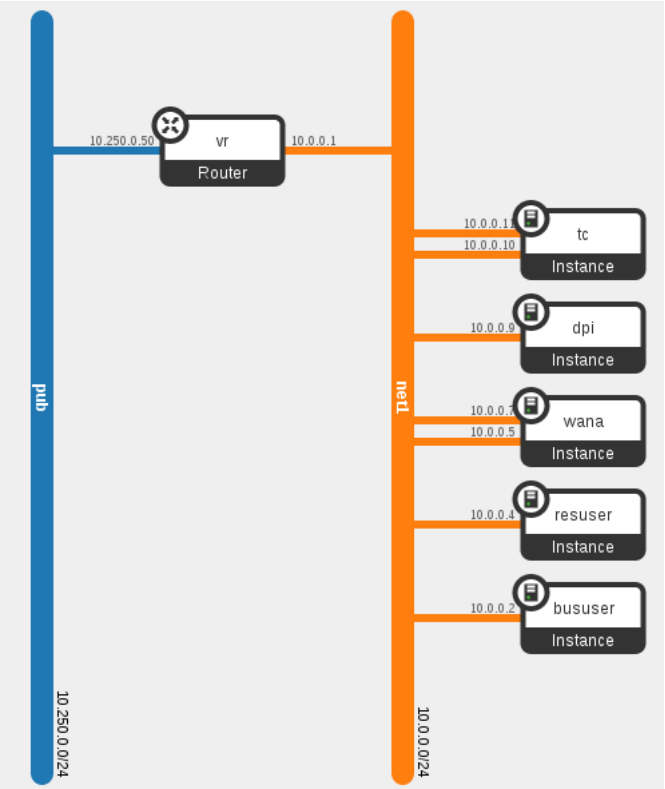

Figura 5.1: Topologia L2: Dashboard OpenStack

Nelle seguenti sezioni verranno aggiunte migliorie alla topologia costruita e si valuteranno e svilupperanno soluzioni ai problemi incontrati.

5.2 Migliorie applicate e risoluzione problemi

5.2.1 Aggiunta delle interfacce di Management

Un concetto molto importante affrontato durante questa fase di progettazione e implementazione è stato quello relativo alla necessità di non perdere la connettività SSH in caso di modifica dei parametri e delle impostazioni di rete.

Da qui dunque si è arrivati alla soluzione: l'aggiunta di un'interfaccia per poter andare in modalità "out of band", con indirizzo sulla rete interna 192.168.122.0/24, che permetterà dunque, una volta collegati via SSH a

questa, di poter agire sulle impostazioni di rete relative alle altre interfacce della Virtual Machine senza perdita di connessione.

Al fine di implementare questo concetto è stato creato un file xml denominato, ad esempio, *int.xml* il cui contenuto rispecchierà un codice simile a:

```
<interface type='network'>
  <source network='default'/>
  <model type='virtio'/>
</interface>
```

Questo file è stato utilizzato dopo aver ottenuto sia gli id che i nomi che OpenStack ha associato alle Virtual Machines; infine, dopo ogni "attach" eseguito è stato ricercato l'indirizzo assegnato automaticamente a quell'interfaccia di management sulla rete 192.168.122.0/24, sfruttando il software Nmap [35]. Da queste risultano i seguenti ultimi 8 bit in formato decimale: .51 (TC), .70 (BusUser), .108 (DPI), .233 (ResUser), .240 (WANA).

```
nova list
nova show 020f3bb8-6058-4b52-ae86-b7f8853ba48c
nova show e283474d-5236-4171-b8cf-bc5b3abdfc7b
nova show 68cd11d0-7ef2-4366-8906-cec4c839baaf
nova show f2cb9a37-5756-48c6-abd4-beb8cd29cc6b
nova show 3d54cff8-8507-4277-9916-1ed7f2cc6748
sudo virsh attach-device istance-000000be.xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device istance-000000bc.xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device istance-000000bf.xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device istance-000000bb.xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device istance-000000ba.xml
sudo nmap 192.168.122.0/24
```

5.2.2 Creazione e gestione dei bridge interni

Proprio sfruttando una connessione SSH in modalità out-of-band si è potuto costruire il bridge internamente alle Virtual Machines WANA e TC, neces-

sario all'implementazione delle corrette Service Chains; questo bridge infatti connette le due interfacce delle Virtual Machines e, sfruttando i meccanismi di steering e gli applicativi livello-kernel che verranno affrontati successivamente si riesce ad ottenere il corretto funzionamento dinamico della rete. Sarà inoltre necessario, ai fini della prova, disabilitare lo Spanning Tree Protocol (STP) di ogni bridge per permettere la creazione di cicli all'interno della topologia e di eliminare gli indirizzi IP alle due interfacce collegate al bridge per obbligare i pacchetti diretti verso la rete internet a passare forzatamente dall'interfaccia di management.

```
ssh -X ubuntu@192.168.122.240
sudo ovs-vsctl add-br ovs1
sudo ovs-vsctl add-port ovs1 eth0
sudo ovs-vsctl add-port ovs1 eth1
sudo ip addr flush dev eth0
sudo ip addr flush dev eth2
exit

ssh -X ubuntu@192.168.122.51
sudo ovs-vsctl add-br ovs2
sudo ovs-vsctl add-port ovs2 eth0
sudo ovs-vsctl add-port ovs2 eth1
sudo ip addr flush dev eth0
sudo ip addr flush dev eth2
exit
```

5.2.3 Superamento problemi relativi ai Security Groups

È importante essere a conoscenza del fatto che OpenStack implementi i suoi Security Groups mediante regole iptables ed ebtables: questi conterranno alcune regole relative al MAC e IP spoofing. Essendo queste deleterie per gli esperimenti, vengono inserite altre regole atte a bypassarle.

```
sudo iptables -I OUTPUT -j ACCEPT
sudo iptables -I INPUT -j ACCEPT
sudo ebtables -t nat -I POSTROUTING -j ACCEPT
sudo ebtables -t nat -I PREROUTING -j ACCEPT
```

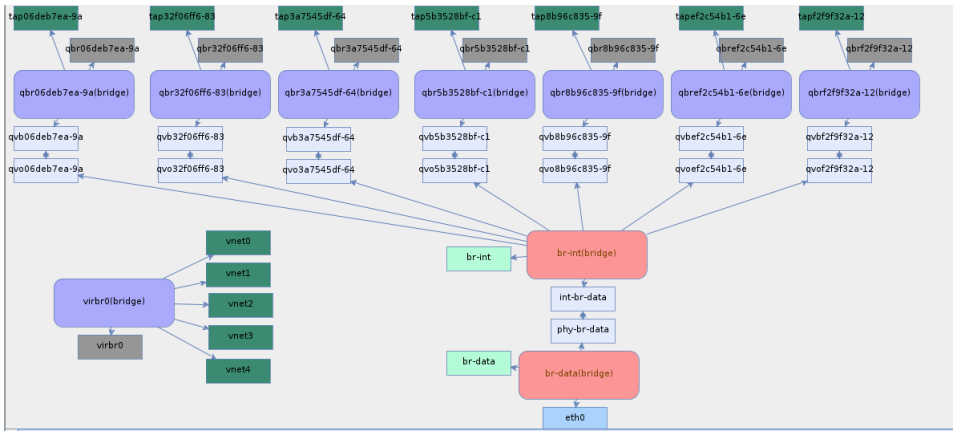

Figura 5.2: Compute Node in Show My Network State: L2

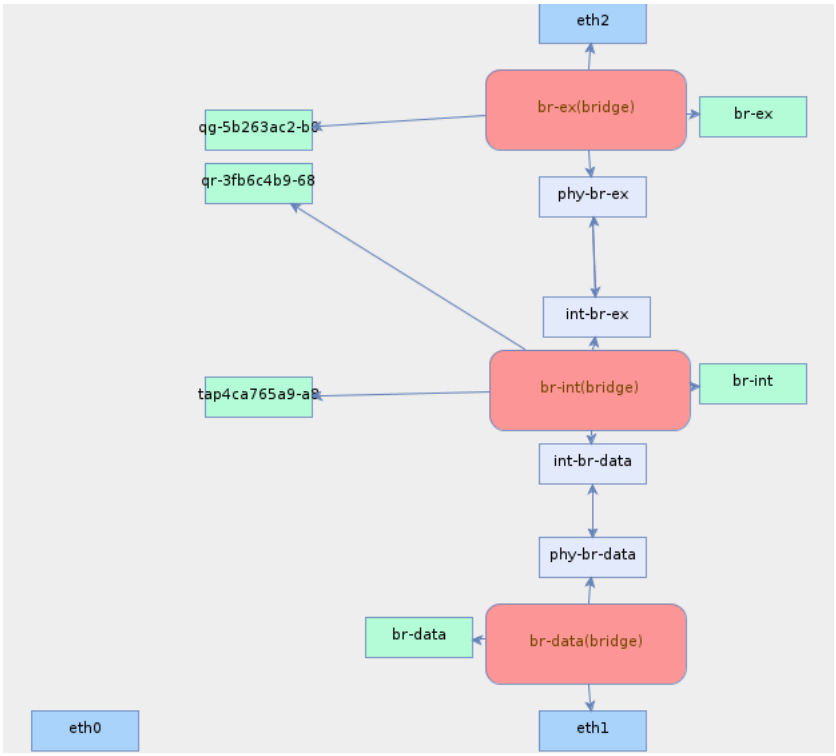

Figura 5.3: Network Node in Show My Network State: L2

5.2.4 Eliminazione di alcune fonti di complessità

Un dettaglio molto importante è che OpenFlow 1.0 non supporta il traffico IPv6: questo è stato attestato anche valutando l'installazione di Mininet e dei pacchetti ad esso associato; si è dunque provveduto a disabilitare ogni fonte di traffico IPv6 all'interno della rete e principalmente su ogni interfaccia; infatti alcuni pacchetti di tipo IPv6 avrebbero generato dei fastidiosi loop di traffico spurio.

Si è provveduto inoltre a inserire tutte le Virtual Machines relative al cluster OpenStack in un unico nodo, per semplificare, di molto, la gestione delle stesse e delle regole OpenFlow che direzionano i flussi.

Un'altra fonte di complessità che si è provveduti a eliminare è stata quella dell'allocazione dinamica degli indirizzi mediante DHCP; gli indirizzi di rete delle Virtual Machines sono stati dunque modificati e posti come statici, direttamente sfruttando le relative configurazioni di rete, in modo da rendere certe modifiche permanenti. Questo è stato fatto anche per una ben precisa necessità: può succedere infatti che le Virtual Machines perdano il proprio IP a causa di problemi della rete (ad esempio un loop di pacchetti); questo impedirebbe loro di contattare il DHCP server per ottenere un altro indirizzo, portando i suddetti pacchetti di DHCP Request ad aumentare ancora di più il numero di pacchetti che entrano nei suddetti loop.

Infine, come già asserito anche nel capitolo 4, viene passato per conosciuto alla rete il Service Level Agreement dell'utente, in modo da semplificare molto il concetto di Service Chaining.

5.2.5 Problema dell'ARP Storming e regolarizzazione traffico ICMP e ARP

È stata affrontata inoltre in questa fase la problematica dell'ARP storming; risulta infatti naturale che in una topologia di rete simile, con dei bridge le cui interfacce sono sulla stessa rete e con STP abilitato, si risenta di un problema di questo tipo.

L'ARP storming è infatti un fenomeno che si verifica a causa del loop che nasce per via del bridge interno configurato in questo modo e che comporta una proliferazione dei pacchetti ARP Request (quindi pacchetti con MAC address broadcast, cioè ff:ff:ff:ff:ff:ff) che ricircolano nel bridge entrando ad esempio su eth0, uscendo su eth1 e rientrando in eth0; questo problema comporta

anche un notevole peggioramento della qualità dei collegamenti: dopo pochi secondi infatti la rete OpenStack risulta irraggiungibile e le Virtual Machines perdono automaticamente il proprio IP senza riuscire a contattare il DHCP per acquisirne uno nuovo.

L'unico modo per continuare a sviluppare questa topologia senza risentirne è stato quello di implementare nel bridge *br-int* del compute node di OpenStack le adeguate regole OpenFlow, assieme ad altre regole volte a regolarizzare il traffico ARP in generale e il traffico di pacchetti ICMP; questo verrà ampiamente trattato nei seguenti paragrafi e nell'Appendice B.

5.2.6 Script per ripulitura della topologia

È stato valutato che, tra i vari problemi che si possono presentare, uno è sicuramente quello determinato da un funzionamento non corretto che si viene a verificare quando i filtering database dei bridge non sono stati ripuliti; sfruttando infatti dispositivi che si basano su una versione di OpenFlow pari a 1.0, spesso si rischia di incorrere in alcuni bug che, con molta probabilità, non esistono nelle successive versioni.

Nonostante ciò, al fine di prevenire questi errori è stato progettato uno script la cui funzionalità è ripulire tutti i dati relativi alle vecchie sperimentazioni, in modo da far ripartire il sistema ogni volta da zero.

```
set -x
set -u

#openflow ports
resuser=4
bususer=3
ponte1_1=10
ponte1_2=11
ponte2_1=5
ponte2_2=6
outport=47
internal_vid=57
bcast="ff:ff:ff:ff:ff:ff"
ip_mgmt_ponte_1="192.168.122.240"
ip_mgmt_ponte_2="192.168.122.51"
ip_mgmt_resuser="192.168.122.233"
ip_mgmt_bususer="192.168.122.70"
ip_resuser="10.0.0.4"
ip_bususer="10.0.0.2"
```

```
ip_router="10.0.0.1"
DPIcredentials='ubuntu@192.168.122.108'
DPI_lbr='qbr3a7545df-64'
h67_creds="root@137.x.y.67"
ip_mgmt_hc01="137.x.y.61"

echo "Current configuration of br-int"
sudo ovs-appctl fdb/show br-int
sudo ovs-ofctl dump-flows br-int

sudo su -c 'echo 1 > /sys/class/net/'${DPI_lbr}'/bridge/
    flush'

#set the controllers for the switches
sudo ovs-vsctl del-controller br-int
sudo ovs-vsctl set-controller br-int  tcp:127.0.0.1:6633

ssh ${h67_creds} ovs-vsctl del-controller br3
ssh ${h67_creds} ovs-vsctl set-controller br3 "tcp:${
    ip_mgmt_hc01}:6633"

echo
echo "Flushing the caches of the switches and hosts..."
echo

#clear the OF table and filtering DB of the br-int virtual
    bridge
sudo ovs-ofctl del-flows br-int
sudo ovs-appctl fdb/flush br-int

ssh ${h67_creds} sudo ovs-ofctl del-flows br3
ssh ${h67_creds} sudo ovs-appctl fdb/flush br3

ssh ${h67_creds} sudo ovs-ofctl del-flows br4
ssh ${h67_creds} sudo ovs-appctl fdb/flush br4

#flush the arp cache in the L2 function and the filtering
    DB in the bridge ovs1
ssh ubuntu@${ip_mgmt_ponte_1} sudo ovs-appctl 'fdb/flush'
    ovs1
ssh ubuntu@${ip_mgmt_ponte_1} sudo ip neigh flush dev ovs1

#the same for ovs2
ssh ubuntu@${ip_mgmt_ponte_2} sudo ovs-appctl 'fdb/flush'
    ovs2
```

```
ssh ubuntu@${ip_mgmt_ponte_2} sudo ip neigh flush dev ovs2

#kill the iperf client instances in the BU and RU
ssh ubuntu@${ip_mgmt_resuser} sudo killall iperf
ssh ubuntu@${ip_mgmt_bususer} sudo killall iperf

#flush the ARP cache in the VR
ssh hc02 sudo ip netns exec qrouter-1a1d1dd7-87dc-4279-9c41
    -e7245968d4cd ip neigh flush dev qr-3fb6c4b9-68

#kill the iperf server instances in the VR
ssh hc02 sudo ip netns exec qrouter-1a1d1dd7-87dc-4279-9c41
    -e7245968d4cd killall iperf

#stop DPI
ssh ${DPIcredentials} 'sudo killall ndpi'
```

5.2.7 Problematiche legate al DPI

È stata valutata una problematica relativa alla singola Virtual Machine DPI: questa infatti, nelle prime sperimentazioni, non riceveva i pacchetti ad essa destinati.

