ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA – SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA DEI SISTEMI ELETTRONICI PER LO SVILUPPO SOSTENIBILE

PIANO DI CONTROLLO SDN PER LA COMPOSIZIONE DINAMICA DI FUNZIONI DI RETE VIRTUALI

Elaborato in

Laboratorio di reti e programmazione dispositivi mobili

Relatore

Prof. Ing. Walter Cerroni

Correlatore

Ing. Chiara Contoli

Presentato da

Elia Leoni

Sessione III

Anno Accademico 2014/2015

Alla mia famiglia e ai miei amici,

per la loro costante presenza.

INDICE

SOMMARIO

Il seguente documento analizza i vantaggi introdotti nel mondo delle telecomunicazioni dai paradigmi di Software Defined Networking e Network Functions Virtualization: questi nuovi approcci permettono di creare reti programmabili e dinamiche, mantenendo alte le prestazioni.

L'obiettivo finale è quello di capire se tramite la generalizzazione del codice del controller SDN , il dispositivo programmabile che permette di gestire gli switch OpenFlow, e la virtualizzazione delle reti, si possa risolvere il problema dell'ossificazione della rete Internet: al giorno d'oggi le applicazioni di rete, le middle-boxes, comportano costi elevati e dipendenza dai fornitori, rendendo impossibile per i providers la sperimentazione di nuovi servizi di rete.

Nella parte finale della tesi viene quindi analizzato in dettaglio il codice del controller, grazie al quale sono stati realizzati interessanti test che dimostrano l'efficienza di tali paradigmi e la possibile svolta che essi possono dare al mondo delle telecomunicazioni.

CAPITOLO 1

Introduzione

Il mondo delle telecomunicazioni negli ultimi anni è stato rivoluzionato grazie all'introduzione del *cloud computing*. L'idea di base di tale paradigma è semplice, ma economicamente efficace: i servizi di rete, le risorse di calcolo e quelle di storage vengono offerte dal provider, che mette a disposizione dei server dedicati, al client, che utilizzerà queste macchine (fisiche o virtuali) per l'archiviazione, la memorizzazione o l'elaborazione di dati a distanza grazie alla velocità di connessione della banda larga. Una volta che l'utente avrà rilasciato la risorsa da esso utilizzata essa tornerà alla configurazione di default e rimessa a disposizione nel pool condiviso di risorse, in modo da poter essere utilizzata da un nuovo client [1].

L'introduzione di tale paradigma ha provocato un aumento esponenziale dei dispositivi presenti sulla rete, dei servizi richiesti e del traffico da essi generato. Questo ha reso necessaria un'evoluzione della rete, ormai resa statica e dipendente dai fornitori dei nodi intermedi, le *middle boxes*, in cui vengono implementate tutte le varie funzioni di rete.

Un *middle-box* tipicamente è un dispositivo hardware costruito su una piattaforma dedicata che supporta solo certe funzioni, ad esempio NAT, firewall … ecc. Tale apparato è quindi altamente prestante, ma molto costoso e per nulla flessibile ai cambiamenti. La presenza di questi dispositivi ha rallentato notevolmente lo sviluppo ed il progresso della rete, rendendo quindi necessaria la ricerca di nuovi metodi atti a rivoluzionare la situazione statica in cui il mercato delle reti di telecomunicazioni risulta essere.

Due grandi innovazioni che contribuiscono a questo cambiamento sono il *Software -Defined Networking* ed il *Network Functions Virtualization*.

Il *NFV* [2] è un paradigma che si basa sulla virtualizzazione delle middle-boxes su hardware generico, con la possibilità quindi di poterle modificare o riprogrammare in modo dinamico a seconda delle funzioni che si desidera implementare. Questo concetto di architettura prevede che le funzionalità di rete,

siano realizzate virtualmente tramite applicazioni software instanziabili direttamente sui server, creando più macchine virtuali all'interno della macchina fisica [3].

Oltre al NFV, come già anticipato si utilizza il *Software-defined networking* [4].Questo è un nuovo approccio al computer networking che permette agli amministratori di rete di separare il piano di controllo del traffico (gestione degli algoritmi di routing) dal piano dati (instradamento dei pacchetti), direttamente via software; si ha così la possibilità di gestire in modo dinamico la rete tramite un protocollo dedicato, il protocollo *Openflow*.

Tale documento si è posto come obiettivo quello di generalizzare ed ottimizzare il codice del controller SDN, al fine di testarlo su di una topologia di rete virtuale realizzata su cluster *OpenStack*, il più vicina possibile ad un caso reale in cui un operatore di rete dà la possibilità a tre utenti diversi, che hanno sottoscritto contratti diversi, di accedere alla rete Internet, utilizzando i meccanismi di NFV e SDN.

Nel prossimo capitolo vengono illustrate le principali caratteristiche dei due paradigmi e le loro implementazioni pratiche: OpenStack per il NFV e OpenFlow per il SDN.

Nel terzo capitolo viene spiegato nel dettaglio il funzionamento del controller Ryu, tramite l'utilizzo di esempi di configurazione di base per meglio comprenderne il funzionamento.

Nel capitolo 4 viene esposto il concetto di *Dynamic Service Chaining*, assieme alla macchina a stati SDN che rappresenta il funzionamento pratico del paradigma. Alla fine del capitolo viene presentata la topologia di rete di riferimento per i test svolti con il controller Ryu e viene descritto il funzionamento teorico della catena di servizi dinamica.

Infine nel quinto capitolo viene mostrato il codice del controller Ryu generalizzato, ne vengono spiegate le parti salienti, vengono mostrati i test svolti e i risultati ottenuti.

CAPITOLO 2

Virtualizzazione delle funzioni di rete e reti definite via software

Come già anticipato nel capitolo 1, i servizi di rete saranno forniti da infrastrutture costruite secondo il protocollo cloud, in modo tale che le funzioni possano essere eseguite su hardware generico, così da poter lentamente sostituire la maggior parte delle middle-boxes, causa prima della staticità della rete.

Risulta quindi ovvia la necessità di studiare nuovi paradigmi di rete da affiancare al cloud, per aumentare la dinamicità della rete stessa, e dare la possibilità ai fornitori di servizi di poter gestire diverse classi di utenza, dalle più prioritarie alle best effort.

Inoltre la possibilità di avere delle interfacce *Open* nei dispositivi darebbe ai ricercatori la possibilità di testare nuove idee in maniera molto più semplice e veloce, direttamente sulla rete, senza dover per forza utilizzare dei programmi di simulazione, che risultano utili, ma che non possono emulare del tutto un caso reale.

I concetti di NFV e SDN risultano quindi innovativi e rappresentano una possibilità di evoluzione per le reti di telecomunicazioni.

Nei prossimi paragrafi vengono quindi analizzati in modo esaustivo, per far meglio comprendere al lettore l'importanza di tali paradigmi.

2.1 Network Function Virtualization (NFV)

L'ossificazione della rete rappresenta uno dei principali problemi di internet, la presenza di una vasta gamma di middle-boxes dipendenti dai venditori non consente di implementare rapidamente nuovi servizi di rete senza costi indifferenti e senza difficoltà nell'integrazione con altri componenti della stessa [5].

Per risolvere questo problema, nell'ottobre del 2012 al "SDN and OpenFlow World Congress" è stato presentato un white paper da un gruppo di rappresentati appartenenti all'European Telecommunications Standards Institute (ETSI), basato sul NFV [6].

Il nuovo approccio proposto dagli autori era quello di sostituire le middle-boxes proprietarie, con dispositivi general purpose poco costosi e riprogrammabili, sfruttando i meccanismi della virtualizzazione. Come si può vedere dalla figura 1, l'idea è quella di sfruttare il concetto di *virtual functions* all'interno di server dedicati.

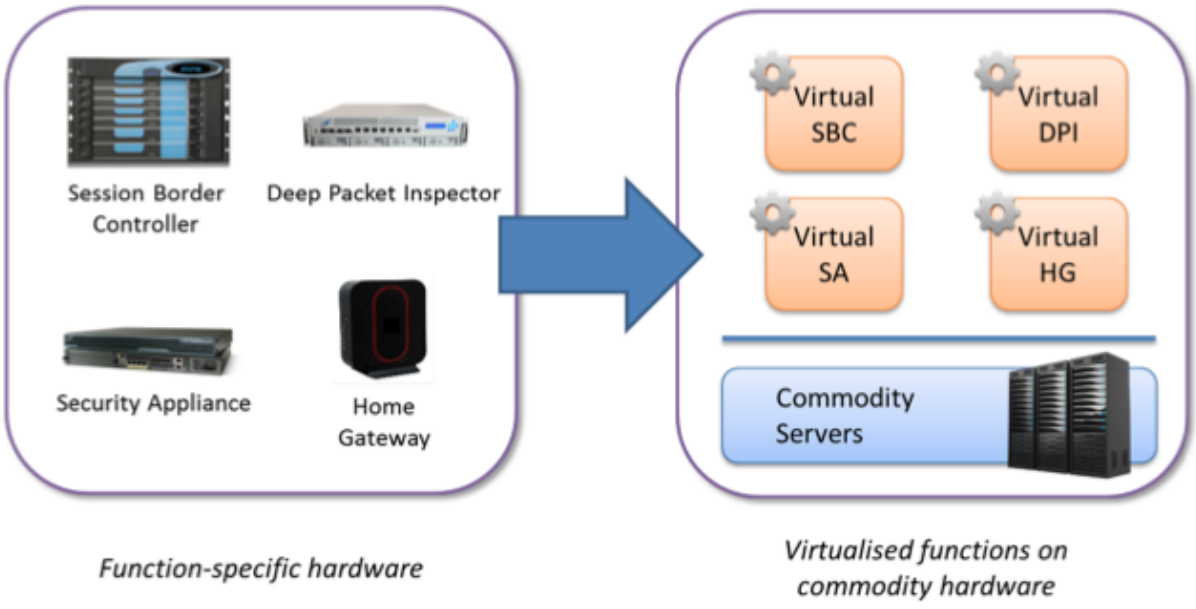

FIGURA 1:NETWORK FUNCTION VIRTUALIZATION [7].

L’utilizzo di questo paradigma comporterebbe numerosi vantaggi per gli operatori e per la rete:

- Una notevole diminuizione dei costi, dovuta all’utilizzo di hardware general purpose poco costoso programmato ad hoc, in grado di offrire nuovi servizi a costi minori per l’utente;

- Miglioramento dell’efficienza dei servizi, grazie all’utilizzo del software che rende la rete più semplice da gestire e più veloce da riparare in caso di malfunzionamento;

- Possibilità di smarcarsi dai venditori, mettendo fine alla dipendenza economica dell’operatore dal proprietario delle middle-boxes;

- Supporto del *multi-tenancy:* l’applicazione software è progettata per partizionare virtualmente i suoi dati e la sua configurazione, in modo che ogni suo client lavori con un’istanza virtuale personalizzata;

- L’utilizzo del software darebbe la possibilità di testare nuovi algoritmi sulla rete stessa, rendendo più veloce il tempo necessario allo sviluppo e alla ricerca;

- Gli operatori potranno ideare nuove offerte e testarle facilmente sul sistema ottenendo così una concreta riduzione del *Time To Market* (tempo che intercorre tra l’ideazione alla commercializzazione di servizi e funzionalità);

Come si può notare, i vantaggi offerti dal NFV sono numerosi. Nel prossimo paragrafo verranno esposti i problemi introdotti da questo paradigma, e le possibili soluzioni proposte.

2.1.1 Sfide e problemi del NFV

Una volta valutati i vantaggi di questo paradigma è importante verificare quali sono i problemi che esso potrebbe introdurre, e le soluzioni proposte al momento.

VIRTUALIZZAZIONE DELLE FUNZIONI

Risulta evidente che, per quanto la virtualizzazione in questi anni si stia evolvendo sempre più, l'hardware costruito ad-hoc presenta performance maggiori, perciò finché ne l'hypervisor ne le virtual machines saranno ottimizzate per ottenere alte performance da server standard, questa risulterà la maggiore sfida per le funzioni virtuali [8].

Per risolvere questo problema sono state proposte diverse soluzioni: DPDK [9] ad esempio è un set di librerie e drivers ideato per processare in modo veloce i pacchetti, che può funzionare su numerosi processori. Questo set viene sfruttato da NetVM [10], una piattaforma software che consente di emulare diverse funzioni di rete in modo virtuale con elevate performance [8]. Infine vorrei citare Click OS[11], un Sistema Operativo ad alte performance ideato per simulare virtualmente le middle boxes.

PORTABILITÁ

Uno dei principi del NFV è quello di instanziare le funzioni virtuali su server generici in un ambiente multi-vendor, il framework NFV dovrà quindi poter essere caricato ed eseguito ovunque . La soluzione potrebbe essere quella di disaccoppiare le virtual machines dal sistema operativo grazie allo strato hypervisor, in modo tale che le VNFs (Virtual Network Functions) siano anch'esse OS-independent. In questo modo l'isolamento delle risorse sarebbe garantito finché le VNFs verranno istanziate su VMs indipendenti [8].

TRAFFIC STEERING

Il paradigma SDN offre numerosi vantaggi per quanto riguarda la dinamicità e l'agilità del traffic steering nelle reti, numerosi passi avanti sono stati fatti sulle middle-boxes hardware, ma su reti virtuali ancora c'è molto lavoro da fare. L'ottimizzazione della architettura NFV software-defined risulta un problema cruciale, la soluzione è lo studio di algoritmi atti a diminuire la complessità computazionale introdotta dal traffic steering nelle reti virtuali [8].

2.1.2 OpenStack

Negli ultimi anni, l'aumento delle applicazioni basate sul cloud-computing ha portato alla nascita di numerose piattaforme software open-source per la gestione dei server cloud in un contesto multi-tenant.

Tra queste piattaforme vi è OpenStack [12], un software ideato originariamente da Rackspace e NASA coadiuvate poi da oltre 200 aziende tra cui Cisco, Dell, Ericsson, Oracle, Yahoo, VMware, Intel, IBM, HP e NetApp [13].

OpenStack è quindi un software per la gestione di piattaforme cloud, cioè dei cluster di macchine fisiche, che ospitano alcuni server, che vengono offerti agli utenti come servizio, secondo il protocollo *Iaas* (*Infrastructure-as-a-Service*).

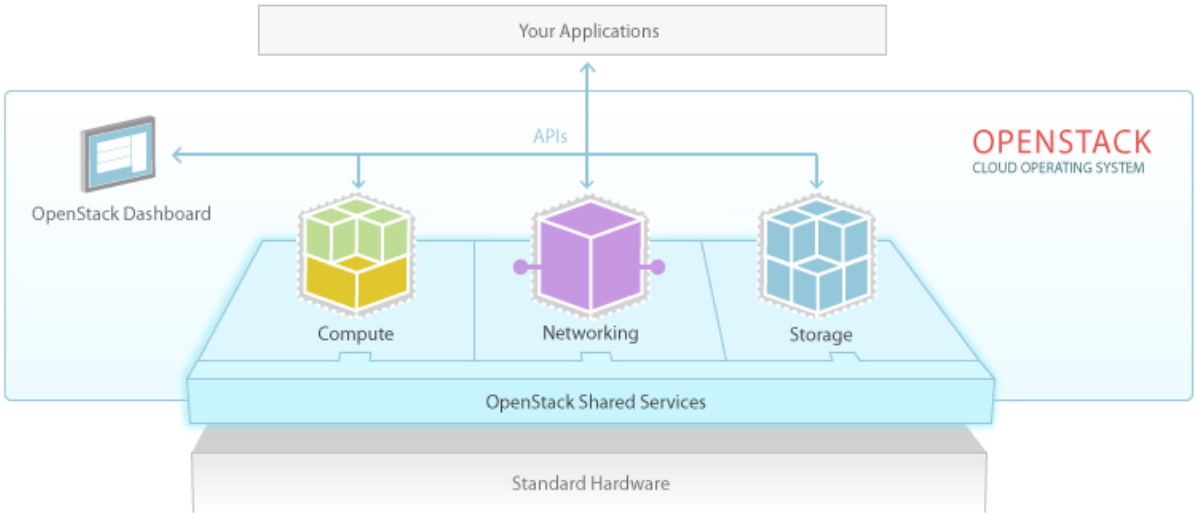

FIGURA 2: STRUTTURA DI OPENSTACK [12].

L'utente tramite l'utilizzo della *Dashboard* grafica (o in alternativa mediante terminale, grazie alle API provviste), ha la possibilità di creare sui server cloud

un'infrastruttura virtuale, composta da server e applicazioni di rete come NAT, Firewall, Deep Packet inspection, Wide Area Network accelerator, completamente virtuali grazie all'utilizzo delle virtual machine che verrano instanziate all'interno delle macchine fisiche.

Struttura e gestione multi-tenancy

Come si può notare dalla figura 2, un cluster OpenStack è composto da quattro elementi fondamentali:

- un *controller node* che si occupa della gestione della piattaforma;
- 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 [14].

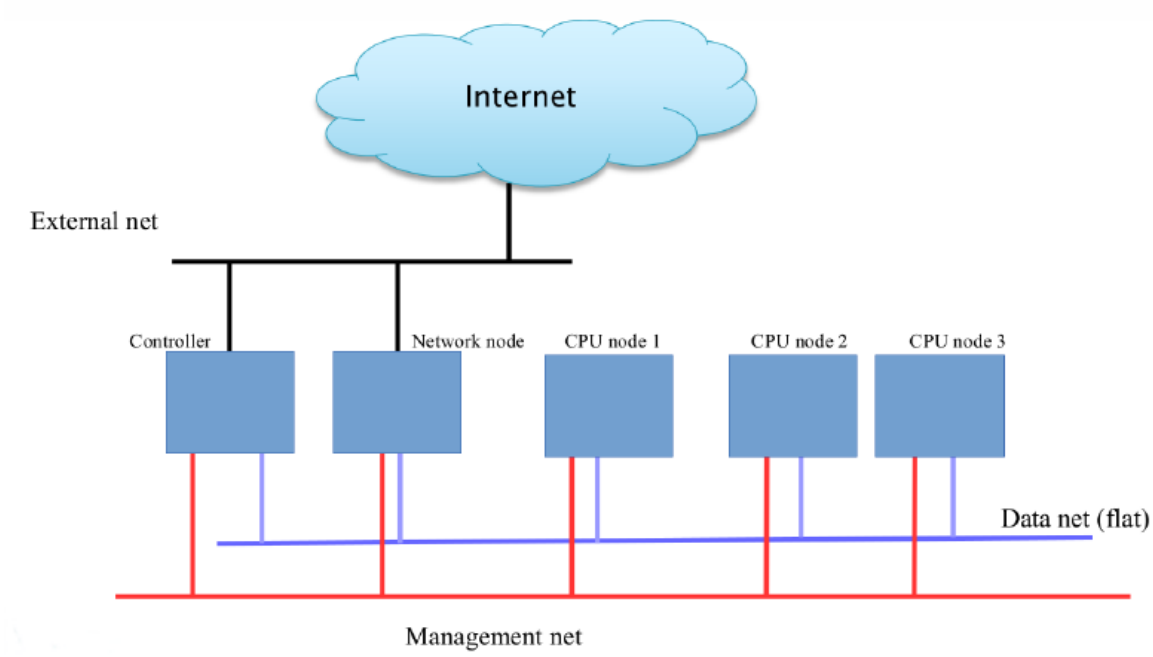

FIGURA 3:TOPOLOGIA DEL CLUSTER OPENSTACK [5].

I suddetti nodi sono collegati tra di loro tramite due reti, una di Management e una di Data (*Management net* e *Data net* rispettivamente). La rete dati viene

utilizzata per far comunicare tra di loro e con la rete esterna (*External net*) le virtual machines, poste all'interno dei CPU node, mentre la rete di management permette all'amministratore di accedere ai nodi del cluster ed è sfruttata per le comunicazioni di servizio.

Come già accennato in precedenza, OpenStack permette il multi-tenancy, cioè la possibilità di avere più utenti sullo stesso cluster. L'isolamento dei tenant avviene tramite l'utilizzo di *VLAN* (o tunnel GRE) e namespaces, i quali permettono di saparare ed isolare più domini di rete all'interno del singolo host. I namespaces garantiscono l'isolamento L3, in modo tale che interfacce relative a tenant diversi possano avere indirizzi IP sovrapposti.

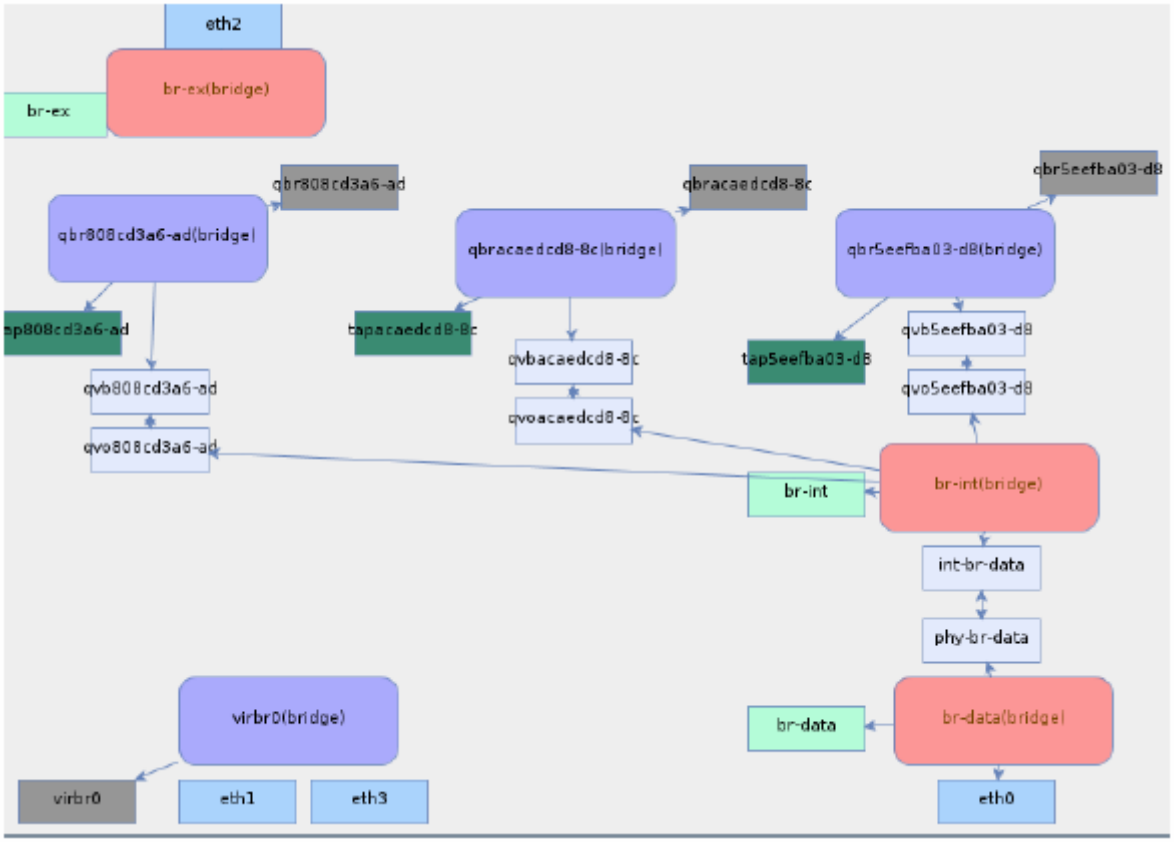

FIGURA 4 ESEMPIO COMPUTE NODE [5].

La figura 4 mostra la struttura interna di un possibile compute node con 3 macchine virtuali istanziate. L'infrastruttura si ottiene sfruttando il tool grafico Show My Network State [14].

Come si può notare il nodo contiene al suo interno diversi elementi :

- *br-int* e *br-data* sono bridge del tipo Open VSwitch [15], all'interno dei quali vengono implementate le regole OpenFlow e inserito o rimosso il VLAN ID per i flussi uscenti o entranti. In particolare all'uscita di br-data si utilizzano dei tunnel GRE;

- *Linux bridge* [16] (bridge in figura 4), è uno switch dentro al quale vengono implementati i *security group*, cioè un insieme di regole iptables per evitare lo spoofing;

- Interfacce di *Tap* connesse ai linux bridge. L'interfaccia di tap è un'entità software che permette di connettere il bridge livello-kernel ad una porta Ethernet della virtual machines che viene eseguita nello user-space [5];

- *Veth pairs*, interfacce Ethernet virtuali utilizzate a coppie che consentono di collegare tra di loro degli switch virtuali. Ognuna di queste coppie consiste in due interfacce che funzionano da endpoints di una pipe [5].

Componenti di OpenStack

Il funzionamento di OpenStack viene realizzato tramite diversi componenti, visibili in figura 5:

Horizon: è la web dashboard, e permette agli utilizzatori di gestire il cluster server in modo semplice;

Nova: rappresenta il controller per il cloud computing, è progettato per gestire e automatizzare il pool di risorse del computer;

Neutron: gestisce il networking, fornisce una API dedicata agli utilizzatori per definire la rete e le connessioni;

Keystone: gestisce le autorizzazioni, fornisce un elenco delgi utenti indicando a quali servizi OpenStack possono accedere;

Glance: rappresenta il gestore delle immagini, contiene una virtual machine "pre-confezionata", necessaria per il boot di nuove istanze;

Swift: è un sistema di storage ridondante, gli oggetti e files vengono memorizzati su diversi dischi distribuiti su diversi server nel centro di calcolo;

Cinder: storage persistente a livello di dispositivi a blocchi per il loro utilizzo da parte delle istanze di nova.

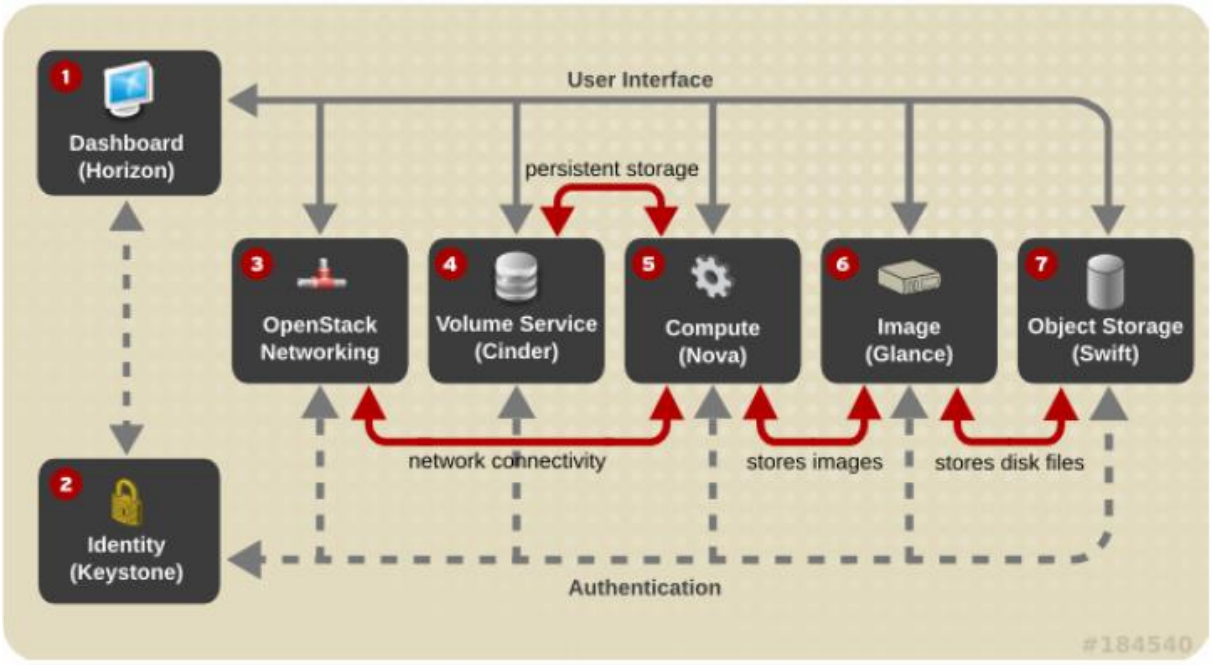

FIGURA 5: COMPONENTI DI OPENSTACK [12].

2.2 Software Defined Networking (SDN)

Il Software Defined Networking è un'architettura di rete altamente dinamica, conveniente e facilmente adattabile, promossa negli ultimi anni dalla Open Networking Foundation. L'idea di base di questo paradigma è quella di separare il piano di controllo dal piano dati, cioè disaccoppiare il sistema che prende le decisioni relative all'invio del traffico, dai sistemi che si occupano dell'inoltro dei pacchetti a destinazione.

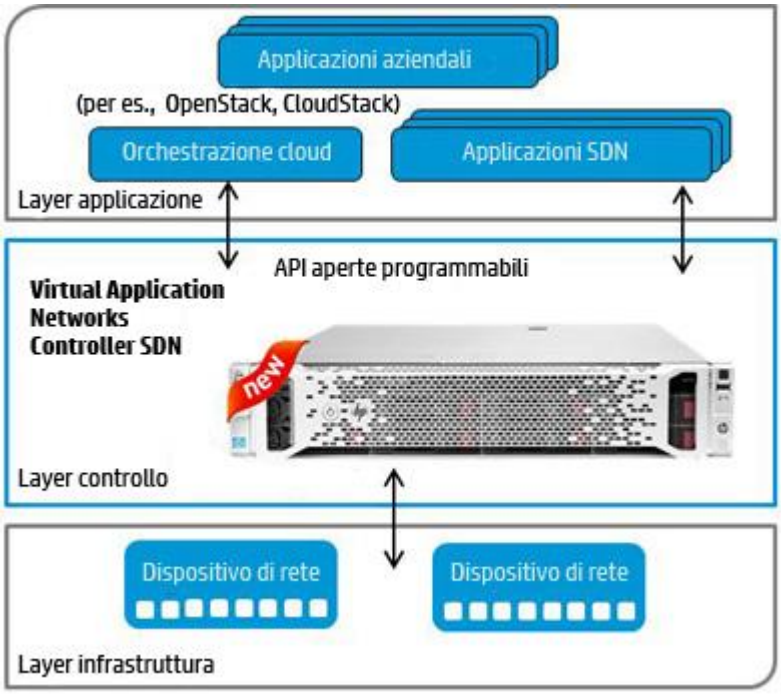

FIGURA 6: ARCHITETTURA SDN [17].

Come si può notare dalla figura 6, ciò che unisce le applicazioni all'infrastruttura di rete è il controller SDN, un dispositivo (fisico o virtuale) programmabile, il cui compito è quello di riempire la *flow table* degli switch SDN, cioè implementare il piano di controllo del sistema.

La flow table altro non è che una tabella di routing che trascende i livelli dell'OSI, quindi contiene tutte le informazioni relative ad ogni flusso di rete passante dallo switch SDN.

Questo tipo di approccio conferisce agli operatori di rete numerosi benefici:

- Rete direttamente programmabile proprio per il disaccoppiamento tra piano di controllo e di inoltro;
- L'astrazione del piano di controllo da quello di inoltro consente agli operatori di regolare direttamente il flusso di traffico a seconda delle esigenze degli utenti;
- Maggiore affidabilità grazie al controllo centralizzato della rete;
- La possibilità di implementare nuovi servizi senza la necessità di configurare i singoli dispositivi;
- Automazione e gestione della rete migliorata;
- Esperienza lato utente migliorata, le applicazioni utilizzano le informazioni sullo stato della rete centralizzata per adattare al meglio il comportamento della rete ai bisogni degli utenti.

Questo tipo di infrastruttura prevede quindi che lo strato di trasporto comunichi con lo strato di controllo tramite un certo tipo di protocollo. Nel Marzo del 2008 un gruppo di ricercatori americani ha proposto il protocollo *OpenFlow* [18], come interfaccia di comunicazione standard definita tra il controller e lo switch SDN. Nella prossima sezione verranno introdotti i concetti base del protocollo, più nello specifico della sua versione 1.3.

2.2.1 OpenFlow

Gestito dalla Open Networking Foundation, il protocollo prevede la presenza di uno switch o di un router e di un controllore, esterno ad esso, che abbia la possibilità di accedere al forwarding plane del dispositivo, modificare le regole di instradamento e determinare il percorso dei pacchetti attraverso una rete di switch in modo dinamico [19].

Lo switch in questione viene denominato *switch openflow*, ed esso consiste in una *flow table* nella quale viene implementato il table lookup, l'inoltro e la modifica dei percorsi che i pacchetti devono seguire, e in un *secure channel*, utilizzato per comunicare con il controller tramite il protocollo OpenFlow.

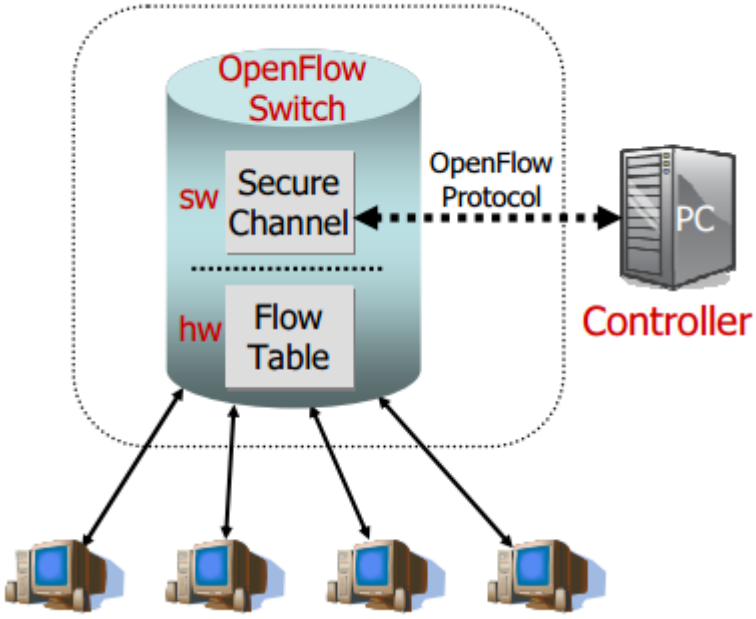

FIGURA 7: OPENFLOW SWITCH E CONTROLLER [20].

La flow table è composta da una lista di flow entries, cioè regole da applicare ai vari flussi di traffico. Se un pacchetto arriva in ingresso allo switch ed esiste una flow entry corrispondente, allora viene eseguita l'azione corrispondente, altrimenti (caso di *table miss*) esso viene inviato al controller tramite il canale sicuro.

Il controller quindi, a seconda di come è stato programmato, aggiungerà o modificherà una flow entry per il nuovo pacchetto.

Il protocollo prevede quindi che tra switch e controller avvenga uno scambio di varie tipologie di pacchetti. Come esempio vengono mostrati i pacchetti scambiati nella fase iniziale di configurazione.

Alla connessione il controller invia un *Hello Packet* allo switch che risponde immediatamente con un pacchetto analogo, in questo modo entrambi sanno che sono collegati.

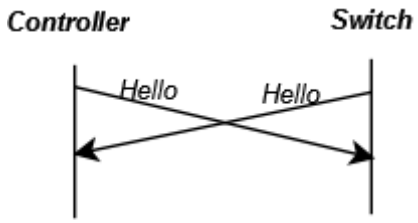

FIGURA 8:HELLO PACKET TRA SWITCH E CONTROLLER [21].

Una volta che tra lo switch e il controller è terminato l'handshake, quindi è stata stabilità una connessione a livello di trasporto (protocollo TCP), il controller manda un pacchetto *FeatureReq* al quale lo switch risponderà con un *FeatureRes*, contenente le sue caratteristiche (lista delle porte, velocità delle porte….eccetera.).

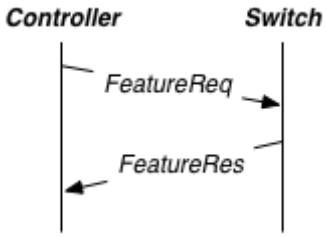

FIGURA 9: SCAMBIO DI PACCHETTI FEATUREREQ E FEATURERES [21].

La fase di configurazione termina con i messaggi da parte del controller che settano il limite per la frammentazione dei pacchetti.

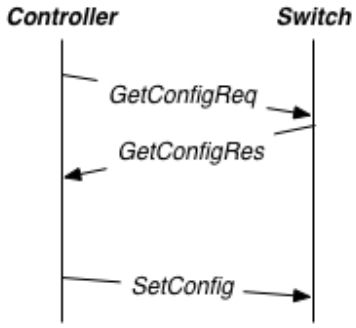

FIGURA 10:SEQUENZA DI MESSAGGI USATA PER IMPOSTARE LA FRAMMENTAZIONE DEI PACCHETTI [21].

Per mantenere attiva la connessione, il controller e lo switch si invieranno reciprocamente degli *Echo Packets*, necessari affinché entrambi sappiano che l'altro dispositivo è ancora attivo e pronto a comunicare.

Esistono altri tipi di pacchetti (aggiungere una flow entry, rimuovere una flow entry….eccetera.) che verranno analizzati nei prossimi capitoli.

Per concludere, esistono diversi tipi di controller, e la scelta è puramente libera. Al momento i più famosi sono POX(scritto in Python) e OpenDaylight (Java) entrambi finanziati da realtà del settore delle telecomunicazioni e con una grande comunità di utenti che li utilizza. Da citare oltre a questi ci sono Floodlight (Java), Beacon (Java) e Ryu (Python); in particolare, Floodlight e Ryu permettono l'integrazione automatica dell'ambiente OpenStack nel loro codice, tramite plug-in realizzati ad-hoc.

CAPITOLO 3

Il controller SDN Ryu

In questo capitolo viene introdotto il controller Ryu e ne vengono analizzati semplici esempi per meglio comprendere nei capitoli successivi il codice generalizzato. La scelta del controller non è stata casuale, Ryu dispone dei plug-in necessari all'integrazione con OpenStack, requisito necessario per i test che sono stati svolti durante la tesi.

3.1 Introduzione a Ryu

Ryu è un software SDN open-source, completamente scritto in Python che fornisce un insieme di componenti che permettono agli sviluppatori di creare nuove applicazioni per la gestione e il controllo della rete.

FIGURA 11:SIMBOLO DI RYU [22].

3.2 Architettura

La figura 12 mostra l'architettura del framework di Ryu e i vari apparati che con esso interagiscono:

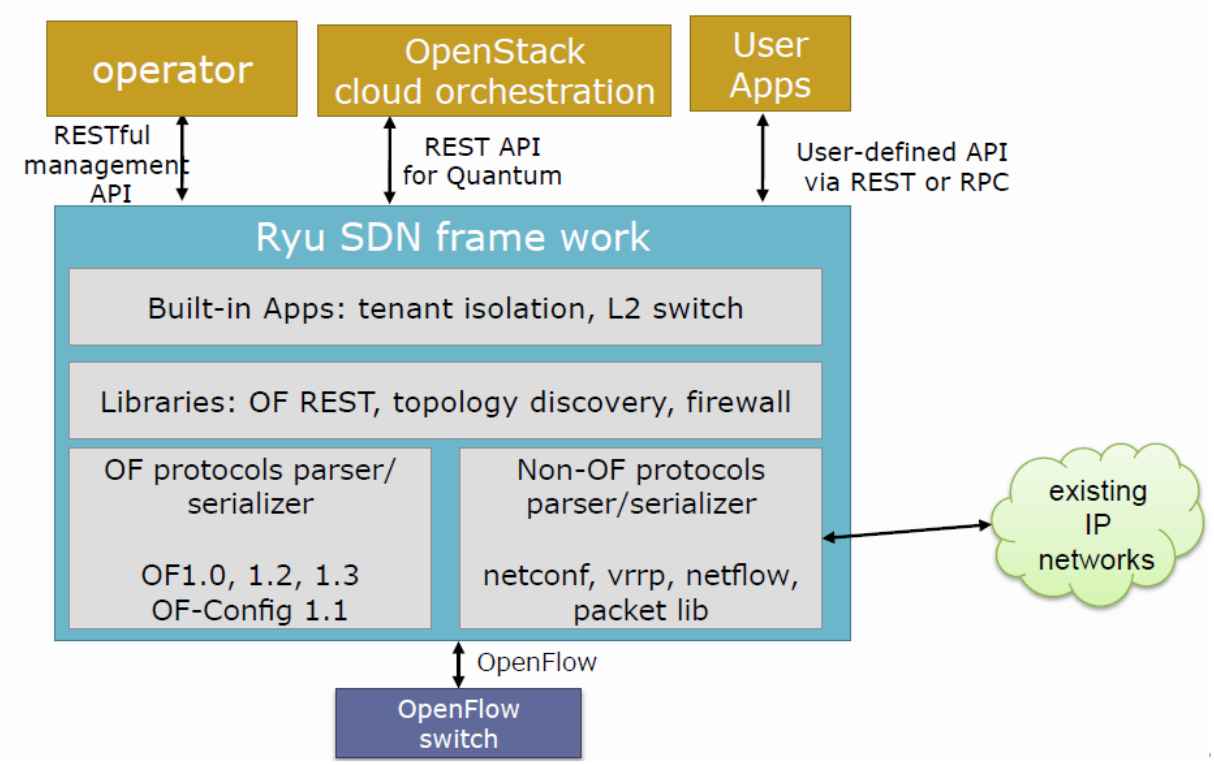

FIGURA 12:ARCHITETTURA RYU [23].

Come si può notare i componenti principali del framework sono:

- Built-in Apps, cioè le applicazioni di Ryu, scritte dagli sviluppatori per creare semplici o complesse funzioni di rete (hub, learning switch, firewall….eccetera.);

- Libreries, le librerie a cui Ryu fa riferimento;

- OF protocols parser, cioè la parte di architettura che riguarda il controller OpenFlow.

3.2.1 Protocollo OpenFlow in Ryu

Ryu contiene al suo interno la libreria di codifica e decodifica del controller OpenFlow.

Il compito principale del controller è quello di gestire le flow table degli switch OpenFlow, scambiando vari tipi di pacchetti con essi. Il controller viene programmato per gestire eventi, ad esempio l'arrivo di un pacchetto allo switch, e in base ad essi modificare o ad aggiungere una regola alla flow table.

Nella tabella seguente vengono riassunti i messaggi principali scambiati tra controller e switch:

Controller to Switch Messages	Asynchronous Messages	Symmetric Messages
Handshake, switch-config, flow-table-config, modify/read state, queue-config, packet-out, barrier, role-request	Packet-in, flow-removed, port-satus, and Error.	Hello, Echo-Request & Reply, Error, experimenter
send_msg API and packet builder APIs	set_ev_cls API and packet parser APIs	Both Send and Event APIs

FIGURA 13: PRINCIPALI MESSAGGI OPENFLOW IN RYU [19].

I messaggi asincroni che lo switch manda al controller vengono gestiti tramite il decoratore set_ev_cls, un particolare tipo di funzione che permette al controller di "sentire" quando si scatena un evento.

3.2.2 Ryu Manager

Prima di spiegare come funziona un'applicazione in Ryu, è doveroso spendere due parole sul Ryu Manager.

Il Ryu Manager è l'eseguibile principale del programma, ogni volta che si vuole far funzionare un'applicazione di Ryu va richiamato. Quando viene avviato, si mette in ascolto tramite un indirizzo IP specifico su una porta nota. Ogni switch in questo modo può connettersi ad esso.

3.2.3 Applicazioni di Ryu

Un'applicazione di Ryu non è altro che un modulo Python nel quale viene definita uno sottoclasse di **ryu.base.app_manager.RyuApp**. Le applicazioni sono entità single-threaded, ciò significa che possono venir eseguite solo una alla volta (ad esempio non è possibile far funzionare uno switch sia da hub che da mac learning switch).

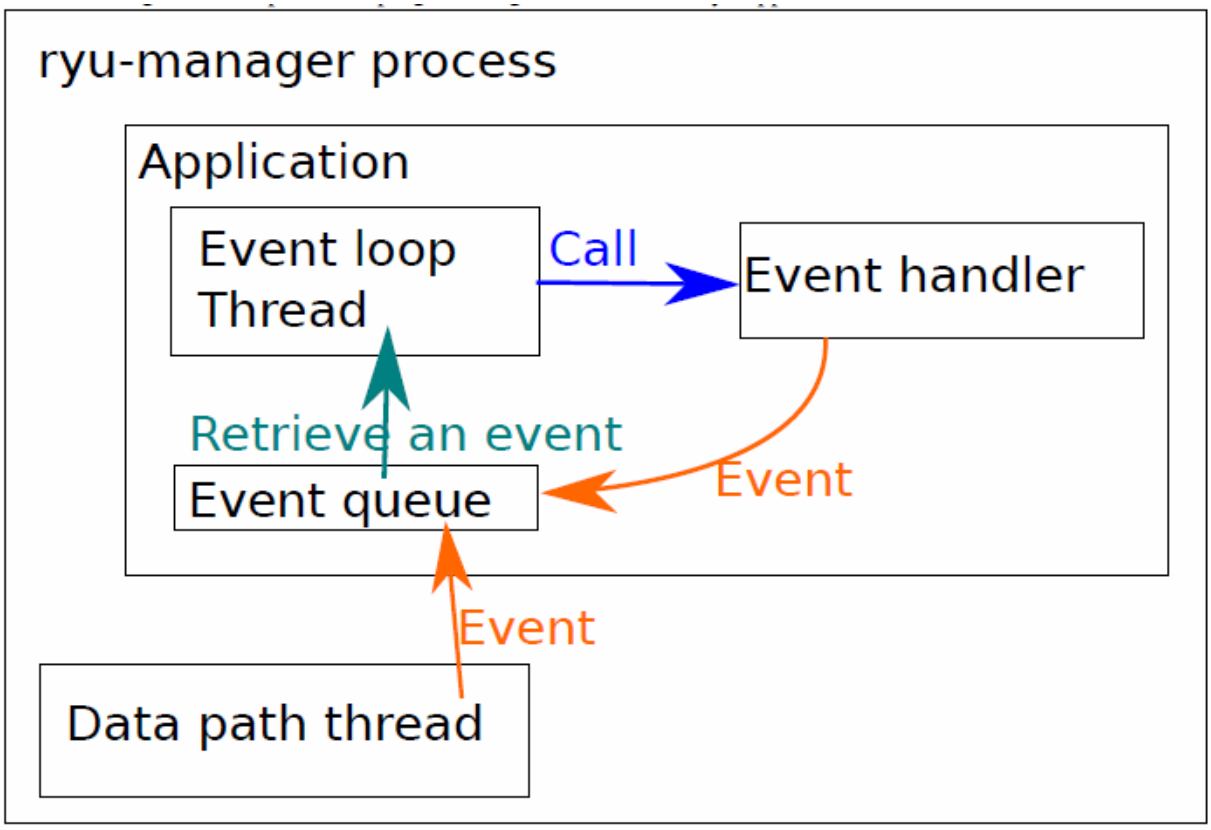

FIGURA 14:FUNZIONAMENTO DI UNA APP IN RYU [24].

Come si può vedere dalla figura 14, l'applicazione è un programma interno al Ryu Manager, le applicazioni dialogano tra di loro tramite eventi asincroni gestiti dall' *Event handler* e mantenuti in ordine tramite la *Event queue*, che funge da pila per gli eventi ricevuti. La pila di eventi funziona con logica FIFO (*First In First Out*) [19].

Ogni volta che l'applicazione riceve un evento, a seconda della sua tipologia viene richiamato un componente che prenderà il controllo del sistema fino al termine della sua funzione, nel frattempo gli altri eventi non possono essere processati.