Si è dunque valutato logicamente il percorso dei pacchetti fatti e si è risaliti alla seguente consequenzialità degli eventi:

1. una Virtual Machine, come ad esempio il BusUser, invia un pacchetto ARP Request (quindi con destinatario broadcast) per conoscere il MAC address del Virtual Router;

2. una volta arrivato a br-int, quest'ultimo associa il MAC sorgente al BusUser (meccanismo di Learning) e farà un'azione di flooding del pacchetto ricevuto;

3. il suddetto pacchetto arriverà al bridge qbr3a che a sua volta farà il learning e successivamente il flooding dello stesso (primo pacchetto ARP che si può vedere catturato in Figura 5.5);

4. contemporaneamente, il pacchetto sarà arrivato, mediante i meccanismi implementati da OpenStack, fino al Virtual Router, che dovrà rispondere con il suo ARP Reply;

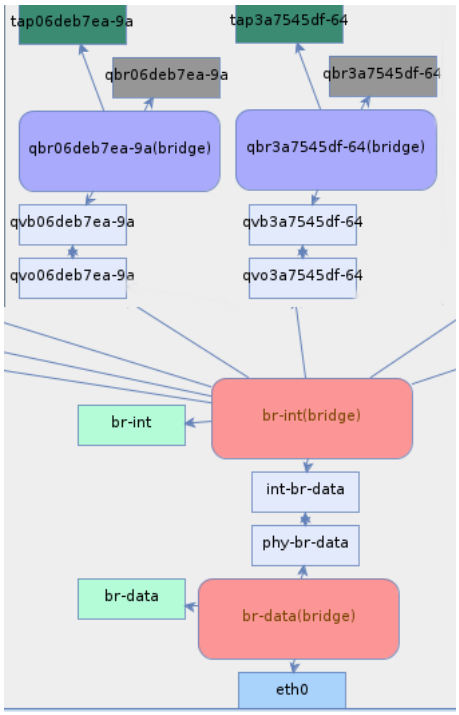

Figura 5.4: Particolare del compute node con BusUser e DPI

```
stack@hc01:~$ []{admin-admin} sudo tcpdump -nnvei tap3a7545df-64
tcpdump: WARNING: tap3a7545df-64: no IPv4 address assigned
tcpdump: listening on tap3a7545df-64, link-type EN10MB (Ethernet), capture size 65535 bytes
14:55:57.751503 fa:16:3e:18:01:cc > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Ether
net (len 6), IPv4 (len 4), Request who-has 10.0.0.1 tell 10.0.0.2, length 28
14:55:58.749033 fa:16:3e:18:01:cc > fa:16:3e:d4:59:0e, ethertype IPv4 (0x0800), length 74: (tos
 0x0, ttl 64, id 43405, offset 0, flags [DF], proto TCP (6), length 60)
    10.0.0.2.60349 > 10.0.0.1.12345: Flags [S], cksum 0x1431 (incorrect -> 0x6ed3), seq 1345163
542, win 29200, options [mss 1460,sackOK,TS val 474691988 ecr 0,nop,wscale 7], length 0
```

Figura 5.5: Test di cattura dei pacchetti sull'interfaccia eth0 del DPI

5. la risposta, che sarà destinata al BusUser, arriverà sul br-int che farà dapprima l'azione di Learning (associando il MAC sorgente al VR) e poi quella di inoltro, inviandola esclusivamente al BusUser, conoscendo già il suo MAC e sapendo su che porta inviare il pacchetto affinchè quest'ultima lo riceva. Ora il BusUser conoscerà il MAC del VR;

6. viene inviato dunque ora il primo pacchetto TCP, contenente il SYN, dalla BusUser verso il VR. Questo pacchetto arriverà al br-int che, in base

alle regole OpenFlow implementate, lo spedisce in output sulla porta del DPI (l'output infatti è un azione che non agisce sul filtering database del bridge) arrivando fino al qbr3a;

7. il qbr3a svolgerà l'azione di Learning, imparando il MAC della BusUser associandolo alla porta 1 del bridge. Non conoscendo però il MAC della VR, svolgerà anche un'azione di flooding (pacchetto TCP visibile in Figura 5.5);

8. il pacchetto arriverà dunque, dopo passaggi simili ai precedenti, al VR che risponderà con il pacchetto TCP contenente i SYN+ACK. Questo però giungerà al qbr3a sempre dalla porta 1, quindi, svolgendo di nuovo l'azione di Learning, si ottiene un risultato ambiguo: alla porta 1 saranno dunque associati entrambi i MAC address di VR e BusUser causando l'impossibilità di indirizzamento dei pacchetti in arrivo verso il DPI;

9. ogni pacchetto destinato alla DPI d'ora in avanti infatti entrerà nel bridge qbr3a e ne uscirà subito dopo dalla stessa porta, in virtù delle associazioni create nelle varie fasi di Learning.

La risoluzione a questo problema è quella di far funzionare il suddetto bridge come un hub, al fine di far arrivare correttamente i pacchetti al DPI:

```
sudo brctl setageing qbr3a7545df-64 0
```

5.3 Implementazione WANA mediante Trafficsqueezer

L'implementazione di Trafficsqueezer è avvenuta mediante interfaccia web: una volta collegati alla Virtual Machine WANA in cui si vuole configurare il suddetto software, precedentemente installato, è stata aperta un'istanza browser (*Firefox*[36] o *lynx*[37]).

Al suo interno è stato eseguito il semplice wizard, scegliendo la modalità bridge, ponendo come porta LAN quella su eth0, mentre la porta WAN è stata posta su eth2; dopo numerosi esperimenti di test è stato valutato che il software Trafficsqueezer, quando lavora in modalità bridge, stranamente non implementa da sè il bridge interno a cui collegare le interfacce. È stato pertanto

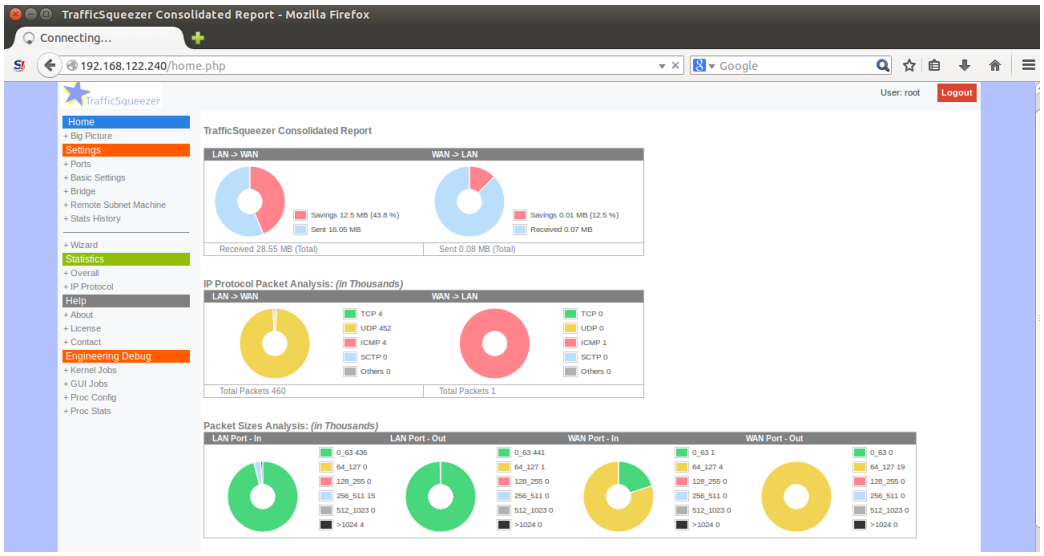

Figura 5.6: Trafficsqueezer: Interfaccia web

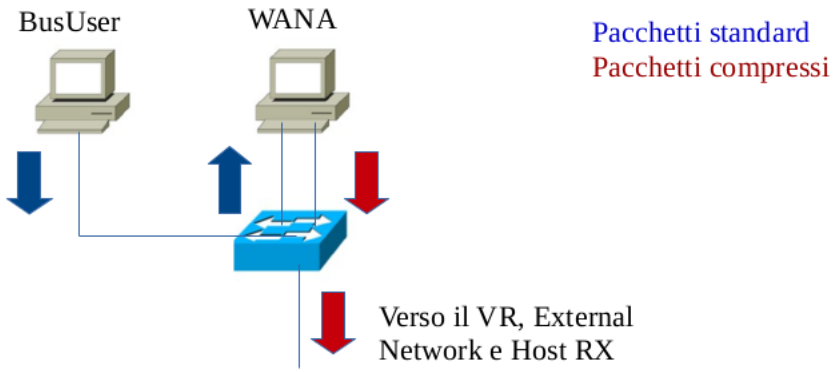

Figura 5.7: Pacchetti compressi e standard in gioco

```
16:07:32.418806 fa:16:3e:d5:07:eb > 52:54:00:7e:2c:27, ethertype IPv4 (0x0800), length 108: (tos 0x0, ttl 63, id 40835, offset 0, flags [DF], pr
oto unknown (201), length 94)
    10.250.0.50 > 10.20.30.1:  ip-proto-201 74
16:07:34.087015 fa:16:3e:d5:07:eb > 52:54:00:7e:2c:27, ethertype IPv4 (0x0800), length 108: (tos 0x0, ttl 63, id 40836, offset 0, flags [DF], pr
oto unknown (201), length 94)
    10.250.0.50 > 10.20.30.1:  ip-proto-201 74
```

Figura 5.8: Cattura di pacchetti TCP compressi

necessario costruirlo, mediante la suite di comandi *ovs-vsctl*, al fine di ottenere il corretto funzionamento.

Questo dispositivo naturalmente, come già sostenuto nei precedenti capitoli, avrà bisogno del suo dispositivo complementare: infatti i pacchetti che attraverseranno il suddetto subiranno delle modifiche, a seconda della rete implementata: nel caso di studio affrontato, venivano rimossi alcuni header ed inserito un diverso campo nel settore Network Protocol del pacchetto IP (200, 201 oppure 202).

Dunque in uscita dal WANA si avranno pacchetti compressi che attraversano la core network fino a giungere all'altra edge network, realizzata in termini di Host 67 dove è stato implementato (vedere Appendice A) il dispositivo complementare WANAdec, che si occupa della decompressione degli stessi pacchetti.

5.4 Progetto di implementazione Traffic Shaper mediante Traffic Control

Come già fatto presente, al fine di configurare il Traffic Control nel kernel Linux si fa uso di *tc*, una potente suite di comandi volta a queste necessità. Tramite questa si possono difatti implementare svariate funzioni relative al traffico:

- shaping: la transmission rate è tenuta sotto controllo, permettendo dunque di abbassare la banda disponibile;
- scheduling: si pianifica la trasmissione dei pacchetti in modo da migliorare l'interattività del restante traffico che richiede la banda garantita;
- policing: si monitorano i flussi di traffico in arrivo;
- dropping: i pacchetti che eccedono di un certo valore settato in banda vengono direttamente scartati.

Per il caso di studio in questione è stato valutato che l'alternativa più semplice e più simile al caso reale che si potrebbe un giorno implementare fisicamente su infrastrutture del provider di servizi, è quella dello Shaping.

L'elaborazione del traffico è controllata da tre tipi di oggetti: *qdiscs*, *classes* e *filters*. Le prime sono intese come "queieing discipline", ovvero quando il kernel ha bisogno di inviare un pacchetto ad un'interfaccia, questo sarà messo

in coda alla qdisc configurata per questa interfaccia; subito dopo il kernel cercherà di ottenere quanti più pacchetti possibili dalla qdisc. Un esempio molto semplice è quello della qdisc *pfifo*, una coda puramente First In, First Out e che quindi non svolge elaborazioni.

Le qdisc possono inoltre contenere le classi (classes), che conterranno a loro volta altre qdisc; il traffico potrebbe dunque essere messo in coda in una di queste qdisc più interne. Una qdisc può dunque per esempio dare maggiore priorità a certi tipi di traffico cercando di togliere pacchetti dalla coda di certe classi piuttosto che di altre.

Infine i filtri (filters) sono utilizzati da qdisc interne alle classi al fine di determinare in quale classe i pacchetti verranno messi in coda.

Tenendo dunque conto del fatto che il funzionamento di tc si basa su una topologia in cui le classi sono disposte ad albero (ogni classe genitore può avere più classi figlie, ma ogni classe figlia ha solo un genitore), si può pensare ad un progetto per cui, impostando i corretti valori di banda massima raggiungibile dal flusso di traffico, si ottenga la funzione di shaping ricercata.

Sfruttando dunque tutto ciò che è stato sopra detto riguardo al comando tc, è stato preparato uno script:

```
#! /bin/bash

if [ $# -lt 3 ]
then
        echo "Usage: $0 <IP dest> <bw limit in Mbps> <
            interface NODE>"
        exit -1
fi

IPdest=$1
BWlimit=$2
DEV=$3

tc qdisc del dev $DEV root
tc qdisc add dev $DEV root handle 1: cbq avpkt 1000
    bandwidth 1000mbit
tc class add dev $DEV parent 1: classid 1:1 cbq rate ${
    BWlimit}mbit allot 1500 prio 5 bounded
tc filter add dev $DEV parent 1: protocol ip prio 16 u32
    match ip dst $IPdest flowid 1:1
tc qdisc add dev $DEV parent 1:1 sfq perturb 10
```

Questo script, come si può vedere, deve essere fatto partire con un comando in cui sono inclusi alcuni dettagli dello shaping: prima di tutto l'IP destinazione, poi il limite in banda (misurato in Mbps) e l'interfaccia su cui si vuole applicare il meccanismo. L'input dato dunque è:

```
# ./setbwlimit.sh 10.20.30.1 10 eth2
```

Il funzionamento rispecchierà dunque ciò che si vuole teoricamente implementare: il traffico della ResUser, nel momento adeguato, attraverserà il Traffic Shaper e, prima di uscire dall'eth2, questo verrà tagliato in banda.

Come si può vedere in Figura 5.9, il traffico TCP viene ridotto una volta trascorsi i 60 secondi necessari al DPI per classificare il traffico.