3.3 Scrivere applicazioni in Ryu

In questo paragrafo viene illustrato un esempio di semplice applicazione Ryu (fornita di default dal software), verranno analizzati e spiegati i punti salienti del codice e qualche possibile modifica.

L'applicazione che andremo ad analizzare si chiama simple_switch_13.py ed è interamente consultabile al seguente link:

https://github.com/osrg/ryu/blob/master/ryu/app/simple_switch_13.py

Lo scopo di questa applicazione è quello di implementare uno *mac learning switch,* cioè uno switch che associa ad ogni porta il numero MAC dell'interfaccia di rete ad essa collegata.

Prima di tutto bisogna importare tramite i comandi *from* e *import* i vari moduli e sottomoduli che si utilizzeranno nel codice :

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
```

Come è già stato anticipato, un'applicazione Ryu è una sottoclasse di app_manager.RyuApp, per cui il programma vero e proprio deve cominciare con la definizione della sottoclasse.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

Il metodo __init__ viene chiamato metodo di inizializzazione, e lo si deve definire ogni volta che si crea un oggetto.

Viene anche definito il dizionario vuoto self.mac_to_port. Un dizionario in python è una struttura dati dove ogni elemento è una coppia chiave-valore. Nel nostro caso la chiave sarà la porta ed il numero mac il valore associato.

Un'applicazione Ryu può mettersi in ascolto riguardo eventi specifici grazie al decoratore **set_ev_cls** importato dal modulo **ryu.controller.handler.** Il primo argomento del decoratore rappresenta l'evento da gestire, mentre il secondo esprime lo stato dello switch:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)
```

Queste righe di codice servono ad aggiungere la regola di table miss, cioè il caso in cui un pacchetto non trova nessun riscontro nella flow table, e quindi va inviato al controller per venire analizzato. Il primo argomento del decoratore rappresenta l'evento di configurazione dello switch, mentre il secondo argomento, **CONFIG_DISPATCHER,** indica che lo stato è quello di attesa di messaggi di configurazione.

Infine si utilizza la funzione **add_flow,** che viene definita di seguito:

```
def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
```

```
    if buffer_id:
        mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                                priority=priority, match=match,
                                instructions=inst)
    else:
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
    datapath.send_msg(mod)
```

Il compito della funzione add_flow è di creare il messaggio che il controller deve mandare allo switch per aggiungere una regola alla flow table. Il messaggio viene costruito tramite il metodo OFPFlowMod, indicando come datapath l'identificativo dello switch e come istruzione **ofproto.OFPIT_APPLY_ACTIONS**, cioè aggiungere la regola alla flow table.

Il datapath è fondamentale in quanto un controller potrebbe avere più switch da gestire, questo richiede che ogni switch abbia un proprio identificativo univoco.

Una volta costruito il messaggio, chiamato **mod** in questo caso, esso viene spedito dal controller allo switch tramite il metodo send_msg della classe datapath.

Resta ora da analizzare il decoratore che gestisce l'evento di *packet-in:*

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        # If you hit this you might want to increase
        # the "miss_send_length" of your switch
        if ev.msg.msg_len < ev.msg.total_len:
            self.logger.debug("packet truncated: only %s of %s bytes",
                              ev.msg.msg_len, ev.msg.total_len)
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']
```

In questo caso il secondo argomento del decoratore, **MAIN_DISPATCHER,** indica che vanno ignorati i messaggi di *packet-in* fino al termine della negoziazione tra il controller Ryu e lo switch.

La funzione **packet_in_handler** svolge due compiti fondamentali:

- Prelevare l'indirizzo MAC dal pacchetto ricevuto ed aggiungerlo al dizionario *mac_to_port.*

```
self.mac_to_port[dpid][src] = in_port

if dst in self.mac_to_port[dpid]:
    out_port = self.mac_to_port[dpid][dst]
else:
    out_port = ofproto.OFPP_FLOOD

actions = [parser.OFPActionOutput(out_port)]
```

- Aggiungere la regola nella flow table e spedire il messaggio di *packet-out.*

```
# install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

Così si è realizzato uno switch in grado di tener conto dei flussi già analizzati, evitando in questo modo il presentarsi del messaggio di *packet-in* ogni volta che un pacchetto lo attraversa.

3.3.1 Pacchetti ARP e ICMP

Una possibile modifica per il codice precedente, potrebbe essere quella di installare a priori delle regole per la gestione dei pacchetti ARP e ICMP. Queste regole vanno installate subito dopo l'handshake tra controller e switch, per cui le seguenti righe di codice vanno inserite all'interno della funzione **switch_features_handler,** assieme alla regola di table miss:

```
self . logger . debug ("** installo regole ARP ")
match = parser . OFPMatch ( eth_type = 2054)
actions = [ parser . OFPActionOutput ( ofproto . OFPP_NORMAL )]
self . add_flow ( datapath , 1, match , actions )
```

```
self . logger . debug ("** installo regole ICMP ")
match = parser . OFPMatch ( eth_type = 2048 , ip_proto =1)
actions = [ parser . OFPActionOutput ( ofproto . OFPP_NORMAL )]
self . add_flow ( datapath , 1, match , actions )
```

Grazie all'aggiunta di queste regole, si evitano i messaggi di packet-in per quanto riguarda tali protocolli [19].

3.3.2 Analisi TCP e UDP

Un'altra aggiunta riguarda il controllo del tipo di protocollo di trasporto utilizzato, in questo modo, si possono inserire nello switch regole altamente selettive che permettono di migliorare la sicurezza e il funzionamento del sistema. Tali regole vanno inserite all'interno della funzione **packet_in_handler** [19]:

```
pkt_ipv4 = pkt. get_protocol ( ipv4 . ipv4 )
ip_src = pkt_ipv4 .src # indirizzo IP sorgente
ip_dst = pkt_ipv4 .dst # indirizzo IP destinazione

actions = []

if pkt_ipv4 :
   if pkt_ipv4 . proto ==6 : # protocollo TCP
      match = parser . OFPMatch ( in_port = in_port ,
               eth_type = eth. ethertype ,ip_proto =6, ipv4_src = ip_src , ipv4_dst = ip_dst )
      actions = [ parser . OFPActionOutput (
      self . add_flow ( datapath , 1, match , actions )
    elif pkt_ipv4 . proto ==17 :      # protocollo UDP
       match = parser . OFPMatch ( in_port = in_port ,
                eth_type = eth. ethertype ,
                    ip_proto =17 , ipv4_src = ip_src ,
                    ipv4_dst = ip_dst )
       actions = [ parser . OFPActionOutput (
                              ofproto . OFPP_NORMAL )]
       self . add_flow ( datapath , 1, match , actions )
else :
   self . logger . debug ("**** FLUSSO NON AMMESSO ****")
   match = parser . OFPMatch ( in_port = in_port ,
                    eth_type = eth. ethertype ,
                    eth_dst =dst )
   actions = []
   if msg . buffer_id != ofproto . OFP_NO_BUFFER :
      self . add_flow ( datapath , 1, match , actions ,
                     msg. buffer_id )
      return
else :
      self . add_flow ( datapath , 1, match , actions )
```

La classe **OFPMatch** permette di utilizzare filtri più o meno selettivi per caratterizzare le regole da salvare nella flow table.

CAPITOLO 4

Composizione dinamica di funzioni di rete virtuali e relativo piano di controllo SDN

Questo capitolo serve ad esplicare il funzionamento teorico del paradigma SDN, applicato a due casi di studio più semplici, il caso data center, cioè in cui il provider dà la connettività alle macchine virtuali instanziate, ed il caso in cui un provider offra connettività ai suoi utenti, rispettivamente reti L2 ed L3. Inoltre viene introdotta la macchina a stati del paradigma SDN, che servirà poi a comprendere meglio il funzionamento del codice del controller generalizzato trattato nel capitolo 5.

Infine, viene mostrata lo topologia di riferimento per il controller generalizzato, che rappresenta una rete simile al caso reale di tipo L3.

In questo capitolo e nei prossimi si farà sempre riferimento a reti virtuali implementate all'interno di server, questo proprio per unire il paradigma NFV con il paradigma SDN. È importante sottolineare che il protocollo SDN può essere applicato anche a reti strettamente fisiche, ma l'unione dei due paradigmi rappresenta un'ottima soluzione all'ossificazione di rete.

4.1 Service Chaining Dinamico

Con Service Chaining Dinamico si intende la catena di servizi che, a seconda degli istanti di tempo considerati, andranno applicati ad uno o più flussi di rete. Quando si parla di servizi si intendono vari tipi di funzioni di rete virtuali, come ad esempio Wana (Wide Area Network Accelerator), TC (Traffic Shaper) e DPI(Deep Packet Inspection).

Queste funzioni, per il protocollo SDN, devono essere applicate ai vari flussi di traffico in modo dinamico, per cui i pacchetti dovranno essere reindirizzati ai vari servizi a seconda delle esigenze e dello SLA, cioè il *Service Level Agreement,* il contratto stipulato tra client ed operatore di rete.

Ad esempio la funzione di classificazione svolta dal DPI dovrà essere applicata solo inizialmente ad un certo flusso dati per determinarne il suo SLA, dopodiché i pacchetti dovranno seguire un percorso diverso, a seconda del servizio richiesto.

Questo dinamismo consente all'operatore di rete di trattare ogni flusso in modo diverso a seconda del momento, e di garantire in ogni istante di tempo la giusta QoS, *Quality of Service*, promessa al client.

È dunque necessaria la presenza di un controller SDN centralizzato che faccia in modo, che ogni flusso di traffico dell'utente attraversi le corrette funzioni di rete virtuali nell'ordine richiesto [5] . Risulta chiaro che queste azioni di re indirizzamento svolte dal controller dovranno essere trasparenti per l'utente, il cui requisito di Qos deve essere soddisfatto.

4.2 SDN Finite State Machine

Per descrivere il funzionamento del Service Chaining Dinamico risulta comodo utilizzare una macchina a stati finiti, in quanto ogni flusso a seconda dell'input e dell'istante di tempo considerato può essere associato ad uno stato diverso.

Le transizioni all'interno dell'automa dipendono dallo stato corrente e dall'input ricevuto, per chiarezza nella figura sottostante è stato riportato solo l'input.

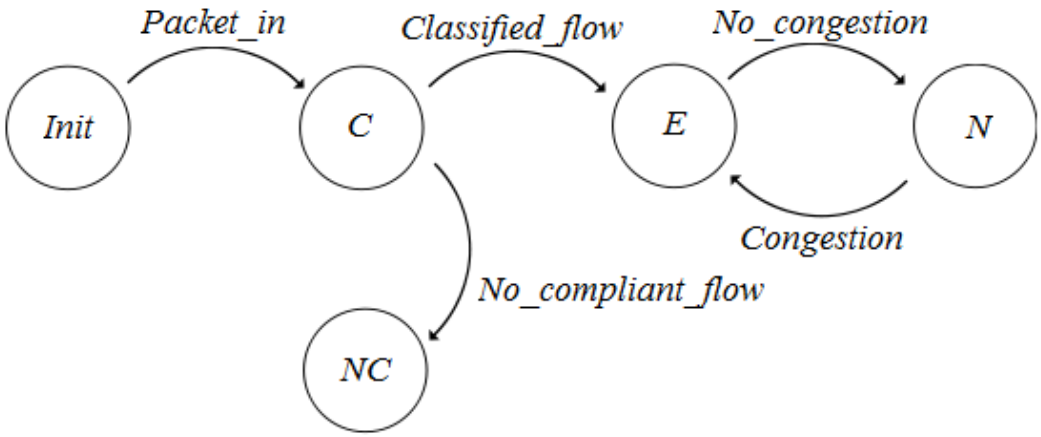

FIGURA 15: FSM CHE RAPPRESENTA IL FUNZIONAMENTO DEL PARADIGMA SDN.

Con riferimento alla figura, definisco gli stati:

- *Init*: stato di inizializzazione nel quale il controller installa regole generiche di forwarding all'interno nel network node. Per ogni nuovo flusso entrante l'automa parte da questo stato;

- *C*: rappresenta lo stato di classificazione del flusso, nel quale il flusso viene reindirizzato al DPI, un dispositivo di rete virtuale che si occupa dell'analisi dei pacchetti, per determinarne il *Service Level Agreement*;

- *D*: l'automa entra in questo stato quando viene individuato un flusso non idoneo ad alcun SLA, per cui viene creata ed aggiunta alla *flow table* una regola di packets drop;

- *E*: stato di *enforcement,* significa che nella rete sono presenti troppi flussi attivi oppure è presente un utente più prioritario, per cui i pacchetti vanno reindirizzati alle rispettive funzioni a seconda dello SLA, per rispettare la QoS;

- *N:* la rete non è congestionata, ciò significa che i pacchetti non devono transitare attraverso funzioni di rete specifiche, ma passare semplicemente dal default gateway.

Descrivo ora il funzionamento della macchina a stati.

Inizialmente, quando il controller viene collegato allo switch OpenFlow si entra nello stato *init*, nel quale il controller installa regole indipendenti dagli specifici flussi (regole per pacchetti ARP, ICMP….eccetera) nella *flow table.*

Quando un flusso *f* arriva allo switch, ho un evento di *packet_in*, per cui il controller passa allo stato di classificazione del flusso. In questo stato, *f* viene reindirizzato al *Deep Packet Inspection* che, esaminandone i pacchetti, deduce se il flusso è conforme o meno ad un *Service Level Agreement.*

Nel caso in cui *f* non fosse conforme, il controller costruirà la regola di drop da mandare allo switch per far scartare i pacchetti relativi a tale flusso.

Altrimenti, se *f* fosse conforme, il controller, in base allo SLA comunicato dal DPI, reindirizzerà il flusso verso la funzione di rete adatta fino a quando non viene confermata la non congestione. Questo stato può essere chiamato di

preventive enforcement, in quanto non è detto che la rete sia congestionata, ma per garantire il QoS la si ipotizza tale.

Una volta che viene confermata la non congestione della rete, la macchina a stati entra nello stato *N* per il flusso *f*. In questo stato non c'è bisogno di reindirizzare i pacchetti verso funzioni di rete specifiche in quanto, il numero di flussi è minore del limite che determina la congestione.

Verranno ora illustrati due semplici topologie di rete, per comprendere dal punto di vista più pratico come agisce il controller SDN.

4.2.1 Topologia L2

La figura 16 rappresenta la topologia L2, cioè il caso tipico di utenti che vogliono connettersi in rete.

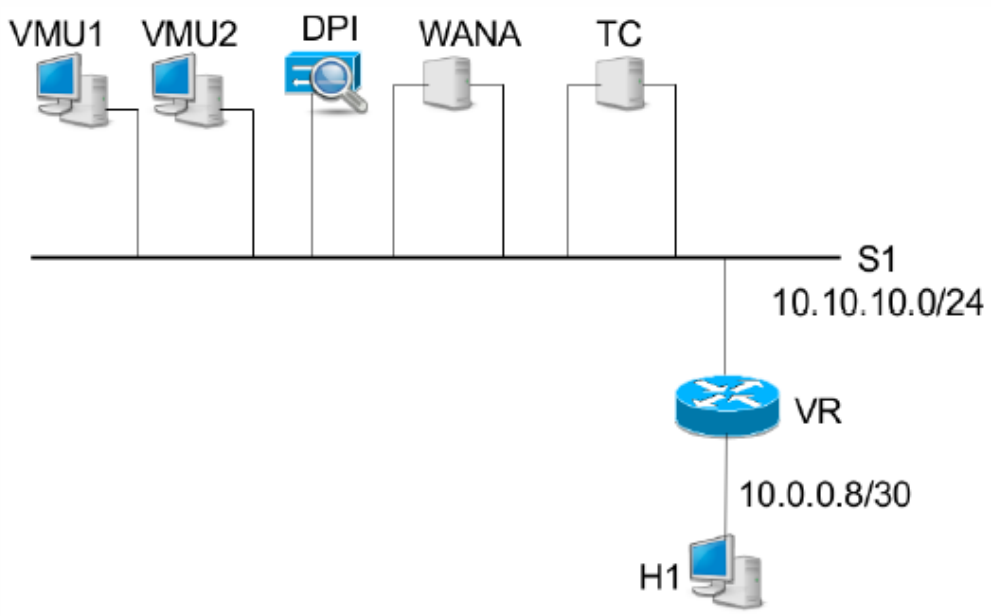

FIGURA 16:TOPOLOGIA L2 [25].

Elenco di seguito e caratterizzo tutti i componenti della rete:

- *VMU1*, rappresenta il *Business User* (BU), cioè l'utente più prioritario. In caso di congestione i flussi di tale utente vanno reindirizzati al WANA, per garantire la larghezza di banda decisa nello SLA. In caso di non congestione il flusso segue le regole tradizionali di forwarding;

- *VMU2* , utente meno prioritario o *Residence User*, caratterizzato da un servizio di tipo *Best-effort.* Quando utenti di priorità maggiore non utilizzano la connessione, il traffico di tale utente segue le regole tradizionali, in caso contrario, il suo flusso va reindirizzato al TC, in modo da modellare la sua banda in funzione della bit rate necessaria agli utenti a maggiore priorità;

- *DPI*, come già anticipato, è un analizzatore di pacchetti, un Application Level Gateway, che consente di classificare i flussi ed associargli il giusto SLA;

- *WANA*, rappresenta il *Wide Area Network Accelerator*, cioè un dispositivo che ottimizza la banda di un utente tramite la compressione e la duplicazione dei pacchetti, in modo da ridurre la quantità di dati da trasmettere;

- *TC*, cioè il *traffic shaper*, un apparato virtuale che consente di ridurre la banda massima degli utenti a minor priorità in caso di congestione, in maniera tale che l'utente più prioritario possa usufruire di banda maggiore.

- *S1*, rappresenta lo switch OpenFlow di rete;

- *VR*, router virtuale che rappresenta il default gateway degli utenti;

- *H1*, simboleggia l'utente destinatario dei flussi uscenti dalla rete L2.

Tutte le funzioni di rete (controller compreso) e gli utenti elencati vengono realizzati tramite l'utilizzo delle Virtual Machine. Le funzioni WANA, TC e DPI in un caso reale sarebbero funzioni messe a disposizione dal gestore di rete.

Analizzo di seguito il funzionamento della rete tramite un approccio a fasi. Per il seguente esempio si suppone che sia *VMU1* a trasmettere per prima, ma tutto quello che viene detto sarebbe valido anche se fosse *VMU2* a trasmettere inizialmente.

FASE 1

Una volta che l'*handshake* tra controller e switch è terminato la rete è pronta, a questo punto quando VMU1 genera un flusso verso H1 ha inizio la prima fase. Il controller riceve un messaggio di packet_in da parte dello switch, contenente le caratteristiche del flusso in ingresso. I pacchetti vengono reindirizzati al DPI per classificare il flusso e contemporaneamente mandati al nodo destinatario.

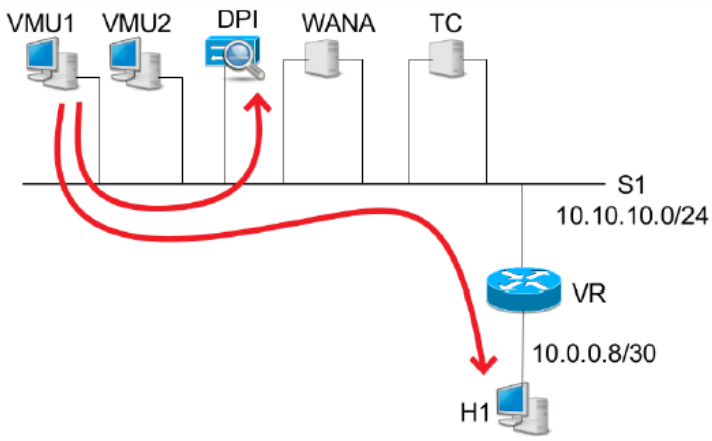

FIGURA 17:FASE 1 TOPOLOGIA L2 [19].

FASE 2

Dopo che è stato riconosciuto che il flusso appartiene all'utente più prioritario, la macchina a stati passa allo stato di enforcement preventivo. In questo stato il flusso di VMU1 viene reindirizzato al WANA che comprime i pacchetti e li spedisce al nodo destinatario. Questa fase ha una durata molto breve.

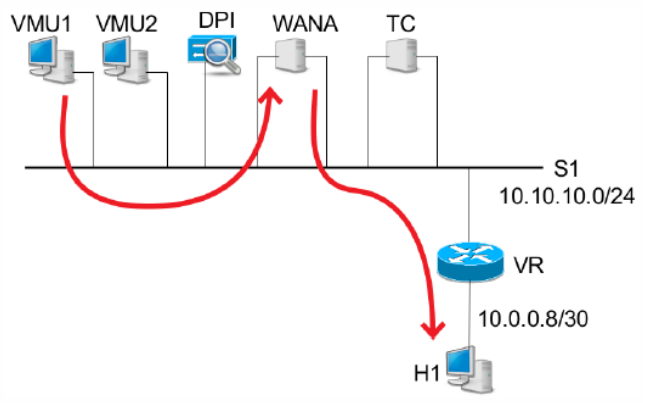

FIGURA 18:FASE 2 TOPOLOGIA L2 [19].

FASE 3

Dopo l'enforcement preventivo, visto che la rete non è congestionata, il controller associa il flusso di VMU1 allo stato di non enforcement, per cui esso verrò reindirizzato direttamente al router virtuale per sfruttare tutta la banda disponibile.

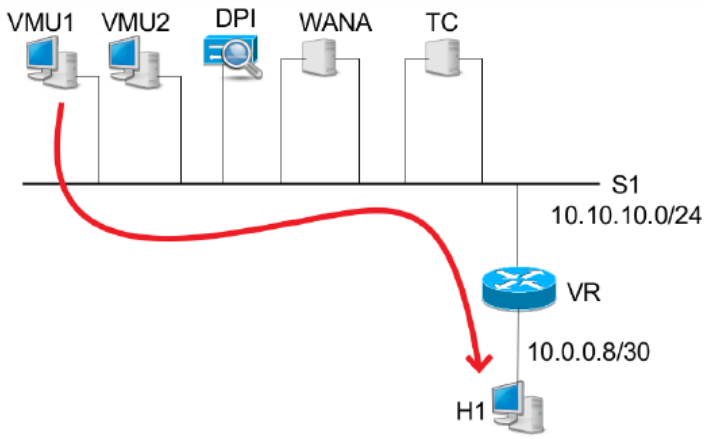

FIGURA 19:FASE 3 TOPOLOGIA L2 [19].

FASE 4

Questa fase ha inizio quando VMU2, cioè l'utente meno prioritario, comincia a trasmettere. Per prima cosa quindi il flusso di VM2 viene inviato al DPI oltre che alla destinazione per procedere alla classificazione. Il flusso di VMU1 per tutta la durata di questa fase non subisce alterazioni.

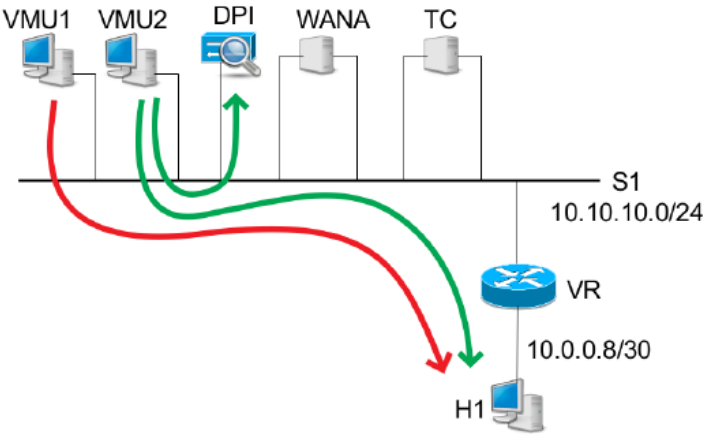

FIGURA 20:FASE 4 TOPOLOGIA L2 [19].

FASE 5

In questa fase il controller, tramite l'analisi eseguita dal DPI, capisce che il nuovo flusso appartiene all'utente meno prioritario e che la rete è congestionata.

La macchina a stati entra perciò nella fase di enforcement per entrambi i flussi, che vengono immediatamente reindirizzati alla virtual network function correlata , WANA per il flusso appartenente a VMU1 E TC per il flusso appartenente a VMU2, per poi giungere a destinazione.

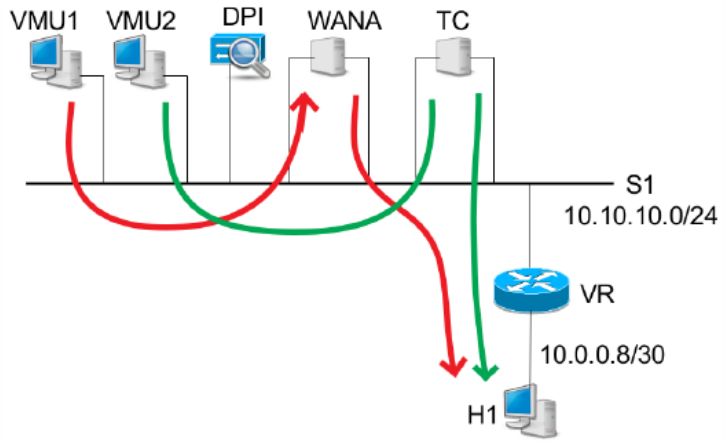

FIGURA 21:FASE 5 TOPOLOGIA L2 [19].

Con questa fase si conclude l'analisi teorica del funzionamento del Service Chaining Dinamico per quanto riguarda la topologia L2.

È importante notare che WANA e TC hanno due interfacce sullo stesso switch, questo implica che si possono verificare problemi di *arp storming*, cioè un loop di pachchetti ARP tra le due interfacce di WANA o TC. Per questo motivo è fondamentale installare sullo switch delle regole per i pacchetti ARP.

Infine voglio sottolineare che quando si parla di reindirizzamento, si intende la modifica dell'indirizzo MAC di destinazione. Questa operazione viene svolta in maniera dinamica dal controller.

Nel prossimo sotto paragrafo verrà mostrata la topologia L3, e ne verranno elencate le differenze rispetto alla L2, sia di rete che di funzionamento.

4.2.2 Topologia L3

La seguente immagine illustra la topologia di rete L3

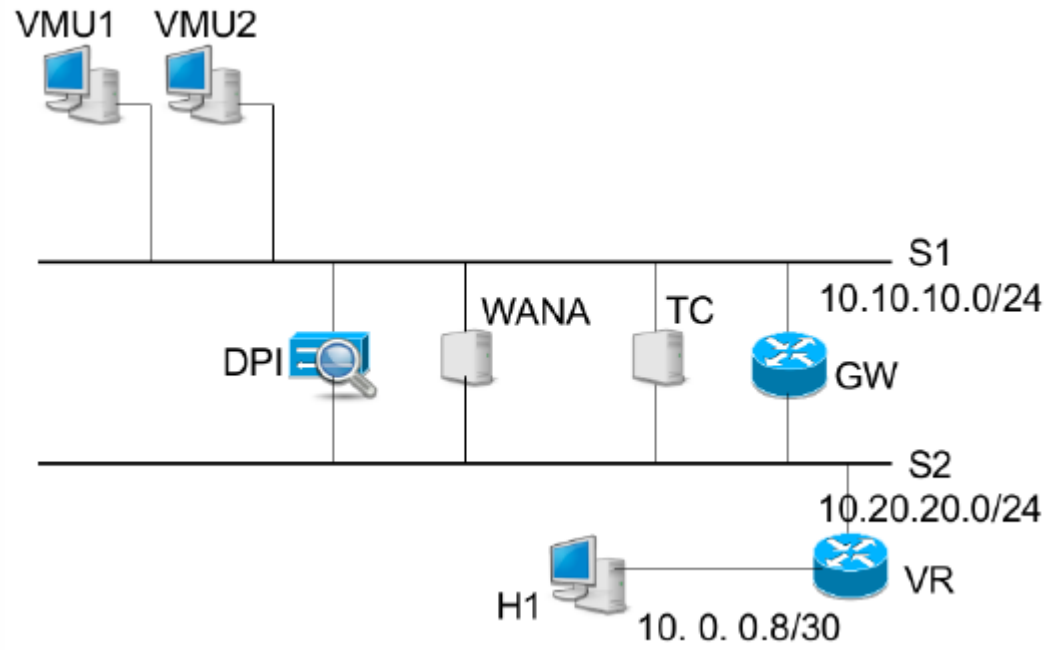

FIGURA 22:TOPOLOGIA L3 [25].

Le principali differenze rispetto al caso L2 sono:

- La presenza del router GW, che sarà il gateway per i flussi che non dovranno transitare attraverso una funzione di rete specifica;
- L'utilizzo di due switch invece di uno, proprio per caratterizzare la topologia;
- Le interfacce di rete di DPI, WANA e TC sono su due switch diversi, al contrario del caso precedente in cui ognuna di queste funzioni virtuali occupava due porte dello switch. Non sussistono quindi problemi legati alla prolificazione di pacchetti ARP.

Il funzionamento della rete è molto simile al caso precedente, ma viene comunque svolta un'analisi per fasi per comprendere appieno tutti i passaggi.

FASE 1

Questa fase ha inizio quando VMU1 comincia a trasmettere pacchetti verso H1

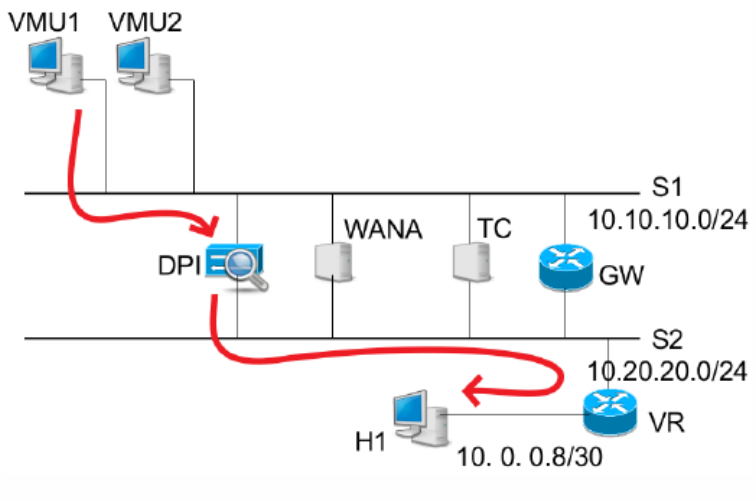

FIGURA 23:FASE 1 TOPOLOGIA L3 [19].

Il flusso di VMU1 deve transitare attraverso il DPI per essere classificato. Rispetto al caso precedente il flusso che attraversa il DPI arriva direttamente a destinazione.

FASE 2

Al termine della classificazione la macchina a stati riferita al flusso di VMU1 entra nella fase di enforcement, in attesa della conferma di non congestione della rete. Il flusso viene perciò indirizzato al WANA per ottenere l'aumento di banda.

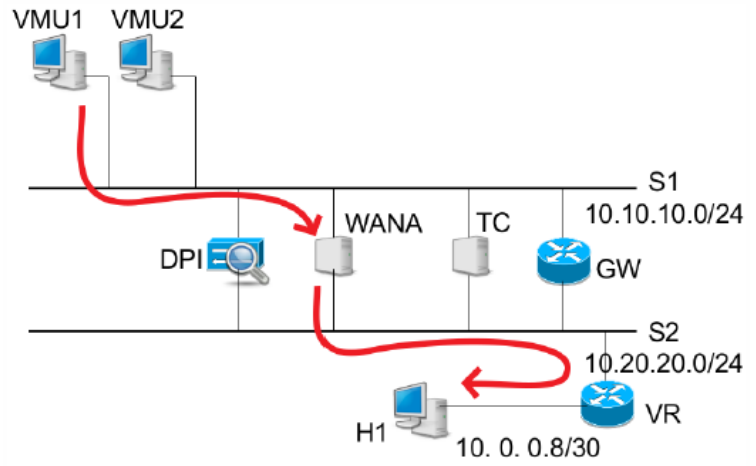

FIGURA 24:FASE 2 TOPOLOGIA L3 [19].

FASE 3

Siccome oltre all'utente più prioritario nessun utente trasmette dati, il controller reindirizza il traffico di VMU1 al router GW, il flusso entra quindi nella fase di non enforcement.

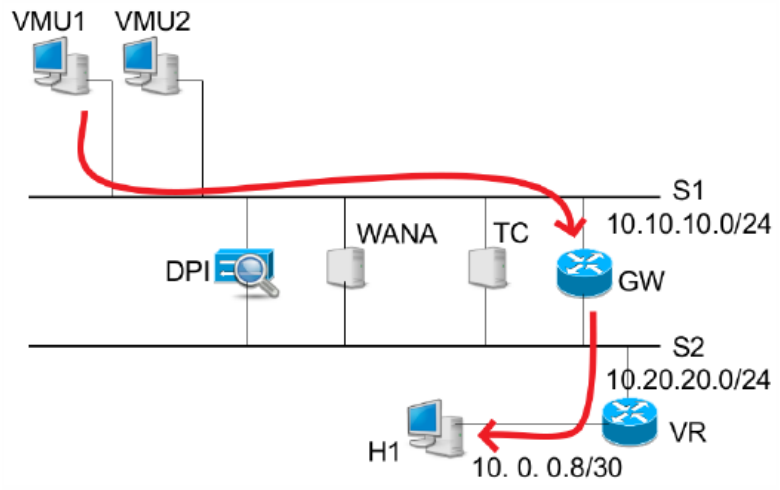

FIGURA 25:FASE 3 TOPOLOGIA L3 [19].

FASE 4

Come per il caso L2, questa fase ha inizio quando VMU2 genera un flusso dati verso il nodo H1. L'arrivo di pacchetti con sorgente IP diversa allo switch, comporta un evento di packet_in per il controller che indirizzerà il nuovo flusso verso il DPI per procedere alla classificazione.

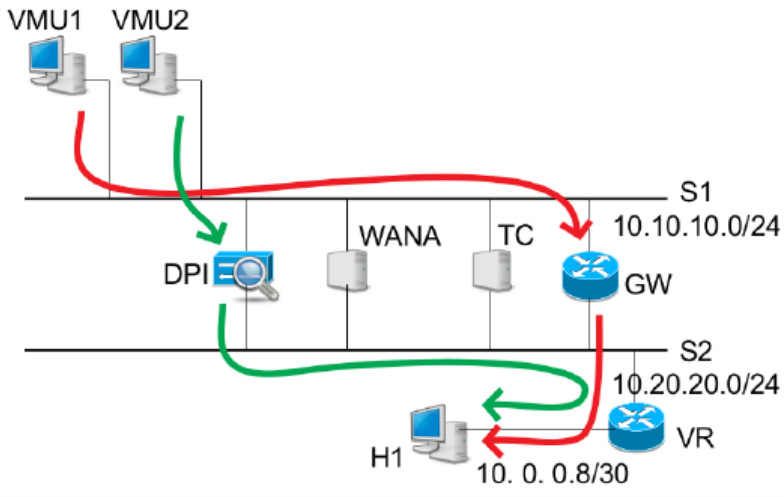

FIGURA 26:FASE 4 TOPOLOGIA L3 [19].

FASE 5

In quest'ultima fase, grazie al DPI il controller riconosce la presenza di congestione di rete, per cui per rispettare le QoS concordate a monte con gli utenti, il flusso dell'utente più prioritario verrà reindirizzato al WANA, mentre quello dell'utente a priorità minore verrà indirizzato al TC per limitarne l'uso di banda.

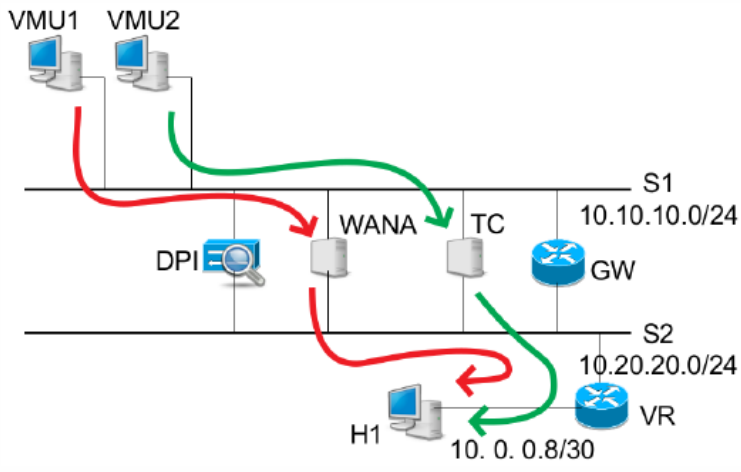

FIGURA 27: FASE 5 TOPOLOGIA L3 [19].

Con questa fase termina lo studio della topologia L3.

Nel prossimo paragrafo viene illustrata una topologia più generale, che rappresenta l'unione delle topologie appena esposte. La seguente architettura sarà di riferimento per il prossimo capitolo, in quanto i test sul codice del controller generalizzato sono stati svolti sulla seguente rete.

4.3 Architettura di rete generalizzata

Le topologie viste in precedenza rappresentavano dei casi particolari di reti, ma se in futuro le telecomunicazioni dovessero utilizzare un approccio software defined, il codice del controller dovrebbe essere generalizzato, in quanto la rete potrebbe essere di qualsiasi tipo.

Per testare il codice, che verrà trattato nel prossimo capitolo, è stata utilizzata la seguente architettura di tipo Layer 3 più complessa, che rispecchia un possibile caso reale.

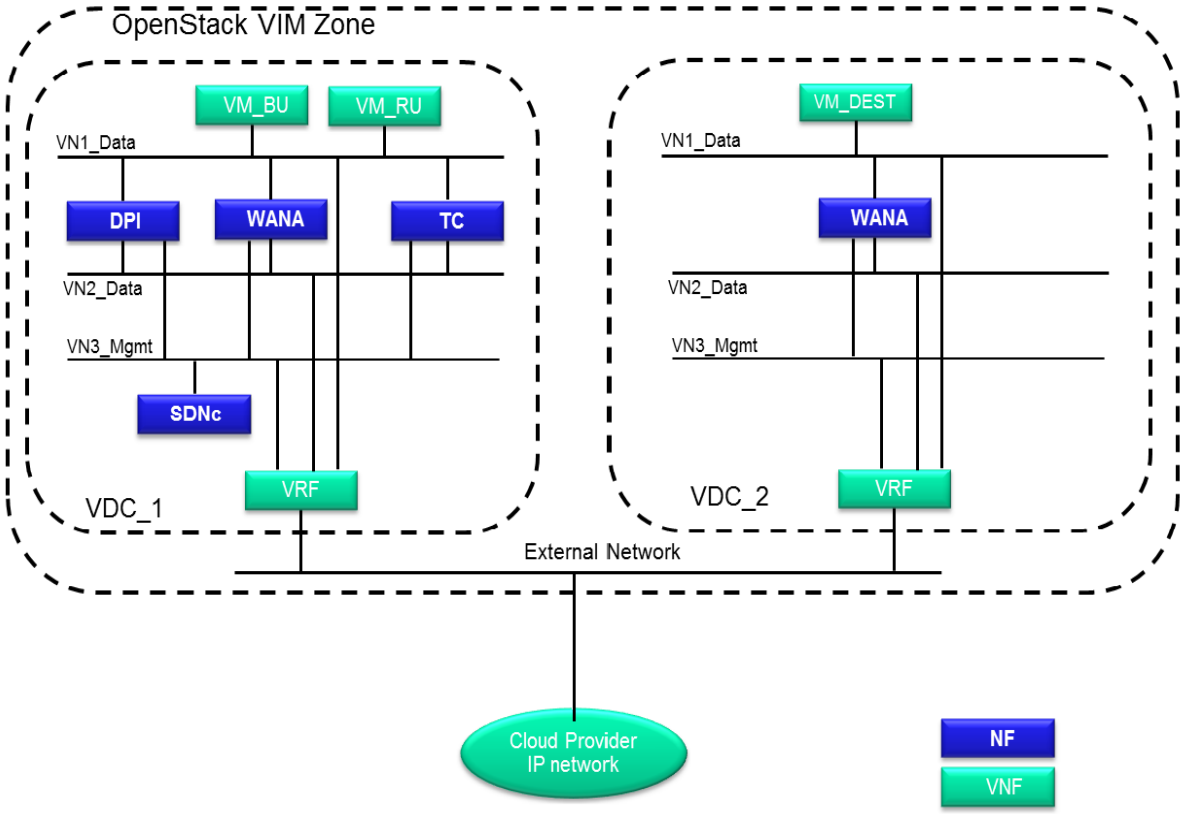

FIGURA 28:TOPOLOGIA DI RETE GENERALIZZATA [26].

La topologia in figura 28 è stata realizzata su OpenStack, ne analizzo di seguito i componenti :

- *VDC_1* e *VDC_2* sono i due data center (*Virtual Data Center*), che possono comunicare solo tramite la rete esterna. Le macchine virtuali presenti all'interno di essi sono state istanziate su due compute node differenti, Nova e hc04. Comunemente i due data center dovrebbero essere dislocati in luoghi diversi, per comodità nel nostro caso sono stati istanziati sullo stesso cluster;

- *VM_BU* e *VM_RU* rappresentano come nel caso precedente rispettivamente utente più prioritario e meno prioritario, sono stati istanziati come virtual machines dentro il compute node nova;

- *DPI*, *WANA* e *TC* rappresentano le funzioni virtuali viste in precedenza. Tutti e tre sono stati istanziati all'interno dello stesso compute node degli users;

- *SDNc* è il controller della rete;
- *VRF* (*Virtual Router Function*) sono i router virtuali che consentono di far comunicare tra loro le LAN. I router sono istanziati all'interno del network node del cluster;
- *VN1_Data* è una LAN per gli utenti, è collegata direttamente al router VRF;
- *VN2_Data* rappresenta la LAN per le funzioni di rete, è collegata direttamente al router VRF;
- *External Network* è la rete esterna che consente al cluster di collegarsi ad internet;
- *VN3_Mgmt* è la rete di gestione. Il controller SDNc trasmette i messaggi OpenFlow su questa rete;
- *VM_DEST* rappresenta l'utente destinatario;
- *WANA* nel VDC2 è la funzione di decodifica dei pacchetti che per la presenza di congestione sono stati compressi dal WANA su VDC1.

Nel prossimo sotto paragrafo vengono analizzati i meccanismi di *traffic steering* della rete dal punto di vista teorico, mentre nel prossimo capitolo viene mostrato il codice del controller, i test effettuati ed i risultati ottenuti.

4.3.1 Traffic Steering Layer 3

Come per le altre topologie viste, anche per questa viene effettuato uno studio diviso in fasi.

Rispetto ai casi precedenti la macchina a stati cambia leggermente: dopo che il DPI ha effettuato la classificazione, se il flusso appartiene all'utente meno prioritario, la macchina a stati riferita a tale flusso entra nello stato di non enforcement, altrimenti nel caso di utente più prioritario, tutti i flussi presenti sulla rete entrano nello stato di enforcement.

FASE 1

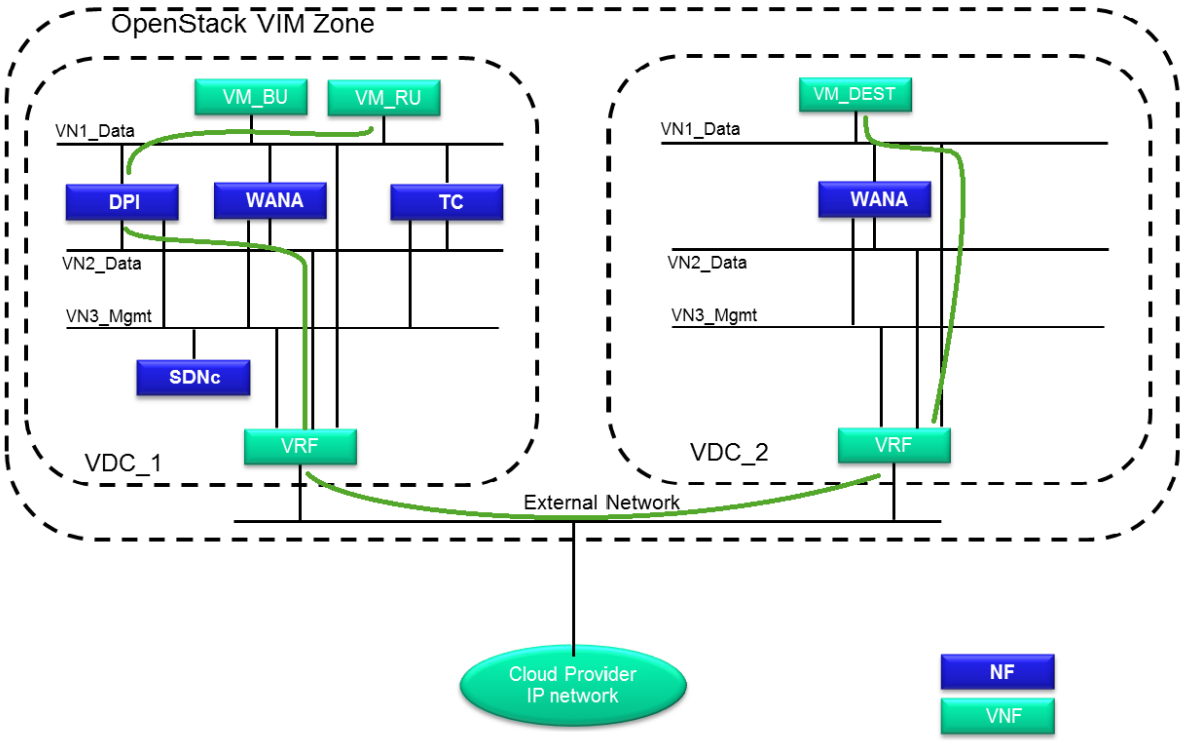

FIGURA 29:TRAFFIC STEERING LAYER 3, FASE 1.

I test che sono stati effettuati su questa topologia ipotizzano che il primo utente a generare traffico sia il residence, per cui la fase uno comincia quando il client meno prioritario trasmettere pacchetti verso il nodo destinatario.

Il flusso generato (colorato di verde in figura 29) viene indirizzato verso il DPI con il cambio dell'indirizzo MAC di destinazione, per procedere alla classificazione. Dal DPI il flusso raggiunge la destinazione tramite i due router.

Questa fase di classificazione nella realizzazione pratica ha una durata di 15 secondi.