Per quanto riguarda invece l'UDP, essendo questo un protocollo connectionless, non si può avere una misura precisa nelle statistiche che iperf stampa in command line: queste saranno infatti relative solo all'invio e verrà mostrata la cifra di circa 101 Mbit/s ad ogni intervallo: ciò è fuorviante, ma, valutando bene i dati che iperf offre si può scoprire che, una volta chiuso il collegamento, il server invia la statistica della media del traffico ricevuto.

Al fine di verificare l'effettiva riduzione di banda è stato dunque svolto un semplice calcolo: infatti si avrà che per 60 secondi si avranno 101 Mbit/s, mentre per i restanti 340 secondi dell'esperimento si avranno circa 10 Mbit/s; il valore che viene dato da iperf al termine del collegamento è 23,7 Mbit/s.

$\frac{(60*101)+(340*10)}{400} = 23,6 Mbit/s$

Come si può vedere, il valore teorico ottenuto è molto simile al valore ottenuto realmente: il Traffic Shaper dunque funziona bene anche con il protocollo UDP.

Testando inoltre il traffico UDP della ResUser senza nessun comando tc implementato, si potrà vedere che la banda media che verrà comunicata alla fine sarà di 100 Mbit/s, confermando dunque che la riduzione di banda teorizzata e vista precedentemente è stata effettivamente realizzata mediante suite di comandi tc del kernel Linux.

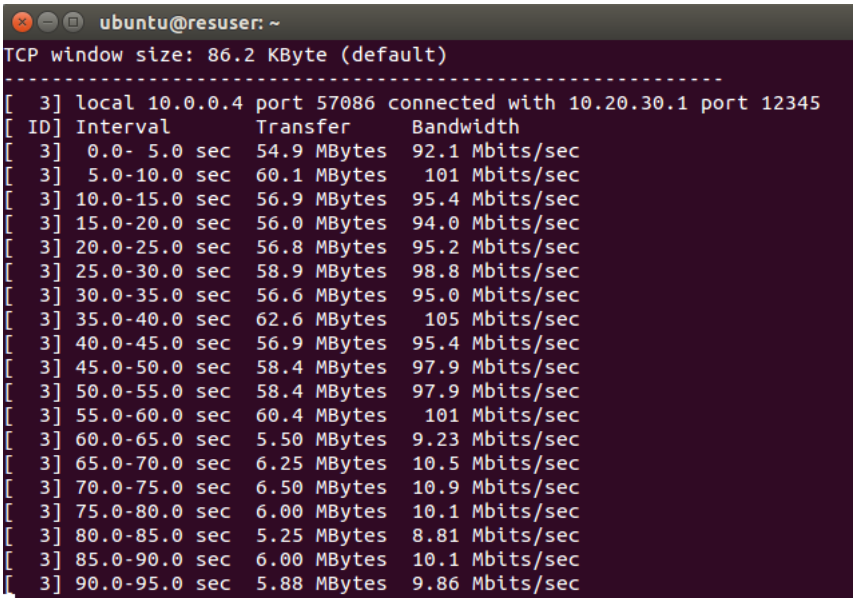

Figura 5.9: Cattura di pacchetti TCP dopo Traffic Shaper

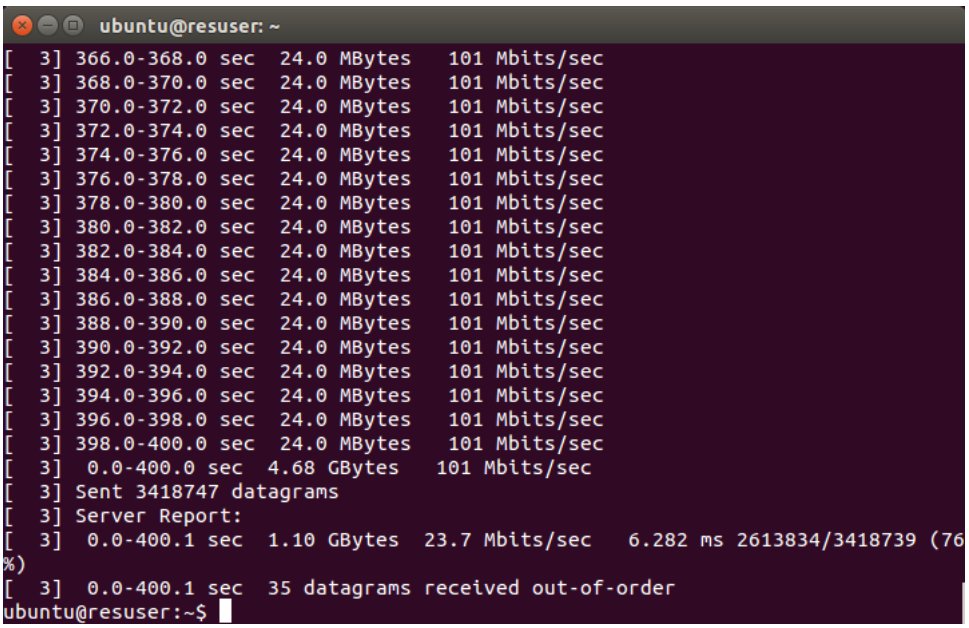

Figura 5.10: Restringimento di banda di pacchetti UDP

5.5 Funzionamento

Il procedimento effettuato è molto semplice, una volta tenuto conto di tutto ciò che è stato implementato precedentemente; il primo passaggio effettuato è stato l'avvio dello script di pulizia dei resti delle precedenti prove:

```
./script_topologia.sh
```

Successivamente si è dunque andato ad avviare il controller POX, attendendo che venissero installate le regole per il corretto funzionamento relative ad ARP e ICMP; a questo punto è dunque possibile, all'interno della VM Sink, far partire un server UDP iperf con porta 12345 e uno sulla porta 12346:

```
iperf -s -u -p 12345 &
iperf -s -u -p 12346 &
```

L'utilizzo di iperf e non di iperf3 è stata valutato in merito a esperienze pratiche che hanno portato alla visualizzazione di problemi relativi alla classificazione dei flussi fatta dal DPI: l'iperf3 infatti porta a flussi che si basasno su più porte contemporaneamente e, considerato che il DPI fornisce in uscita un file dpi.json in cui sono riportati i flussi che l'hanno attraversato, il codice scritto per il controller (che carica questo file e lo elabora al fine di comprendere a quale flusso applicare quali chains) non riconosceva questa possibilità.

È quindi possibile infine far partire un flusso iperf dalla Virtual Machine BusUser, diretto al Sink, dopo aver cancellato la ARP Table:

```
sudo ip neigh flush dev eth0 ; iperf -c 10.20.30.1
    -u -b 100m -p 12345 -i 2 -t 400
```

Una volta classificato il flusso UDP del BusUser mediante il DPI (e l'utilizzo del file dpi.json) e trascorsi 60 secondi, il flusso verrà indirizzato verso il WANA.

Dopo aver atteso circa 30 secondi dall'installazione delle regole di steering per il BusUser, si fa partire il flusso UDP iperf del ResUser:

```
sudo ip neigh flush dev eth0 ; iperf -c 10.20.30.1
    -u -b 100m -p 12346 -i 2 -t 310
```

Il funzionamento ora sarà quindi del tutto simile a quello che è stato per la BusUser: dopo 60 secondi di contesa del traffico in cui il DPI valuterà il flusso utente, questo verrà mandato attraverso il TC al fine di comprimerne la banda occupata.

Nel nodo ricevente il pacchetto compresso viene riconosciuto e fatto passare mediante regole OpenFlow implementate su br3, attraverso il WANAdec,

mentre il pacchetto UDP proveniente dalla ResUser verrà spedito direttamente al Sink senza ulteriori intermediari.

Il codice del controller POX relativo a questo livello di rete è visualizzabile nell'Appendice B.

5.6 Misure

Sfruttando i comandi della suite di *ovs-vsctl*, è stato sfruttato uno script ad-hoc che permette di ricavare il numero di byte dei pacchetti che transitano sulle interfacce di un determinato bridge. Questo script funziona campionando ogni numero di secondi il valore dei byte sulle porte, in ricezione e trasmissione. Per funzionare necessita anche di un ulteriore programma, scritto in C, che permette di ottenere una misura approssimativa del tempo per svolgere le varie opzioni: questo è postosotto forma di commento, per sola visione, in fondo al codice seguente.

```
#! /bin/bash

# gettimestamp (see bottom) must be present in the same
        directory as this script

if [ $# -lt 2 ]
then
        echo "Usage: $0 <OVS switch name> <sampling period
            in seconds>"
        exit -1
fi

ovsname=$1
period=$2
bwscale=$(echo "scale=6; 1/$period" | bc)

portlist=`ovs-vsctl list-ports $ovsname`

echo "# Sampling received data (in bytes) every $period
    seconds"
echo "# Bandwidth scale factor = $bwscale"
echo
echo -n "# time                    "
for p in $portlist
do
        echo -n "$p-byte-rx      $p-byte-tx       "
```

```
done
echo
echo

while true
do
        t=`./gettimestamp`
        echo -n "$t      "
        for p in $portlist
        do
                Brx=`ovs-ofctl dump-ports $ovsname $p |
                    grep rx | cut -d, -f2 | cut -d= -f2`
                Btx=`ovs-ofctl dump-ports $ovsname $p |
                    grep tx | cut -d, -f2 | cut -d= -f2`
                echo -n "$Brx    $Btx     "
        done
        echo

        sleep $period
done

# gettimestamp.c
#
##include <sys/time.h>
##include <stdio.h>
#
#main (void) {
#
#       struct timeval currentTime;
#
#       gettimeofday(&currentTime,NULL);
#       printf("%.6f\n",(double)currentTime.tv_sec+
    currentTime.tv_usec/1000000.0);
#
#}
#

#! /bin/bash
```

Quando questo script viene eseguito, va specificato, oltre all'intervallo di campionamento, anche il bridge su cui ovviamente si vuole fare questa misura; inoltre, il suo output si può poi redirigere su un file di testo:

```
sudo ./rcvdatafw.sh br-int 1 | tee prova3-br-int-2
   UDP.txt
```

Considerando che i contatori di ovs-ofctl non possono essere ripristinati se non cancellando e ricreando lo switch (cosa ovviamente poco agevole, soprattutto per il bridge br-int, essendo stato creato da OpenStack), si è proceduto in seconda istanza ad una elaborazione dei dati raccolti nel file di testo in ambiente di fogli di calcolo: ad esempio, nel caso specifico, LibreOffice Calc. L'obiettivo è stato quello di ottenere gli effettivi valori del tempo di campionamento, del numero dei byte dei pacchetti realmente transitati sulle porte di interesse e, dopo un'attenta valutazione di come elaborare questi dati, è stato trovato il throughput effettivo, tenendo soprattutto conto delle diverse grandezze degli intervalli di tempo.

È anche vero che questo metodo comporta un problema: infatti il conteggio su una porta tiene conto anche di pacchetti non riferiti al caso in esame; al fine di ottenere quindi una stima di quanti fossero realmente questi, si è tenuta sotto controllo, durante le numerose prove, la rete in vari punti. È stato valutato che in un intervallo di tempo di circa 400 secondi (e cioè il tempo in cui la sperimentazione avviene) transitano, oltre ai pacchetti che verranno poi valutati, un numero compreso tra 15 e 30 pacchetti, tutti di tipo ARP. Di conseguenza si è valutato che un numero così esiguo di pacchetti non dovrebbe compromettere la misura effettiva realizzata.

Successivamente, riportando questi dati in un documento di testo in formato *.dat sono stati ottenuti i seguenti grafici, utilizzando il software GNUplot.

Il primo grafico, rappresentato in Figura 5.11 mostra le due fasi di mirroring: nella prima onda quadra è rappresentato dunque il BusUser, nella seconda il ResUser.

In Figura 5.12 si può avere un'effettiva dimostrazione del corretto funzionamento del Traffic Shaper: in ingresso infatti si avranno circa 100 Mbps, mentre in uscita si vede un valore pari a 10 Mbps. In modo simile si può vedere anche la Figura 5.13, riferita al WANA: in ingresso infatti si hanno 100 Mbps, mentre in uscita i pacchetti saranno stati compressi arrivando ad un valore di circa 2,5 Mbps: questo è un risultato molto interessante in quanto porta a vedere come un dispositivo WAN Optimizer possa essere performante.

A fronte di queste prove, di fatto la BusUser non richiede quindi troppa banda per funzionare a regime: è anche vero però che questo dipende dal tipo di traffico generato. Essendo i pacchetti iperf probabilmente composti da un gran numero di zeri (o da pattern replicate) e con delle intestazioni che costuiscono la stragrande maggioranza dei dati, il risultato che è stato ottenuto può essere verosimile, ma al contempo leggermente diverso dal caso reale.

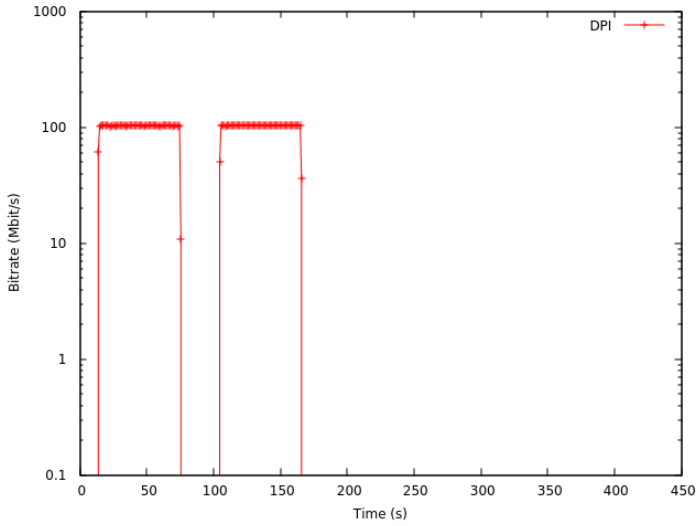

Figura 5.11: Pacchetti in ingresso al DPI

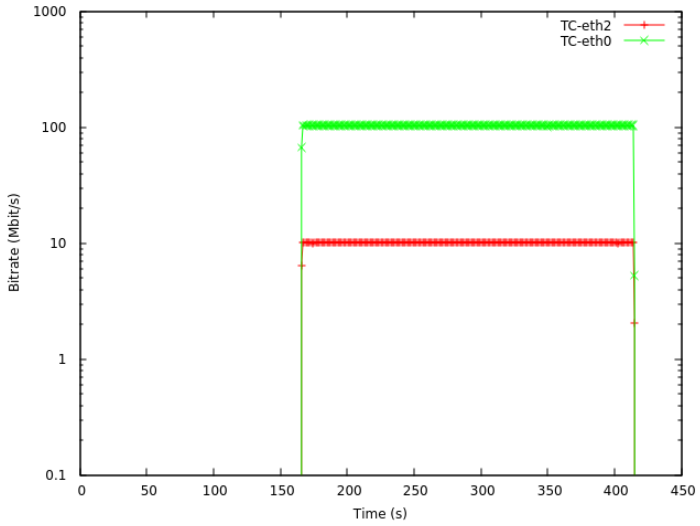

Figura 5.12: Confronto fra pacchetti in ingresso e in uscita al TC

Infatti prendendo ad esempio un flusso video compresso, questo non avrebbe un valore così basso di banda come lo è stato per i pacchetti iperf di questa sperimentazione, ma comunque si andrebbe a ottenere un risultato intermedio molto apprezzabile.