FASE 2

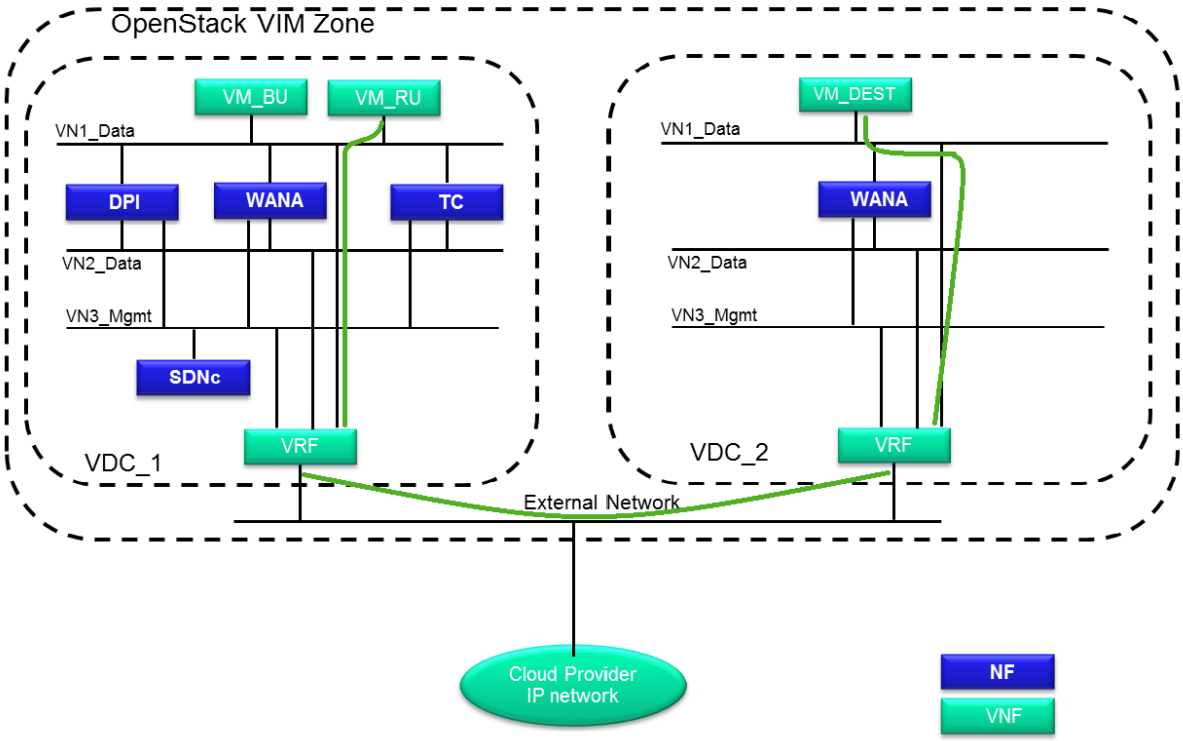

FIGURA 30:TRAFFIC STEERING LAYER 3, FASE 2.

Una volta che il flusso è stato riconosciuto e non sussiste una condizione di congestione di rete, si procede allo stato di non enforcement (a differenza degli esempi precedenti in cui si entrava nello stato di enforcement preventivo) per il flusso. Il traffico verrà indirizzato direttamente verso il router, senza dover quindi variare indirizzo MAC di destinazione.

FASE 3

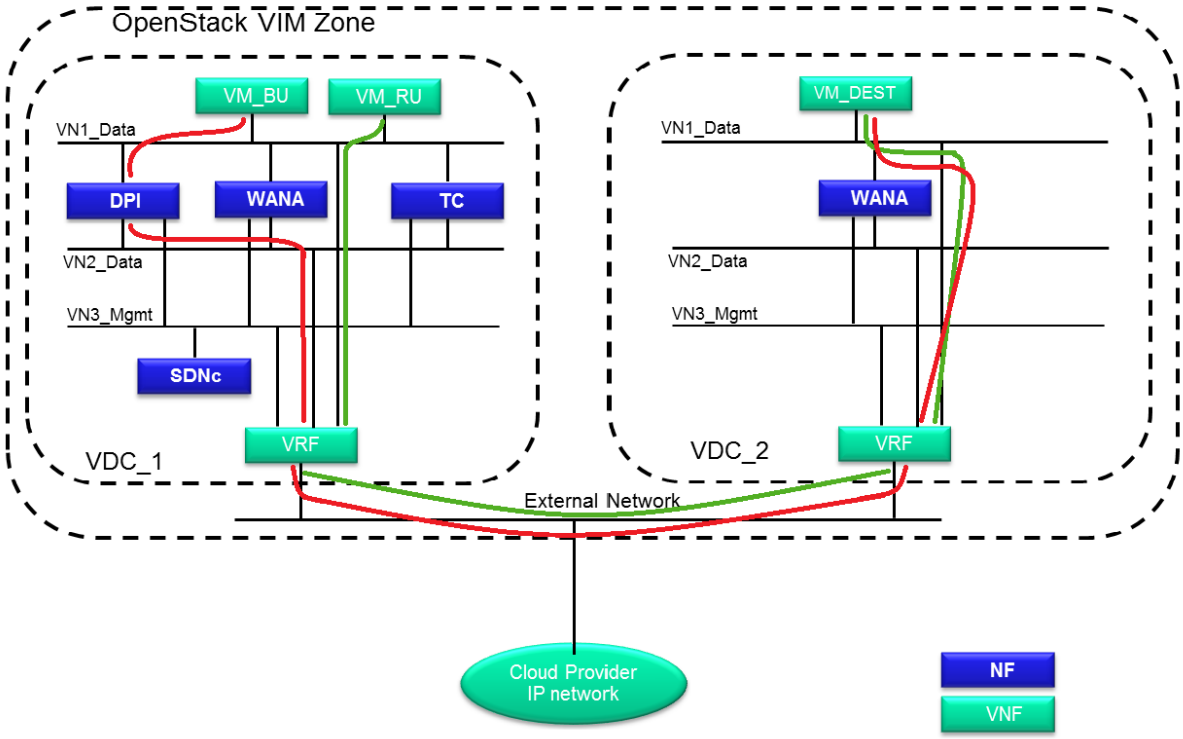

FIGURA 31:TRAFFIC STEERING LAYER 3, FASE 3.

Questa fase ha inizio quando il business user genera pacchetti di traffico verso il nodo destinatario, il flusso (colorato di rosso in figura 31) entra perciò nella fase di classificazione. Nel frattempo il flusso generato dall'utente residence continua a seguire il percorso definito in fase 2.

Come nel caso precedente la classificazione richiede 15 secondi di tempo per essere completata.

FASE 4

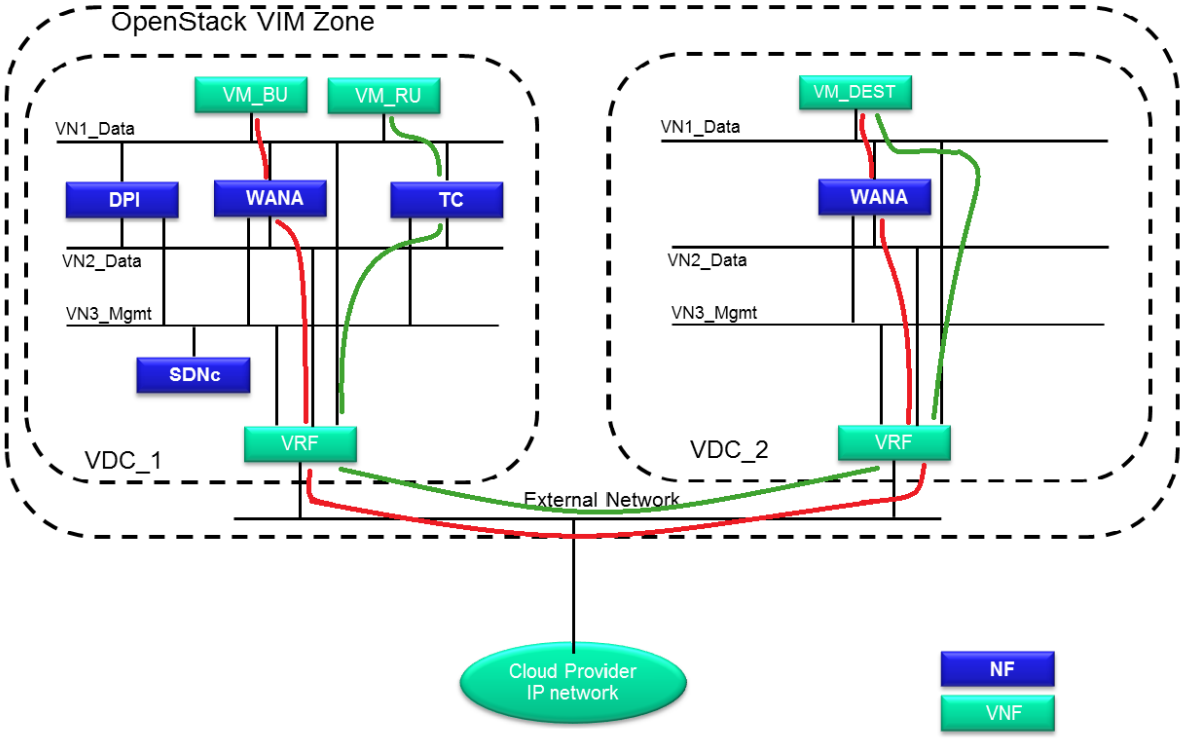

FIGURA 32:TRAFFIC STEERING LAYER 3, FASE 4.

Una volta che l'utente più prioritario viene classificato, tutti i flussi presenti nella rete entrano nello stato di enforcement, per cui il flusso del business user viene reindirizzato al WANA , mentre quello dell'utente meno prioritario al TC.

Nel data center di destinazione il flusso del business è costretto a passare attraverso la funzione di decodifica dei pacchetti compressi dal WANA.

Nei casi visti in precedenza, l'esempio terminava quando entrambi i flussi entravano in enforcement, nel caso generalizzato, al contrario si ipotizza che in modo dinamico si possa passare da stato di enforcement a non enforcement, o viceversa, a seconda della presenza o meno dell'utente più prioritario.

È presente per questo motivo un'altra fase.

FASE 5

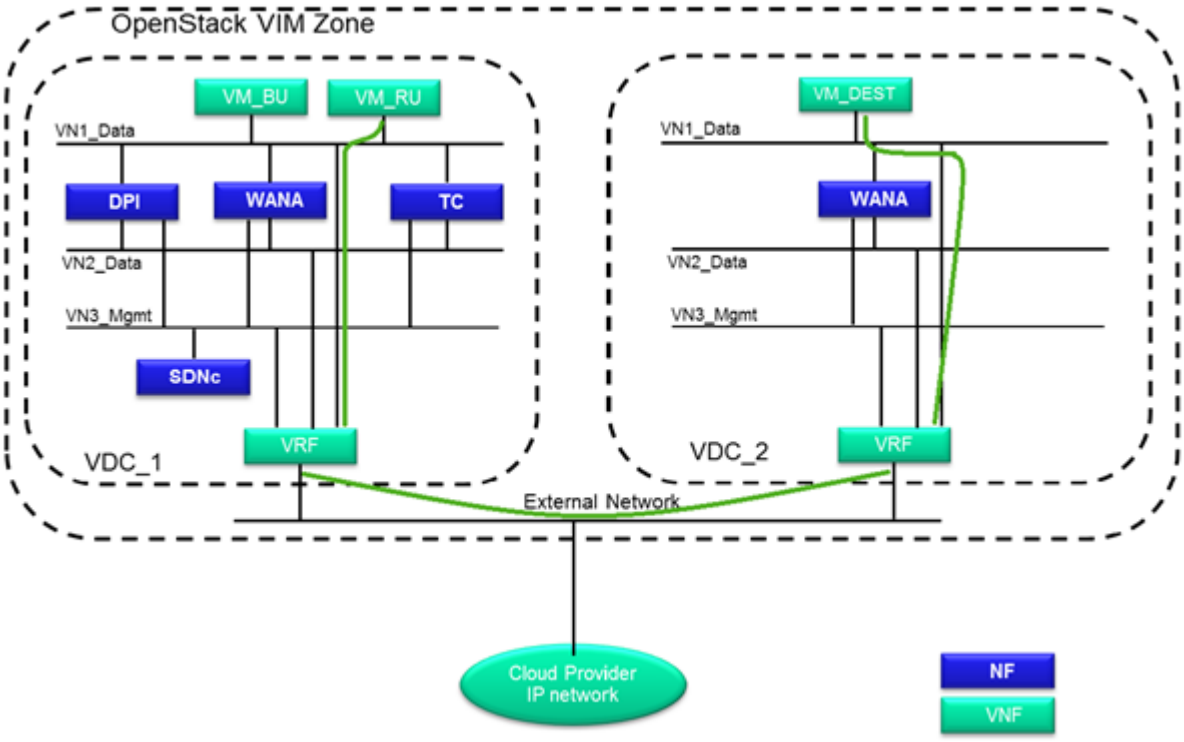

FIGURA 33:TRAFFIC STEERING LAYER 3, FASE 5.

Ipotizziamo perciò che l'utente più prioritario smetta di generare traffico.

Lo switch si accorge che l'utente più prioritario interrompe il proprio flusso dati grazie all'utilizzo di un *idle_timeout* che verrà spiegato nel prossimo capitolo.

Sarà dunque compito del controller riportare tutti i flussi presenti nella rete nello stato di non enforcement, per non limitare inutilmente la banda dei flussi ancora attivi, ed ottenere una situazione identica alla fase 3.

Questa fase rappresenta l'ultimo dei possibili casi che si possono ottenere dalla topologia L3 generalizzata.

Il codice che verrà trattato nel seguito è fatto in modo tale che possano essere presenti sia più utenti meno prioritari che più prioritari nella rete.

I test sono stati effettuati con l'utilizzo di due residence user, ma il funzionamento resta identico a quello appena descritto.

CAPITOLO 5

Implementazione e verifica

In quest'ultimo capitolo viene trattato il codice del controller generalizzato riferito alla topologia di rete vista precedentemente.

L'idea di base è quella di realizzare un controller completamente indipendente dalla rete che deve gestire, in modo da rendere il paradigma SDN una soluzione concreta all'ossificazione di Internet.

Prima di tutto vengono analizzati i punti salienti del codice e le funzioni più importanti, dopodiché vengono mostrati i test svolti ed i risultati ottenuti.

5.1 Implementazione del controller SDN Ryu

La prima parte del codice (dalla riga 1 alla 602) contiene le import delle librerie necessarie, le definizioni delle variabili globali e tutte le definizioni di funzioni utili per far funzionare la macchina a stati del controller.

Subito dopo la definizione della classe, sono presenti le variabili globali:

```
# INITIALIZE GLOBAL VARIABLES

    #logging.basicConfig(level=logging.INFO)
    global curr_flow_id
    curr_flow_id = 0
    global flows_state
    flows_state = [] # List of flows
    global active_flows
    active_flows = [] # List of active flows
    global  hipriousers
    hipriousers = [] #List of high priority users
    global flowlimit
    flowlimit = 2    #Max no. of flows in Non-Enforcement case
    global wanaproto
    wanaproto = [200, 201, 202] #Protocols of compressed packets
    global C_state_prio
    C_state_prio = 34500
    global E_state_prio
    E_state_prio = 34502
    global N_state_prio
    N_state_prio = 34502
```

```
global rules_list
rules_list = []
```

- **curr_flow_id** è un numero n che identifica univocamente un flusso x. I curr_flow _id vengono assegnati ai flussi in ordine di tempo (di ingresso nella rete);

- **flows_state** rappresenta una lista di dizionari, dove ogni dizionario contiene tutte le caratteristiche di un flusso (*flow_id*, *in_port* , *ip_src*, *ip_dst*….eccetera). Tramite il flow_id è possibile individuare il singolo flusso (cioè il singolo dizionario);

- **active_flows** è una lista contenente i flow_id dei flussi attivi in rete;

- La lista **hipriousers** contiene gli indirizzi IP degli utenti più prioritari. Nel nostro caso un unico indirizzo;

- **flowlimit** è un valore che indica il numero massimo di flussi consentiti in non enforcement;

- Le variabili **C_state_prio**, **E_state_prio**, **N_state_prio**, identificano lo stato di un certo flusso.

Successivamente si procede al recupero dei valori noti presenti nella rete attraverso il file sdncInit.py che contiene gli indirizzi IP,MAC e le porte dello switch collegate alle virtual machines.

```
def __init__(self, *args, **kwargs):
    super(BasicOpenStackL3Controller, self).__init__(*args, **kwargs)
    self.dpset = kwargs['dpset'] #NOTE. dpset (argument of kwargs) is the name specified in the contexts
variable
    #VARIABLES
    self.switch_dpid_name = {} #Keep track of each switch by mapping dpid to name
    self.connections_name_dpid = {} #Keep track name and connection
    self.net_topo = {}
    #
    self.users = ['BusUser', 'ResUser'] # U
    self.net_func = ['DPI', 'TC', 'Wana', 'WanaDec', 'vr', 'vr_dest'] # NF
    #self.net_func = ['DPI', 'TC', 'Wana', 'WanaDec', 'gw', 'gw_dest', 'vr', 'vr_dest'] # NF
    #switch_ports = {'br-int1': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 'br4': [1, 2, 3], 'br3': [1, 2, 3]} # SWj
```

```
#self.int_network_vid = 9
#self.gw_network_vid = 10
# Not used??????
#self.sink = 3
#self.bcast = "ff:ff:ff:ff:ff:ff"

# Datapath ID
self.ovs_datapath_id = sdncInit.ovs_datapath_id
self.net_topo[self.ovs_datapath_id] = 'VDC1'
self.ovs_dest_datapath_id = sdncInit.ovs_dest_datapath_id
self.net_topo[self.ovs_dest_datapath_id] = 'VDC2'

# VM PORT DICTIONARY
self.outport = sdncInit.outport
self.outport_dest = sdncInit.outport_dest
self.bususer_port = sdncInit.bususer_port
self.resuser_port = sdncInit.resuser_port
self.dpi_port = sdncInit.dpi_port
self.tc_port = sdncInit.tc_port
self.wana_port = sdncInit.wana_port
self.wana_dest_port = sdncInit.wana_dest_port
self.sink_port = sdncInit.sink_port

# VM IP DICTIONARY
self.ip_bususer = sdncInit.ip_bususer
self.ip_resuser = sdncInit.ip_resuser
self.ip_dpi = sdncInit.ip_dpi
self.ip_nat_bu = sdncInit.ip_nat_bu
self.ip_nat_ru = sdncInit.ip_nat_ru
self.ip_sink = sdncInit.ip_sink
self.ip_nat_sink = sdncInit.ip_nat_sink
self.ip_sdnc = sdncInit.ip_sdnc
self.ip_nat_sdnc = sdncInit.ip_nat_sdnc

# VM MAC DICTIONARY
self.dpi_mac = sdncInit.dpi_mac
self.tc_mac = sdncInit.tc_mac
self.wana_mac = sdncInit.wana_mac
self.wana_dest_mac = sdncInit.wana_dest_mac

# DPI execution parameters
self.dpiExePath = '/home/ubuntu/nDPI_Tool/nDPI/example/ndpiReader'
self.dpiCapPath = '/tmp/ndpiresult_'
self.a = 0
self.b = 0

# HIGH PRIORITY USERS LIST
hipriousers.append(self.ip_bususer)

# JUST PLACEHOLDERS, LEAVE COMMENTED
#self.gw_port={'port1': , 'port2': }
#self.ip_gw = "10.10.0.1"
#self.ip_router_vr = "10.0.0.1"
```

```
        #self.gw_mac={'eth1':"fa:16:3e:f1:cc:7b", 'eth2':"fa:16:3e:a4:16:7a"}
        #self.DPIcredentials='ubuntu@192.168.122.64'
```

Analizzo ora le principali funzioni utilizzate dal controller.

La funzione *get_in_mac_address* consente di ottenere l'indirizzo MAC dell'interfaccia di rete relativa ad una virtual machine, specificando l'host e la direzione del flusso:

- Direzione *outbound* significa che il flusso è uscente rispetto agli utenti.
- Direzione *inbound* singifica che il flusso è entrante rispetto agli utenti.

```
def get_in_mac_address(self, host, direction):
        intf=None
        if direction == 'outbound':
            intf='eth1'
        elif direction == 'inbound':
            intf='eth2'
        if host =='DPI' :
            return self.dpi_mac[intf]
        elif host =='Wana' :
            return self.wana_mac[intf]
        elif host =='TC' :
            return self.tc_mac[intf]
#         elif host =='GW' :
#             return self.gw_mac[intf]
        elif host =='WanaDec' :
            return self.wana_dest_mac[intf]
#         elif host =='GWDest' :
#             return self.gw_dest_mac[intf]
```

Successivamente viene definita la funzione *get_in_port* che utilizza lo stesso concetto, ma che restituisce una lista contenente il nome dello switch (*VDC1-br-int* per gli host appartenenti al data center uno e *VDC2-br-int* per il nodo destinazione) ed il numero di porta relativi. I flussi outbound identificano la porta 1 mentre i flussi inbound la porta 2.

```
def get_in_port(self, host, direction):

    result_tuple = []

    if host =='DPI' :
        result_tuple.append('VDC1-br-int')
        if direction == 'outbound':
            result_tuple.append(self.dpi_port['port1'])
        elif direction == 'inbound':
            result_tuple.append(self.dpi_port['port2'])
    elif host =='Wana' :
```

```
            result_tuple.append('VDC1-br-int')
            if direction == 'outbound':
                result_tuple.append(self.wana_port['port1'])
            elif direction == 'inbound':
                result_tuple.append(self.wana_port['port2'])
        elif host =='TC' :
            result_tuple.append('VDC1-br-int')
            if direction == 'outbound':
                result_tuple.append(self.tc_port['port1'])
            elif direction == 'inbound':
                result_tuple.append(self.tc_port['port2'])
        elif host =='WanaDec' :
            result_tuple.append('VDC2-br-int')
            if direction == 'outbound':
                result_tuple.append(self.wana_dest_port['port1'])
            elif direction == 'inbound':
                result_tuple.append(self.wana_dest_port['port2'])
        return result_tuple
```

Un aspetto fondamentale del controller è la cancellazione delle regole relative ai flussi inattivi per mantenere la coerenza tra *flow_table* interna allo switch e *flows_state*.

Per fare ciò, il controller deve richiedere allo switch lo stato dei flussi e confrontare con una matching se i flussi nella *flow_table* coincidono con quelli presenti nella lista *flow_state.* La funzione *_request_stats* serve a creare il messaggio di OFPFlowStatsRequest, al quale lo switch risponderà con la propria flow_table.

```
# ANALIZE ACTIVE FLOWS REQUEST
    def _request_stats(self):
        self.logger.info('FLOW CONTROL request')
        datapath = self.connectionForBridge('VDC1-br-int')
        # Schedule next request
        threading.Timer(60.0, self._request_stats).start()
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)
```

Il controller ha bisogno del decoratore *set_ev_cls* per intercettare il messaggio contenente i flussi attivi nello switch e farne il matching con *flows_state.*

All’interno del decoratore viene quindi definita la funzione incaricata di confrontare le regole, *_flow_stats_reply_handler*.

```
# ANALIZE ACTIVE FLOWS REPLY
    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def _flow_stats_reply_handler(self, ev):
```

```
        global active_flows
        global classified_flows
        global flows_state
        body = ev.msg.body
        dp = ev.msg.datapath
        self.logger.info('FLOW CONTROL reply at
%s',datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%S'))
        for flow_id in active_flows :
            valid = 0
            for stat in sorted([flow for flow in body if flow.priority >= 34502 ],
                               key=lambda flow: (flow.priority)):
                #self.logger.info('[DEBUG] Flow stats: %s',stat)
                try:
                    if (stat.match['ipv4_src'] == flows_state[flow_id]['ipv4_src']) and (stat.match['ipv4_dst'] ==
flows_state[flow_id]['ipv4_dst']) and (stat.match['tcp_src'] == flows_state[flow_id]['port_src']) and
(stat.match['tcp_dst'] == flows_state[flow_id]['port_dst']):
                        valid = 1
                except:
                    pass
                try:
                    if (stat.match['ipv4_src'] == flows_state[flow_id]['ipv4_src']) and (stat.match['ipv4_dst'] ==
flows_state[flow_id]['ipv4_dst']) and (stat.match['udp_src'] == flows_state[flow_id]['port_src']) and
(stat.match['udp_dst'] == flows_state[flow_id]['port_dst']):
                        valid = 1
                except:
                    pass
                try:
                    if (stat.match['ipv4_dst'] == flows_state[flow_id]['ipv4_src']) and (stat.match['ipv4_src'] ==
flows_state[flow_id]['ipv4_dst']) and (stat.match['tcp_dst'] == flows_state[flow_id]['port_src']) and
(stat.match['tcp_src'] == flows_state[flow_id]['port_dst']):
                        valid = 1
                except:
                    pass
                try:
                    if (stat.match['ipv4_dst'] == flows_state[flow_id]['ipv4_src']) and (stat.match['ipv4_src'] ==
flows_state[flow_id]['ipv4_dst']) and (stat.match['udp_dst'] == flows_state[flow_id]['port_src']) and
(stat.match['udp_src'] == flows_state[flow_id]['port_dst']):
                        valid = 1
                except:
                    pass
            if valid == 1 :
                self.logger.info('flow %s is active', flow_id)
            elif valid == 0 :
                self.logger.info('flow %s is not active', flow_id)
                active_flows.remove(flow_id)
                self.logger.info('flow %s removed from active_flows', flow_id)
                #try:
                #    classified_flows.remove(flow_id)
                #    self.logger.info('flow %s removed from classified_flows', flow_id)
                #except:
                #    pass
        self.logger.info('[ %s ]: flows active: %s',datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-
%d %H:%M:%S'), active_flows)
```

```
        # ANALIZE IF CONGESTION ARISE
        if len(active_flows) <= flowlimit :
            self.logger.info('Non-Enforcement State')
            self._handle_NonEnforcement_N_State()
        else :
            self.logger.info('WARNING! - Too many active flows: Enforcement State')
            self._handle_Enforcement_E_State()
```

La parte finale della funzione serve a controllare se il numero di flussi attivi presenti nella rete è minore del flowlimit (impostato a 2). Se il numero di flussi attivi è maggiore la rete entra in enforcement, altrimenti in non enforcement. Questo è fondamentale in quanto permette di far passare i flussi degli utenti non prioritari, se il numero di flussi è minore del flow limit, dallo stato di enforcement allo stato di non enforcement quando il business user smette di generare traffico.

A questo punto vengono definite le funzioni relative al DPI. Tali funzioni hanno lo scopo di far cominciare l'analisi del DPI ed una volta conclusa leggerne l'output contenente la classificazione del flusso considerato, che viene generato in formato JSON.

```
# DPI FUNCTIONS
    def startDPI(self, ip_src, f_id):
        os.system("ssh ubuntu@%s 'sudo nohup %s -i eth1 -f \"ip host %s\" -v 2 -s 10 -j %s%s-%s.json  > foo.out
2> foo.err < /dev/null & '" % (self.ip_dpi, self.dpiExePath, ip_src, self.dpiCapPath, ip_src, f_id))
        self.logger.info("[CONTROLLER - TIMER (%s)] DPI started for IP address %s\n",
datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%S'), ip_src)



def obtain_DPI_output(self,flow_id):
        #return subprocess.check_output(['ssh', 'ubuntu@10.15.0.1', '-p', '44444', 'sudo cat %s' % self.dpiCapPath ])
        result = None
        try:
            result = subprocess.check_output(['ssh', 'ubuntu@%s' % self.ip_dpi, 'sudo cat %s%s-%s.json' %
(self.dpiCapPath, flows_state[flow_id]['ipv4_src'], flow_id) ])
        except subprocess.CalledProcessError as e:
            result = e.output
        return result



#function executed after DPI analysis
    def dpi_analysis_finished(self, flow_id):
            switch_port = ['None', 'None']
        ## Step 1: stop DPI
        #self.stopDPI()
        #time.sleep(0.15)
```

```
        self.logger.info("[DEBUG - READING JSON FILE FOR FLOW ID %d] ", flow_id)
        # Step 2: read json output file (DPI classification)
        json_data = self.obtain_DPI_output(flow_id)
        #self.logger.info("[DEBUG - JSON]\n %s \n", 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)
                self.logger.info("[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'])

                # UPDATING FLOW INFORMATION
                if host_a == self.ip_bususer or host_a == self.ip_resuser or host_b == self.ip_bususer or host_b ==
self.ip_resuser:
                    if i['protocol'] == "TCP":
                        ip_prot = inet.IPPROTO_TCP
                    elif i['protocol'] == "UDP":
                        ip_prot = inet.IPPROTO_UDP
                    self._memFlow(flow_id,switch_port, ipv4_src=host_a, port_src=port_a, ipv4_dst=host_b,
port_dst=port_b, ip_proto=ip_prot)
                    time.sleep(0.10)
```

L’ultima parte della funzione *dpi_analysis_finished* ha lo scopo di aggiornare la lista flows_state, tramite la funzione _memFlow che viene definita di seguito.

#Prima parte

```
def _memFlow(self, flow_id, switch_port ,msg = None, mac_addr = {} , ipv4_src="0.0.0.0", port_src=0,
ipv4_dst="0.0.0.0", port_dst=0, in_port=-1, ip_proto=0, state="X") :
        tmp_dict_opts = {}
        tmp_list_actions = []
        tmp_dict_match = {}
        tmp_dict_actions = {}
        global flows_state
        global active_flows
        global rules_list

        if flow_id in active_flows :
            if ipv4_src != "0.0.0.0" :
                flows_state[flow_id]['ipv4_src'] = ipv4_src
             if port_src != 0 :
```

```
            flows_state[flow_id]['port_src'] = port_src
        if ipv4_dst != "0.0.0.0" :
            flows_state[flow_id]['ipv4_dst'] = ipv4_dst
        if port_dst != 0 :
            flows_state[flow_id]['port_dst'] = port_dst
        if in_port != -1 :
            flows_state[flow_id]['in_port'] = in_port
        if ip_proto != 0 :
            flows_state[flow_id]['ip_proto'] = ip_proto
        if state != "X" :
            flows_state[flow_id]['state'] = state
        if rules_list != [] :
            flows_state[flow_id]['rules'] = rules_list
    else:  #NEW FLOW
        flow={}
        flow['flow_id'] = flow_id
        flow['ipv4_src'] = ipv4_src
        flow['port_src'] = port_src
        flow['ipv4_dst'] = ipv4_dst
        flow['port_dst'] = port_dst
        flow['in_port'] = in_port
        flow['ip_proto'] = ip_proto
        flow['state'] = state
        flow['rules'] = rules_list
        if ipv4_src in hipriousers :
            flow['ip_nat']=self.ip_nat_bu
        else:
            flow['ip_nat']=self.ip_nat_ru
        flows_state.append(flow)
        active_flows.append(flow_id)
```

#Seconda parte

```
    # Add the previous rule to internal memory
    if msg is not None:        # Options
        if msg.idle_timeout == 0 :    #default value
            tmp_dict_opts['hard_timeout'] = msg.hard_timeout
        else:
            tmp_dict_opts['idle_timeout'] = msg.idle_timeout
        tmp_dict_opts['priority'] = msg.priority
       # print(msg.idle_timeout)
    # Matching rule
        tmp_dict_match['in_port'] = msg.match['in_port']
        tmp_dict_match['eth_type'] = msg.match['eth_type']
        tmp_dict_match['ip_proto'] = msg.match['ip_proto']
        tmp_dict_match['ipv4_src'] = msg.match['ipv4_src']
        tmp_dict_match['ipv4_dst'] = msg.match['ipv4_dst']
        if msg.match['ip_proto'] == inet.IPPROTO_TCP: # TCP CASE
            tmp_dict_match['port_src'] = msg.match['tcp_src']
            tmp_dict_match['port_dst'] = msg.match['tcp_dst']
        else: # UDP CASE
            tmp_dict_match['port_src'] = msg.match['udp_src']
```

```
        tmp_dict_match['port_dst'] = msg.match['udp_dst']
    # Actions
        #  if flows_state[]['ipv4_src'] in hipriousers :   #HIGH PRIORITY CASE
          if not mac_addr :
          #self.logger.info("need to change mac address, bus user")
             switch_port.append('OFPP_NORMAL')
          else:
          #self.logger.info("no change bus user")
             tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
             tmp_dict_actions['dl_addr'] = mac_addr # string format: remember to convert to EthAddr()
             tmp_list_actions.append(tmp_dict_actions) # Add actions to the list of actions
             tmp_dict_actions.clear()

          tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
          tmp_dict_actions['port'] = switch_port[1]
          tmp_list_actions.append(tmp_dict_actions) # Add actions to the list of actions
          tmp_dict_actions.clear()
    #    else:
        #LOW PRIORITY CASE
    #       if not mac_addr :
          #self.logger.info("need to change mac address, res user")

        #              switch_port.append('OFPP_NORMAL')
        #       else:
           #self.logger.info("no change res user")
        #              tmp_dict_actions['type'] = 'OFPAT_SET_DL_DST'
    #          tmp_dict_actions['dl_addr'] = mac_addr # string format: remember to convert to EthAddr()
    #          tmp_list_actions.append(tmp_dict_actions) # Add actions to the list of actions
    #          tmp_dict_actions.clear()

        #tmp_dict_actions['type'] = 'OFPAT_OUTPUT'
        #tmp_dict_actions['port'] = switch_port[1]
        #tmp_list_actions.append(tmp_dict_actions) # Add actions to the list of actions
        #tmp_dict_actions.clear()


        rule = {'switch': switch_port[0], 'op': 'ADD', 'options': tmp_dict_opts, 'match': tmp_dict_match, 'actions':
tmp_list_actions}
        rules_list.append(rule)
```

Questa funzione è divisa in due parti:

- La prima parte si occupa dell'aggiornamento della lista flows_state. Se il flow_id relativo ad un certo flusso è presente nella lista active_flows, significa che il flusso è già noto per cui l'unica operazione che si deve fare è cambiare o no qualche campo, ad esempio un cambio di stato da C (classificazione) ad N (non enforcement). Altrimenti si deve aggiungere alla flows_state un nuovo dizionario contenente tutte le informazioni note di tale flusso.

- La seconda parte si occupa del matching dei valori relativi ad un certo flusso in modo da salvare le regole anche nella memoria interna. Viene per cui creato un dizionario *rule*, contenente i valori presenti dentro il *msg* (il pacchetto da mandare allo switch per salvare le regole nella flow_table). Questo rule viene poi aggiunto alla lista *rules_list* contente tutte le rule di tutti i flussi attivi. La rules_list viene poi aggiunta alla flows_state.

Per ogni flusso vengono salvate le seguenti informazioni:

- **Flow_id**;
- **Ipv4_src**, cioè l'indirizzo IP sorgente;
- **Ipv4_dst**, indirizzo IP destinazione;
- **port_src**, porta sorgente;
- **port_dst**, porta di destinazione;
- **ip_proto**, numero che identifica il tipo di protocollo IP;
- **state**, rappresenta lo stato dell'automa associato al flusso considerato;
- **rules**, lista delle regole associate al flusso considerato, installate nella flow_table dello switch;
- **ip_nat**, indirizzo *floating IP* , viene assegnato alle virtual machines degli utenti, ed alla virtual machine del controller SDN, permette di installare regole su dispositivi posti a monte del NAT.

Questa funzione risulta molto importante in quanto, viene richiamata all'interno di ogni stato dell'automa per ogni singolo flusso per procedere al salvataggio della regola sulla memoria interna.

Un'importante novità presente all'interno di questo codice è la funzione *flow_removed_handler*:

#Handle FlowRemoved event

```
@set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
def flow_removed_handler(self, ev):
    self.logger.info('[DEBUG] FLOW_REMOVED_HANDLER')
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    self.logger.info('msg.reason = %d',msg.reason)
    if msg.reason == ofp.OFPRR_IDLE_TIMEOUT:
        reason = 'IDLE TIMEOUT'
    elif msg.reason == ofp.OFPRR_HARD_TIMEOUT:
        reason = 'HARD TIMEOUT'
    elif msg.reason == ofp.OFPRR_DELETE:
        reason = 'DELETE'
    elif msg.reason == ofp.OFPRR_GROUP_DELETE:
        reason = 'GROUP DELETE'
    else:
        reason = 'unknown'

    self.logger.debug('OFPFlowRemoved received: '
                      'cookie=%d priority=%d reason=%s table_id=%d '
                      'duration_sec=%d duration_nsec=%d '
                      'idle_timeout=%d hard_timeout=%d '
                      'packet_count=%d byte_count=%d match.fields=%s',
                      msg.cookie, msg.priority, reason, msg.table_id,
                      msg.duration_sec, msg.duration_nsec,
                      msg.idle_timeout, msg.hard_timeout,
                      msg.packet_count, msg.byte_count, msg.match)
    self._request_stats()
```

Questa funzione, che viene invocata ogni volta che un flusso viene rimosso dalla flow_table, ha il compito di segnalare il motivo della rimozione tramite l'attributo di msg, *reason* e di conseguenza invocare la funzione *_request_stats*, spiegata precedentemente, per mantenere la coerenza tra flow_table dello switch e regole salvate nella memoria interna.

Infine è presente la funzione incaricata di gestire i pacchetti in ingresso, *_packet_in_handler*. Il concetto di base è lo stesso della funzione vista nel capitolo 3, chiaramente in questo caso sono presenti ulteriori operazioni interne.

```
#Handle PacketIn event
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    global curr_flow_id
    self.logger.info("[DEBUG] NEW FLOW!!!!!!!")
    msg = ev.msg
    pkt = packet.Packet(msg.data)

    datapath = msg.datapath
```

```
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        self.logger.info("[DEBUG] datapath = %s in_port=%d", self.net_topo[datapath.id], msg.match['in_port'])
        eth_header = pkt.get_protocol(ethernet.ethernet)
        self.logger.info("[DEBUG] ethertype=0x%04x mac_src=%s mac_dst=%s", eth_header.ethertype,
eth_header.src, eth_header.dst)
        header = pkt.get_protocol(ipv4.ipv4)
        self.logger.info("[DEBUG] ip_proto=%s ip_src=%s ip_dst=%s", header.proto, header.src, header.dst)
        if header.proto == inet.IPPROTO_UDP:
            udpheader = pkt.get_protocol(udp.udp)
            self.logger.info("[DEBUG] udp.src_port=%s udp.dst_port=%s", udpheader.src_port, udpheader.dst_port)
        elif header.proto == inet.IPPROTO_TCP:
            tcpheader = pkt.get_protocol(tcp.tcp)
            self.logger.info("[DEBUG] tcp.src_port=%s tcp.dst_port=%s", tcpheader.src_port, tcpheader.dst_port)

        # Normal forwarding if packet is not involving sink
        if (header.dst != self.ip_nat_sink) and (header.src != self.ip_nat_sink):
            self.logger.info("[DEBUG] Flow not used for testing: use normal forwarding")
            eth = pkt.get_protocols(ethernet.ethernet)[0]
            if eth.ethertype == ether.ETH_TYPE_IP:
                match = parser.OFPMatch(eth_type=eth.ethertype, ip_proto=header.proto, ipv4_src=header.src,
ipv4_dst=header.dst)
            elif eth.ethertype == ether.ETH_TYPE_8021Q:
                vltag = pkt.get_protocols(vlan.vlan)[0]
                if vltag.ethertype == ether.ETH_TYPE_IP:
                    match = parser.OFPMatch(eth_type=vltag.ethertype, ip_proto=header.proto, ipv4_src=header.src,
ipv4_dst=header.dst)
                else:
                    match = parser.OFPMatch(eth_type=vltag.ethertype)
            else:
                match = parser.OFPMatch(eth_type=eth.ethertype)
            actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
            self.add_flow(datapath, 0, 33000, match, actions)
        # Otherwise, start finite-state machine processing
        else:
            flow_id = curr_flow_id;
            curr_flow_id = curr_flow_id + 1;
            self.logger.info("[DEBUG] Flow to be processed with ID = %s", flow_id)
            np = len(pkt.get_protocols(ethernet.ethernet))
            self.logger.info("[PKT-HANDLER]  (%s) Number of detected protocols: %d",
datetime.datetime.fromtimestamp(time.time()).strftime('%Y-%m-%d %H:%M:%S'), np)
            if np == 1:

                self._handle_Classification_C_State(flow_id, msg)

                # Wait 15 seconds for DPI to finish
                self.logger.info("Waiting for DPI...")
                time.sleep(15.0)
                self.logger.info("Done!")
                self.dpi_analysis_finished(flow_id)

                if flows_state[flow_id]['ipv4_src'] in hipriousers:
```

```
            self.logger.info("******* BU *******")
            self._handle_Enforcement_E_State()
        else:
            self.logger.info("******* RU *******")
            self._handle_NonEnforcement_N_State(flow_id)



        # Move new flow to enforcement state
        #self._handle_Enforcement_E_State(flow_id)

        # FLOW CONTROL STARTS WITH FIRST FLOW
        #time.sleep(15.0)
        #if flow_id == 0 :
        #    self._request_stats()

    else:
        self.logger.info("More than one protocol detected")
```

Ogni volta che un nuovo flusso attraversa lo switch, la funzione di packet_in viene invocata.

La parte iniziale è di debug e serve per mostrare a schermo le caratteristiche del flusso, dopodiché si controlla se i pacchetti coinvolgono il nodo *sink* (cioè il nodo destinatario) oppure no, ad esempio nel caso siano pacchetti DHCP o ARP.

Se il flusso non è diretto al sink si impone il normale forwarding tramite il comando actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)], altrimenti si associa il flusso allo stato di classificazione, invocando la funzione self._handle_Classification_C_State(flow_id, msg) e si attendono 15 secondi per l'analisi del DPI.

Ad analisi conclusa, se l'indirizzo IP sorgente appartiene alla lista *hipriousers*, tutti i flussi entrano nello stato di enforcement grazie alla funzione self._handle_Enforcement_E_State, altrimenti il flusso analizzato entra nello stato di non enforcement tramite self._handle_NonEnforcement_N_State.

Con questa funzione termina la prima parte del codice.

Nei sotto paragrafi successivi vengono analizzate le funzioni che hanno il compito di implementare la macchina a stati SDN descritta nel precedente capitolo.

5.1.1 Init

In questo primo stato vengono aggiunte alla flow_table tramite la funzione *_add_flow* (la stessa spiegata a pagina 25 del capitolo 3), le regole inerenti ai protocolli ARP, ICMP,DHCP, DNS ed SSH.

Queste regole vengono installate durante questa fase perché valgono in generale e sono indipendenti dai flussi diretti al sink.

5.1.2 Classification

Come è già stato anticipato, questa fase viene invocata all'interno della funzione *_packet_in_handler*.

Questo stato serve a creare le regole di classificazione del flusso, che dovrà essere reindirizzato al DPI per ottenerne le caratteristiche. In particolare la regola presenterà nel campo *state* la stringa "C" che sta proprio ad indicare questo stato.

All'interno della funzione vengono trattati distintamente i possibili flussi, in modo da installare regole dedicate a seconda del caso. Per esempio di seguito viene mostrata la parte di codice che si occupa di creare e salvare la regola relativa al flusso diretto dall'user al DPI:

```
# VDC1-br-int internal network, outbound traffic, from User to DPI, MAC_DST is changed
        switch_port = []
        action=[]
        switch_port = self.get_in_port('DPI', 'outbound')  #2 elements vector, at 0 switchname, 1 port
        mac_addr = self.get_in_mac_address('DPI', 'outbound')
        dp = self.connectionForBridge(switch_port[0])
        ofproto = dp.ofproto
        parser = dp.ofproto_parser
        action.append( parser.OFPActionSetField(eth_dst = mac_addr) ) # Change MAC_DST because packets
must go through...
        action.append( parser.OFPActionOutput(switch_port[1]) ) #...DPI
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, action)]
        if pkt_tcp: # TCP CASE
            msg = parser.OFPFlowMod(datapath = dp, hard_timeout=270, priority=C_state_prio, match =
parser.OFPMatch(in_port = coming_port, eth_type = ether.ETH_TYPE_IP, ip_proto = pkt_ipv4.proto, ipv4_src
= nw_src, ipv4_dst = nw_dst, tcp_src = pkt_tcp.src_port, tcp_dst = pkt_tcp.dst_port), instructions = inst)
        else: # UDP CASE
            msg = parser.OFPFlowMod(datapath = dp, hard_timeout=270, priority=C_state_prio, match =
parser.OFPMatch(in_port = coming_port, eth_type = ether.ETH_TYPE_IP, ip_proto = pkt_ipv4.proto, ipv4_src
= nw_src, ipv4_dst = nw_dst, udp_src = pkt_udp.src_port, udp_dst = pkt_udp.dst_port), instructions = inst)
        self.logger.info(action)
            # self.logger.info('[DEBUG] Matching fields for E state :in_port: %s ,eth_type: %s ,ip_proto: %s'
```

```
        #                ' ,ip_src:  %s ,ip_dst:  %s tp_src: %s, tp_dst: %s,command: %s ',
str(flows_state[flow]['in_port']), str(ether.ETH_TYPE_IP), str(flows_state[flow]['ip_proto']),
        #                str(flows_state[flow]['ipv4_src']), str(flows_state[flow]['ipv4_dst']),
str(flows_state[flow]['port_src']),str(flows_state[flow]['port_dst']),str(msg.command))
        dp.send_msg(msg)

        self._memFlow(f_id,switch_port,msg = msg,mac_addr= mac_addr, in_port = coming_port, ipv4_src =
nw_src, ipv4_dst = nw_dst, state = 'C')
        self.logger.info('[DEBUG] C-state: %d OpenFlow rules added for flow %d', len(rules_list), f_id)
```

Inizialmente vengono definite le liste vuote *switch_port* e *action*: all'interno della prima, tramite la funzione *get_in_port* si inseriscono il nome dello switch relativo (VDC1-br-int o VDC2-br-int) e la porta associata, mentre la seconda dovrà contenere l'azione da applicare al flusso, in questo caso il cambio dell'indirizzo MAC di destinazione, per indirizzare i pacchetti verso il DPI.

Successivamente tramite OFPFlowMod si crea il messaggio da mandare allo switch contenente la regola, e mediante il metodo *send_msg* si spedisce. Infine si invoca la funzione *_memFlow* per salvare la regola anche all'interno della flows_state.

Le regole di classificazione vengono installate con un *hard_timeout*, ciò significa che dopo il periodo indicato vengono eliminate dallo switch.

5.1.3 Non-Compliant

Questo stato viene invocato qualora un flusso venga definito non conforme e quindi debba essere scartato. Si procede dunque alla creazione della regola di drop:

```
# packet dropping rules
    switch_port = []
    action=[]
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, action)]

    if flows_state[flow_id]['ip_proto'] == inet.IPPROTO_TCP: # TCP CASE
        msg = parser.OFPFlowMod(datapath = datapath, priority = 30000, match = parser.OFPMatch(in_port =
flows_state[flow_id]['in_port'], eth_type = ether.ETH_TYPE_IP, ip_proto = inet.IPPROTO_TCP, ipv4_src =
flows_state[flow_id]['ipv4_src'], ipv4_dst = flows_state[flow_id]['ipv4_dst'], tcp_src =
flows_state[flow_id]['port_src'], tcp_dst = flows_state[flow_id]['port_dst'] ), instructions = inst)
    elif flows_state[flow_id]['ip_proto'] == inet.IPPROTO_UDP: # UDP CASE
        msg = parser.OFPFlowMod(datapath = datapath, priority = 30000, match = parser.OFPMatch(in_port =
flows_state[flow_id]['in_port'], eth_type = ether.ETH_TYPE_IP, ip_proto = inet.IPPROTO_UDP, ipv4_src =
flows_state[flow_id]['ipv4_src'], ipv4_dst = flows_state[flow_id]['ipv4_dst'], udp_src =
flows_state[flow_id]['port_src'], udp_dst = flows_state[flow_id]['port_dst'] ), instructions = inst)
```

```
datapath.send_msg(msg)

# SAVING CURRENT FLOW IN flows_state
self._memFlow(flow_id,switch_port,msg=msg , state = 'D')
```

Come si può notare la regola viene creata e salvata seguendo gli stessi passaggi visti per la funzione di classificazione, ma in questo caso lo stato è “D” e la lista *action* resta vuota.