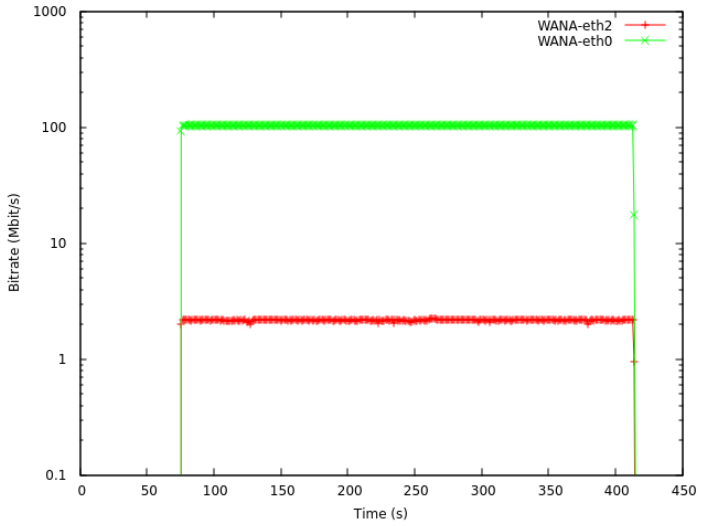

Figura 5.13: Confronto fra pacchetti in ingresso e in uscita al WANA

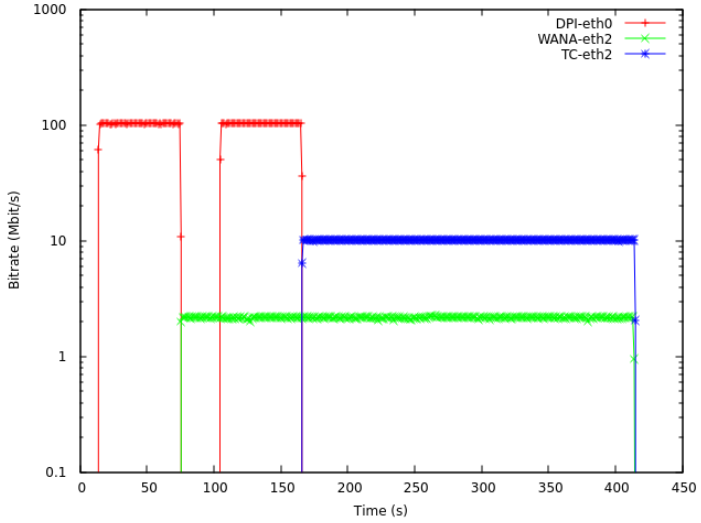

Figura 5.14: Traffico in ingresso al DPI e in uscita da WANA e TC

Infine, per completezza, si può vedere in Figura 5.14 l'andamento dei flussi in entrata al DPI assieme a quelli in uscita dal WANA e dal TC: si ha dunque maggiormente la percezione di come lo steering avvenga e di come siano dunque state costituite le service chains.

Capitolo 6

Caso di studio: Livello L3

Nel Capitolo 5 è stata valutata la Topologia di rete L2, mentre nel seguente si farà riferimento a quella di livello L3.

È importante comprendere sin dall'inizio che anche in questo caso la topologia fisica del cluster OpenStack non è uguale alla topologia virtuale, ora ancora più drasticamente rispetto a come lo è stato per la L2. Infatti questo dipende dalle astrazioni di rete implementate da Neutron, che comporta tra le varie cose, nel caso di livello L3, una topologia fisica molto simile a quella del livello L2 con la differenza che le diverse networks sono implementate in termini di differenti VLAN ID applicati alle porte.

6.1 Realizzazione della Topologia OpenStack

Considerando la Figura 4.3 si può avere un'idea di come questa topologia verrà implementata; in modo simile alla topologia L2 i componenti utilizzati saranno sempre WANA, TC, DPI a cui si aggiungerà un ulteriore router chiamato GW, oltre naturalmente alle due Virtual Machines ResUser e BusUser.

In questo caso si avranno i due utenti connessi alla net1, mentre DPI, WANA, TC e GW avranno un'interfaccia a testa sulla rete net1 e una sulla net2; infine il VR avrà un'interfaccia sulla net2 e l'altra andrà verso l'Host 67, che per questo specifico caso di studio è stato reimplementato al fine di ottenere una topologia speculare a quella appena riportata e quindi quanto più possibile realistica.

Nemmeno la costruzione della topologia stessa risulta diversa dal caso L2, come si può di seguito valutare. Essendo questa una topologia più semplice, in pochi passaggi è possibile ricrearla anche mediante Web Dashboard.

Una diversità degna di nota è però quella per cui, in questo livello di rete, non c'è bisogno di regole per prevenire l'ARP storming: nonostante per la topologia OpenStack infatti le Virtual Machines hanno le interfacce collegate entrambe al bridge br-int, le relative porte sono associate a diversi tag VLAN, comportando dunque l'immunità da quel tipo di problematica che invece colpiva il caso L2.

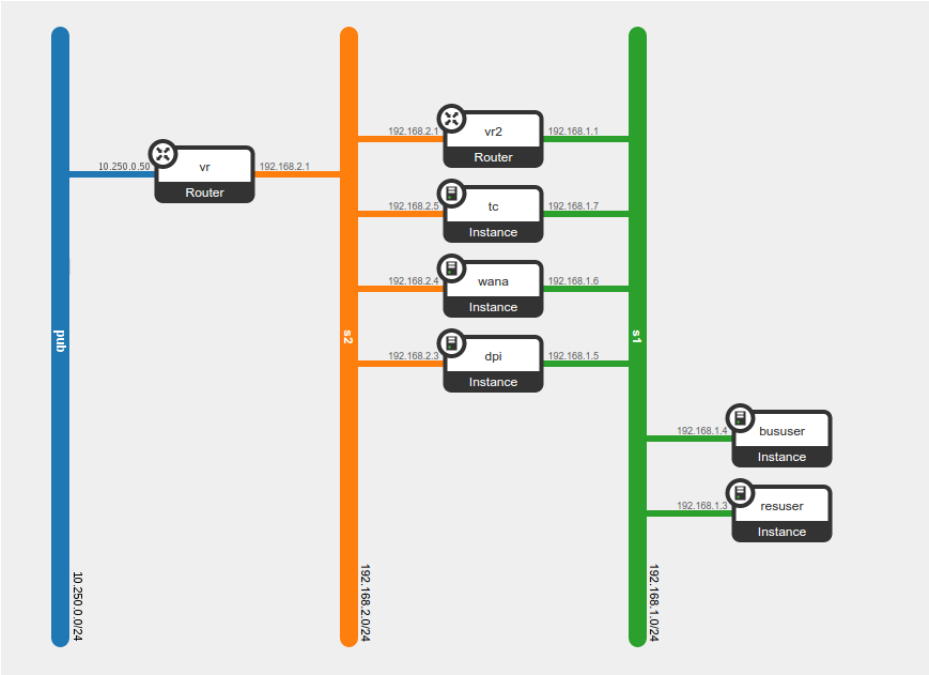

Figura 6.1: Topologia L3: Dashboard OpenStack

6.1.1 Progetto di Realizzazione mediante CLI

La connessione sfruttata per entrare nel cluster OpenStack è sempre quella SSH, come lo è stato per la costruzione della topologia di rete L2.

Una volta costruite le reti net1 e net2 e le relative subnets,

```
neutron net-create net1
```

```
neutron subnet-create subnet1_net1 10.10.10.0/24
neutron net-create net2
neutron subnet-create subnet1_net2 10.20.20.0/24
```

si possono aggiungere le Virtual Machines; in virtù della semplicità della topologia di rete, non è necessario ricorrere a meccanismi esterni per la creazione della doppia interfaccia, come per il caso L2. Sarà infatti sufficiente inserire il net-id di entrambe le reti net1 e net2 a cui le Virtual Machines DPI, WANA e TC dovranno essere collegate.

Essendo questo un progetto non ancora implementato praticamente, si fa riferimento ai net-id della topologia come ipotetiche variabili di uno script bash.

```
nova image-list #recupero image
nova flavor-list #recupero flavor
neutron net-list #recupero net-id delle due reti
nova boot --flavor 2 --image fd23f971-582a-4d6b-
   a9be-dd6d6487bda9 --nic net-id=$(net_id1)
   bususer
nova boot --flavor 2 --image fd23f971-582a-4d6b-
   a9be-dd6d6487bda9 --nic net-id=$(net_id1)
   resuser
nova boot --flavor 2 --image fd23f971-582a-4d6b-
   a9be-dd6d6487bda9 --nic net-id=$(net_id1) --
   nic net-id=$(net_id2) dpi
nova boot --flavor 2 --image fd23f971-582a-4d6b-
   a9be-dd6d6487bda9 --nic net-id=$(net_id1) --
   nic net-id=$(net_id2) wana
nova boot --flavor 2 --image fd23f971-582a-4d6b-
   a9be-dd6d6487bda9 --nic net-id=$(net_id1) --
   nic net-id=$(net_id2) tc
nova list #controllo le vm create
```

Il passo successivo è la creazione del Virtual Router e del GW, con i conseguenti collegamenti necessari; viene fatto presente che in OpenStack non è possibile inserire come gateway di un virtual router una rete creata da OpenStack stesso, ma solo reti esterne ad esse (ad esempio la rete pub 10.250.0.50/24 va bene). Nel caso in cui si voglia collegare un virtual router (ed è il caso del GW) tra due reti di OpenStack, bisogna ricorrere all'inserimento di due inter-

facce, senza poter impostare un gateway predefinito; è da valutare se questo comporti poi delle limitazioni in fase di sperimentazione. In quel caso sarà necessario sostituire il GW con una Virtual Machine ad-hoc, così come fatto all'interno dell'host 67.

```
neutron router-create vr #creazione del virtual
   router
neutron subnet-list #recupero il subnet-id di
   subnet1_net1
neutron router-interface-add vr $(subnet1_net1_id)
    eth0 # creazione interfaccia
neutron router-gateway-set vr pub # impostazione
   gateway del vr
neutron router-create gw #creazione del gw
neutron router-interface-add gw $(net_id1) eth0 #
   creazione interfaccia eth0
neutron router-interface-add gw $(net_id2) eth1 #
   creazione interfaccia eth1
```

6.2 Possibili migliorie applicabili

6.2.1 Aggiunta delle interfacce di Management

In modo del tutto simile al livello L2, anche qui viene affrontata la necessità di aggiungere ad ogni Virtual Machine un'interfaccia di management al fine di non perdere la connettività SSH nel momento in cui si sta lavorando su di essa e sulle sue interfacce.

Il procedimento è identico a quello affrontato per il livello L2: si crea dapprima il file xml,

```
<interface type='network'>
  <source network='default'/>
  <model type='virtio'/>
</interface>
```

si cercano dapprima gli id che OpenStack ha associato alle Virtual Machines, successivamente si vanno a cercare i nomi delle instances presenti in OpenStack per poi fare l'attachment dell'interfaccia.

Si vanno infine a ricercare quali indirizzi sulla rete 192.168.122.0/24 sono stati associati a ciascuna di esse mediante Nmap.

```
nova list
nova show $(bu_id)
nova show $(ru_id)
nova show $(wana_id)
nova show $(tc_id)
nova show $(dpi_id)
sudo virsh attach-device $(bu_istance).xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device $(ru_istance).xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device $(wana_istance).xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device $(tc_istance).xml
sudo nmap 192.168.122.0/24
sudo virsh attach-device $(dpi_istance).xml
sudo nmap 192.168.122.0/24
```

6.2.2 Superamento problemi relativi ai Security Groups

Il progetto prevede inoltre le stesse regole anti-spoofing per bypassare i Security Groups implementati da OpenStack.

```
sudo iptables -I OUTPUT -j ACCEPT
sudo iptables -I INPUT -j ACCEPT
sudo ebtables -t nat -I POSTROUTING -j ACCEPT
sudo ebtables -t nat -I PREROUTING -j ACCEPT
```

6.3 Progetto di implementazione avanzata

Non essendoci sostanziali differenze, il TC è implementabile sfruttando sempre lo script utilizzato per la topologia L2; per quanto riguarda il WANA invece le cose cambiano.

Una volta eseguito l'accesso mediante interfaccia web al software, si dovranno inserire le adeguate configurazioni; essendo ora in una topologia di

livello L3, come è tra l'altro possibile immaginare guardando la Figura 4.3, il Trafficsqueezer dovrà essere configurato in modalità Router.

Anche in questo caso sono eliminabili diverse fonti di complessità: il DHCP, ovviamente in coordinato all'inserimento di indirizzi statici all'interno delle Virtual Machines, le route statiche e l'impostazione, come default gateway, dell'interfaccia di management (in modo tale da mantenere all'interno della rete solo i pacchhetti effettivamente riferiti ad essa) e altri ancora molto simili a quelli per il caso L2.

Infine, anche la scrittura del controller riprenderà molto la forma del corrispettivo per la rete L2: tutti i metodi definiti possono essere infatti riutilizzati, mentre il cambio di regole necessario non comporta grosse difficoltà se non quella dell'effettiva valutazione dell'inserimento o della rimozione del tag vlan in riferimento alle porte verso cui questi pacchetti sono destinati.

Capitolo 7

Conclusioni

Questo documento ha affrontato l'argomento del Service Chaining dinamico in ambienti cloud, sfruttando i paradigmi SDN e NFV, applicati sulla piattaforma OpenStack, mediante l'utilizzo del protocollo OpenFlow. La struttura di rete è stata realizzata anche esternamente al cluster al fine di ottenere una topologia di rete quanto più possibilmente realistica.

L'implementazione pratica delle stesse strutture ha fatto notare come questa possa essere una soluzione possibile e al contempo performante, con le dovute attenzioni ad ogni dettaglio relativo ad esse.

Inoltre, i risultati pratici ottenuti hanno dimostrato l'efficacia di una soluzione del genere: si è dunque valutato ulteriormente come SDN e NFV possano convivere al fine di creare nuove possibilità di sviluppo in ambito di networking.

Le prospettive future in questo ambito sono molte, fermo restando la necessità di testare anche il funzionamento a livello L3.

Un'idea è quella di modellare il meccanismo che rende consapevole la rete del Service Level Agreement di un utente sfruttando il motore semantico Smart-M3 [38], in modo da ottenere una maggiore context-awareness del sistema in studio, contemporaneamente al progredire degli studi basati su NGSON [39].

Un'altra idea, sempre relativa al context-aware, è quella di sfruttare un diverso tipo di controller SDN, come Ryu o OpenDaylight: questo permetterebbbe di sfruttare un dispositivo che, installato a dovere nel cluster OpenStack e programmato in maniera efficiente, porterebbe ad una maggiore autonomia all'interno dello stesso cluster e ad infinite nuove possibilità.

Capitolo 8

Ringraziamenti

Se sono arrivato qua, devo assolutamente dire grazie ad alcune persone.

Vorrei prima di tutto ringraziare Walter Cerroni, Giuliano Santandrea e Chiara Contoli che mi hanno seguito nell'ambito di tirocinio prima, nel lavoro sperimentale poi e infine nella stesura di questo documento, dandomi preziosi consigli che spero di aver colto nel migliore dei modi.

È da ringraziare naturalmente la mia famiglia, per l'immenso sostegno datomi assieme alla continua fiducia e all'affetto. Tra questi è importante ringraziare in particolare i miei genitori Gennaro e Monica, mia sorella Fabiana e la mia metà Federica: siete sempre stati i miei punti fissi, i miei sostegni, coloro su cui so di poter sempre contare, giorno dopo giorno.

Infine, volevo spendere una menzione per quattro persone che, nonostante la grande lontananza, mi sono state sempre vicino: i miei nonni. Un enorme grazie dunque deve andare a Immacolata, o Pupetta, la donna forte e affettuosa e ad Antonio, dal sorriso gioioso che dopo ogni esame ha sempre chiesto scherzando del "cachiss" (ossia la lode); un altrettanto grande ringraziamento deve andare a Dora, dallo sguardo perso nei suoi ricordi da ragazza e Francesco, o Franco, dolce e splendida persona che è purtroppo venuta a mancare quando al mio traguardo mancavano due esami. Lui mi ha sempre detto, ogni volta che ci salutavamo sapendo di non poterci vedere per molti mesi, di rendere onore al nome e cognome che porto (e cioè, anche al suo). In questi mesi un po' più bui degli altri l'ho sempre portato con me, oltre che nel nome, anche nel cuore. Le ultime fatiche fatte per arrivare a concludere questo percorso sono state fatte con l'orgoglio che mi ha sempre detto di avere, per chi sono e per quello che faccio. E sono certo di non essere mai venuto meno a tutto ciò.

Quindi dico grazie a tutti voi; se oggi sono arrivato fin qua il merito non è solo mio, ma anche vostro.

Francesco

Appendice A

Nodo ricevente

Tenendo conto della topologia delle reti sfruttate e, considerando in essa anche l'intero cluster OpenStack (vedere Figura 4.8), l'implementazione del nodo ricevente è stata effettuata nell'host denominato Host 67.

Il suddetto ha l'interfaccia fisica eth0 sulla rete pubblica di ateneo, chiamata in codice rete x.y.z.0/24, il cui IP è x.y.z.67; è proprio attraverso questa che avviene la connessione SSH.

Al suo interno andranno create le Virtual Machines relative al WANA decompressor e al Sink, assieme ai bridge necessari allo steering dei pacchetti.

A.1 Implementazione del nodo ricevente

A.1.1 Livello L2

Sfruttando *libvirt* si ha la creazione dello switch virtuale virbr0, che possiede un'interfaccia con indirizzo 192.168.122.1; assieme ad essa viene creato un NAT che funziona in modalità masquerade, cosa che può essere verificata mediante l'utilizzo del comando:

```
# iptables -t nat -nvL
```

All'interno di questo nodo andranno implementati tre bridge: il primo è *brf*, bridge che si sfrutterà per pacchetti non strettamente relativi al caso di studio (ad es. ping verso siti internet), il secondo è *br3*, il bridge che gestirà le regole OpenFlow di steering relative al WANA decompressor e al Sink, mentre

il terzo è *br4*, ossia quel bridge creato per essere interfacciato all'interfaccia fisica eth2 e con una delle due interfacce del Virtual Router di destinazione.

Il primo passo è stato impostare una nuova regola iptables che permetta di applicare la regola di masquerade del nat anche per la rete 192.168.123.0/24 che si andrà poi a sfruttare:

```
# iptables -t nat -I POSTROUTING -s
    192.168.123.0/24 !
-d 192.168.123.0/24 -j MASQUERADE
```

È stato successivamente creato un file chiamato wanadec.xml (ad esempio a partire dal dumpxml di una qualsiasi altra Virtual Machine) nel quale, oltre alle modifiche dei dettagli implementativi a seconda delle esigenze e all'uso di un'immagine in formato qcow2, verranno inserite tre interfacce. È stato scelto in questo caso di non inserire nessun campo ulteriore a quello del"target dev", ossia il nome delle stesse interfacce, per non incorrere in problemi relativi a MAC address e slots virtuali già presenti all'interno della topologia in uso; queste tre interfacce sono state denominate tapmia2, tapmia3 e tapmia5.

```
<domain type='kvm'>
  <name>wanadec</name>
  <uuid>6da0c4ec-6b7b-04cf-0ad2-4d8737acb87c</uuid>
  <memory>1048576</memory>
  <currentMemory>1048576</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='x86_64' machine='rhel6.3.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/libexec/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' cache='none'/>
      <source file='/root/img2.qcow2'/>
```

```
      <target dev='hda' bus='ide'/>
      <address type='drive' controller='0' bus='0' unit='0'
          />
    </disk>
    <controller type='ide' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x01' function='0x1'/>
    </controller>
    <interface type='ethernet'>
      <mac address='52:54:00:bb:55:f5'/>
      <target dev='tapmia3'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x09' function='0x0'/>
    </interface>
    <interface type='ethernet'>
      <mac address='52:54:00:12:2e:56'/>
      <target dev='tapmia5'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x08' function='0x0'/>
    </interface>
    <interface type='ethernet'>
      <mac address='52:54:00:5d:cd:11'/>
      <target dev='tapmia2'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x04' function='0x0'/>
    </interface>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='-1' autoport='yes'/>
    <video>
      <model type='cirrus' vram='9216' heads='1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x02' function='0x0'/>
    </video>
    <memballoon model='virtio'>
      <address type='pci' domain='0x0000' bus='0x00' slot='
          0x03' function='0x0'/>
    </memballoon>
  </devices>
</domain>
```

La Virtual Machine è stata successivamente fatta partire sfruttando la suite di comandi di virsh con

```
# virsh define wanadec.xml
# virsh start wanadec
```

Entrando dunque nel WANAdec tramite *virt-manager* sono stati modificati direttamente i valori degli IP assegnati alle interfacce modificando i relativi file di configurazione eth*.cfg; quest'operazione è necessaria per eliminare gradi di complessità aggiuntivi derivanti dall'utilizzo di indirizzi IP dinamici.

Relativamente alle interfacce, la tapmia2 e la tapmia3 sono state connesse al bridge br3 mediante le istruzioni

```
# ovs-vsctl add-port br3 tapmia2

# ovs-vsctl add-port br3 tapmia3
```

mentre la restante, tapmia5, verrà connessa al bridge brf:

```
# ovs-vsctl add-port brf tapmia5
```

Ai fini della sperimentazione, la Virtual Machine WANAdec avrà le due interfacce eth0 ed eth1 (corrispondenti alle tapmia2 e tapmia3), messe in bridge tra loro internamente; si è andato dunque a creare un bridge *bri* interno alla WANAdec e ad esso sarebbero dovute essere aggiunte le due interfacce eth0 ed eth1. Questo però, in virtù di quanto già asserito nell'implementazione del livello L2, avrebbe comportato un loop di pacchetti, denominato ARP storming e già affrontato nel Capitolo 5, risolvibile solo mediante il corretto inserimento di regole apposite che bloccano il ricircolo dei pacchetti broadcast. A differenza di quanto succede con i loop all'interno della rete OpenStack, qui la situazione sarebbe stata anche più complicata: la valutazione di questo fenomeno, applicato al caso in esame e con le rispettive prove, è stato svolto nella sezione A.2.

Per i motivi che possono dunque essere visti successivamente è stato creato un Virtual Router, denominato VRdest; questo è stato implementato utilizzando una Virtual Machine per motivi di necessità, ma si sarebbe ottenuto un risultato non dissimile anche mediante Linux Namespaces.

La modalità con cui è stata implementata la Virtual Machine VRdest è del tutto simile a come è stato per il WANAdec, creando dunque un file xml e sfruttando virsh; anche questa avrà tre interfacce, tapmia7, tapmia8 e tapmia9, che sono state collegate rispettivamente a br4, br3 e brf. Una volta

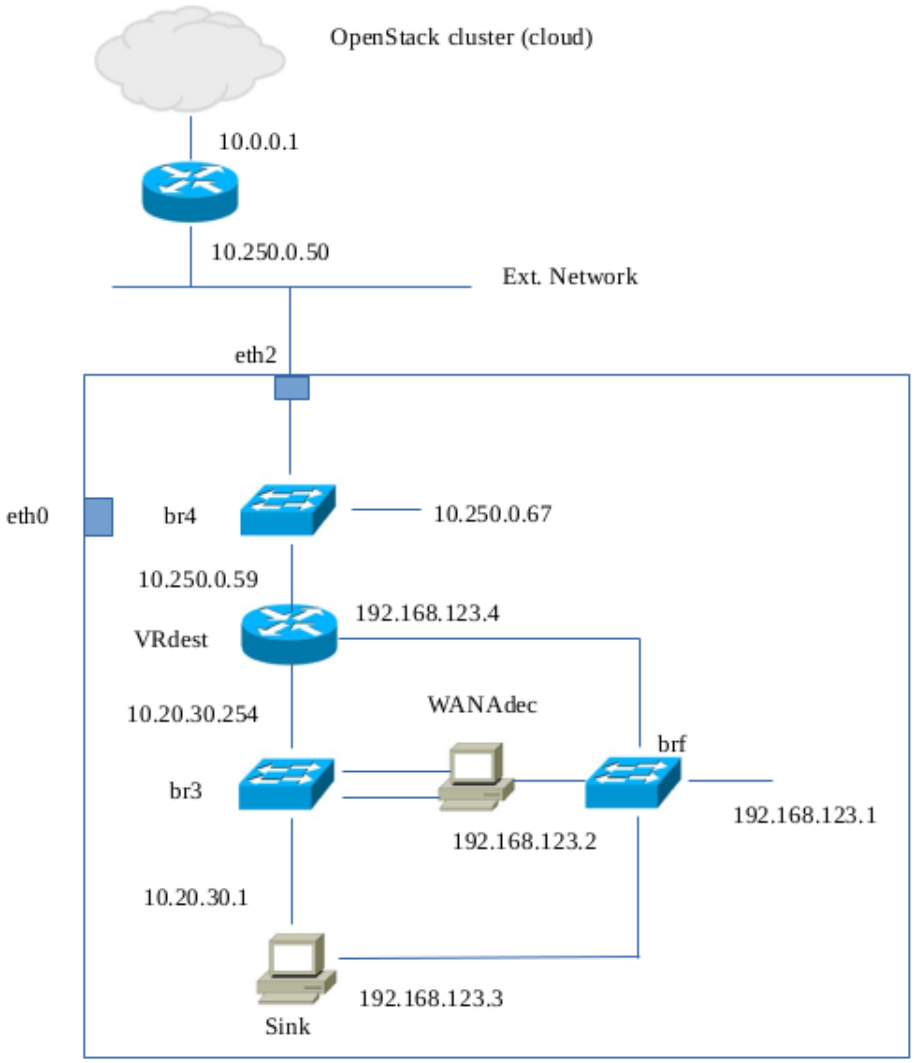

Figura A.1: Implementazione pratica L2 dell'host 67

abilitato l'IP forwarding, si è proceduto alla modifica degli indirizzi IP sulle sue interfacce; è piuttosto importante notare come la presenza del VRdest imponga la creazione di una nuova rete, identificata dagli indirizzi 10.20.30.0/24, a cui farà riferimento la Virtual Machine di destinazione Sink, che è stata successivamente creata.

L'aggiunta della Virtual Machine ricevente (chiamata Sink) è simile al caso qui sopra riportato relativo al WANAdec: si utilizza sempre un file xml modificato per l'occasione con due interfacce, tapmia4 e tapmia6, che virsh utilizzerà per farla partire; si configurano successivamente le tap con i bridge (tapmia4 collegata a brf e tapmia6 a br3) e si iniziano a fare le relative prove di connessione e cattura dei pacchetti inviati.

In questa situazione naturalmente il percorso dei pacchetti dalla ResUser o dalla BusUser non sarebbe completo: sono state infatti aggiunte delle route statiche mediante la suite di comandi *ip route* in vari nodi della rete; i comandi inseriti, rispettivamente nel ResUser e nel BusUser, nel VR e nel Sink sono:

```
sudo ip route add 10.20.30.0/24 via 10.0.0.1 #RU
    ,BU

sudo ip route add 10.20.30.0/24 via 10.250.0.59
    #VR

sudo ip route add 10.250.0.50/24 via
    10.20.30.254 #SINK
```

Un caso interessante può essere quello di ping (e dunque pacchetti ICMP) tra le Virtual Machines di OpenStack e il Sink: se questi vengono inviati dal ResUser o dal BusUser, non ci sono problemi di sorta; viceversa, inviati dal Sink, non tornerà mai indietro una risposta.

Questa casistica è stata valutata e la causa è stata additata alle funzioni di nat implementate dal Virtual Router relativo a OpenStack: questo svolge infatti una funzionalità di SNAT e quindi il singolo pacchetto arriverà senza problemi all'utente desiderato (BusUser o ResUser), ma con un indirizzo sorgente modificato e relativo all'interfaccia esterna a OpenStack del Virtual Router. Quindi il pacchetto ICMP arriverà all'utente, ma questo risponderà con un pacchetto che verrà spedito verso l'interfaccia del Virtual Router sulla rete 10.250.0.0/24, comportando una perdita totale dei pacchetti di ritorno relativamente al Sink.

A.1.2 Livello L3

In modo simile è possibile implementare anche la topologia di rete relativamente all'host 67 per il Livello L3; questa, visibile in Figura A.2, rispecchia la topologia di rete L3 stessa.

Infatti per quanto, con i dovuti accorgimenti, si sarebbe pure potuta mantenere la topologia sfruttata per il Livello L2, la scelta è ricaduta su una creata ad-hoc. Questo è stato fatto perchè il riutilizzo dei componenti sfruttati per la topologia L2 avrebbe portato all'utilizzo di dispositivi di rete L2 in una topologia L3. Per coerenza con il concetto prefissato di voler quindi implementare qualcosa che possa essere fisicamente realizzabile, la differenziazione che è stata fatta per i livelli L2 ed L3 per la edge network sorgente viene applicata anche alla edge network destinazione.

Le operazioni svolte per ricreare questa topologia non sono dissimili da quelle effettuate per la topologia L2: la differenza infatti risiede principalmente nella presenza di un ulteriore bridge, del GW e dei differenti collegamenti.