5.1.4 Enforcement

La funzione relativa a tale stato viene invocata un volta che è stato classificato un flusso appartenente al Business user.

Prima di tutto viene fatto un controllo dello stato precedente del flusso:

```
if flows_state[flow]['state'] == 'N' or flows_state[flow]['state'] == 'C':
        if flows_state[flow]['state'] == 'N':
            command = ofproto_v1_3.OFPFC_MODIFY_STRICT
            command1= ofproto_v1_3.OFPFC_ADD
            self.logger.info('[DEBUG]MODIFY FLOWS' )
        else:
            command = ofproto_v1_3.OFPFC_ADD
            self.logger.info('[DEBUG]ADD FLOWS')
        if flows_state[flow]['ipv4_src'] in hipriousers :   #HIGH PRIORITY CASE
            vnf='Wana'
        else:
            vnf='TC'
```

Se il flusso ha come stato “N” significa che sono già presenti delle regole relative ad esso all’interno della flow_table, per cui si devono modificare le regole già presenti per lo stato di non enforcemet tramite il comando OFPFC_MODIFY_STRICT ed aggiungerne altre, in quanto le regole di non enforcement sono in numero minore rispetto alle regole di enforcement.

Al contrario lo stato “C” implica che il flusso è appena stato classificato e non sono presenti regole (a parte quelle di classificazione), per cui si dovranno aggiungere alla flow_table nuove regole mediante il comando .OFPFC_ADD.

Il secondo if serve a distinguere il caso prioritario, cioè in cui la vnf (*virtual network function*) è il WANA, dal meno prioritario in cui il flusso va reindirizzato al TC.

La creazione della regola è similare a quelle già viste in precedenza:

```
# VDC1-br-int internal network, outbound traffic, from User to VNF (Wana or TC), MAC_DST is changed
```

```
            switch_port = []
            action = []
            switch_port = self.get_in_port(vnf, 'outbound') # 2 elemnts vector, at 0 switchname, 1 port
            mac_addr = self.get_in_mac_address(vnf, 'outbound')
            dp = self.connectionForBridge(switch_port[0])
            ofproto = dp.ofproto
            parser = dp.ofproto_parser
            action.append( parser.OFPActionSetField(eth_dst = mac_addr) ) # Change MAC_DST because
packets must go through ...
            action.append( parser.OFPActionOutput(switch_port[1]) ) #...VNF
            inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, action)]
            if flows_state[flow]['ip_proto'] == inet.IPPROTO_TCP: # TCP CASE
               msg = parser.OFPFlowMod(datapath = dp, idle_timeout=10, priority=E_state_prio,
match=parser.OFPMatch( in_port = flows_state[flow]['in_port'], eth_type = ether.ETH_TYPE_IP,
ip_proto=flows_state[flow]['ip_proto'], ipv4_src = flows_state[flow]['ipv4_src'], ipv4_dst =
flows_state[flow]['ipv4_dst'], tcp_src = flows_state[flow]['port_src'], tcp_dst = flows_state[flow]['port_dst'] ),
instructions=inst, flags = ofproto.OFPFF_SEND_FLOW_REM, command = command )
               self.logger.info('[DEBUG] TCP case ')
            elif flows_state[flow]['ip_proto'] == inet.IPPROTO_UDP: # UDP CASE
               msg = parser.OFPFlowMod(datapath = dp, idle_timeout=10, priority=E_state_prio,
match=parser.OFPMatch( in_port = flows_state[flow]['in_port'], eth_type = ether.ETH_TYPE_IP,
ip_proto=flows_state[flow]['ip_proto'], ipv4_src = flows_state[flow]['ipv4_src'], ipv4_dst =
flows_state[flow]['ipv4_dst'], udp_src = flows_state[flow]['port_src'], udp_dst = flows_state[flow]['port_dst'] ),
instructions=inst, flags = ofproto.OFPFF_SEND_FLOW_REM, command = command )
               self.logger.info('[DEBUG] UDP case ')
            dp.send_msg(msg)
            self.logger.info('[DEBUG] Matching fields for E state :in_port: %s ,eth_type: %s ,ip_proto: %s'
                     ' ,ip_src:  %s ,ip_dst:  %s tp_src: %s, tp_dst: %s,command: %s ',
str(flows_state[flow]['in_port']), str(ether.ETH_TYPE_IP), str(flows_state[flow]['ip_proto']),
                     str(flows_state[flow]['ipv4_src']), str(flows_state[flow]['ipv4_dst']),
str(flows_state[flow]['port_src']),str(flows_state[flow]['port_dst']),str(msg.command))
            self.logger.info(action)
            self._memFlow(flow,switch_port,msg=msg,mac_addr=mac_addr, state = 'E')
```

Le regole che vengono installate in questo stato, al contrario dello stato di classificazione, presentano un *idle_timeout*, ciò implica che la regola venga eliminata dallo switch se il flusso resta inattivo per il periodo indicato dal timeout.

Per segnalare al controller l'eliminazione della regola è necessario includere nel messaggio il flag OFPFF_SEND_FLOW_REM.

All'interno di msg è presente anche il campo command che, a seconda dell'esito dell'if analizzato in precedenza, può assumere il valore ofproto_v1_3.OFPFC_MODIFY_STRICT oppure ofproto_v1_3.OFPFC_ADD. Il comando ADD è presente di default all'interno del msg (non andrebbe specificato il campo command) ed implica che venga creata una nuova regola, mentre se viene specificato il comando MODIFY_STRICT, va identificata

tramite matching la regola già presente e modificata in modo che diventi una regola di enforcement.

5.1.5 Non-Enforcement

L'ultima parte del codice riguarda le regole di non-enforcement.

Analogamente al caso di enforcement, inizialmente si utilizza un if per distinguere il caso in cui sia presente una regola di enforcement relativa al flusso considerato oppure esista solo la regola di classificazione.

```
if flows_state[flow]['state'] == 'E' or flows_state[flow]['state'] == 'C':
        if flows_state[flow]['state'] == 'E':
            command =ofproto_v1_3.OFPFC_MODIFY_STRICT
        else:
            command = ofproto_v1_3.OFPFC_ADD
```

In questo caso la modifica della regola avviene se lo stato precedente del flusso era "E".

In modo del tutti similare agli stati precedenti vi è la creazione delle regole, in questo caso, di non-enforcement, che avranno quindi come stato la stringa "N".

```
# VDC1-br-int internal network, outbound traffic (from User to VRF)
        switch_port = []
        switch_port.append('VDC1-br-int')
        dp = self.connectionForBridge(switch_port[0])
        ofproto = dp.ofproto
        parser = dp.ofproto_parser
        action = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, action)]
        if flows_state[flow]['ip_proto'] == inet.IPPROTO_TCP: # TCP CASE
            msg = parser.OFPFlowMod(datapath = dp, idle_timeout=0, priority=N_state_prio,
match=parser.OFPMatch( in_port = flows_state[flow]['in_port'], eth_type = ether.ETH_TYPE_IP,
ip_proto=flows_state[flow]['ip_proto'], ipv4_src = flows_state[flow]['ipv4_src'], ipv4_dst =
flows_state[flow]['ipv4_dst'], tcp_src = flows_state[flow]['port_src'], tcp_dst = flows_state[flow]['port_dst'] ),
instructions=inst, command = command   )
        elif flows_state[flow]['ip_proto'] == inet.IPPROTO_UDP: # UDP CASE
            msg = parser.OFPFlowMod(datapath = dp, idle_timeout=0, priority=N_state_prio,
match=parser.OFPMatch( in_port = flows_state[flow]['in_port'], eth_type = ether.ETH_TYPE_IP,
ip_proto=flows_state[flow]['ip_proto'], ipv4_src = flows_state[flow]['ipv4_src'], ipv4_dst =
flows_state[flow]['ipv4_dst'], udp_src = flows_state[flow]['port_src'], udp_dst = flows_state[flow]['port_dst'] ),
instructions=inst, command = command )
        dp.send_msg(msg)
        self.logger.info(action)
        self.logger.info('[DEBUG] Matching fields for N state :in_port: %s ,eth_type: %s ,ip_proto: %s'
                ' ,ip_src:  %s ,ip_dst:  %s tp_src: %s, tp_dst: %s,command: %s ',
str(flows_state[flow]['in_port']), str(ether.ETH_TYPE_IP), str(flows_state[flow]['ip_proto']),
```

```
                str(flows_state[flow]['ipv4_src']), str(flows_state[flow]['ipv4_dst']),
str(flows_state[flow]['port_src']),str(flows_state[flow]['port_dst']),str(msg.command))
            self._memFlow(flow,switch_port,msg=msg, state = 'N')
```

In questo esempio viene installata la regola che riguarda il flusso che va dall'user al router virtuale. Si può notare che non è necessario il cambio di indirizzo MAC, per cui l'azione da far compiere sarà ofproto.OFPP_NORMAL.

Termina così lo studio del codice del controller Ryu. Nel prossimo paragrafo vengono mostrati i test svolti sulla topologia.

5.2 Verifica di funzionamento

Per mettere in pratica l'esempio visto nel paragrafo 4.3 del capitolo precedente è stato utilizzato il comando *iperf* della Shell di Linux. Tramite questo comando è possibile creare una connessione TCP client server ad una porta specifica. Nel nostro caso il server è il nodo destinatario (DEST) mentre i client sono due residence users ed un business user (RU, RU, BU rispettivamente).

Espongo di seguito tutti i passaggi svolti:

- Prima di tutto si deve attivare il controller e collegarlo agli switch. Per fare ciò si deve invocare lo script *go.sh* ed attendere che il controller si configuri:

```
ubuntu@hcgw:~$ ./go.sh
loading app ryu_controller_l3_fsm_2016-03-02.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
instantiating app ryu_controller_l3_fsm_2016-03-02.py of BasicOpenStackL3Controller
instantiating app ryu.controller.ofp_handler of OFPHandler
[DEBUG] Found output port no. 17 for VDC2
[TIMER (2016-03-02 12:40:41)] [SC-HANDLER] Switch VDC2-br-int registered on the controller, datapath.id  =  b20d30cfc44a
[DEBUG] Found output port no. 37 for VDC1
[TIMER (2016-03-02 12:40:42)] [SC-HANDLER] Switch VDC1-br-int registered on the controller, datapath.id  =  b68db7143348
```

FIGURA 34: CONFIGURAZIONE CONTROLLER.

- Una volta che il controller è stato inizializzato si può mettere in ascolto il server, cioè il nodo destinazione. Per fare questo si mette in ascolto su tre porte diverse (5001, 5002, 5003) tramite il comando **iperf –s –p 5001/5002/5003**

- Inizialmente si fa entrare il primo residence user utilizzando il comando **iperf –t 180 –i 2 –c 10.250.0.117 –p 5001**, che consente di creare un canale TCP con la porta 5001 del nodo destinazione. Il residence user

all'inizio ha a disposizione la banda massima, ed il flusso entra nella fase di classificazione per 15 secondi. La flow table dello switch a questo punto contiene le regole installate in fase di inizializzazione e quelle relative alla classificazione dell'utente.

Per visualizzare la flow table relativa allo switch VDC1-br-int, bisogna collegarsi via SSH al compute1, cioè il compute del cluster OpenStack che contiene le virtual machine degli utenti, ed utilizzare il comando **sudo ovs-ofctl dump-flows br-int**:

```
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=65000,tcp,nw_src=192.168.10.59,tp_src=6633 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=32000,ip,in_port=30,nw_dst=192.168.9.1 actions=drop
 cookie=0x0, duration=2.814s, table=0, n_packets=26, n_bytes=3907, idle_age=17,
priority=32000,udp,tp_src=53 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=124, n_bytes=21299, idle_age=5,
priority=32000,tcp,tp_src=22 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=32000,icmp actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=32000,udp,tp_src=68,tp_dst=67 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=32000,udp,tp_src=67,tp_dst=68 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=22, n_bytes=924, idle_age=2,
priority=32000,arp actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=139, n_bytes=23037, idle_age=5,
priority=32000,tcp,tp_dst=22 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=26, n_bytes=2220, idle_age=17,
priority=32000,udp,tp_dst=53 actions=NORMAL
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=67,
priority=65000,tcp,nw_dst=192.168.10.59,tp_dst=6633 actions=NORMAL
 cookie=0x0, duration=27.468s, table=0, n_packets=409078, n_bytes=27194592,
hard_timeout=270, idle_age=27,
priority=34500,tcp,in_port=37,vlan_tci=0x1000/0x1000,nw_src=10.250.0.117,nw_dst
=192.168.8.8,tp_src=5001,tp_dst=52022
actions=strip_vlan,mod_dl_dst:fa:16:3e:ad:81:3b,output:24
 cookie=0x0, duration=2.814s, table=0, n_packets=2, n_bytes=148, idle_age=43,
priority=2 actions=CONTROLLER:65535
 cookie=0x0, duration=2.814s, table=0, n_packets=0, n_bytes=0, idle_age=70,
priority=1 actions=NORMAL
 cookie=0x0, duration=27.468s, table=0, n_packets=25316, n_bytes=1176351052,
```

hard_timeout=270, idle_age=27,
priority=34500,tcp,in_port=24,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=520
22,tp_dst=5001 actions=NORMAL
cookie=0x0, duration=27.468s, table=0, n_packets=419288, n_bytes=27868452,
hard_timeout=270, idle_age=27,
priority=34500,tcp,in_port=23,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
1,tp_dst=52022 actions=NORMAL
cookie=0x0, duration=27.468s, table=0, n_packets=24867, n_bytes=1147364510,
hard_timeout=270, idle_age=27,
priority=34500,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=520
22,tp_dst=5001 actions=mod_dl_dst:fa:16:3e:b4:c5:58,output:23
cookie=0x0, duration=2.814s, table=23, n_packets=0, n_bytes=0, idle_age=70,
priority=0 actions=drop

Le regole di classificazione hanno priorità 34500, in questo modo si possono distinguere dalle altre, per semplicità le ho sottolineate. In particolare si nota la presenza delle due regole di reindirizzamento del flusso (in una caso *outbound* e nell'altro *inbound*) alle porte 23 e 24 dello switch, collegate al DPI.

- Dopo la classificazione vengono installate le regole relative allo stato di non enforcement per il residence user:

cookie=0x0, duration=36.854s, table=0, n_packets=7750, n_bytes=399347166,
idle_age=32,
priority=34502,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=520
22,tp_dst=5001 actions=NORMAL

cookie=0x0, duration=36.854s, table=0, n_packets=141461, n_bytes=9354762,
idle_age=32,
priority=34502,tcp,in_port=37,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
1,tp_dst=52022 actions=NORMAL

- Si connette ora anche il secondo residence tramite l'iperf alla porta 5002 di DEST. La flow table conterrà oltre alla regole precedenti, anche le regole di classificazione del secondo utente best effort, che riporto di seguito:

cookie=0x0, duration=4.706s, table=0, n_packets=6980, n_bytes=143567058,
hard_timeout=270, idle_age=0,

priority=34500,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=411
40,tp_dst=5002 actions=mod_dl_dst:fa:16:3e:b4:c5:58,output:23

cookie=0x0, duration=4.706s, table=0, n_packets=51938, n_bytes=3448040,
hard_timeout=270, idle_age=0,
priority=34500,tcp,in_port=23,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
2,tp_dst=41140 actions=NORMAL

cookie=0x0, duration=4.706s, table=0, n_packets=6980, n_bytes=143567058,
hard_timeout=270, idle_age=0,
priority=34500,tcp,in_port=24,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=411
40,tp_dst=5002 actions=NORMAL

cookie=0x0, duration=4.706s, table=0, n_packets=51938, n_bytes=3448040,
hard_timeout=270, idle_age=0,
priority=34500,tcp,in_port=37,vlan_tci=0x1000/0x1000,nw_src=10.250.0.117,nw_dst
=192.168.8.8,tp_src=5002,tp_dst=41140
actions=strip_vlan,mod_dl_dst:fa:16:3e:ad:81:3b,output:24

- In analogia con il caso precedente vengono installate all'interno dello switch anche le regole di non enforcement per il secondo residence user:

cookie=0x0, duration=2.772s, table=0, n_packets=3519, n_bytes=106980534,
idle_age=0,
priority=34502,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=411
40,tp_dst=5002 actions=NORMAL

cookie=0x0, duration=2.772s, table=0, n_packets=37942, n_bytes=2504172,
idle_age=0,
priority=34502,tcp,in_port=37,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
2,tp_dst=41140 actions=NORMAL

- A questo punto si connette l'utente business, sfruttando la porta 5003 del DEST. Inizialmente vengono installate regole di classificazione, come nei casi precedenti, ma successivamente, ad avvenuta classificazione, le precedenti regole di non enforcement relative ai residence vengono modificate e trasformate in regole di enforcement e vengono aggiunte le regole di enforcement del utente business:

cookie=0x0, duration=2.598s, table=0, n_packets=38587, n_bytes=2549222,
idle_timeout=10, idle_age=0,
priority=34502,tcp,in_port=37,vlan_tci=0x1000/0x1000,nw_src=10.250.0.117,nw_dst
=192.168.8.80,tp_src=5003,tp_dst=43176
actions=strip_vlan,mod_dl_dst:fa:16:3e:6d:b1:33,output:30

cookie=0x0, duration=109.675s, table=0, n_packets=7750, n_bytes=399347166,
idle_age=105,
priority=34502,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=520
22,tp_dst=5001 actions=NORMAL

cookie=0x0, duration=2.598s, table=0, n_packets=4502, n_bytes=115261132,
idle_timeout=10, idle_age=0,
priority=34502,tcp,in_port=21,nw_src=192.168.8.80,nw_dst=10.250.0.117,tp_src=43
176,tp_dst=5003 actions=mod_dl_dst:fa:16:3e:cd:3a:a8,output:29

cookie=0x0, duration=2.598s, table=0, n_packets=6758, n_bytes=16263346,
idle_timeout=10, idle_age=0,
priority=34502,tcp,in_port=27,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=520
23,tp_dst=5001 actions=NORMAL

cookie=0x0, duration=2.598s, table=0, n_packets=35319, n_bytes=2333534,
idle_timeout=10, idle_age=0,
priority=34502,tcp,in_port=29,nw_src=10.250.0.117,nw_dst=192.168.8.80,tp_src=50
03,tp_dst=43176 actions=NORMAL

cookie=0x0, duration=30.099s, table=0, n_packets=329332, n_bytes=21983816,
idle_age=0,
priority=34502,tcp,in_port=37,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
2,tp_dst=41140 actions=NORMAL

cookie=0x0, duration=30.099s, table=0, n_packets=48976, n_bytes=919619108,
idle_age=0, hard_age=2,
priority=34502,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=4114
0,tp_dst=5002 actions=mod_dl_dst:fa:16:3e:cb:9e:04,output:26

cookie=0x0, duration=2.598s, table=0, n_packets=4026, n_bytes=106069048,
idle_timeout=10, idle_age=0,
priority=34502,tcp,in_port=30,nw_src=192.168.8.80,nw_dst=10.250.0.117,tp_src=43
176,tp_dst=5003 actions=NORMAL

cookie=0x0, duration=60.442s, table=0, n_packets=1116181, n_bytes=73919026,
idle_age=0,
priority=34502,tcp,in_port=37,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=500
1,tp_dst=52023 actions=NORMAL

cookie=0x0, duration=109.675s, table=0, n_packets=141461, n_bytes=9354762, idle_age=105, priority=34502,tcp,in_port=37,nw_src=10.250.0.117,nw_dst=192.168.8.8,tp_src=5001,tp_dst=52022 actions=NORMAL

cookie=0x0, duration=2.598s, table=0, n_packets=7121, n_bytes=14934084, idle_timeout=10, idle_age=0, priority=34502,tcp,in_port=27,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=41140,tp_dst=5002 actions=NORMAL

cookie=0x0, duration=60.442s, table=0, n_packets=93628, n_bytes=3153389790, idle_age=0, hard_age=2, priority=34502,tcp,in_port=20,nw_src=192.168.8.8,nw_dst=10.250.0.117,tp_src=52023,tp_dst=5001 actions=mod_dl_dst:fa:16:3e:cb:9e:04,output:26

Ho riportato solo le regole relative allo stato di enforcement, per evitare troppa ridondanza. Le regole che sono state modificate si possono notare guardando il campo *duration*, un timer che indica da quanto tempo una certa regola è stata installata nella flow table. Le regole appena aggiunte hanno un valore di duration di pochi secondi, mentre le regole modificate un valore più elevato.

- L'ultimo step consiste nell'interrompere il traffico dati dell'utente business ed attendere 10 secondi (valore dell'*idle_timeout*), che è il tempo necessario affinchè le regole di enforcement relative ad esso vengano eliminate dalla flow table per inattività del flusso.
A questo punto il controller, utilizzando le funzioni viste nel paragrafo precedente, confronta la flow table con la lista flows_state, si accorge che il flusso business non è presente nello switch, per cui elimina il flow_id relativo ad esso dalla lista active_flows e fa tornare allo stato di non enforcement i residence user.

```
[DEBUG] FLOW_REMOVED_HANDLER
msg.reason = 0
FLOW CONTROL request
FLOW CONTROL reply at 2016-03-02 12:50:05
flow 0 is active
flow 1 is active
flow 2 is not active
flow 2 removed from active_flows
[ 2016-03-02 12:50:05 ]: flows active: [0, 1]
Non-Enforcement State
[DEBUG] Entering N state for all flows...
```

FIGURA 35:MESSAGGIO DI FLOW_REMOVED AL CONTROLLER.

Con riferimento alla figura 35, si può notare che il flow_id relativo all'utente business vale due.

Con questo l'esempio termina, vengono ora mostrati gli screen relativi al traffico dei tre utenti:

```
ubuntu@ru:~$ iperf -t 300 -i 2 -c 10.250.0.117 -p 5001
------------------------------------------------------------
Client connecting to 10.250.0.117, TCP port 5001
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.8.8 port 52023 connected with 10.250.0.117 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec   165 MBytes   692 Mbits/sec
[  3]  2.0- 4.0 sec   166 MBytes   696 Mbits/sec
[  3]  4.0- 6.0 sec   132 MBytes   556 Mbits/sec
[  3]  6.0- 8.0 sec   148 MBytes   623 Mbits/sec
[  3]  8.0-10.0 sec   145 MBytes   607 Mbits/sec
[  3] 10.0-12.0 sec   138 MBytes   579 Mbits/sec
[  3] 12.0-14.0 sec   164 MBytes   688 Mbits/sec
[  3] 14.0-16.0 sec   166 MBytes   697 Mbits/sec
[  3] 16.0-18.0 sec   166 MBytes   694 Mbits/sec
[  3] 18.0-20.0 sec   169 MBytes   709 Mbits/sec
[  3] 20.0-22.0 sec   165 MBytes   693 Mbits/sec
[  3] 22.0-24.0 sec   166 MBytes   698 Mbits/sec
[  3] 24.0-26.0 sec   168 MBytes   704 Mbits/sec
[  3] 26.0-28.0 sec   167 MBytes   702 Mbits/sec
[  3] 28.0-30.0 sec   166 MBytes   696 Mbits/sec
[  3] 30.0-32.0 sec   104 MBytes   437 Mbits/sec
[  3] 32.0-34.0 sec  96.5 MBytes   405 Mbits/sec
[  3] 34.0-36.0 sec  94.9 MBytes   398 Mbits/sec
[  3] 36.0-38.0 sec  99.6 MBytes   418 Mbits/sec
[  3] 38.0-40.0 sec  93.6 MBytes   393 Mbits/sec
[  3] 40.0-42.0 sec  90.8 MBytes   381 Mbits/sec
[  3] 42.0-44.0 sec  93.4 MBytes   392 Mbits/sec
[  3] 44.0-46.0 sec  90.8 MBytes   381 Mbits/sec
[  3] 46.0-48.0 sec  89.4 MBytes   375 Mbits/sec
[  3] 48.0-50.0 sec  89.1 MBytes   374 Mbits/sec
[  3] 50.0-52.0 sec  88.1 MBytes   370 Mbits/sec
[  3] 52.0-54.0 sec  89.5 MBytes   375 Mbits/sec
[  3] 54.0-56.0 sec  98.9 MBytes   415 Mbits/sec
[  3] 56.0-58.0 sec  84.2 MBytes   353 Mbits/sec
[  3] 58.0-60.0 sec  48.8 MBytes   204 Mbits/sec
[  3] 60.0-62.0 sec  49.8 MBytes   209 Mbits/sec
[  3] 62.0-64.0 sec  48.6 MBytes   204 Mbits/sec
[  3] 64.0-66.0 sec  51.5 MBytes   216 Mbits/sec
[  3] 66.0-68.0 sec  67.1 MBytes   282 Mbits/sec
[  3] 68.0-70.0 sec  64.1 MBytes   269 Mbits/sec
[  3] 70.0-72.0 sec  63.6 MBytes   267 Mbits/sec
[  3] 72.0-74.0 sec  32.0 MBytes   134 Mbits/sec
[  3] 74.0-76.0 sec  10.2 MBytes  43.0 Mbits/sec
[  3] 76.0-78.0 sec  11.4 MBytes  47.7 Mbits/sec
[  3] 78.0-80.0 sec  12.4 MBytes  51.9 Mbits/sec
[  3] 80.0-82.0 sec  14.2 MBytes  59.8 Mbits/sec
[  3] 82.0-84.0 sec  15.0 MBytes  62.9 Mbits/sec
[  3] 84.0-86.0 sec  11.1 MBytes  46.7 Mbits/sec
[  3] 86.0-88.0 sec  11.6 MBytes  48.8 Mbits/sec
[  3] 88.0-90.0 sec  11.6 MBytes  48.8 Mbits/sec
[  3] 90.0-92.0 sec  62.1 MBytes   261 Mbits/sec
[  3] 92.0-94.0 sec  94.9 MBytes   398 Mbits/sec
[  3] 94.0-96.0 sec  92.4 MBytes   387 Mbits/sec
```

FIGURA 36:IPERF RESIDENCE 1.