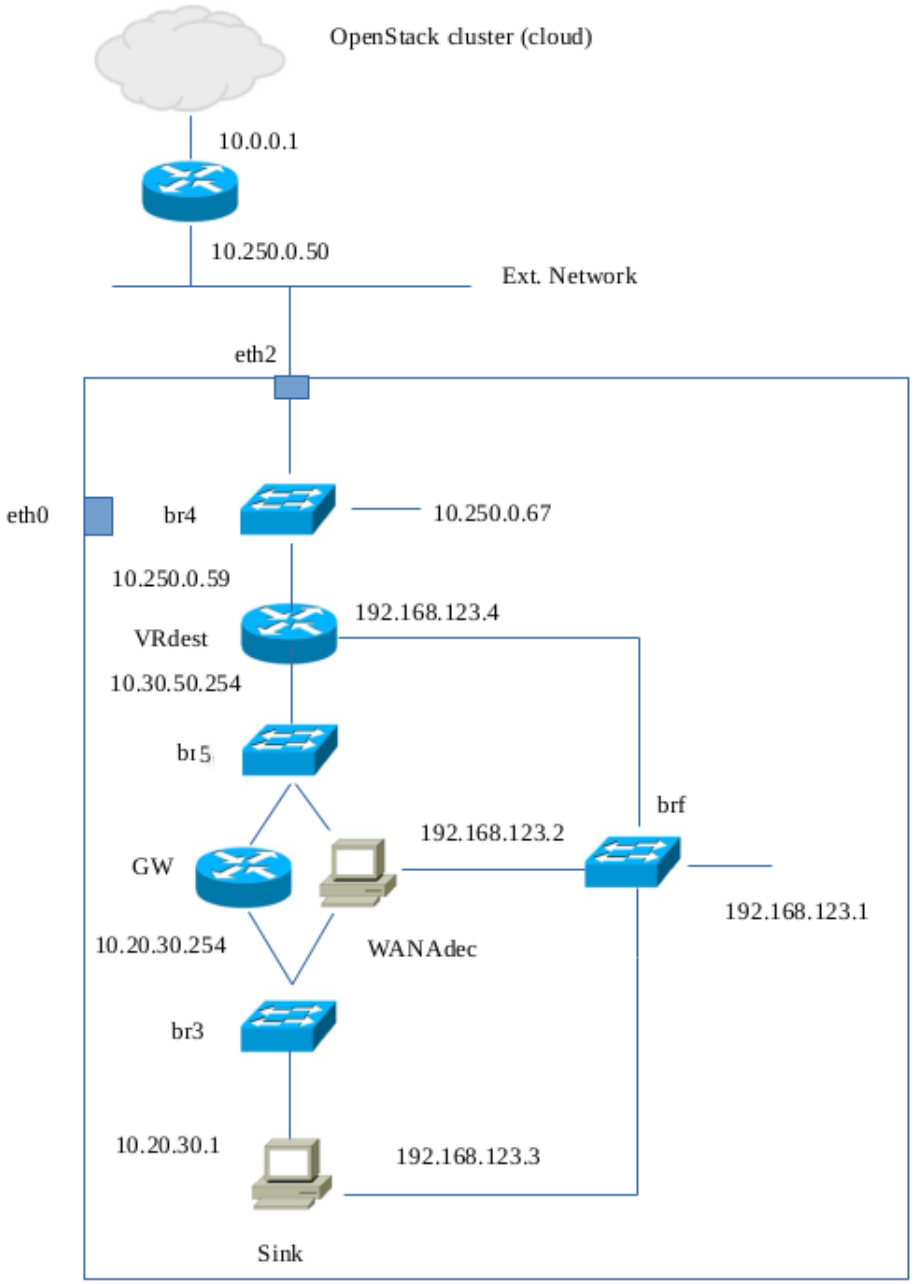

Figura A.2: Progetto di implementazione pratica L3 dell'host 67

A.2 Test sui domini di broadcast

Come già visto nel capitolo 5, un loop L2 di pacchetti ARP e ICMP può generare vari problemi, tra cui la perdita di connettività per gli utenti connessi via SSH; allo stesso modo è stato fatto vedere come questo possa essere evitato, mediante regole OpenFlow adeguate, scritte per gestire un'eventualità di questo caso.

Se si applica lo stesso discorso relativamente al nodo ricevente, nel caso in cui non venisse inserito il VRdest, si può vedere che il dominio di broadcast [1] non viene spezzato fino al Virtual Router di OpenStack; quindi, nel caso in cui le due interfacce del WANAdec venissero accidentalmente connesse tra di loro senza le adeguate regole OpenFlow per gestire il loop, quest'ultimo avrebbe conseguenze in tutta la rete compresa la parte relativa a OpenStack.

Al fine di valutare questo fenomeno con le precauzioni del caso, si è fatto connettere un computer con IP fisso via SSH al cluster OpenStack mediante una rete specifica alla porta 61, mentre un altro è stato fatto connettere con modalità SSH standard alla porta 63, tramite il nat.

Sono state poi aggiunte le due interfacce del WANAdec al bridge e, mediante catture, si è visto il proliferare dei pacchetti ARP e ICMP in loop; dopo una trentina di secondi infatti si è verificata la perdita di connettività all'interno del compute node di OpenStack nel secondo computer, mentre il primo valutava l'effettivo rovinamento della rete stessa.

Eliminando poi una delle due interfacce il loop è ovviamente cessato e, dopo circa 60 secondi la connettività sul primo computer era stata ristabilita senza necessità di manovre aggiuntive.

Come previsto dunque dallo studio teorico il dominio di broadcast così esteso definisce un dominio di fallimento altrettanto esteso [40], portando in caso di guasto o di funzionamento non corretto delle regole OpenFlow ad una perdita di connettività globale al sistema in studio comprendendo anche la struttura di rete di OpenStack.

Per questo motivo, nella sezione precedente, è stato aggiunto un Virtual Router secondario in entrambe le topologie: la sua funzione è infatti quella di spezzare il dominio di broadcast efficacemente e cercare di limitare i danni in caso di guasto, proteggendo il cluster di OpenStack; questo comporterà quasi

[1] Insieme di computer in una rete che possono scambiare dati a livello datalink (L2), senza che questi debbano risalire fino al livello di rete (L3)

sicuramente un calo delle prestazioni end-to-end, ma risulta un compromesso necessario.

Appendice B

Controller

Il sistema necessita di un adeguato controller atto a modificare dinamicamente le regole OpenFlow installate negli switch br-int e br3; la scelta è ricaduta sul controller POX scritto in Python per mantenere una sorta di dualità con il caso di Mininet. Viene da sè che, ad esempio, il codice che gestisce il livello di rete L2 in Mininet non è poi del tutto simile a quello poi realmente implementato, così come i codici L2 e L3 sono profondamente diversi tra di loro.

Nonostante tutto si è valutato come, sfruttando alcuni standard creati ad-hoc, si potrebbe mantenere lo stesso codice per entrambe le casistiche, arrivando ad un risultato molto interessante: il codice scritto per la rete emulata infatti sarebbe equivalente a quello scritto per la sperimentazione fisica, portando dunque ad una completa compatibilità tra le due.

B.1 Codice del controller relativo a rete L2

```
#!/usr/bin/python

'''
Controller for the Openstack cluster - L2 scenario dynamic service
    chaining

'''

#Packages
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
import pox.lib.packet as pkt
from pox.openflow.of_json import *
from pox.lib.recoco import Timer
```

```
import string
import os
import json
import time
import datetime
import subprocess
#Allow logging messages to the console
log = core.getLogger()

switch_dpid_name = {} #Keep track of each switch by mapping dpid to
    name


#OpenFlow ports in the br-int switch
resuser = 4
bususer = 3
ponte1_1 = 10 #ITC
ponte1_2 = 11
ponte2_1 = 5 #AVANA
ponte2_2 = 6
ponte3_1 = 5 #AVANA DECOMPRESSOR
ponte3_2 = 6
sink = 7
outport_dest = 8
outport = 47
bcast = "ff:ff:ff:ff:ff:ff"
ip_mgmt_ponte_1 = "192.168.122.240"
ip_mgmt_ponte_2 = "192.168.122.51"
ip_mgmt_resuser = "192.168.122.233"
ip_resuser = "10.0.0.4"
ip_bususer = "10.0.0.2"
ip_router = "10.0.0.1"
ip_nat = "10.250.0.50"
ip_sink = "10.20.30.1"
internal_vid = 57
external_vid = 1000
dpiExePath = '/root/nDPI/example/ndpiReader'
dpiCapPath = '/root/nDPI/example/filecattura1.json'
a = 0
b = 0
analizzatore = 8
DPIcredentials='ubuntu@192.168.122.108'

def startDPI():
    os.system("ssh %s 'sudo nohup  %s -i eth0 -v 2 -j %s  > foo.out 2>
         foo.err < /dev/null & '" % (DPIcredentials, dpiExePath,
        dpiCapPath))
    log.debug("[CONTROLLER - TIMER (%s)] DPI started!\n\n\n\n",
        datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-%
        d %H:%M:%S'))

def stopDPI():
    os.system("ssh %s 'sudo kill -2 $(pgrep ndpi) '" % DPIcredentials)
    log.debug("[CONTROLLER - TIMER (%s)] DPI stopped!", datetime.
        datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%
```

```
        S'))

def cleanDPI():
    os.system("ssh %s 'sudo killall  ndpi '" % DPIcredentials)

def obtain_DPI_output():
    return subprocess.check_output(['ssh', "%s" % DPIcredentials , '
        sudo cat %s' % dpiCapPath ])

def connectionForBridge(bridge):
    myConnection = None
    for currentDPID in core.openflow.connections.keys():
        if switch_dpid_name[dpid_to_str(currentDPID)] == bridge:
            myConnection= core.openflow.connections[currentDPID]
    return myConnection

#The POX controller controls br-int (in hc01) and br3 (in h67)
class MyController(object):

    def __init__(self, connection):
        self.connection = connection
        connection.addListeners(self)

        log.debug("[CONTROLLER - (%s)] Installing ARP STORM AVOIDANCE,
             ICMP/ARP rules", datetime.datetime.fromtimestamp(time.
            time()).strftime('%Y-%m-%d %H:%M:%S'))

        if self.connection == connectionForBridge("br-int"):

            #br-int (hc01)
            #ARP storm avoidance
            for p in [ponte1_1, ponte1_2, ponte2_1, ponte2_2]:
                msg = of.ofp_flow_mod(priority=32050, match=of.
                    ofp_match(in_port = p , dl_type = pkt.ethernet.
                    ARP_TYPE , dl_dst = pkt.ETHERNET.ETHER_BROADCAST )
                    )
                connectionForBridge("br-int").send(msg)

            #ARP traffic
            msg = of.ofp_flow_mod( action=of.ofp_action_output(port=of
                .OFPP_NORMAL), priority=32000, match=of.ofp_match(
                dl_type=pkt.ethernet.ARP_TYPE ) )
            connectionForBridge("br-int").send(msg)

            #Special case for ARP frames coming from outport
            msg = of.ofp_flow_mod( priority=32001, match=of.ofp_match(
                 in_port=outport, dl_type=pkt.ethernet.ARP_TYPE ) )
            msg.actions.append(of.ofp_action_vlan_vid(vlan_vid =
                internal_vid))
            msg.actions.append(of.ofp_action_output(port = of.
                OFPP_NORMAL ))
            connectionForBridge("br-int").send(msg)

            #ICMP traffic
            msg = of.ofp_flow_mod( action=of.ofp_action_output(port=of
                .OFPP_NORMAL), priority=32000, match=of.ofp_match(
```

```
                dl_type=pkt.ethernet.IP_TYPE, nw_proto=1 ) )
            connectionForBridge("br-int").send(msg)

            #Special case for ICMP packets coming from outport
            msg = of.ofp_flow_mod( priority=32001, match=of.ofp_match(
                in_port = outport, dl_type=pkt.ethernet.IP_TYPE,
                nw_proto = 1) )
            msg.actions.append(of.ofp_action_vlan_vid(vlan_vid =
                internal_vid))
            msg.actions.append(of.ofp_action_output(port = of.
                OFPP_NORMAL ))
            connectionForBridge("br-int").send(msg)

        if self.connection == connectionForBridge("br3"):

            #br3 (h67)
            #ARP storm avoidance
            for p in [ponte3_1, ponte3_2]:
                msg = of.ofp_flow_mod(priority=32050, match=of.
                    ofp_match(in_port = p , dl_type = pkt.ethernet.
                    ARP_TYPE , dl_dst = pkt.ETHERNET.ETHER_BROADCAST )
                    )
                connectionForBridge("br3").send(msg)

            #ARP traffic
            msg = of.ofp_flow_mod( action=of.ofp_action_output(port=of
                .OFPP_NORMAL), priority=32000, match=of.ofp_match(
                dl_type=pkt.ethernet.ARP_TYPE ) )
            connectionForBridge("br3").send(msg)

            #ICMP traffic
            msg = of.ofp_flow_mod( action=of.ofp_action_output(port=of
                .OFPP_NORMAL), priority=32000, match=of.ofp_match(
                dl_type=pkt.ethernet.IP_TYPE, nw_proto=1 ) )
            connectionForBridge("br3").send(msg)

    #Convert an HEX to DEC
    def convertToInt(self, hexval):
        dpid_split = string.split(hexval, '-')
        concat_hex = ''
        for elem in dpid_split:
            concat_hex += elem
        intval = int(concat_hex, 16)
        return intval

    #function executed after DPI analysis
    def dpi_analysis_finished (self):
        # Step 1: stop DPI
        stopDPI()
        time.sleep(0.05)

        # Step 2: read json output file (DPI classification)
        json_data = obtain_DPI_output()
        log.debug(str(json_data))
        data = json.loads(json_data)
        list_of_flows = data["known.flows"]
```

```
# Cycle over the known flows captured by nDPI
for i in list_of_flows:
    if i['protocol'] == "TCP" or i['protocol'] == "UDP":
        host_a = i["host_a.name"]
        host_b = i["host_b.name"]
        port_a = i["host_a.port"]
        port_b = i["host_n.port"]
        str_host_a = str(host_a)
        str_host_b = str(host_b)
        str_port_a = str(port_a)
        str_port_b = str(port_b)
        log.debug("[CONTROLLER - TIMER (%s)] Host %s:%s is
            exchanging packets with Host %s:%s, via %s ",
            datetime.datetime.fromtimestamp(time.time()).
            strftime('%Y-%m-%d %H:%M:%S'), i["host_a.name"], i
            ["host_a.port"], i["host_b.name"], i["host_n.port
            "], i['protocol'])

        #BUSINESS USER
        #check if the flow is the bususer's flow
        #NOTE: when the resuser starts its traffic, the
            bususer OpenFlow rules are injected again
        if host_a == ip_bususer or host_b == ip_bususer:
            log.debug("[CONTROLLER ] DPI found bususer flow!
                Installing rules for steering to WANA")

            # STEERING RULES FOR UDP
            # UDP STEERING OUTGOING TRAFFIC ENTERING WANA
            msg = of.ofp_flow_mod(priority = 34501, match = of
                .ofp_match(in_port = bususer, dl_type = pkt.
                ethernet.IP_TYPE, nw_proto = pkt.ipv4.
                UDP_PROTOCOL, nw_src = ip_bususer ) )
            msg.actions.append(of.ofp_action_output(port =
                ponte2_1))
            connectionForBridge("br-int").send(msg)

            # traffic steering for outgoing traffic, exiting
                WANA
            msg = of.ofp_flow_mod(priority=34501, match=of.
                ofp_match(in_port = ponte2_2, dl_type = pkt.
                ethernet.IP_TYPE, nw_proto = 201, nw_src =
                ip_bususer) )
            msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
                = internal_vid))
            msg.actions.append(of.ofp_action_output(port =
                outport))
            connectionForBridge("br-int").send(msg)

            msg = of.ofp_flow_mod(priority=34501, match=of.
                ofp_match(in_port = ponte2_2, dl_type = pkt.
                ethernet.IP_TYPE, nw_proto = 202, nw_src =
                ip_bususer) )
            msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
                = internal_vid))
            msg.actions.append(of.ofp_action_output(port =
```

```
    outport))
connectionForBridge("br-int").send(msg)

msg = of.ofp_flow_mod(priority=34501, match=of.
    ofp_match(in_port = ponte2_2, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = 200, nw_src =
    ip_bususer) )
msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
     = internal_vid))
msg.actions.append(of.ofp_action_output(port =
    outport))
connectionForBridge("br-int").send(msg)

#traffic steering for incoming traffic, entering
    WANA
msg = of.ofp_flow_mod(priority=34501, match=of.
    ofp_match(in_port = outport, dl_vlan =
    external_vid, dl_type = pkt.ethernet.IP_TYPE,
    nw_proto = 201, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    ponte2_2))
connectionForBridge("br-int").send(msg)

msg = of.ofp_flow_mod(priority=34501, match=of.
    ofp_match(in_port = outport, dl_vlan =
    external_vid, dl_type = pkt.ethernet.IP_TYPE,
    nw_proto = 202, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    ponte2_2))
connectionForBridge("br-int").send(msg)

msg = of.ofp_flow_mod(priority=34501, match=of.
    ofp_match(in_port = outport, dl_vlan =
    external_vid, dl_type = pkt.ethernet.IP_TYPE,
    nw_proto = 200, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    ponte2_2))
connectionForBridge("br-int").send(msg)

#UDP STEERING INCOMING TRAFFIC EXITING WANA
msg = of.ofp_flow_mod( priority=34501, match=of.
    ofp_match(in_port = ponte2_1, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_output(port =
    bususer))
connectionForBridge("br-int").send(msg)

# STEERING RULES FOR TCP
# traffic steering for outgoing traffic entering
    WANA
msg = of.ofp_flow_mod(priority = 34501, match = of
    .ofp_match(in_port = bususer, dl_type = pkt.
```

```
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_src = ip_bususer ) )
    msg.actions.append(of.ofp_action_output(port =
        ponte2_1))
    connectionForBridge("br-int").send(msg)

    #traffic steering for incoming TCP traffic,
        exiting WANA
    msg = of.ofp_flow_mod( priority=34501, match=of.
        ofp_match(in_port = ponte2_1, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_dst = ip_bususer) )
    msg.actions.append(of.ofp_action_output(port =
        bususer))
    connectionForBridge("br-int").send(msg)

    #traffic steering for Traffic-Squezed packets br3
    #from br3 to sink
    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = outport_dest, dl_type =
        pkt.ethernet.IP_TYPE, nw_proto = 201, nw_dst =
         ip_sink) )
    msg.actions.append(of.ofp_action_output(port =
        ponte3_1))
    connectionForBridge("br3").send(msg)

    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = outport_dest, dl_type =
        pkt.ethernet.IP_TYPE, nw_proto = 202, nw_dst =
         ip_sink) )
    msg.actions.append(of.ofp_action_output(port =
        ponte3_1))
    connectionForBridge("br3").send(msg)

    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = outport_dest, dl_type =
        pkt.ethernet.IP_TYPE, nw_proto = 200, nw_dst =
         ip_sink) )
    msg.actions.append(of.ofp_action_output(port =
        ponte3_1))
    connectionForBridge("br3").send(msg)

    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = ponte3_2, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_dst = ip_sink) )
    msg.actions.append(of.ofp_action_output(port =
        sink))
    connectionForBridge("br3").send(msg)

    #UDP TRAFFIC GOING TO THE SINK
    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = ponte3_2, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        UDP_PROTOCOL, nw_dst = ip_sink) )
    msg.actions.append(of.ofp_action_output(port =
```

```
            sink))
        connectionForBridge("br3").send(msg)

        #from sink to br3
        msg = of.ofp_flow_mod(priority=34501, match=of.
            ofp_match(in_port = sink, dl_type = pkt.
            ethernet.IP_TYPE, nw_proto = pkt.ipv4.
            TCP_PROTOCOL, nw_dst = ip_nat) ) #IF WE WANT
            TO DISCRIMINATE FROM BUSUSER AND RESUSER, WE
            NEED TO SPECIFY TCP/UDP PORT
        msg.actions.append(of.ofp_action_output(port =
            ponte3_2))
        connectionForBridge("br3").send(msg)

        msg = of.ofp_flow_mod(priority=34501, match=of.
            ofp_match(in_port = ponte3_1, dl_type = pkt.
            ethernet.IP_TYPE, nw_proto = 201, nw_dst =
            ip_nat) )
        msg.actions.append(of.ofp_action_output(port =
            outport_dest))
        connectionForBridge("br3").send(msg)

        msg = of.ofp_flow_mod(priority=34501, match=of.
            ofp_match(in_port = ponte3_1, dl_type = pkt.
            ethernet.IP_TYPE, nw_proto = 200, nw_dst =
            ip_nat) )
        msg.actions.append(of.ofp_action_output(port =
            outport_dest))
        connectionForBridge("br3").send(msg)

        msg = of.ofp_flow_mod(priority=34501, match=of.
            ofp_match(in_port = ponte3_1, dl_type = pkt.
            ethernet.IP_TYPE, nw_proto = 202, nw_dst =
            ip_nat) )
        msg.actions.append(of.ofp_action_output(port =
            outport_dest))
        connectionForBridge("br3").send(msg)

        #UDP TRAFFIC GOING TO THE BUSUSER
        msg = of.ofp_flow_mod(priority=34501, match=of.
            ofp_match(in_port = sink, dl_type = pkt.
            ethernet.IP_TYPE, nw_proto = pkt.ipv4.
            UDP_PROTOCOL, nw_dst = ip_nat) )
        msg.actions.append(of.ofp_action_output(port =
            ponte3_2))
        connectionForBridge("br3").send(msg)

    #RESIDENTIAL USER
    #Check if the flow is resuser's flow
    if  host_a == ip_resuser or host_b == ip_resuser:
        log.debug("[CONTROLLER ] DPI found bususer flow!
            Installing rules for steering to TC")

        #TCP traffic steering for outgoing traffic
            entering TC
        msg = of.ofp_flow_mod(priority = 34501, match = of
```

```
        .ofp_match(in_port = resuser, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_src = ip_resuser ) )
    msg.actions.append(of.ofp_action_output(port =
        ponte1_1))
    connectionForBridge("br-int").send(msg)

    #TCP traffic steering for outgoing traffic,
        exiting TC
    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = ponte1_2, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_src = ip_resuser) )
    msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
         = internal_vid))
    msg.actions.append(of.ofp_action_output(port =
        outport))
    connectionForBridge("br-int").send(msg)

    #TCP traffic steering for incoming traffic,
        entering TC
    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = outport, dl_vlan =
        external_vid, dl_type = pkt.ethernet.IP_TYPE,
        nw_proto = pkt.ipv4.TCP_PROTOCOL, nw_dst =
        ip_resuser) )
    msg.actions.append(of.ofp_action_strip_vlan())
    msg.actions.append(of.ofp_action_output(port =
        ponte1_2))
    connectionForBridge("br-int").send(msg)

    #TCP traffic steering for incoming traffic,
        exiting TC
    msg = of.ofp_flow_mod( priority=34501, match=of.
        ofp_match(in_port = ponte1_1, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        TCP_PROTOCOL, nw_dst = ip_resuser) )
    msg.actions.append(of.ofp_action_output(port =
        resuser))
    connectionForBridge("br-int").send(msg)

    #UDP traffic steering for outgoing traffic
        entering TC
    msg = of.ofp_flow_mod(priority = 34501, match = of
        .ofp_match(in_port = resuser, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
        UDP_PROTOCOL, nw_src = ip_resuser ) )
    msg.actions.append(of.ofp_action_output(port =
        ponte1_1))
    connectionForBridge("br-int").send(msg)

    #UDP traffic steering for outgoing traffic,
        exiting TC
    msg = of.ofp_flow_mod(priority=34501, match=of.
        ofp_match(in_port = ponte1_2, dl_type = pkt.
        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
```

```
    UDP_PROTOCOL, nw_src = ip_resuser) )
msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
    = internal_vid))
msg.actions.append(of.ofp_action_output(port =
    outport))
connectionForBridge("br-int").send(msg)

#UDP traffic steering for incoming traffic,
    entering TC
msg = of.ofp_flow_mod(priority=34501, match=of.
    ofp_match(in_port = outport, dl_vlan =
    external_vid, dl_type = pkt.ethernet.IP_TYPE,
    nw_proto = pkt.ipv4.UDP_PROTOCOL, nw_dst =
    ip_resuser) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    ponte1_2))
connectionForBridge("br-int").send(msg)

#UDP traffic steering for incoming traffic,
    exiting TC
msg = of.ofp_flow_mod( priority=34501, match=of.
    ofp_match(in_port = ponte1_1, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_dst = ip_resuser) )
msg.actions.append(of.ofp_action_output(port =
    resuser))
connectionForBridge("br-int").send(msg)

#rules to manage the sink traffic (ip_nat)
# outgoing from br3 to sink TCP packets
msg = of.ofp_flow_mod(priority = 34501, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)

# outgoing from sink to br3 TCP packets
msg = of.ofp_flow_mod(priority = 34501, match = of
    .ofp_match( in_port = sink, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_sink))
msg.actions.append(of.ofp_action_output(port =
    outport_dest))
connectionForBridge("br3").send(msg)

# outgoing from br3 to sink UDP packets
msg = of.ofp_flow_mod(priority = 34501, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)
```

```
                    # outgoing from sink to br3 UDP packets
                    msg = of.ofp_flow_mod(priority = 34501, match = of
                        .ofp_match( in_port = sink, dl_type = pkt.
                        ethernet.IP_TYPE, nw_proto = pkt.ipv4.
                        UDP_PROTOCOL, nw_src = ip_sink))
                    msg.actions.append(of.ofp_action_output(port =
                        outport_dest))
                    connectionForBridge("br3").send(msg)

#Handler for PacketIn event
def _handle_PacketIn(self, event):
    packet = event.parsed
    src_dpid = dpid_to_str(event.dpid) #Read the dpid of the
        switch that fired the PacketIn event
    int_dpid = self.convertToInt(src_dpid)
    log.debug("[CONTROLLER - _HPI_ (%s)] Receiving packet from
        switch %s (DPID=%s) ...", datetime.datetime.fromtimestamp(
        time.time()).strftime('%Y-%m-%d %H:%M:%S'),
        switch_dpid_name[src_dpid], src_dpid)

    #Creating the match object
    match = of.ofp_match.from_packet(packet)

    if packet.type == packet.LLDP_TYPE or packet.type == packet.
        IPV6_TYPE:
        log.debug("[CONTROLLER - _HPI_ (%s)] LLDP=%s or IPv6=%s ;
            PACKET is = %s hex=%x", datetime.datetime.
            fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%S
            '), packet.LLDP_TYPE, packet.IPV6_TYPE, packet.type,
            packet.type)
        log.debug("[DROP-DEBUG] in_port= %s , paket_type= %x ,
            nw_proto= %s , nw_tos= %s", event.port, packet.type,
            match.nw_proto, match.nw_tos)

    elif match.dl_type == pkt.ethernet.IP_TYPE:
        dst_ip = match.nw_dst #dst_ip is an object of type IPAddr
        dst_ip_str = dst_ip.toStr() #Convert the IP destination
            address to a string

        log.debug("[CONTROLLER - _HPI_ (%s)] WHAT IS? TRAFFIC WITH
             NW_PROTO %s", datetime.datetime.fromtimestamp(time.
            time()).strftime('%Y-%m-%d %H:%M:%S'), match.nw_proto)

        if match.nw_proto == 6 or match.nw_proto == 17: # TCP or
            UDP protocol
            src_ip = match.nw_src #src_ip is an object of type
                IPAddr
            # Send traffic to DPI
            log.debug("[CONTROLLER - _HPI_ (%s)] INITIAL TCP/UDP
                TRAFFIC FROM %s and %s", datetime.datetime.
                fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%
                M:%S'), match.nw_proto, src_ip.toStr())
            # Tech. 1)

            if src_ip.toStr() == ip_bususer :
```

```
log.debug("[CONTROLLER - _HPI_] ... FROM bususer
    %s", src_ip.toStr())

global a
if a == 1 :
    log.debug('ERROR!!!! first event already
        happened (BU traffic generated)')
    return
a = 1

#rules to manage the bususer traffic mirroring
# outgoing bususer TCP packets
msg = of.ofp_flow_mod( hard_timeout=270, priority
    =34500, match=of.ofp_match( in_port = bususer,
     dl_type = pkt.ethernet.IP_TYPE, nw_proto =
    pkt.ipv4.TCP_PROTOCOL, nw_src = ip_bususer ) )
msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
    = internal_vid)) #attach the VLAN header
    because outport is a VLAN trunk port and we
    use OUTPUT
msg.actions.append(of.ofp_action_output(port =
    outport)) # Packet goes to VR and ...
msg.actions.append(of.ofp_action_strip_vlan()) #
    remove the VLAN tag before mirroring to the
    DPI
msg.actions.append(of.ofp_action_output(port =
    analizzatore)) # ... DPI
connectionForBridge("br-int").send(msg)

# incoming bususer TCP packets
msg = of.ofp_flow_mod(hard_timeout = 270, priority
    = 34500, match = of.ofp_match( in_port =
    outport, dl_vlan = external_vid, dl_type = pkt
    .ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    bususer))
msg.actions.append(of.ofp_action_output(port =
    analizzatore))
connectionForBridge("br-int").send(msg)

# outgoing bususer UDP packets
msg = of.ofp_flow_mod( hard_timeout=270, priority
    =34500, match=of.ofp_match( in_port = bususer,
     dl_type = pkt.ethernet.IP_TYPE, nw_proto =
    pkt.ipv4.UDP_PROTOCOL, nw_src = ip_bususer ) )
msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
    = internal_vid)) #attach the VLAN header
    because outport is a VLAN trunk port and we
    use OUTPUT
msg.actions.append(of.ofp_action_output(port =
    outport)) # Packet goes to VR and ...
msg.actions.append(of.ofp_action_strip_vlan()) #
    remove the VLAN tag before mirroring to the
    DPI
```

```
msg.actions.append(of.ofp_action_output(port =
    analizzatore)) # ... DPI
connectionForBridge("br-int").send(msg)

# incoming bususer UDP packets
msg = of.ofp_flow_mod(hard_timeout = 270, priority
    = 34500, match = of.ofp_match( in_port =
    outport, dl_vlan = external_vid, dl_type = pkt
    .ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_dst = ip_bususer) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    bususer))
msg.actions.append(of.ofp_action_output(port =
    analizzatore))
connectionForBridge("br-int").send(msg)

# rules to manage the sink traffic
# outgoing from br3 to sink TCP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)

# outgoing from sink to br3 TCP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = sink, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_sink))
msg.actions.append(of.ofp_action_output(port =
    outport_dest))
connectionForBridge("br3").send(msg)

# outgoing from br3 to sink UDP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)

# outgoing from sink to br3 UDP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = sink, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_src = ip_sink))
msg.actions.append(of.ofp_action_output(port =
    outport_dest))
connectionForBridge("br3").send(msg)

log.debug('[CONTROLLER - _HPI_] DPI mirroring
    rules for bususer installed! Starting DPI for
```

```
            bususer traffic')
        startDPI()
        t = Timer(60, self.dpi_analysis_finished,
            recurring=False)

    if src_ip.toStr() == ip_resuser :
        log.debug("[CONTROLLER - _HPI_] ...FROM resuser %s
            ", src_ip.toStr())

        # resuser traffic mirroring
        # outgoing resuser TCP packets
        msg = of.ofp_flow_mod( hard_timeout = 270,
            priority = 34500, match = of.ofp_match(in_port
             = resuser, dl_type = pkt.ethernet.IP_TYPE,
            nw_proto = pkt.ipv4.TCP_PROTOCOL, nw_src =
            ip_resuser ) )
        msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
             = internal_vid)) #attach the VLAN header
            because outport is a VLAN trunk port and we
            use OUTPUT
        msg.actions.append(of.ofp_action_output(port =
            outport)) # Packet goes to VR and ...
        msg.actions.append(of.ofp_action_strip_vlan()) #
            remove the VLAN tag before mirroring to the
            DPI
        msg.actions.append(of.ofp_action_output(port =
            analizzatore)) # ... DPI
        connectionForBridge("br-int").send(msg)

        # incoming resuser TCP packets
        msg = of.ofp_flow_mod( hard_timeout = 270,
            priority = 34500, match = of.ofp_match(
            in_port = outport, dl_vlan = external_vid,
            dl_type = pkt.ethernet.IP_TYPE, nw_proto = pkt
            .ipv4.TCP_PROTOCOL, nw_dst = ip_resuser) )
        msg.actions.append(of.ofp_action_strip_vlan())
        msg.actions.append(of.ofp_action_output(port =
            resuser))
        msg.actions.append(of.ofp_action_output(port =
            analizzatore))
        connectionForBridge("br-int").send(msg)

        # outgoing resuser UDP packets
        msg = of.ofp_flow_mod( hard_timeout = 270,
            priority = 34500, match = of.ofp_match(in_port
             = resuser, dl_type = pkt.ethernet.IP_TYPE,
            nw_proto = pkt.ipv4.UDP_PROTOCOL, nw_src =
            ip_resuser ) )
        msg.actions.append(of.ofp_action_vlan_vid(vlan_vid
             = internal_vid)) #attach the VLAN header
            because outport is a VLAN trunk port and we
            use OUTPUT
        msg.actions.append(of.ofp_action_output(port =
            outport)) # Packet goes to VR and ...
        msg.actions.append(of.ofp_action_strip_vlan()) #
            remove the VLAN tag before mirroring to the
```

```
    DPI
msg.actions.append(of.ofp_action_output(port =
    analizzatore)) # ... DPI
connectionForBridge("br-int").send(msg)

# incoming resuser UDP packets
msg = of.ofp_flow_mod( hard_timeout = 270,
    priority = 34500, match = of.ofp_match(
    in_port = outport, dl_vlan = external_vid,
    dl_type = pkt.ethernet.IP_TYPE, nw_proto = pkt
    .ipv4.UDP_PROTOCOL, nw_dst = ip_resuser) )
msg.actions.append(of.ofp_action_strip_vlan())
msg.actions.append(of.ofp_action_output(port =
    resuser))
msg.actions.append(of.ofp_action_output(port =
    analizzatore))
connectionForBridge("br-int").send(msg)

#rules to manage the sink traffic (ip_nat)
# outgoing from br3 to sink TCP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)

# outgoing from sink to br3 TCP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = sink, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    TCP_PROTOCOL, nw_src = ip_sink))
msg.actions.append(of.ofp_action_output(port =
    outport_dest))
connectionForBridge("br3").send(msg)

# outgoing from br3 to sink UDP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = outport_dest, dl_type =
    pkt.ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_src = ip_nat) )
msg.actions.append(of.ofp_action_output(port =
    sink))
connectionForBridge("br3").send(msg)

# outgoing from sink to br3 UDP packets
msg = of.ofp_flow_mod(priority = 34500, match = of
    .ofp_match( in_port = sink, dl_type = pkt.
    ethernet.IP_TYPE, nw_proto = pkt.ipv4.
    UDP_PROTOCOL, nw_src = ip_sink))
msg.actions.append(of.ofp_action_output(port =
    outport_dest))
connectionForBridge("br3").send(msg)

global b
```

```
                    if b == 1:
                        print 'ERROR!!!! second event already happened
                            (RU traffic generated)'
                        return
                    b = 1
                    log.debug('starting the DPI for RESUSER traffic')
                    startDPI()
                    t = Timer(60, self.dpi_analysis_finished,
                        recurring = False)

            else:
                log.debug("[CONTROLLER - _HPI_] PACKET UNKNOWN: DL_TYPE=%s
                    ", match.dl_type)

# ********** MAIN CLASS **********
class my_Controller(object):
    def __init__(self):
        core.openflow.addListeners(self)

    #Define EVENT HANDLER for each OVS switch
    def _handle_ConnectionUp(self, event):
        log.debug("[CONTROLLER - _HCU_ (%s)] Switch %s has come up.",
            datetime.datetime.fromtimestamp(time.time()).strftime('%Y
            -%m-%d %H:%M:%S'), dpid_to_str(event.dpid))
        feature = event.ofp
        nports = len(feature.ports)
        global switch_dpid_name
        switch_dpid_name[dpid_to_str(event.dpid)] = feature.ports[
            nports-1].name
        log.debug("[CONTROLLER - _HCU_] INFO: %s found", feature.ports
            [nports-1].name)
        #initiate connection handler
        MyController(event.connection)

def launch():
    core.registerNew(my_Controller)
    log.info("External controller is running!")
    cleanDPI()
```