Dalla figura 36 si può notare la limitazione della banda dovuta alla fase di enforcement, al secondo 74. La banda viene limitata a 100 Mbits/sec, ma visto che i res user sono due, entrambi se la dividono.

```
ubuntu@ru:~$ iperf -t 300 -i 2 -c 10.250.0.117 -p 5002
------------------------------------------------------------
Client connecting to 10.250.0.117, TCP port 5002
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.8.8 port 41140 connected with 10.250.0.117 port 5002
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  71.1 MBytes   298 Mbits/sec
[  3]  2.0- 4.0 sec  70.5 MBytes   296 Mbits/sec
[  3]  4.0- 6.0 sec  68.5 MBytes   287 Mbits/sec
[  3]  6.0- 8.0 sec  73.8 MBytes   309 Mbits/sec
[  3]  8.0-10.0 sec  72.1 MBytes   303 Mbits/sec
[  3] 10.0-12.0 sec  72.6 MBytes   305 Mbits/sec
[  3] 12.0-14.0 sec  75.9 MBytes   318 Mbits/sec
[  3] 14.0-16.0 sec  77.5 MBytes   325 Mbits/sec
[  3] 16.0-18.0 sec  78.9 MBytes   331 Mbits/sec
[  3] 18.0-20.0 sec  78.9 MBytes   331 Mbits/sec
[  3] 20.0-22.0 sec  77.8 MBytes   326 Mbits/sec
[  3] 22.0-24.0 sec  76.9 MBytes   322 Mbits/sec
[  3] 24.0-26.0 sec  69.0 MBytes   289 Mbits/sec
[  3] 26.0-28.0 sec  60.2 MBytes   253 Mbits/sec
[  3] 28.0-30.0 sec  49.8 MBytes   209 Mbits/sec
[  3] 30.0-32.0 sec  51.9 MBytes   218 Mbits/sec
[  3] 32.0-34.0 sec  53.2 MBytes   223 Mbits/sec
[  3] 34.0-36.0 sec  53.1 MBytes   223 Mbits/sec
[  3] 36.0-38.0 sec  46.8 MBytes   196 Mbits/sec
[  3] 38.0-40.0 sec  50.9 MBytes   213 Mbits/sec
[  3] 40.0-42.0 sec  53.0 MBytes   222 Mbits/sec
[  3] 42.0-44.0 sec  23.6 MBytes  99.1 Mbits/sec
[  3] 44.0-46.0 sec  8.25 MBytes  34.6 Mbits/sec
[  3] 46.0-48.0 sec  12.5 MBytes  52.4 Mbits/sec
[  3] 48.0-50.0 sec  12.9 MBytes  54.0 Mbits/sec
[  3] 50.0-52.0 sec  7.12 MBytes  29.9 Mbits/sec
[  3] 52.0-54.0 sec  10.1 MBytes  42.5 Mbits/sec
[  3] 54.0-56.0 sec  11.8 MBytes  49.3 Mbits/sec
[  3] 56.0-58.0 sec  11.2 MBytes  47.2 Mbits/sec
[  3] 58.0-60.0 sec  11.2 MBytes  47.2 Mbits/sec
[  3] 60.0-62.0 sec  41.5 MBytes   174 Mbits/sec
[  3] 62.0-64.0 sec  70.0 MBytes   294 Mbits/sec
[  3] 64.0-66.0 sec  74.0 MBytes   310 Mbits/sec
[  3] 66.0-68.0 sec  76.9 MBytes   322 Mbits/sec
[  3] 68.0-70.0 sec  75.9 MBytes   318 Mbits/sec
[  3] 70.0-72.0 sec  78.0 MBytes   327 Mbits/sec
[  3] 72.0-74.0 sec  68.6 MBytes   288 Mbits/sec
[  3] 74.0-76.0 sec  66.1 MBytes   277 Mbits/sec
[  3] 76.0-78.0 sec  66.1 MBytes   277 Mbits/sec
[  3] 78.0-80.0 sec  78.2 MBytes   328 Mbits/sec
[  3] 80.0-82.0 sec  82.9 MBytes   348 Mbits/sec
[  3] 82.0-84.0 sec  85.8 MBytes   360 Mbits/sec
[  3] 84.0-86.0 sec  57.6 MBytes   242 Mbits/sec
^C[  3]  0.0-87.1 sec  2.44 GBytes   241 Mbits/sec
```

FIGURA 37:IPERF RESIDENCE 2

```
ubuntu@bu:~$ iperf -t 200 -i 1 -c 10.250.0.117 -p 5003
------------------------------------------------------------
Client connecting to 10.250.0.117, TCP port 5003
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.8.80 port 43176 connected with 10.250.0.117 port 5003
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec  40.4 MBytes   339 Mbits/sec
[  3]  1.0- 2.0 sec  33.5 MBytes   281 Mbits/sec
[  3]  2.0- 3.0 sec  32.2 MBytes   271 Mbits/sec
[  3]  3.0- 4.0 sec  32.2 MBytes   271 Mbits/sec
[  3]  4.0- 5.0 sec  30.8 MBytes   258 Mbits/sec
[  3]  5.0- 6.0 sec  32.0 MBytes   268 Mbits/sec
[  3]  6.0- 7.0 sec  32.1 MBytes   269 Mbits/sec
[  3]  7.0- 8.0 sec  30.9 MBytes   259 Mbits/sec
[  3]  8.0- 9.0 sec  29.5 MBytes   247 Mbits/sec
[  3]  9.0-10.0 sec  25.5 MBytes   214 Mbits/sec
[  3] 10.0-11.0 sec  27.2 MBytes   229 Mbits/sec
[  3] 11.0-12.0 sec  28.0 MBytes   235 Mbits/sec
[  3] 12.0-13.0 sec  24.4 MBytes   204 Mbits/sec
[  3] 13.0-14.0 sec  27.1 MBytes   228 Mbits/sec
[  3] 14.0-15.0 sec  23.0 MBytes   193 Mbits/sec
[  3] 15.0-16.0 sec  36.6 MBytes   307 Mbits/sec
[  3] 16.0-17.0 sec  36.8 MBytes   308 Mbits/sec
[  3] 17.0-18.0 sec  43.2 MBytes   363 Mbits/sec
[  3] 18.0-19.0 sec  39.9 MBytes   334 Mbits/sec
[  3] 19.0-20.0 sec  42.0 MBytes   352 Mbits/sec
[  3] 20.0-21.0 sec  46.5 MBytes   390 Mbits/sec
[  3] 21.0-22.0 sec  48.9 MBytes   410 Mbits/sec
[  3] 22.0-23.0 sec  48.4 MBytes   406 Mbits/sec
^C[  3]  0.0-23.1 sec   794 MBytes   289 Mbits/sec
```

FIGURA 38:IPERF BUSINESS.

A differenza degli utenti precedenti, come possiamo vedere dalla figura 38, la banda utilizzata dall'utente business resta sempre elevata, in media di 289 Mbits/sec.

Per concludere questo capitolo vengono ora mostrati i grafici relativi all'andamento dello *Throughput* alle porte dello switch nel tempo.

Per calcolare lo throughput innanzitutto è stato necessario ottenere il numero di byte per secondo che nel tempo attraversavano le porte dello switch. Per fare ciò si è utilizzato uno script dedicato, rcvdatafw.sh, eseguito tramite il comando **sudo ./rcvdatafw.sh br-int 1**, dove 1 sta ad indicare che ogni secondo viene mostrato a schermo il numero di byte rx/tx per ogni porta dello switch.

Con rx si intende i byte che dalla virtual machine entrano nello switch, mentre con tx i byte che dallo switch entrano nella virtual machine.

Una volta ottenuti i byte per secondo, per calcolare lo throughput si è utilizzata l'espressione seguente:

$$\frac{(byte_{succ} - byte_{prec}) * 8}{10^6}$$

Dove $byte_{prec}$ indica i byte rx o tx al tempo n, mentre $byte_{succ}$ al tempo n + 1. Con questa espressione si ricava il valore dello throughput in Mb/Sec.

Il primo grafico che viene mostrato è quello relativo al DPI, dove si possono notare le tre fasi di classificazione dei flussi:

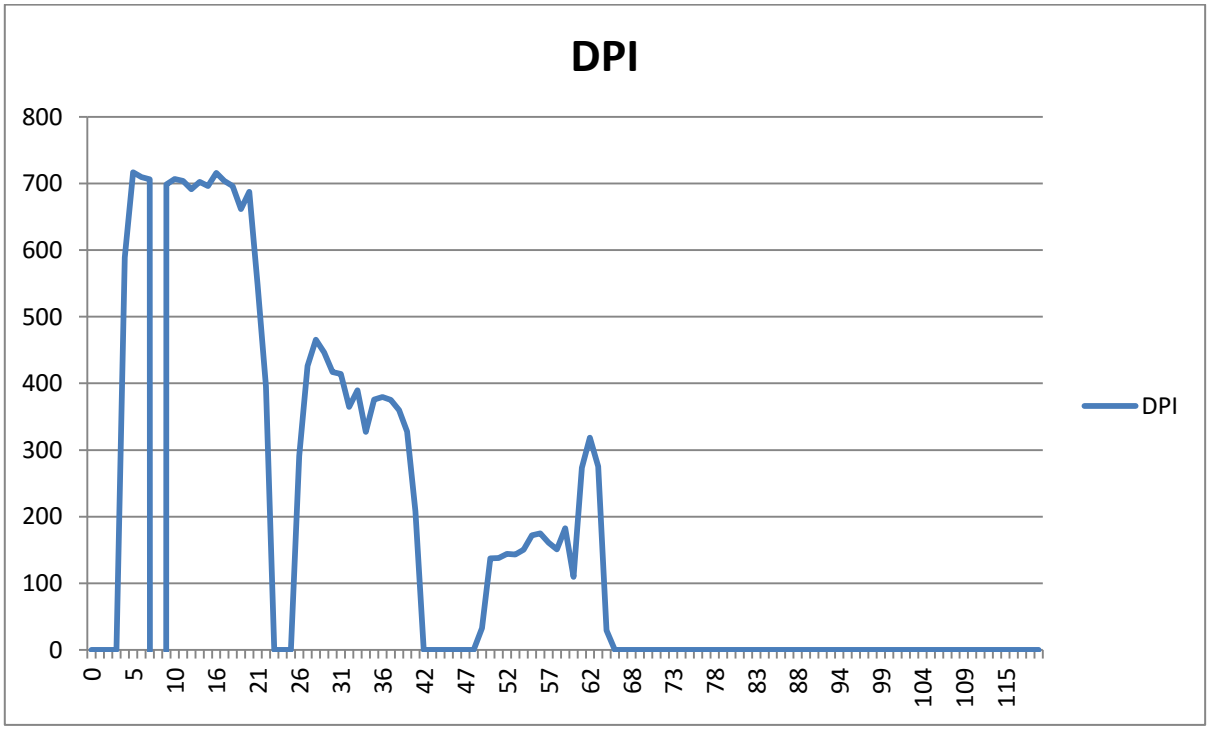

FIGURA 39:ANDAMENTO NEL TEMPO DELLO THROUGHPUT ALLE PORTE DEL DPI

In ascisse è presente il tempo, calcolato tramite lo script *gettimestamp.sh* richiamato all'interno dello script precedente, mentre in ordinate lo throughput in Mb/Sec. Dall figura 39 si può notare che ogni fase di classificazione ha durata di circa 15 secondi come ci aspettavamo.

Infine viene mostrato il grafico contenente lo throughput relativo al traffico generato dagli utenti e dal TC nella fase di enforcement:

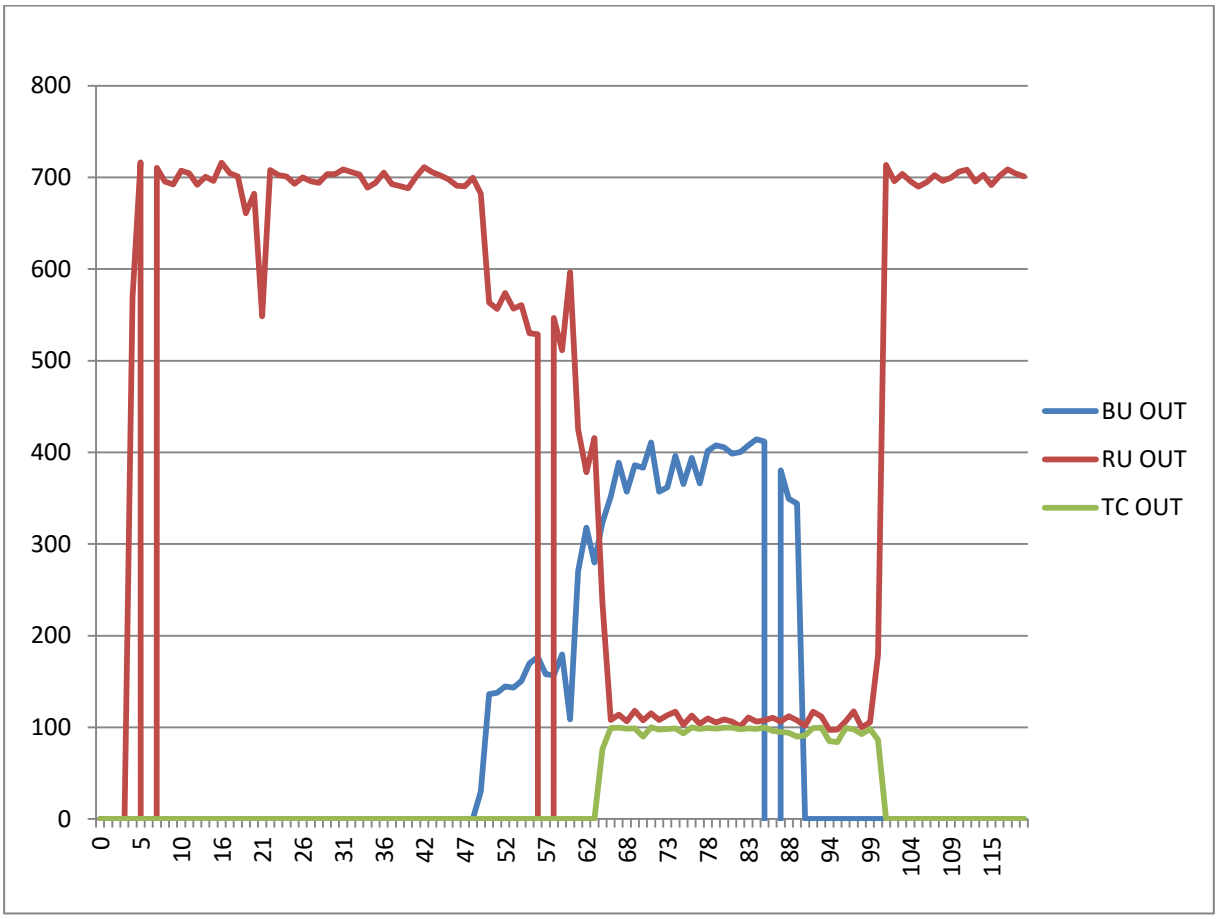

FIGURA 40:ANDAMENTO NEL TEMPO DELLO THROUGHPUT ALLE PORTE DEI TRE UTENTI E DEL TC

Con riferimento alla figura 40 si nota che inizialmente la banda è completamente occupata dall'utente residence. Questa situazione resta invariata fino all'ingresso dell'utente business a circa 48 secondi dall'inizio del test, che determina, prima che venga classificato, una spartizione della banda.

Al termine della fase di classificazione dell'utente prioritario, circa 65 secondi dall'inizio (con riferimento anche alla figura 39), il controller fa passare i flussi nello stato di enforcement, per cui la bit rate dell'utente residence cala drasticamente e viene limitata dal TC a 100 Mb/Sec, mentre quella dell'utente business aumenta. In realtà lo throughput del business dovrebbe essere maggiore, ma per ragioni di incompatibilità il WANA non funziona come dovrebbe, ma implementa il semplice forwarding.

CAPITOLO 6

Conclusioni

Al giorno d'oggi Internet risulta sempre più parte integrante della quotidianeità di tutti, questo grazie all'ampio sviluppo tecnologico delle reti di telecomunicazioni degli ultimi anni. Ora però, per colpa dell'ossificazione della rete i ricercatori vanno spesso incontro a grandi difficoltà per quanto riguarda la sperimentazione di nuove tecnologie.

I paradigmi SDN e NFV rappresentano una soluzione tangibile alla staticità della rete odierna, portando vantaggi sia ai gestori di rete che ai ricercatori, consentendo ai primi di abbattere i costi (molto elevati per colpa delle middle-boxes vendor dependant) e sviluppare nuove offerte di rete grazie all'elasticità della stessa, e dando la possibilità ai secondi di sviluppare e mettere in pratica velocemente le nuove idee.

I test pratici visti nello scorso capitolo hanno dimostrato, che l'implementazione del protocollo OpenFlow sul cluster OpenStack rende la rete fortemente dinamica, ma anche che il codice richiede ancora tanto lavoro, in quanto il controller Ryu funziona come dovrebbe finchè si applica l'esempio visto, altrimenti si comporta in maniera errata. Risulta quindi fondamentale passo dopo passo rendere il codice sempre più generale e performante, in modo che in un futuro non troppo lontano la rete possa implementare qualsiasi tipo di topologia reale.

CAPITOLO 7

Ringraziamenti

Arrivati a questo punto mi sembra doveroso ringraziare coloro che mi hanno sostenuto durante il percorso universitario.

Prima di tutto voglio ringraziare la mia famiglia, le mie due famiglie in realtà: da una parte mia mamma Carla, suo marito Maurizio in arte mauri, mio fratello Luca e mia nonna Teresa e dall'altra mio babbo Werter, sua moglie Cristina, in arte cri e mia nonna Mariella. Senza il loro supporto, sia affettivo che monetario, tutto questo non sarebbe stato possibile.

Voglio inoltre ringraziare i miei amici, che nei miei momenti di crisi hanno sempre trovato il modo per farmi distrarre, grazie a loro questi anni di università sono passati molto velocemente. In particolare vorrei ringraziare Fabio e Leo che da 15 anni mi supportano e sopportano.

Infine un grandissimo ringraziamento va a Walter Cerroni e Chiara Contoli, per avermi seguito durante tutto il percorso della tesi,per i loro preziosi consigli e la loro costante disponibilità.

Bibliografia

[1] Wikipedia. Cloud Computing: https://en.wikipedia.org/wiki/Cloud_computing.

[2] Wikipedia. Network functions virtualization: https://en.wikipedia.org/wiki/Network Functions Virtualization.

[3] Telecom Italia. Notiziario tecnico: SDN e NFV: quali sinergie? : http :// www .telecomitalia .com /tit /it/ notiziariotecnico /numeri /2014 -2/ capitolo -05. html.

[4] Wikipedia. Software-defined networking. http://en.wikipedia.org/wiki/Software-dened networking.

[5] F. Foresta, Composizione dinamica di funzioni di rete virtuali in ambienti Cloud, Alma Mater Studiorum, ingegneria elettronica informatica e delle telecomunicazioni, 2015.

[6] The European Telecommunications Standards Institute. Network functions virtualization: An introduction, benets, enablers, challenges & call for action. In ETSI White Paper, editor, SDN and OpenFlow WorldCongress, Ottobre 2012. Darmstadt-Germany.

[7] Information and Communication technology Brazil-Israel.Network Functions Virtualization. http://vega-bi.blogspot.it/2014/08/nfv-network-functions-virtualization.html.

[8] Yong Li and Min Chen, Software-Defined Network Function Virtualization: A Survey, IEEE Access, vol. 3, pp. 2542-2553, 2015.

[9] Intel Data Plane Development Kit, Intel, Santa Clara, CA, USA, 2015.

[10] J. Hwang, K. K. Ramakrishnan, and T. Wood, ‘’NetVM: High performance and exible networking using virtualization on commodity platforms,’’ in Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2014, pp. 1-12.

[11] 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.

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

[13] G. Santandrea. OpenStack: network internals.

[14] G. Santandrea. Show my network state project website, 2014.

[15] Wikipedia. Openvswitch. http://en.wikipedia.org/wiki/Open vSwitch.

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

[17] HP Networking. OpenFlow: Rendere possibile la tecnologia di rete software-defined (SDN). http://pro-networking-h17007.external.hp.com/it/it/solutions/technology/openflow/index.aspx.

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

[19] L. Ponti, Virtualizzazione di funzioni di rete su piattaforme per cloud computing, Alma Mater Studiorum, ingegneria elettronica informatica e delle telecomunicazioni, 2015.

[20] Greg Sowel Consulting.OpenFlow and Mikrotik. http://gregsowell.com/?p=4442.

[21] FlowProgrammable. Message Layer. http://flowgrammable.org/sdn/openflow/message-layer/#tab_ofp_1_3.

[22] Fujita Tomonori. Introduction to Ryu SDN framework. NTT, Software Innovation Center. 2013. http://osrg.github.io/ryu/slides/ONS2013-april-ryu-intro.pdf.

[23] Kei Ohmura. OpenStack/Quantum SDNbased network virtulization with Ryu. NTT, Software Innovation Center. Maggio 2013. http://osrg.github.io/ryu/slides/LinuxConJapan2013.pdf.

[24] Ryu Project Team. Ryu SDN framework (Using OpenFlow 1.3). https://osrg.github.io/ryu-book/en/Ryubook.pdf.

[25] F. Callegati, W. Cerroni, C. Contoli, G. Santandrea, Dynamic Chaining of Virtual Network Functions in Cloud-Based Edge Networks.

[26] Ericsson Cloud Lab e Università di Bologna, EIT Digital – SDN@Edge UNIBO-ERICSSON joint experiment on dynamic service chaining.