B.2 Alcuni chiarimenti sul codice

B.2.1 Differenze fra StopDPI() e CleanDPI()

Come si può visualizzare nel codice e pienamente riscontrabile nel man del comando *kill*, le procedure seguite per fare una pulizia del DPI non sono le stesse utilizzate per stopparlo.

Infatti stopDPI(), manda un segnale di interruzione (SIGINT) al processo che si sta eseguendo: questo permette quindi al processo di nDPI di fermarsi e di salvare su file il contenuto dell'analisi, per poi fermarsi; diversamente, cleanDPI() comporta l'invio di un segnale di chiusura immediata

(TERM): questo porta dunque il DPI a doversi chiudere immediatamente, senza nemmeno poter salvare l'analisi fatta fino a quel momento sul file.

Da qui dunque la differenza tra stopDPI(), che dà il segnale al processo nDPI di spegnersi concludendo le mansioni a cui è destinato (simile al comando CTRL+C da CLI) e cleanDPI() che lo chiude direttamente.

B.2.2 Realizzazione di connectionForBridge(bridge)

Nel caso di studio sono presenti due switch da pilotare, br-int (interno al cluster di OpenStack) e br3; al fine di discriminare i due si sfrutta l'OpenFlow Nexus, che altro non è che un manager che permette di gestire le OpenFlow Connections.

In questo si ha switch_dpid_name che contiene la chiave e il nome associato ad essa, relativi allo swtich; dal nome (ad esempio br-int), si può ottenere il dpid (datapath id), a priori sconosciuto e diverso ad ogni avvio del controller; infine dal dpid si può recuperare la connection relativa.

Una volta compreso questo modo di ragionare, trovare la connection è dunque molto semplice: basterà infatti fare un ciclo ed associare a myConnection la connection trovata relativa al bridge di interesse.

Il connectionForBridge è di vitale importanza all'interno del controller che è stato realizzato, avendo il caso di studio due switch da pilotare.

B.2.3 OpenFlow: Differenze tra actions

Come già valutato nel Capitolo 3, basandosi sul protocollo OpenFlow le flow entries hanno una o più actions: l'ultima di queste dovrà essere output oppure NORMAL.

Quando si fa uso dell'azione *mod_vlan_vid*, ovvero di quell'azione che permette di moficare le informazioni sulla VLAN di un determinato pacchetto, questa avrà differenti conseguenze a seconda di quale azione finale viene eseguita.

Eseguendo l'action NORMAL infatti le conseguenze saranno quella del MAC learning delle porte da parte dello switch (vedere anche Capitolo 5) e la successiva scelta di dove inviarlo, a seconda della presenza nel suo filtering database di quegli specifici indirizzi. È importante inoltre notare come l'action NORMAL tenga conto anche della gestione delle porte (access o

trunk) su cui questo pacchetto dovrà essere inviato, facendo uscire il pacchetto rispettivamente senza o con il tag della VLAN.

L'action output invece non gestisce tutti questi parametri: non svolge meccanismi di MAC Learning Switch e non controlla su che tipo di porta farà uscire il pacchetto; il significato fisico di questa azione è infatti una modifica o la creazione del tag VLAN presente nell'header Ethernet.

Bibliografia

[1] S. Venugopal J. Broberg I. Brandic R. Buyya, C. S. Yeo. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616, 2009.

[2] Wikipedia. Cloud computing. Disponibile: http://en.wikipedia.org/wiki/Cloud_computing.

[3] Sun Microsystems. Sun cloud-computing technology. Technical report, Sun Microsystems, 2009. Internal Report.

[4] Wikipedia. Network functions virtualization. Disponibile: http://en.wikipedia.org/wiki/Network_Functions_Virtualization.

[5] F. Callegati W. Cerroni A. Manzalini, R. Minerva and A. Campi. Clouds of virtual machines in edge networks. *IEEE Communications Magazine*, 51(7):63–70, Luglio 2013.

[6] Wikipedia. Software-defined networking. Disponibile: http://en.wikipedia.org/wiki/Software-defined_networking.

[7] Open Networking Foundation. Openflow: Enabling innovation in your network. Disponibile: http://archive.openflow.org/.

[8] N. Shishodia. Software defined networks: Overview, concepts and examples. Technical report, ECODE Networks, Ottobre 2012.

[9] The European Telecommunications Standards Institute. Network functions virtualisation: An introduction, benefits, enablers, challenges & call for action. In ETSI White Paper, editor, *SDN and OpenFlow World Congress*, Ottobre 2012. Darmstadt-Germany.

[10] C. Prakash R. Grandl J. Khalid S. Das A. Gember-Jacobson, R. Viswanathan and A. Akella. Opennf: Enabling innovation in network function control. In ACM, editor, *SIGCOMM 2014*, pages 163–174, Agosto 2014. Chicago, IL, USA.

[11] D.H.C. Du L. Zhang H. Guan J. Chen Y. Zhao X. Ge, Y. Liu and X. Hu. Openanfv: Accelerating network function virtualization with a consolidated framework in openstack. In ACM, editor, *ACM SIGCOMM 2014*, pages 163–174, Agosto 2014. Chicago, IL, USA.

[12] The European Telecommunications Standards Institute. *NFV Performance & Portability Best Practises*, Giugno 2014. Group Specification, v 1.1.1.

[13] V. Olteanu M. Honda R. Bifulco F. Huici J. Martins, M. A. C. Raiciu. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 459–473. USENIX Association, Aprile 2014.

[14] T. Wood J. Hwang, K. K. Ramakrishnan. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 445–458. USENIX Association, Aprile 2014.

[15] Open Networking Foundation. Software-defined networking: The new norm for networks. Technical report, ONF White Paper, Aprile 2012.

[16] Plexxi. Sdn market to reach $35b by 2018. Disponibile: http://www.plexxi.com/2013/04/sdn-market-to-reach-35b-by-2018/.

[17] Greg Ferro. Sdn is business, openflow is technology. Disponibile: http://www.networkcomputing.com/networking/sdn-is-business-openflow-is-technology/a/d-id/1233978?

[18] M. Ulema F. Paganelli and B. Martini. Context-aware service composition and delivery in ngsons over sdn. *IEEE Communications Magazine*, 52(8):97–105, Luglio 2014.

[19] Pate Prayson. Nfv and sdn: What's the difference? Disponibile: https://www.sdxcentral.com/articles/contributed/nfv-and-sdn-whats-the-difference/2013/03/.

[20] Hari Balakrishnan Guru Parulkar Larry Peterson Jennifer Rexford Scott Shenker Jonathan Turner Nick McKeown, Tom Anderson. Openflow: Enabling innovation in campus network. Technical report, White Paper, Marzo 2008.

[21] Toshal Dudhwala. The industry status of sdn/openflow: An ixia webinar. Disponibile: http://blogs.ixiacom.com/ixia-blog/the-industry-status-of-sdn-openflow-webinar/.

[22] Chris Swan. Software defined networking and network function virtualization. Disponibile: http://insights.wired.com/profiles/blogs/software-defined-networking-and-network-function-virtualization.

[23] OpenStack Foundation. Openstack: Open source cloud computing software. Disponibile: http://www.openstack.org/.

[24] Google. Google trends. Disponibile: http://www.google.com/trends/.

[25] G. Santandrea. Openstack: Network internals. Disponibile: http://www.slideshare.net/lilliput12/openstack-networking-internals-first-part.

[26] G. Santandrea. Show my network state project website, 2014. Disponibile: https://sites.google.com/site/showmynetworkstate.

[27] Wikipedia. Openvswitch. Disponibile: http://en.wikipedia.org/wiki/Open_vSwitch.

[28] The Linux Foundation. Linux bridge, 2009. Disponibile: http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.

[29] C. Contoli G. Santandrea F. Callegati, W. Cerroni. Performance of network virtualization in cloud computing infrastructures: The openstack case. In *Proc. of 3rd IEEE International Conference on Cloud Networking (CloudNet 2014)*, Ottobre 2014. Luxemburg.

[30] C. Contoli G. Santandrea F. Callegati, W. Cerroni. Performance of multi-tenant virtual networks in openstack-based cloud infrastructures. In *Proc. of 2nd IEEE Workshop on Cloud Computing Systems, Networks, and*

Applications (CCSNA 2014), in conjunction with IEEE Globecom 2014, Dicembre 2014. Austin, TX.

[31] C. Contoli G. Santandrea F. Callegati, W. Cerroni. Dynamic chaining of virtual network functions in cloud-based edge networks. In *Proc. of the 1st IEEE Conference on Network Softwarization (NetSoft 2015)*, 2015. Londra, UK.

[32] nDPI. ndpi: Open and extensible lgplv3 deep packet inspection library. Disponibile: http://www.ntop.org/products/ndpi/.

[33] Kiran Kankipati. Trafficsqueezer - open-source wan optimization. Disponibile: http://www.trafficsqueezer.org/.

[34] Mininet. Mininet vm images. Disponibile: https://github.com/mininet/mininet/wiki/Mininet-VM-Images.

[35] Nmap. Nmap: Free security scanner. Disponibile: http://nmap.org/.

[36] Mozilla Inc. Firefox. Disponibile: https://www.mozilla.org/it/firefox/new/.

[37] Lynx. Lynx browser. Disponibile: http://lynx.browser.org/.

[38] Wikipedia. Smart m3. Disponibile: http://en.wikipedia.org/wiki/Smart-M3.

[39] NGSON Working Group of the IEEE Standards Committee. Next generation service overlay network. Technical report, IEEE White Paper, Maggio 2008.

[40] Cisco. Switching infrastructure. Disponibile: http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Security/Baseline_Security/securebasebook/sec_chap7.html.